

Черняев Александр

ДЕНДИ КОД



Как писать код с аккуратностью, уважением
к читателю и стилем — даже если вы
новичок

ДЕНДИ-КОД

Рабочий код сам по себе не повод для гордости. Его можно написать быстро и наспех. Но код, с которым действительно приятно работать, требует внимания, аккуратности и внутренней дисциплины.

Почему «Денди»?

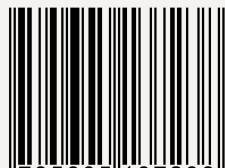
Потому что код — это не просто инструкции для машины, а твоя репутация. Как денди следит за стилем, так и ты должен следить за кодом — он должен быть аккуратным, понятным и держать марку. Без грязи, без костылей, без «потом исправлю».

О книге

Эта книга — практический справочник в формате cookbook. В каждой главе — примеры типичных ошибок и более удачные решения, которые помогут писать чище и понятнее.

**Не будь просто разработчиком.
Будь денди.**

ISBN 978-5-6051372-3-8



9 785605 137238 >

УДК 004.415

ББК 32.973.201

Ч49

ДЕНДИ-КОД

Черняев Александр

ISBN 978-5-6051372-3-8

Коммит 5761c81

Липецк, 2025

Формат: 148×210 мм (А5)

Бумага: офсетная 80 г/м², печать цифровая

Шрифты: EB Garamond, JetBrains Mono

Тираж: 76 экз. | Заказ № 140574

Издательство: Полиграфия «Колор», ИП Москалёва Э. В.

Содержание

Предисловие	6
Всё начинается с README	11
Великий монолит	21
Код как средство коммуникации	25
Форматирование	31
Код должен дышать	38
Именованье	43
Магические значения	62
Размер имеет значение	71
Без лишних движений	78
Ранний выход	95
Управляющие конструкции	102
Аргументы	112
Обработка ошибок	121
Комментарии	132

Не бойся удалять код	138
Тесты	143
Играй по правилам	157
Не отказывайтесь от будущего	162
Второй пилот — не капитан	165
Послесловие	174



Предисловие

Одна из трудностей, с которой мы сталкиваемся, — это поиск кратких и содержательных выжимок без лишней воды. Часто приходится выбирать между чтением громоздких книг по 500 страниц или бесконечным пролистыванием блогов, лент и статей в интернете. Это разочарование подтолкнуло меня к созданию сжатой, практической книги — о том, как писать понятный, опрятный и уважающий коллег код. Всё — на конкретных примерах, преимущественно с использованием PHP.

Тема качественного кода сложна и многогранна. Некоторые её аспекты неизбежно вызывают споры и сводятся к вопросам вкуса и личных предпочтений. Такие вопросы в книге опущены. В центре внимания — конкретные, проверенные практикой правила, которые помогают писать опрятный на вид код.

Важно понимать, что темы чистого кода, архитектуры и объектно-ориентированного программирования — это по-настоящему глубокие области. Углубляться в них можно бесконечно. Однако в этой книге мы сознательно сосредотачиваемся на визуальной читаемости и внешней аккуратности кода. Почему? Потому что именно с этого всё начинается.

Это как человек-денди: он входит в комнату — и ты сразу замечаешь, что он ухожен, аккуратен, элегантен. Ты ещё не знаешь, о чём он будет говорить, но уже понимаешь — перед тобой тот, кто уважает себя и окружающих.

Так и код: даже если ты не вникал в его логику, по внешнему виду видно, что автору не всё равно. Это уважение — к себе, к коллеге, к команде.

Эта книга — о том, как сделать код чуть приятнее глазу. И пусть это станет вашим первым шагом к более глубокому пониманию чистоты и структуры.

Для кого предназначена эта книга

Предполагается, что читателем книги в первую очередь станет специалист начального уровня, уже принимавший участие хотя бы в одном коммерческом, личном или учебном проекте. Но и опытные специалисты найдут здесь полезные идеи — особенно если их профессиональный путь был связан с одной командой, похожими задачами и устойчивыми привычками. Она поможет лучше понять, как работать в команде и избегать раздражения у коллег. Для руководителей она будет полезна как пособие для команды и как источник рекомендаций при ревью кода.

Если вы сомневаетесь, «Подходит ли это мне?», попробуйте ответить на короткий чек-лист:

- ☐ Я имею базовые знания программирования.
- ☐ Я участвовал в создании хотя бы одного коммерческого, личного или учебного проекта.
- ☐ Я хочу улучшить свой код и работать в команде более эффективно.

Кому не подойдёт эта книга

Если вы ищете глубокое погружение в теорию объектно-ориентированного программирования, архитектуру приложений или сложные паттерны проектирования, возможно, вам стоит начать с трудов Мартина Фаулера, Роберта Мартина, Стива Макконнелла и других авторов.

Здесь вы не найдёте пространных рассуждений о том, почему `static` — это зло, `null` — ошибка на миллиард долларов, или как строго следовать принципам SOLID. Мы не будем явно обсуждать coupling и cohesion, рассчитывать показатели maintainability или разбирать уровни абстракций.

Вместо этого книга предлагает практические рекомендации, основанные на существующих идеях и личном опыте, — чтобы помочь вам писать понятный, последовательный и дружелюбный к другим код в реальных проектах.

Как извлечь максимум из этой книги

Вы можете прочитать её залпом за вечер, но вряд ли это принесёт ощутимую пользу. Гораздо эффективнее — читать по одной-две главы за раз. Применять на практике. Обсуждать с коллегами. Спорить. Переделывать старый код. И главное — почувствовать разницу.

Если спор зашёл в тупик или у вас возник вопрос по содержанию книги, пожалуйста, создайте issue в GitHub-репозитории: <https://github.com/tabuna/dandy-code>

Искренне надеюсь, что чтение будет для вас интересным и полезным — и что эта книга поможет строить приложения проще, чище и эффективнее. Спасибо, что читаете.

Рецензенты и консультанты

Эта книга прошла проверку рядом специалистов, чей опыт и знания помогли повысить её качество. Они поддерживали меня, делились обратной связью, участвовали в обсуждениях и внимательной вычитке текста. Именно благодаря их вкладу книга стала лучше. Я с удовольствием увековечиваю их имена на этих страницах:

Владислав Абросимов, Дмитрий Афанасьев, Игорь Бабаев, Егор Бугаенко, Зайнулла Жумаев, Иван Сорокин, Сергей Ивлев, Денис Лопухин, Сергей Предводителей, Денис Тарков, Евгений Уткин, Андрей Хэллдар, Владимир Чепелев.

Особые слова благодарности я хочу выразить моим друзьям:

- Илье Чубарову
- Дмитрию Скирте

Хватит предисловий. Пора наводить порядок.



Всё начинается с README

Первое, что видит разработчик, открывая репозиторий — это небольшой файл, лежащий в корне проекта под именем **README**, именно он формирует первое впечатление о коде и подходе команды к работе. Пустой или отсутствующий **README** — красноречивый сигнал: с этим проектом, скорее всего, будет непросто.

В open-source это особенно важно: если документация неясна, вряд ли кто-то захочет разбираться, не говоря уже о внесении вклада. В коммерческой разработке мотивацией служит зарплата, но это не означает, что вход в проект должен быть запутанным и неприятным.

На практике **README** часто либо отсутствует, либо существует лишь как формальность. Такое встречается в репозиториях компаний любого масштаба.

Например, в корне может лежать пустой или шаблонный файл **README**, созданный автоматически при инициализации — и на этом всё. Иногда файл содержит только заголовок **# ProjectName**, без единого слова о сути проекта. А в худшем случае **README** вообще нет, зато рядом находятся десятки разрозненных скриптов и несколько почти одинаковых конфигурационных файлов: **config_old.php**, **config_bak.php**, **config_real_final.php**.

Что с этим делать? Начать с простого — представить, что репозиторий открыл разработчик, который впервые видит этот проект. Что ему нужно знать в первую очередь? Вот минимальный список разделов, которые стоит включить:

Описание проекта

Несколько строк, что делает проект и зачем он нужен. Без маркетинга, по делу. Помогите читателю быстро понять, о чём речь.

Weather – сервис для приёма погодных данных через REST API, их обработки и дополнения вычисленными показателями. Итоговые агрегированные отчёты доступны через REST API и веб-интерфейс для просмотра и анализа.

Если проект достаточно крупный, добавьте ссылки на сопутствующую документацию — это сильно упростит жизнь тем, кто только начинает с ним работать:

- Staging/dev-окружение (URL)
- API-документация (например, Swagger)
- Описание CI/CD pipeline
- Отчёт о покрытии кода

Это минимизирует вопросы и сделает процесс вхождения в проект значительно быстрее и комфортнее.

Установка и запуск

Чёткие инструкции по установке зависимостей и запуску проекта локально или в тестовой среде. Команды должны быть проверены и воспроизводимы.

Пример:

```
make install  
make up
```

Если не предусмотрена возможность запуска проекта локально, то укажите, как получить доступ к персональному тестовому окружению или выделенному стенду разработчика.

Вполне нормально, что проект после установки будет чистым/голым, без каких-либо данных. Но разработчику нужно починить баг или внести изменения. Заставлять его заполнять какие-либо данные вручную — плохая практика. Скорее всего, они будут неполными, а возможно, и запутанными — например, иметь названия «Test» или «Test 1» для первого и второго пользователя.

Вместо этого предоставьте и задокументируйте на этом этапе возможность разработчику заполнить тестовые данные автоматически, например с помощью команды вида:

```
make seed  
make reset-db
```

Если тестовые данные генерируются, то укажите, как это сделать. Если они импортируются из файла, то укажите, где его взять и как использовать.

Тестирование

Если в проекте есть тесты — это прекрасно. Но мало просто иметь их, важно объяснить, как их запускать.

Пример:

```
# Для запуска всех тестов:  
vendor/bin/phpunit  
  
# или для запуска тестов в определённой тестовой группе:  
vendor/bin/phpunit --testsuite=Browser
```

Опишите, какие тесты есть (юнит, интеграционные), где их искать и какие требования нужны для запуска.

Структура каталогов

Хорошая структура — это договорённость, благодаря которой каждый член команды с первой секунды понимает, куда пойти за нужным кодом и куда сохранить новый. Представьте, что вы пришли в библиотеку, где книги разбросаны по полу: поиск нужного тома займёт вечность. То же самое и с проектом: без чёткого каталога любой свой вклад вы будете оформлять в нервном режиме.

Допустим, вы работаете над внутренним проектом Weather — платформой для анализа метеоданных. Вам поручили реализовать класс, вычисляющий лунную фазу, чтобы добавить его в блок астрономического прогноза.

В какой директории вы создадите этот класс:

- В `components`?
- В `modules`?
- В `services`?
- А может, в `utils`?

Если у вас нет описания структуры — быстрого ответа вы не найдёте.

Потрудитесь коротко объяснить, что в них лежит. Даже если вам кажется, что «и так понятно»:

```
project
├ components // Переиспользуемые куски UI
├ modules    // Отдельные бизнес-модули (оплата, доставка)
├ services    // Работа с API и хранилищами
└ utils      // Вспомогательные функции без состояния
```

Это уберёжёт от вопросов в духе: «а куда это класть?» или ещё хуже — от ситуации, когда каждый кладёт куда ему хочется. Это не только про порядок — это способ синхронизировать мышление всей команды.

Задайте жёсткую структуру. Мягкие договорённости не работают. Слова вроде «примерно тут», «по смыслу ближе сюда», «у нас гибкий подход» — признак инженерной слабости. Структура либо определена, либо её нет.

А если ваш репозиторий большой и над ним работают несколько команд или отделов, рассогласованность в структуре — не исключение, а скорее норма. Даже если код работает, поддерживать и развивать его в таких условиях всё сложнее.

Посмотрите, как может выглядеть типичная «естественно выросшая» структура:

```
repository
├── core
│   ├── cfg
│   ├── lib
│   └── domain
├── dashboard
│   ├── components
│   ├── conf
│   └── stuff
├── api
│   ├── config
│   ├── handlers
│   └── logic
└── cli
    ├── etc
    └── src
```

Каждый проект организован по-своему. Где-то **config**, где-то **conf**, где-то **cfg**, где-то **etc**. В одном месте **handlers**, в другом — **logic**, в третьем — **lib**. Даже если каждый разработчик понимает свою часть, **в целом репозиторий превращается в поле догадок.**

Создание единой структуры каталогов помогает всей команде **наглядно увидеть разногласия и перейти к общей договорённости.** Вместо бессистемного подхода появляется чёткая архитектура, в которой каждый понимает, где что находится и зачем.

Сравните с вариантом, где команды **договорились о едином стиле**:

```
repository
├─ weather-core
│   ├── config
│   ├── modules
│   └─ ...
├─ weather-dashboard
│   ├── config
│   ├── ui
│   └─ ...
├─ weather-api
│   ├── config
│   ├── routes
│   └─ services
└─ weather-cli
    ├── config
    └─ commands
```

В такой структуре каталоги становятся не просто способом хранить код, а **единым языком команды**. Всё предсказуемо, согласовано и масштабируется без лишних вопросов. Подключение новых разработчиков, автоматизация, CI/CD, документация — всё это упрощается, когда структура работает на вас, а не против.

Но даже если вы не сможете договориться — это тоже хорошо. Это значит, что между вами нет общего архитектурного видения. А значит, и не должно быть общего репозитория.

Ответственные лица

Каждый проект, как самолёт, должен иметь экипаж. Особенно если это корпоративная разработка, где репозиторий в десятки или сотни. Открыв README, любой разработчик должен сразу понимать: кто отвечает за этот код и к кому можно подойти с вопросом, предложением или проблемой.

Укажите одного или нескольких мейнтейнеров — это может быть ведущий разработчик, архитектор или просто человек, который хорошо знает проект и готов принимать решения. Добавьте способ связи: email или внутреннюю ссылку на профиль.

Это может казаться формальностью, но на самом деле — это фундамент доверия и ответственности. Проект, под которым стоит имя, внушает уважение. Даже у новичков появляется ощущение, что этот код — не брошен. Его кто-то любит. За него кто-то отвечает.

В советских КБ — будь то Ильюшина, Туполева или Сухого — под каждым самолётом стояло имя главного конструктора.

Когда ты ставишь под проект своё имя, он перестаёт быть просто папкой с файлами. Он становится частью тебя. Это меняет отношение к деталям. Люди начинают не просто писать код — они становятся авторами.

Такой подход работает на всех уровнях. Люди гордятся кодом, за который отвечают. Они вовлечены. Они стараются. Не потому что кто-то требует, а потому что на проекте теперь есть лицо.

Когда имя ответственного указано прямо в репозитории, не нужны матрицы компетентности, диаграммы HR и долгие поиски «а кто в этом шарит». Всё видно сразу: имя есть — значит, человек в теме. Имени нет — не трогай, найди владельца.

Ответственный	Контакт	Статус
-----	-----	-----
@ivanov	ivanov@corp	active
@smirnov	Jira: UI-32	maintenance
—	—	archived

*В **README** написано **Owner: @petrov** — значит, Петров отвечает. Не «вроде Петров», не «Петров что-то там писал», не «спрашивай у Петрова». Он указан — он и есть интерфейс проекта.*

Ответственный — это не контролёр. Это интеграционная точка. Он не обязан всё чинить или лично писать весь код. Но именно он может подсказать, помочь или принять решение.

В идеале, если в проекте уже есть файл **CODEOWNERS**, который используется не только для понимания кода, но и для автоматизации — назначения ревьюверов, интеграции с CI, поддержки документации, — стоит либо встроить ключевую информацию из него в **README**, либо просто сослаться на него.



Великий монолит

После прочтения **README** в любом проекте возникает естественное желание оценить масштаб работы. Насколько велика система? Насколько быстро я смогу в ней разобраться и начать вносить изменения? Эти вопросы важны не только для новичка, но и отражают уровень прозрачности архитектуры и зрелости проекта.

Во многих экосистемах по умолчанию доминирует монолитный подход. Это классика, проверенная временем.

Его по умолчанию используют:

- Laravel (PHP)
- Django (Python)
- Ruby on Rails (Ruby)
- Phoenix Framework (Elixir)
- Spring (Java)
- Sails (Node.js)

И этот список можно продолжать долго — монолиты надёжны, удобны и, что важно, имеют широкую поддержку в виде инструментов и сообществ. Это безопасная отправная точка практически для любого проекта.

Однако столкнуться с **20 000+** файлов и папок в репозитории — для любого разработчика может стать демотивирующим шоком. Это словно стоять перед огромным валуном и пытаться его сдвинуть с места.

Даже опытный разработчик теряет мотивацию, когда не видит чёткой структуры, границ ответственности, понятных точек входа. Это не просто психологический барьер — это профессиональная фрустрация: ты не понимаешь, где начать, как ничего не сломать, как внести изменения безопасно.

В любой работе, будь то программный код, список дел или физическая работа — есть одно универсальное правило: разбивай большую задачу на маленькие части.

Это вовсе не значит, что каждый монолит обязательно нужно дробить на десятки микросервисов. Нет, не стоит драматизировать. Речь о том, что даже внутри монолита обязательно нужно выделять и изолировать компоненты, которые можно вынести в отдельные репозитории.

Например, в приложении для прогноза погоды есть класс температуры, который может автоматически записываться как в градусах Цельсия, так и в градусах Фаренгейта, его можно выделить отдельно от монолита:

```
class Temperature
{
    // ...
}
```

Этот компонент можно вынести в собственный репозиторий, покрыть тестами, добавить документацию и подключать через Composer как внешнюю зависимость.

Такой подход упрощает основную кодовую базу и снижает когнитивную нагрузку: вместо тысячи связанных между собой файлов разработчик имеет дело с чётко очерченным, изолированным модулем.

Кроме того, переиспользуемые и опубликованные компоненты не просто сокращают дублирование — они создают эффект «внешней границы», когда ответственность модуля очевидна и проверяется временем.

Разработчику становится проще работать с проектом. Показав небольшой модуль, он скажет и подумает: «Да, я могу разобраться с этим и внести изменения», — а потом постепенно расширять понимание всего проекта и его частей. Это сильно снижает тревожность и прокрастинацию. Чем понятнее и локальнее задача — тем выше вовлечённость.

Кроме того, отдельные небольшие пакеты часто можно и нужно выкладывать в open source — это не то, что подпадает под ограничения или коммерческую тайну. Это чистый, полезный, часто общепринятый код.

Они могут стать отличным инструментом для профессионального роста и демонстрации своих навыков.

Часто можно встретить талантливых разработчиков, которые годами работают внутри одной компании, в огромном монолите, но не могут продемонстрировать ни одной строчки своего кода. Почему? Потому что весь их труд спрятан за корпоративным VPN, внутри безликой и плохо структурированной массы.

Поэтому небольшие компоненты и пакеты — это одновременно и технический, и карьерный инструмент, который стоит использовать каждому разработчику.



Код как средство коммуникации

Помните, что вы делали вчера? А позавчера? Четыре дня назад? Месяц назад? Вряд ли вы сможете дословно вспомнить, что мелькало на 69-й странице вашей любимой книги или какие точные реплики звучали в той серии того сериала. И это нормально — наш мозг сохраняет только то, что считает действительно важным, а всё остальное отбрасывает, чтобы не перегружаться.

Точно так же обстоят дела и с кодом. Когда вы возвращаетесь к проекту, чтобы починить баг или добавить новую функциональность, вы сначала читаете код заново: знакомые приёмы встречаются, но в целом вы анализируете каждую строчку.

Часто можно услышать, что код — это средство общения с компьютером: набор инструкций, например: «положи из X в Y». Но после компиляции в байт-код и многочисленных оптимизаций от вашего исходника остаётся лишь упрощённый набор команд. Машине безразличны стиль, комментарии и даже длина имён — она выполнит любую версию кода.

А вот людям всё это куда важнее, поэтому правильнее сказать, что код мы пишем не только для компьютера (хотя это тоже важно), но и для коллег, для себя и для будущего «Я».

История знает немало исследований: например, ещё в 1989 году в IBM подсчитали, что больше половины рабочего времени разработчиков уходит не на новое кодирование, а на чтение и понимание чужого кода. Поэтому чем чётче выражена мысль в коде, тем быстрее команда поймет, как он работает, тем проще сопровождать и развивать проект.

Давайте сменим установку: думать не о том, как код поймёт машина, а о том, как его воспримут коллеги и мы сами в будущем.

Код — это тоже текст

Наша работа — это умственный труд, результатом которого становятся тексты: код, документация, сообщения в коммитах и комментарии. Вокруг нас много примеров того, как может быть написан текст по-разному: Научная статья — строгая. Пост в блоге — разговорный. Сообщение в мессенджере — короткое и по делу. Код — это тоже текст, просто со своей спецификой. И у него тоже есть читатель.

Чтобы понять, насколько важно, **как написан текст**, представьте себе такое письмо:

Здрастуйте Я ПРОГРАМИСТ с БОЛЬШИМ опытом делал сайты и приложения для интирнета. Пишу код на джаве и пхп еще умею с питон и руби. Работал в разных места делал всё што гаварили. Харашо разбираюсь в компьютерах и серверах, могу делать апдейт и чинить баги. Если вазьмёте меня, не пажселеете, я осен-осен стараюс.

Дочитывать не хочется, а приглашать на собеседование — тем более. Даже если письмо будет оформлено без ошибок, оно всё равно может произвести не лучшее впечатление:

Многоуважаемый господин! Сим доношу, что имею значительный опыт в делах программных: создавал сайты и приложения для сети, пишу на Java и PHP, равно как и с Python и Ruby знаком не понаслышке. Служил в разных местах, поручения исполнял добросовестно. В компьютерах и серверах разбираюсь, умею обновлять системы и исправлять неполадки. Аще примете — не пожалеете. Труд люблю, дело своё знаю.

Теперь это грамотно. Но стиль — перегружен, устаревший и странно напыщенный. Такое письмо вызывает уже не раздражение, а недоумение.

Теперь представьте: это не кандидат, а **ваш код**. Что будет, когда два таких специалиста объединятся в одну команду? Скорее всего, в кодовой базе появится нечто подобное:

Здрастуйте, господин! Сим доношу, что Я ПРОГРАМИСТ с БОЛЬШИМ опытом делал сайты и приложения для сети. Пишу код на Java и PHP еще умею с питон и рубли не понаслышке. В компьютерах и серверах разбираюсь, Харашо разбираюсь в компьютерах и серварах, могу делать андейт и чинить баги. Если вазьмёте меня, не пажелеете. Труд люблю, дело своё знаю.

Так выглядит проект, где два разработчика и полная свобода. А теперь добавьте к ним в команду фаната «Звёздных войн», любителя Йоды, обожателя аббревиатур и экономного комментатора. Результат — код, который... ну, *«вроде работает»*, но никто не понимает, как именно.

И дело не в грамматике, не в запятых. Проблема глубже: **код теряет свою речь** и становится полностью неестественным.

Ожидания и реальность

При этом менеджеры и продукт-оунеры оценивают таких кандидатов по совершенно другим критериям: затраченное время и стоимость труда. Я ни разу не слышал, чтобы разработчиков штрафовали или увольняли **за работающий, но плохо написанный код**. Порой даже некомпетентные руководители могут подталкивать к этому, говоря: «Закостыли», «Делай хоть как-то», «Потом перепишем». А со временем сами начинают смеяться над своими решениями от безысходности.

Единственный, кому по-настоящему нужен ясный код — это высокомотивированный разработчик, который хочет разобраться и сделать хорошо. Такой, как вы. Никто, кроме таких людей, как вы, не будет действительно заботиться о коде. Тимлиду проще настроить линтеры, ввести формальные код-ревью и регламенты, по которым без особого вникания будут ставить друг другу галочки. Но только увлечённый разработчик задаётся вопросом: удобно ли это читать, сопровождать, дорабатывать? Поэтому очень важно отстаивать своё мнение — ведь завтра именно вам придётся разбираться в том, что вы написали сегодня.

Стремление важнее совершенства

Не существует самого красивого человека на свете для всех, лучшей книги, лучшего сериала. Так же не существует и идеального кода. Даже в Open-Source проектах, за которыми следят тысячи разработчиков, всегда найдутся спорные решения или места, где можно было бы сделать лучше.

Но цель не в том, чтобы достичь идеала. Цель — в намерении. В желании писать так, чтобы код был понятен, аккуратен, приятен в работе.

Когда рядом есть люди, которые пишут с вниманием, с уважением к будущим читателям — это вдохновляет. Хочется и самому придумать название переменной чуть удачнее, структурировать код так, чтобы он читался легко. Такая культура — заразительна, и она создаёт особую атмосферу в команде.

Если вы возьмёте на вооружение практики, описанные в этой книге, — вы уже на голову выше большинства. Потому что стараетесь. Потому что старание видно в коде. И это не выдумка — это то, что действительно ощущается теми, кто читает и поддерживает проект.



Форматирование

Вспомните, как в школе у вас были тетрадки в клетку и в линейку — для разных предметов свои. На уроках математики вы аккуратно писали « $2 + 2 = 4$ », размещая каждый знак в отдельной клетке. Оставляли пару строк между задачами, чтобы всё выглядело опрятно и не сливалось.

Когда вы стали старше, то преподаватель просил оформлять работы определённым образом: использовать определённый шрифт, выделять заголовки, проставлять нумерацию страниц и вставлять колонтитулы. Возможно, вы считали, что это подавляет индивидуальность, но делалось это не просто так.

Единый формат облегчает чтение, проверку и сравнение. Представьте: каждая работа оформлена по-разному. Кто-то пишет в клетку, кто-то — в линейку, а кто-то и вовсе на белом листе, без разметки. Сколько бы времени ушло только на то, чтобы «вникнуть» в текст? Выглядели бы вычисления по математике на листе в линейку красиво? Конечно же, нет.

Так и с кодом. Когда в Google внедрили единый стиль кодирования, время прохождения ревью сократилось на 30%. Это не значит, что программисты стали умнее, а потому что перестало мешать оформление.

Но форматирование кода не преподают, не снижают оценки при обучении и на него вообще не обращают внимания.

Рассмотрим небольшой фрагмент кода:

```
// Плохо [X]
class BlipController extends Controller {
  public function index (){
    $blips=Blip::with('user')->latest()->get() ;
    return view( 'blips.index',[
'blips' => $blips] );
  }

  public function update(Request $request , Blip $blip)
{
  $blip->update($request->validated());

  return redirect()->route('blips.index');}
}
```

Код рабочий, синтаксических ошибок нет. Но посмотрите внимательнее: уже на двух методах видно, что стиль гуляет — лишние пробелы, где-то отступ есть, где-то нет, скобки кто как поставил. Это создаёт ощущение небрежности и усложняет чтение.

Исправить это — дело пары минут. И чем раньше вы начнёте обращать внимание на такие детали, тем легче будет работать с вашим кодом — вам и другим.

Забудь привычки

У каждого разработчика со временем формируются свои привычки. Один любит писать скобки на той же строке, другой — на новой. Один ставит пробел перед скобкой, другой считает это дурным тоном. Всё это — дело вкуса.

Но вот в чём загвоздка: **вкусы у всех разные, а кодовая база — одна.**

Допустим, в проекте работают два разработчика. Каждый оформляет код по-своему:

```
extends Controller
{
    // ...
}
```

```
extends Controller {
    // ...
}
```

Оба уверены в правильности своего стиля и могут привести весомые аргументы. Но код — это общее пространство. Здесь важнее не то, чей стиль «красивее», а то, чтобы он выглядел так, как будто его писал один человек. Даже если над ним работают пятнадцать разработчиков. Кому-то придётся уступить — это нормально.

Отнеситесь к этому как к государственной форме документа. Есть определённый шаблон, единый для всех граждан. Неважно, нравится ли шрифт, цвет или размер бумаги — **важно, чтобы стиль был единым.**

Если язык программирования достаточно зрел, скорее всего, у него есть общепринятые стандарты. Например, для PHP есть **PSR-12** и основанный на нём **PER** — *PHP Extended Recommendation*.

Все эти стандарты охватывают очень большое количество аспектов оформления кода, включая отступы, пробелы, переносы строк, скобки, длину строки и многое другое.

Возможно, сначала будет непривычно, но вы быстро привыкнете и перестанете замечать отступы и переносы, сосредоточившись на содержимом.

Как внедрять форматирование?

Как же тогда это внедрить? Не писать же комментарий коллеги на каждую строчку кода в ревью? Это отнимет слишком много времени и превратится в абсурд. На самом деле — это одна из самых простых практик в этой книге. Потому что уже есть множество инструментов автоматизации, например, **Laravel Pint** или **PHP-CS-Fixer**, которые будут автоматически исправлять код:

```
// Хорошо [✓]
class BlipController extends Controller
{
    public function index()
    {
        $blips = Blip::with('user')->latest()->get();

        return view('blips.index', [
            'blips' => $blips,
        ]);
    }

    public function update(Request $request, Blip $blip)
    {
        $blip->update($request->validated());

        return redirect()->route('blips.index');
    }
}
```

Пусть работает машина

Очень часто всё заканчивается полумерой: «Обязательно запускать линтер перед каждым коммитом!» и добавляют проверку в CI. Вроде разумно. Но это как просить студентов самостоятельно проверять орфографию на экзамене.

Разработчик отправил код и ушёл на обед, надеясь, что когда вернётся — получит верификацию и, может быть, ревью. Но, вернувшись, он видит: тесты не прошли. Линтер упал из-за пробела в конце строки. Смешно? Нет. Просто глупо.

Можно, конечно, в очередной раз напомнить: «Проверь перед коммитом», «Запусти линтер вручную», «Не забудь отформатировать». Но зачем? Если это можно делегировать машине — пусть делает она. Без напоминаний. Без обсуждений.

Это всё равно что зайти на Госуслуги и увидеть форму для входа, где можно ввести в свободное поле СНИЛС или номер телефона. Который можно ввести по-разному:

- 89512345678,
- +79512345678,
- 9512345678,
- +7 (951) 234-56-78
- и ещё множество вариантов.

Было бы странно, если бы где-то внизу было сказано: «Если вы вводите номер телефона, вводите его строго в формате XXX». Сайт заботится о пользователе и сам стандартизирует ввод, приводя ваш номер к единому формату.

Two examples of phone number input forms are shown side-by-side.

Left Form: The title is "Номер телефона". The input field contains the text "+7()- - - -", where the digits are represented by brackets and dashes.

Right Form: The title is "Номер телефона". The input field contains the placeholder text "Введите номер телефона". Below the input field, there is a red asterisk followed by the text: "* Номер телефона должен быть указан обязательно в формате +7(123)-123-12-22."

Поведение должно быть аналогичным: разработчик отправил код. Линтер в CI автоматически привёл его в порядок. После этого — запустились тесты. Всё прошло. Разработчик спокойно обедает. Команда работает. Никто не отвлекается на ерунду.

Вот что по-настоящему эффективно. Не заставляйте разработчика думать об этом вовсе. Не возвращайте его к коду из-за пробела, скобки, пустой строки. Это абсурд.

Пусть он думает о логике. Об архитектуре. О том, за что его на самом деле наняли.



Код должен дышать

Форматирование делает код опрятнее: выравнивает отступы и расставляет пробелы. Но автоматические инструменты не понимают его смысла. Они не знают, где заканчивается одна логическая структура и начинается другая.

А программисту важно видеть не строки, а блоки логики. Каждое завершённое действие — новая мысль, её нужно отделять пустой строкой, чтобы код было проще читать и понимать.

Если не дать коду «дышать», он превращается в сплошной поток текста. Такой код давит. Утомляет. Изматывает. В нём легче ошибиться — потому что невидна структура.

Плохая новость:

Ни один автоматический инструмент форматирования не научит ваш код дышать.

Рассмотрим пример:

```
// Плохо [X]
$user = $request->user();
$zone = ClimateZone::find($id);
$zone->assign($user);
$zone->save();
return $zone;
```

Плотность текста не даёт глазу сделать паузу и понять, где что происходит. Всё сливается в один поток. Лучше отделять каждую завершённую мысль пустой строкой. Пусть будет пауза. Это нужно не для компьютера, а для человека, чтобы это выглядело так:

```
// Хорошо [✓]
$user = $request->user();

$zone = ClimateZone::find($id);
$zone->assign($user);
$zone->save();

return $zone;
```

Такой код «дышит»:

- первый блок — получение пользователя;
- второй блок — работа с климатической зоной;
- третий блок — возврат результата.

Ещё пример с условием и несколькими действиями:

```
// Плохо [X]
if ($user->isAdmin()) {
    $settings->loadDefaults();
    $settings->setTheme('dark');
    $settings->applyUserPreferences();
    $settings->enableNotifications();
    $settings->save();
    Log::info('Настройки для администратора обновлены');
}
```

Разобьём на осмысленные части:

```
// Хорошо [✓]
if ($user->isAdmin()) {
    $settings->loadDefaults();
    $settings->setTheme('dark');

    $settings->applyUserPreferences();
    $settings->enableNotifications();

    $settings->save();

    Log::info('Настройки для администратора обновлены');
}
```

Теперь каждое действие выделено, и мозгу легче уловить логику. Но тут важно не переборщить. Код должен дышать, но не задыхаться.

Иногда пауза мешает, их избыток разрушает поток. Бывает код, где всё держится на ритме — строка за строкой, быстро, без отступов. И вставленная пустая строка рвёт это ощущение, как случайный абзац посреди строки диалога.

Например, вот так:

```
// Плохо [X]
$logger->debug('start');

$service->prepare();

$service->run();

$logger->debug('done');
```

Здесь пустые строки разрывают логическую цепочку, которая воспринимается лучше как единое целое — это последовательность действий, которую хочется видеть непрерывной.

```
$logger->debug('start');

$service->prepare();
$service->run();

$logger->debug('done');
```

Если сомневаетесь, нужно ли ставить пустую строку, попробуйте вместо неё вставить комментарий. Если он логично завершает блок, значит, и отступ уместен. А если не уверены — всё равно отделите. Хуже точно не будет.

Код должен читаться так же, как текст — с абзацами, паузами и интонацией. Если убрать структуру — рушится восприятие.



Именование

Иногда достаточно открыть структуру проекта, чтобы многое понять о команде. И не в лучшую сторону. Например, можно встретить такие каталоги:

```
project
├─ old_config
├─ mordor
├─ BlackMagic
└─ ...
```

Причин для их появления масса. Кто-то просто скопировал старую директорию. Кто-то решил временно «вынести в сторону» непонятный или страшный код. Но так и не разобрался, что с ним делать. А потом это «временно» прижилось и стало частью архитектуры. Это симптом того, что именованию в проекте никто не уделяет внимания, а ведь имена — это первая линия коммуникации.

Теперь представьте, что вы открыли чужой код и наткнулись на переменные:

```
$pogoda;  
$veter;  
$solnce;
```

Такие имена мгновенно выдают новичка.

Это напоминает, как я писал СМС-сообщения в нулевых: тогда сообщения имели ограничение по количеству символов, а на латинской раскладке в одно сообщение помещалось намного больше текста, чем при использовании кириллицы. Поэтому, если не удавалось уложиться в лимит, я писал транслитом:

```
Privet. Mi segodnya vtretimsya v parke?  
Ya vzal s soboy...
```

Это была вынужденная мера, но та эпоха давно закончилась. В программном коде большинство фреймворков, библиотек, документаций — на английском. Если в коде появляется нечто вроде:


```
class Order extends Controller
{
    public function ...()
    {
        // ...
        foreach ($zakazy as $tovar) {
            $product->otpravka($tovar);
        }
        // ...
    }
}
```

Создаётся разрыв контекста, фреймворк говорит на одном языке, твой код — на другом. Переключаться между языками утомительно, особенно в больших проектах. Это снижает читаемость и замедляет понимание.

Если имена переменных, файлов, классов, папок не передают смысл — они становятся ментальным мусором. Чем их больше — тем труднее читать, понимать и поддерживать код.

Поэтому нужно заботиться об именовании. Но что делает имена хорошими? И как начать исправлять это прямо сейчас?

Некоторым разработчикам нравится использовать сокращения в именах, что кажется для них удобным и помогает ускорить написание кода. Некоторые языки даже рекомендуют подобный подход — например, в языке Go советуют:

Имя объекта, для которого вызывается метод, должно отражать его суть; часто достаточно одной или двух букв, обозначающих тип (например, «с» или «cl» для «Client»). Не используйте общие имена вроде «me», «this» или «self».

Сокращения могут быть как однобуквенными, так и более длинными или смешанными, например, итерация цикла как `$i`, запрос как `q`, интерфейс как `IComponent`. Однако зачастую подобные сокращения лишь приводят к путанице и усложняют поддержку кода.

Это происходит потому, что, взяв небольшой фрагмент кода, невозможно сразу понять, что происходит. Разработчику придется либо возвращаться к месту объявления переменной, чтобы разобраться, либо открывать исходники метода или класса. В итоге такой код начинает выглядеть как шифровка: смотришь на него — не понимаешь, что значит каждая переменная. Потом приходится искать «ключ», чтобы расшифровать смысл, и так по кругу. Но мы ведь не должны заниматься разведкой.

Давайте рассмотрим следующий пример:

```
// Плохо [X]
$usr = User::find($id);

// Хорошо [✓]
$user = User::find($id);
```

Здесь переменная `$usr` представляет объект пользователя. Однако, сокращённое имя `$usr` не даёт понимания того, что именно хранится в этой переменной.

```
// Плохо [X]
class Usrctrl extends Ctrl {
    public function f() {
        // ...
    }
}
```

В данном примере имя класса `Usrctrl` недостаточно информативно. Разработчику, сталкивающемуся с этим классом впервые, будет трудно понять его назначение. Название класса должно чётко отражать его функциональность, например, `ProfileController`.

```
// Хорошо [✓]
class ProfileController extends Controller
{
    public function show()
    {
        // ...
    }
}
```

В некоторых книгах по программированию можно встретить утверждения, что код должен быть самодокументируемым. Это значит, что имена переменных, методов и классов должны объяснять, что происходит — без комментариев, без документации, без менторов.

Некоторые даже добавляют:

«Пусть имя будет длинным — это сделает код самодокументированным и более понятным».

Нет. Не делает. Особенно если это имя — дымовая завеса над тем, что в коде нет ни логики, ни смысла.

Я встречал разработчиков, воспитанных на строгих правилах: «Имена должны быть максимально подробными, чтобы не нужны были комментарии».

В теории звучит благородно, но на практике часто рождает чудовищные конструкции вроде:

```
public function retrieveUserAccountByEmailAddress(  
    string $email  
): ?UserAccount
```

Да, здесь всё предельно описательно. Но вместе с этим — чрезмерно длинно, тяжело читается и мешает восприятию кода. А ещё хуже, когда длинные имена тратят всю эту длину не на конкретику, а на расплывчатые абстракции:

```

abstract class AbstractContextHandler
{
    use SemanticMapper;

    public string $moduleScopeIdentifier = 'reporting';

    public function process(
        array $contextualizedComponentUnitPayload
    ): array
    {
        $moduleScopedUnits = [];

        foreach ($contextualizedComponentUnitPayload as $contextBoundSemanticUnit) {
            $moduleScopedResponseUnits[] = $this->transformContextUnit($contextBoundSemanticUnit);
        }

        return $moduleScopedResponseUnits;
    }

    protected function transformContextUnit(
        $contextBoundSemanticUnit
    ): array
    {
        return [
            'encodedPayloadFragment' => $this->map($contextBoundSemanticUnit),
            'operationalModuleDomain' => $this->moduleScopeIdentifier,
        ];
    }
}

```

Что делает этот класс? Не ясно. Что он обрабатывает? Какой «контекст»? Какой «модуль»? Что за «единицы компонентов»?

Это типичный корпоративный анти-паттерн: взять простую задачу, обернуть её в кучу терминов, и сделать вид, что это архитектура.

Этот код невозможно понять. Не потому что он глупый. А потому что этот код никогда ничего конкретного не делал.

Конкретика

Предыдущий пример лишён конкретики, но она может отсутствовать и в именах переменных вроде таких:

```
// Плохо [X]
$data;
$var;
$info;
$item;
```

На первый взгляд выглядят нормально. И правда, в крошечных методах, где весь контекст на виду, такие имена вполне читаемы. Но стоит методу хоть немного разрастись — и смысл переменной начинает размываться.

Что именно скрывается за `$data`? Это может быть пользователь, список заказов, JSON или какая-нибудь внутренняя структура. Мы не знаем, пока не полезем внутрь: смотреть, что туда присваивается, как используется, что откуда приходит. А даже если разберёмся — не факт, что в другом месте кода `$data` не означает уже совсем другое.

Это относится и к методам, иногда вполне нормально иметь метод `run` для классов, которые выполняют одну единственную функцию (так называемые action-классы). Но это совершенно не информативно для масштабных объектов, например:

```
// Плохо [X]
$user->run();
$user->handleData();
$user->process();
```

Старайтесь использовать информативные имена, которые отражают суть того, что они представляют, например:

```
// Хорошо [✓]
$user->posts();
$user->notify(...);
$user->deactivate();
```

Логические значения

Переменные и методы, которые содержат логическое значение (`true` или `false`), часто называют непонятно. Например:

```
// Плохо [X]
$admin;
$retry;
$user->access();
```

В этих примерах трудно понять назначение переменной или метода: `$admin` может быть флагом или объектом пользователя, `$retry` — числом попыток или логическим состоянием, а метод `access()` — проверкой доступа или действием. Чтобы сразу было понятно, что значение **логическое**, используют префиксы `is`, `has` и `should`:

```
// Хорошо [✓]
$isAdmin = true;
$shouldRetry = false;
$user->hasAccess();
```

Единицы измерения

Рассмотрим пример именования переменных с указанием единиц измерения температуры:

```
// Плохо [X]
$temperature = 98.6;
```

На первый взгляд всё выглядит нормально — просто число. Но что это за температура? Фаренгейты? Градусы Цельсия? Кельвины?

Ситуация усложняется, если в другом месте кода встречается:

```
$temperature = 37;
```


Чтобы избежать путаницы, можно явно указывать единицы измерения:

```
// Хорошо [✓]
// Мы явно указываем, что это температура в фаренгейтах
$temperatureInFahrenheit = 98.6;

// Хорошо [✓]
// Или в градусах Цельсия
$temperatureInCelsius = 37;
```

Другой способ справиться с этим — создать специальные объекты. Создадим объект `Temperature` со статическими конструкторами, каждый из которых явно указывает единицу измерения:

```

class Temperature
{
    public static function fromCelsius(float $degrees): self
    {
        return new self($degrees);
    }

    public static function fromFahrenheit(float $degrees): self
    {
        $celsius = self::convertFahrenheitToCelsius($degrees);

        return new self($celsius);
    }

    private function __construct(
        public float $valueInCelsius,
    ) {}
}

```

Использование класса `Temperature` поясняет, что ожидается:

```

// Хорошо [✓]
$temperature = Temperature::fromFahrenheit(98.6); // 37.0°C
$temperature = Temperature::fromCelsius(37.0);    // 37.0°C

```

Таким образом, единица измерения становится несущественной — объект скрывает детали и позволяет получить значение в нужном формате.

Будь кратким

Не стоит пытаться расписать всё длинными именами в надежде, что это сделает код понятнее. Вместо этого давайте ровно столько информации, чтобы можно было уверенно принимать решения. А всё лишнее — уберите.

Рассмотрим пример:

```
// Плохо [X]
class PostItemCollection
{
    public function addPost(Post $post)
    {
        // Добавляем пост в коллекцию
    }

    public function hasPost(Post $post): bool
    {
        // Проверяем, есть ли пост в коллекции
    }

    public function clearPost()
    {
        // Очищаем коллекцию
    }
}
```

Здесь много повторов и избыточных уточнений в именах методов. Ведь всё понятно из контекста класса — это коллекция постов. При этом слово **Item** в имени класса ничего не добавляет и скорее мешает.

Поэтому лучше упростить:

```
// Хорошо [✓]
class PostCollection
{
    public function add(Post $post)
    {
        // Добавляем пост в коллекцию
    }

    public function has(Post $post): bool
    {
        // Проверяем, есть ли пост в коллекции
    }

    public function clear()
    {
        // Очищаем коллекцию
    }
}
```

Теперь вместо длинных имён — простой класс с тремя понятными методами: `add`, `has` и `clear`. Каждый делает ровно то, что ожидаешь, без лишних слов.

Такой подход помогает избежать длинных и сложных имён, при этом сохраняя ясность и понятность. Код становится чище, короче и проще для восприятия.

Использование суффикса `-er`

В мире объектно-ориентированного программирования слишком часто встречаются имена классов вроде:

- Manager
- Controller
- Formatter
- Presenter

Эти имена плохи не потому, что они технически неверны. Они плохи потому, что не говорят ничего конкретного, слишком абстрактны.

Такая абстракция хороша для высокоуровневых концепций типа фреймворков, но не для конкретных классов в вашем приложении.

А ведь имя класса — это первый и, зачастую, единственный источник информации о его ответственности.

Эти имена — дымовая завеса. Они скрывают детали, замыливают смысл, делают код нечитаемым, а архитектуру — расплывчатой. Они подменяют суть интерфейсом, упрощая названия до абсурда. Да, это удобно. Да, так делают все. Но именно поэтому ваш проект через год превращается в грудку мусора.

```
// Плохо [X]  
class ReportManager { /* ... */ }  
class StringFormatter { /* ... */ }
```

```
// Хорошо [✓]  
class StringTruncatedToLength { /* ... */ }
```

Если вы не можете придумать конкретное имя — это сигнал, что саму ответственность объекта стоит пересмотреть.

Парные имена

Методы в паре работают лучше, когда звучат как единое целое. Они напоминают диалог: начало перекликается с концом.

Рассмотрим пример, где имена не согласованы:

```
// Плохо [X]  
$object->startProcess();  
$object->completeTask();
```

Здесь методы словно из разных рассказов — они не складываются в цельный образ. Такой код заставляет остановиться и задуматься, как эти действия связаны между собой. Это — явный признак слабого дизайна и плохой коммуникации через код.

Гораздо эффективнее, когда методы звучат как пара, поддерживают одну мысль и логически соответствуют друг другу:

```
// Хорошо [✓]  
$object->startProcess();  
$object->finishProcess();
```

Или так:

```
$object->beginTask();  
$object->completeTask();
```

Тогда логика воспринимается как диалог — вызов и ответ. Такой подход существенно облегчает понимание кода: у читателя возникает естественный и логичный поток мысли.

Но чтобы такой «диалог» методов работал ещё лучше, класс должен быть сфокусирован на одной задаче или сущности. Например:

```
// Хорошо [✓]  
$task->begin();  
$task->complete();
```

```
// Хорошо [✓]  
$task->start();  
$task->finish();
```

Здесь класс чётко определяет свою зону ответственности — работу с задачей. Методы — естественные этапы жизненного цикла этой задачи. Такой фокус значительно упрощает тестирование, поддержку и развитие кода.

Не обманывай

Мы верим, что метод делает то, что обещает его имя. Мы верим, что класс отражает свою роль. Мы верим, что код говорит правду. Но стоит нарушить это правило — и доверие рушится. Имя, которое вводит в заблуждение, оставляет ощущение предательства: разработчик ожидал одно поведение, а получил другое. После этого каждый следующий метод он будет читать уже с подозрением.

Пример метода с вводящим в заблуждение названием:

```
// Плохо [X]
public function saveModels(array $item): void
{
    $model = new Model();
    $model->setAttributes($item);
    // ...
}
```

Метод заявляет, что он сохраняет модели — во множественном числе, и принимает массив. Если не заглядывать внутрь, любой прочитавший его разработчик подумает, что использовать его можно примерно так:

```
$models = [
    new Model(),
    new Model(),
    new Model(),
];

$object->saveModels($models);
```

Но вместо этого — разочарование от предательства. Метод берёт массив, который на самом деле описывает атрибуты одной модели. Название подтолкнуло к ложной ментальной модели. Название обещало одно действие, а сделало совершенно другое.

Это подрывает базовый инструмент командной работы — язык. Когда названия перестают соответствовать коду, разработчики очень быстро теряют к нему доверие, начинают бояться ошибиться и реже вносят изменения.

Соглашение об именовании

Имена — это не только классы, объекты и методы. Они затрагивают всё вокруг. Например, в веб-приложении важно выбрать понятные имена для адресов:

```
https://example.com/WeatherReport
https://example.com/weather_report
https://example.com/weather-report
```

То же касается ключей в JSON-переводах:

```
{
  "WeatherReport": "Отчёт о погоде",
  "weather_report": "Отчёт о погоде",
  "weather-report": "Отчёт о погоде"
}
```

Или файлов и директорий, не связанных напрямую с языком или фреймворком:

```
project
├─ WeatherReports
├─ weather_reports
├─ weather-reports
└─ ...
```

Чтобы всё оставалось предсказуемым, выработайте в команде своё соглашение и используйте его там, где общие правила не помогают.



Магические значения

Никто не начинает изучение проекта с вдумчивого чтения всей вики. Разработчик открывает редактор кода, запускает поиск по имени метода или класса — и вперёд, прямо в код. Только когда поведение становится неочевидным, когда логика не складывается — он обращается к документации, вики, базе знаний, если такая вообще есть, или, ещё хуже, к коллеге, выпрашивая информацию по чайной ложке.

Рассмотрим классическую ситуацию:

```
// Плохо [X]
if ($status == 1) {
    // ...
}
```

На первый взгляд — ничего страшного. Просто проверка статуса. Но что значит это число **1**? Почему именно оно?

Разработчик, который это писал, наверняка знает, что **1** здесь означает «активный статус». Но для всех остальных — это магическое число, появившееся из ниоткуда. Код превращается в загадку: *почему не 0? А может быть значение 2? А в статус приходит 1 или true?*

Теперь представьте, что этот код читает кто-то вроде меня — из мира Linux. В Unix-системах код **0** означает успешное завершение, а **1** — **ошибку**. И я интуитивно читаю этот код иначе: «О, тут проверяется, что была ошибка?» Мои привычки вступают в конфликт с чужими соглашениями — и я начинаю сомневаться в логике самого кода.

А если значение гораздо больше, например:

```
// Плохо [X]
if ($status == 24) {
    // ...
}
```

Что это значит? День рождения начальника? Номер ошибки? Идентификатор тайного соглашения?

Загадочными могут быть не только числа. Иногда код может притаить за собой набор символов:

```
// Плохо [X]
if ($char === '%!') {
    // ...
}
```

Что значит `%!`? Если значение несёт смысл, пусть оно громко заявляет о себе именем. Тогда станет ясно, зачем оно здесь и как его использовать дальше — с помощью константы:

```
// Хорошо [✓]
const STATUS_ACTIVE = 1;

if ($status === STATUS_ACTIVE) {
    // ...
}
```

Теперь код стал более понятным и поддерживаемым. При его чтении сразу становится ясно, что проверяется в условии.

Можно пойти дальше и использовать перечисления для явного определения различных значений:

```
// Хорошо [✓]
enum Status: string
{
    case ACTIVE = 'active';
    case INACTIVE = 'inactive';
    case ARCHIVED = 'archived';
}

if ($status === Status::ACTIVE) {
    // ...
}
```

Или числовым значением:

```
// Хорошо [✓]
enum Status: int
{
    case ACTIVE = 1;
    case INACTIVE = 2;
    case ARCHIVED = 3;
}

if ($status === Status::ACTIVE) {
    // ...
}
```

Такой подход делает код более читаемым и позволяет явно указать доступные значения статуса, а также использовать типизированное значение в методах, например:

```
function canBePublished(Status $status): bool
{
    // ...
}
```

Используя именованные константы или перечисления, мы делаем код более понятным и поддерживаемым, ведь нам не нужно обращаться ни к документации, ни к коллеге за пояснениями, что важно для разработки масштабируемых приложений.

Переизбыток констант

Когда речь заходит о «магических значениях», первый инстинкт многих начитанных только первой частью разработчиков — немедленно заменить каждое из них на именованную константу. Как было показано ранее, это логично, но не всегда разумно. Проблема не в самих константах, а в том, как читается код и насколько понятна его суть.

Рассмотрим реальный пример:

```
// Плохо [X]
class Order
{
    public function daysSinceLastUpdate(): float
    {
        return $this->updated_at / 1_000_000 / 60 / 60 / 24;
    }
}
```

Здесь мы видим цепочку арифметических операций, и каждый разработчик, читающий этот код, вынужден мысленно раскручивать её: «Так, это микросекунды, потом секунды, потом минуты... ага, значит, это перевод времени в дни». Это усложняет чтение и отвлекает от сути метода.

После слепого ввода многочисленных констант пример начинает выглядеть так:

```
// Плохо [X]
class Order
{
    private const MICROSECONDS_IN_SECOND = 1_000_000;
    private const SECONDS_IN_MINUTE = 60;
    private const MINUTES_IN_HOUR = 60;
    private const HOURS_IN_DAY = 24;

    public function daysSinceLastUpdate(): float
    {
        return $this->updated_at
            / self::MICROSECONDS_IN_SECOND
            / self::SECONDS_IN_MINUTE
            / self::MINUTES_IN_HOUR
            / self::HOURS_IN_DAY;
    }
}
```

Формально код стал «говорящим». Но читается он по-прежнему тяжело. Мы заменили магию чисел на большое количество деталей.

Вместо того чтобы «расшифровывать» механику времени вручную, лучше полностью передать заботу об этом классу, который создан именно для этой работы. Например, **Carbon**:

```
// Хорошо [✓]
use Carbon\Carbon;

class Order
{
    public function daysSinceLastUpdate(): float
    {
        return Carbon::create($this->updated_at)
            ->floatDiffInDays(now());
    }
}
```

Не всякое «магическое значение» нужно заменять на константу. Иногда **лучший способ устранить магию — не объяснять детали вообще**. Спрячьте реализацию за выразительным интерфейсом. Пусть код говорит, что он делает, а не как.

Ещё лучше будет, если наши свойства сразу будут объектами:

```
// Хорошо [✓]
class Order
{
    public function daysSinceLastUpdate(): float
    {
        return $this->updated_at->floatDiffInDays(now());
    }
}
```


Если кажется, что время слишком простой пример, то вот похожий с размером файла:

```
// Плохо [X]
class File
{
    public function humanReadableSize(): string
    {
        return $this->size / 1024 / 1024 . ' MB';
    }
}
```

Для которого добавили константы:

```
// Плохо [X]
class File
{
    private const SHORT_MEGABYTE = 'MB';
    private const BYTES_IN_MEGABYTE = 1024 * 1024;

    public function humanReadableSize(): string
    {
        $megabytes = $this->size / self::BYTES_IN_MEGABYTE;

        return sprintf(
            '%.2f %s',
            $megabytes,
            self::SHORT_MEGABYTE
        );
    }
}
```

Вместо таких констант лучше всего делать классы, которые будут скрывать все эти вычисления. К тому же они сразу же будут переиспользованы в вашем проекте в других местах, чем вводить новые приватные константы или, ещё хуже, объявлять публичными у `File` и ещё больше увеличивать связность.

```
// Хорошо [✓]
class File
{
    public function humanReadableSize(): string
    {
        return $this->size->toHumanReadable();
    }
}
```



Размер имеет значение

В детстве мы играли в простую, но удивительно поучительную игру. Дети становились в круг, и один из них начинал перекидывать мяч. Но это был не просто мяч — это была «горячая картошка». И правила были предельно ясны: поймал — тут же бросай дальше. Максимум одна секунда. Кто задержал — проиграл. Никаких пауз, планов и стратегий. Только действие. Только передача.

Если ты хоть на мгновение задумался — обжёгся. Здесь нет времени для сомнений: нужно полагаться на интуицию, играть легко и не мешать ходу игры.

Так и с кодом. Каждый класс, каждый метод, каждая строка — это не долгий монолог, а быстрый пас, моментальный результат, передача задачи следующему игроку. Код не должен «держат мяч» в руках подолгу, копаться в себе, раздувать внутренние сложности, мешать движению.

У каждого должна быть одна цель — передать задачу и не тормозить процесс. Ассоциируйте это как:

- Класс — это игрок.
- Метод — это пас. Он может быть левой рукой, правой, можно схватить или отбросить мяч.
- Строки — это момент перед броском.

Когда момент перед пасом выглядит вот так:

```
// Слишком длинный метод [X]
public function export(string $key)
{
    // ...
    // ...
    // 1000 строк кода
    return $result;
}
```

То получается, что игрок ловит мяч и не бросает. Он встал посреди круга и начал делать кувырки, включил музыку, рассказал стихотворение и только потом — спустя долгие секунды — наконец передал мяч дальше.

Это раздражает не только других игроков при игре, ведь то же самое происходит с кодом, когда его размер выходит за разумные пределы. Длинные методы и классы начинают запутывать, а вместо ясности мы получаем неразбериху, с которой сложно работать.

Точно так же, как перегруженные предложения, огромные блоки кода перегружают восприятие. Читая их, трудно понять, о чём конкретно идёт речь, и приходится возвращаться к началу, чтобы разобраться, что вообще происходит.

Худшие разработчики гордятся таким кодом: «Он сложный», «Он умный», «Он крутой». А если его трудно читать, советуют лучше разобраться в основе. Но на самом деле это просто неумение передать мяч как можно быстрее. Некоторые разработчики пытаются исправить проблему, формально дробя код на отдельные методы:

```
// Слишком длинный метод [X]
public function export(string $key)
{
    // Загрузка данных
    // Валидация
    // Преобразование
    // Генерация отчёта
    // Сохранение в файл
    // Отправка по почте
    // И ещё десяток шагов...
    $this->step();
    $this->step();
    $this->step();
    $this->load();
    $this->validate();
    $this->transform();
    $this->generateReport();
    // И ещё десяток шагов...
    $this->sendMail();
    return $result;
}
```

На первый взгляд — красиво, ведь метод `export()` записан условно в пять строчек. Но где тут само «сердце»? Вам приходится прыгать из метода в метод, искать смысл: «а, здесь что-то подгружается, а вот здесь валидируется, а вот здесь ещё что-то происходит...». Глаз бежит по коду без чувства завершённости.

Правильно дробить — значит давать каждому этапу собственную осмысленную ответственность, а не делать «пустую оболочку» ради экономии строк. Если метод публичный, он должен отражать высокоуровневый шаг, понятный «с первого взгляда». А приватные методы должны решать действительно отдельный логический блок, а не просто «задёргивать» следующий вызов без собственной логики.

Хороший публичный метод должен вызывать у вас реакцию: «Да, это целостный шаг!» Например:

```
// Хорошо [✓]
$document = Document::find(1);

$content = $document->export(Excel::class);

$user->notify(ExportNotification::class, [
    'content' => $content->toString(),
]);
```

Каждая строка — как законченный абзац. Здесь нет прыжков по стеку. Всё перед глазами и мы наглядно видим, что сделали пас.

Но даже когда мы избавились от процедурного стиля «шаг1», «шаг2», «шаг3», очень легко попасть в ловушку: кто должен принимать решения и в какой форме объекты должны взаимодействовать друг с другом?

Например:

```
// Плохо [X]
$document = Document::find(1);

if($document->isPublished()) {
    $content = $document->export(Excel::class);
}
```

В этом примере происходит запрос данных у объекта, после чего на их основе принимается решение, то есть ответственность фактически переносится на внешний код. Вместо этого лучше переложить эту ответственность на сам объект: внутри метода может быть выброшено исключение или выполнено иное поведение.

Объект самостоятельно определяет, с кем и как ему взаимодействовать — внешний код лишь описывает намерение. Такой подход повышает модульность и гибкость архитектуры.

```
// Плохо [X]
if ($user->isAdmin() || $user->hasRole('manager')) {
    $content = $document->export(Excel::class);
}
```

```
// Хорошо [✓]
if ($user->canExport($document)) {
    $content = $document->export(Excel::class);
}
```


Это один из важных принципов, который помогает сделать код объектов лаконичным, звучит так:

Не спрашивай объект о его данных, чтобы принять решение — скажи объекту, что делать.

Игра «горячая картошка» научила нас — не задерживать ответственность. А принцип объектно-ориентированного проектирования — говорить, а не спрашивать; не задавать лишних вопросов, а формулировать намерения. В следующих разделах разберём конкретные техники, которые позволяют быстрее передать «мяч» дальше.



Без лишних движений

Что отличает настоящего мастера от любителя? Не скорость или количество работы, а умение делать только то, что действительно нужно — без лишних действий, без излишних усложнений.

Лишние операции съедают время, отвлекают от сути и увеличивают вероятность ошибок. Настоящий мастер знает: чем меньше шагов, тем меньше точек отказа и тем проще менять и улучшать код.

Не создавай переменные без необходимости

Каждая переменная — это дополнительный элемент, который надо держать в голове и читать. Если переменная не улучшает читаемость и не нужна для повторного использования, от неё стоит отказаться.

```
// Плохо [X]  
$tmp = $user->name;  
echo $tmp;
```

В этом примере переменная `$tmp` не даёт никакой дополнительной пользы: она лишь усложняет код добавляя лишнее имя, которое нужно запомнить.

```
// Хорошо [✓]  
echo $user->name;
```

Прямой вывод значения из объекта будет проще и понятнее.

Не меняй тип данных для переменной

Переменная может легко поменяться, это заложено в самом определении переменной, но смена типа в процессе её жизни означает, что код плохо структурирован. Это снижает предсказуемость и усложняет отладку кода.

```
// Плохо [X]
function (array $user) {
    $user = new User($user); // Был массив, стал объект
}
```

Но если оказались в такой ситуации, намного лучше будет уточнить и дать другое имя:

```
// Хорошо [✓]
function (array $userData) {
    $user = new User($userData);
}
```

Лучше всего сразу работать с объектами и при необходимости — извлекать из них нужные данные, например, преобразовывать в массив:

```
// Хорошо [✓]
function (User $user) {
    // ...
}
```

Особенно часто такое изменение можно заметить с переменными `$value`, `$item` и `$result`, когда конечный результат меняется:

```
// Плохо [X]
$value = [1, 2, 3];           // массив
$value = (object) $value;     // теперь объект
$value = json_encode($value); // теперь строка
```

Избегай повторных вычислений

Если результат операции нужен несколько раз, лучше вычислить его один раз и сохранить в переменную, чем повторять вычисления. Это особенно важно внутри циклов и условий: лишний вызов метода или обращение к внешнему ресурсу может стоить дорого.

```
// Плохо [X]
foreach ($users as $user) {
    $moon = MoonPhase::forToday();
    $user->notify(new MoonPhaseNotification($moon));
}
```

Метод `MoonPhase::forToday()` вызывается на каждой итерации, хотя результат каждый раз одинаковый. В реальной жизни этот метод может обращаться к внешнему API или выполнять тяжёлые вычисления.

```
// Плохо [X]
foreach ($users as $user) {
    $moon ??= MoonPhase::forToday();

    $user->notify(new MoonPhaseNotification($moon));
}
```

Этот способ вызова метода выполнится только один раз, но может быть неочевиден для неподготовленного читателя. Оператор `??=` (null coalescing assignment) означает: «если переменная `$moon` ещё не определена или равна `null`, присвой ей значение». Хотя это удобно, внутри цикла такая запись может восприниматься как «магия» — к тому же важно помнить, что переменная с таким именем не должна быть объявлена ранее. Чтобы избежать недопониманий, лучше сделать намерение явным:

```
// Хорошо [✓]
$moon = MoonPhase::forToday();

foreach ($users as $user) {
    $user->notify(new MoonPhaseNotification($moon));
}
```

Сохранение результата в переменную делает код не только читаемее, но и предсказуемее.

Используй подходящие структуры данных

Выбирайте правильную структуру данных с самого начала, чтобы не пришлось преобразовывать и адаптировать её в процессе. Часто начинают с базовых типов, например, хранение данных в массивах кажется очень заманчивым и удобным решением: можно просто взять нужное значение по ключу. Но затем в коде начинают появляться конструкции вроде:

```
// Плохо [X]
if (isset($user['address']['city'])) {
    $city = $user['address']['city'];
} else {
    $city = 'Неизвестно';
}
```

Сначала это кажется безобидным. Но когда таких вложенных ключей становится много, и данные разбросаны по всему коду, всё усложняется. Вместо того, чтобы описывать логику поведения, мы продолжаем манипуляции с данными на низком уровне. Чтобы получить ещё одно значение, снова и снова приходится писать `isset` с длинной цепочкой:

```
// Плохо [X]
if (
    isset($user['address']['city']) &&
    isset($user['preferences']['language'])
) {
    $city = $user['address']['city'];
    $language = $user['preferences']['language'];
}
```

Такая практика раскрывает детали хранения данных во всех местах их использования и делает код процедурным: каждый шаг работы с «сырыми» данными прописывается вручную, вместо того чтобы описать желаемое поведение

Например, фильтрация заказов пользователя по городу при использовании массивов выглядит так:

```
// Плохо [X]
if (isset($user['id'], $user['address']['city'])) {
    $city = $user['address']['city'];

    $userOrders = array_filter(
        $orders,
        function ($order) use ($user, $city) {
            return $order['user_id'] === $user['id']
                && $order['city'] === $city;
        }
    );
} else {
    $userOrders = [];
}
```


Здесь постоянно приходится повторять проверки и раскрывать детали — где у пользователя лежит город, как устроен заказ. Если структура данных изменится, код придётся менять во многих местах. Это сложно и рискованно. Использование объектов и инкапсуляция данных позволяют скрыть детали и упростить взаимодействие:

```
// Хорошо [✓]
$city = $user->address?->city() ?? 'Неизвестно';
$language = $user->preferences?->language ?? 'ru';
```

Код становится линейным и более читаемым, а изменения внутренней структуры данных требуют правок только внутри соответствующих классов.

Для полного устранения проверок на `null` можно применить паттерн **Null Object** — объект-заглушку, возвращающий значения по умолчанию, что дополнительно упрощает код:

```
// Хорошо [✓]
class NullAddress {
    public function city()
    {
        return 'Неизвестно';
    }
}

$city = $user->address->city();
```

Если же продолжать использовать массивы с `isset` повсюду, проект превратится в спагетти из проверок и длинных цепочек ключей. Это затруднит чтение и увеличит количество ошибок — они всегда рядом там, где много ручных проверок.

Паттерн `Null Object` полезен не только для возврата значений по умолчанию, но и для реализации методов, которые не должны выполнять никаких действий.

Даже для простых типов данных — строк и чисел — стоит использовать объекты. Мы уже ранее рассматривали пример класса `Temperature`, который скрывает от нас работу с единицами измерения. Теперь рассмотрим аналогичный подход на примере строк.

Часто обработку строки записывают через вложенные вызовы:

```
// Плохо [X]
echo strtoupper(trim(substr($input, 0, 10)));
```

Код работает, но превращается в «матрёшку» из-за их чрезмерного количества вложенных функций. Читать его приходится справа налево, что совсем не свойственно латинице, на которой мы пишем код. Это увеличивает когнитивную нагрузку и скрывает намерение.

Для чтения удобнее сделать класс:

```
// Хорошо [✓]
class Text implements Stringable {
    public function __construct(private string $value) {}

    public function cut(int $length): static
    {
        return new static(substr($this->value, 0, $length));
    }

    public function trim(): static
    {
        return new static(trim($this->value));
    }

    public function value(): string
    {
        return $this->value;
    }

    // ...
}
```

С таким классом обработка становится читаемой и выразительной:

```
echo (new Text($input))
    ->cut(10)
    ->trim()
    ->upper()
    ->value();
```

Теперь мы читаем цепочку шагов, а не пытаемся расшифровать вложенные функции.

Обратите внимание на иммутабельность. Каждый шаг возвращает новый объект при котором не будет скрытых побочных эффектов.

Конкатенация строк

Соединение строк с помощью оператора `.` кажется простым, но быстро теряет читаемость, особенно при длинных сообщениях или множестве переменных:

```
// Плохо [X]
$message = 'Hello, ' . $name . '! Today is ' . date('Y-m-d');
```

При большом количестве переменных сложно сразу понять, как будет выглядеть итоговая строка. Чтобы избежать этого, можно использовать шаблон с подстановкой, например `sprintf`:

```
// Хорошо [✓]
$message = sprintf(
    'Hello, %s! Today is %s',
    $name,
    date('Y-m-d')
);
```

Такой подход делает структуру строки прозрачной и упрощает её поддержку.

Регулярные выражения

Регулярные выражения часто превращаются в «магические строки»:

```
// Плохо [X]
if (preg_match('/^(\d{4})-(\d{2})-(\d{2})$/', $date, $matches))
{
    $year = $matches[1];
    $month = $matches[2];
    $day = $matches[3];
}
```

Через месяц уже непонятно, что означает каждая группа. И код начинает становиться путанным и явно обращаться к числовому индексу.

По возможности, следует использовать именованные группы:

```
// Хорошо [✓]
if (preg_match(
    '/^(?<year>\d{4})-(?<month>\d{2})-(?<day>\d{2})$/',
    $date,
    $matches
)) {
    $year = $matches['year'];
    $month = $matches['month'];
    $day = $matches['day'];
}
```

Теперь каждая часть регулярного выражения имеет понятное имя, и при чтении кода сразу ясно, что именно извлекается.

Ссылки делают код хрупким

Передавать переменную по ссылке кажется удобным: функция сразу меняет её — меньше кода, меньше присваиваний. Вот пример:

```
// Плохо [X]
function celsiusToFahrenheit(float &$celsius): void
{
    $celsius = $celsius * 9 / 5 + 32;
}

$temp = 25;
celsiusToFahrenheit($temp);
echo $temp; // 77 — значение изменилось «внутри» функции
```

На первый взгляд это экономия кода, но есть подвох: после вызова функции уже не понятно, изменится переменная или нет. Изменения происходят «за кадром», что усложняет чтение и поддержку кода.

Из-за этого код становится хрупким — любое забытое или неожиданное изменение может сломать логику программы.

Лучшей практикой считается писать функции, которые принимают значение и возвращают новый результат, не изменяя исходные данные:

```
// Хорошо [✓]
function celsiusToFahrenheit(float $celsius): float
{
    return $celsius * 9 / 5 + 32;
}

$temp = 25;
$tempInFahrenheit = celsiusToFahrenheit($temp);
echo $temp; // 25 — значение не изменилось
```

Такой код прозрачный и предсказуемый: переменные не меняются «вдруг», а результат возвращается явно и используется там, где нужно.

Место для расширения

Иногда нужно немного изменить поведение класса — добавить одно условие, поменять одну строчку, подставить другую функцию. Казалось бы, мелочь, но уже появляются соблазны.

Часто мы думаем: «Создам новый класс, унаследуюсь от старого и переопределю нужный метод». Быстро, просто и вроде аккуратно.

Например, есть простой класс который группирует новостные сводки погоды по городам:

```
$news = [  
    "В Москве температура побила рекорд 2013 года",  
    "В Москве жарко и солнечно",  
    "В Липецке дожди идут без остановки",  
    "В Липецке на этой неделе дождливая погода",  
];  
  
$grouper = new NewsGrouper($news);  
// ['Москва' => [...], 'Липецк' => [...]]
```

И нужно изменить алгоритм сравнения двух заголовков. Часто мы думаем: «Создам новый класс, унаследуюсь от старого и переопределим нужный метод». Быстро, просто и вроде аккуратно.

Например, так:

```
class NewsGrouperBySimilarText extends NewsGrouper  
{  
    protected function similar(string $a, string $b): bool  
    {  
        similar_text($a, $b, $percent);  
  
        return $percent > 51;  
    }  
}
```


А затем еще один:

```
class NewsgrouperBySoundex extends Newsgrouper
{
    protected function similar(string $a, string $b): bool
    {
        return soundex($a) === soundex($b);
    }
}
```

На первый взгляд — разумно. Но через месяц таких потомков будет уже пять, потом десять. Затем понадобится изменить в части наследников нормализацию имен для городов. Кто-то добавит изменения в родительский класс, чтобы «не трогать потомков» — и структура начинает шататься.

Вы смотрите на этот зоопарк и задаётесь вопросом: где живёт нужная логика? В родителе? В потомке? В обоих? В каком из пятнадцати классов?

Всё стало сложнее, чем было. Хотя вы просто хотели добавить одно небольшое изменение.

Почему так происходит? Потому что в коде нет места для изменений без разрушений. Нет точки расширения — шва (seam). В результате единственный способ изменить поведение — наследование, переопределение и копипастинг.

Вместо наследования можно сделать в классе точку расширения — шов, который позволяет изменить поведение без правки существующего кода. Например, передать колбэк или другой объект. Главное — не трогать внутренности класса, всё меняется снаружи.

Простой пример:

```
class Newsgrouper {
    public function groupBy(callable $similar): static {
        $this->similar = $similar;
        return $this;
    }
}
```

Теперь вы определяете, как именно форматировать отчёт:

```
$grouper = new Newsgrouper($news);
$grouper->groupBy(function (string $a, string $b): bool {
    similar_text($a, $b, $percent);

    return $percent > 51;
});
```

Никаких наследников. Всё поведение — в одном месте. Тестировать такой класс легко, расширять — ещё проще.

Когда его будет недостаточно или он разрастётся, замените `callable` на интерфейс и передавайте объект, реализующий его.

```
$grouper = new Newsgrouper($news);
$groups = $grouper->groupBy(new SimilarComparator());
```

Идея очень проста: Не спешите писать `extends`. Спросите себя: можно ли здесь оставить шов? Если да — вы только что сделали код гибче, проще и чище.



Ранний выход

Когда рассказываешь историю и постоянно перебиваешь себя новыми деталями, слушатель теряется, забывает, с чего всё началось и ему становится всё труднее следить за ходом мысли. В итоге рассказ превращается в запутанную смесь, которая теряет смысл.

В программировании аналогичная ситуация возникает, когда в функции слишком глубоко вкладываются условия и циклы. Это снижает читаемость и усложняет понимание логики.

В прошлой главе мы говорили о важности отдавать результат как можно быстрее. Одним из препятствий для этого является глубокая вложенность кода. Часто мы мысленно строим блок-схему:

«Если X равно Y, тогда выполнить действие, если Y равно Z, выполнить другое действие» — и так далее.

Пример глубокой вложенности:

```
// Плохо [X]
if($condition) { // уровень 1
    foreach($users as $user) { // уровень 2
        if($user->isActive()) { // уровень 3
            // Обработка
        }
    }
}
```

Каждый новый уровень вложенности усложняет чтение — нелинейно, а экспоненциально. На третьем уровне вложенности уже приходится держать в голове весь предыдущий контекст. Это утомляет и увеличивает когнитивную нагрузку.

Стоит стремиться минимизировать глубину вложенности и предпочитать располагать основные сценарии выполнения функции без вложенных условий.

```
// Плохо [X]
if ($condition) {
    // много кода
} else {
    throw new Exception('Условие не выполнено');
}
```

Лучше написать так:

```
if (! $condition) {
    throw new Exception('Условие не выполнено');
}

// много кода
```

Это называется **ранним выходом (early return)**. Мы сразу обрабатываем граничные случаи, а затем плавно движемся по основному сценарию — без вложенности и лишних условий.

Такой подход упрощает чтение и понимание кода, делает его более структурированным и лёгким для поддержки. Ранний выход актуален не только для исключений, но и для возвращений значений в методах, например:

```
// Плохо [X]
public function hasAssign(User $user): bool
{
    if ($condition) {
        return // ... много кода;
    }
    return false;
}
```

Основной сценарий спрятан внутри условия. Лучше развернуть его наружу:

```
// Хорошо [✓]
public function hasAssign(User $user): bool
{
    if (! $condition) {
        return false;
    }

    // ... много кода
    return $result;
}
```

В хорошем варианте мы сначала отсекаем негативные случаи, а затем идём по основному сценарию, который расположен в основной части функции. Это делает код чище и проще для понимания.

Ранний выход полезен не только для условий, но и для циклов:

```
// Плохо [X]
foreach ($orders as $order) {
    if ($order->isPaid()) {
        foreach ($order->items as $item) {
            if ($item->isInStock()) {
                // обработка
            }
        }
    }
}
```

Можно использовать `continue` для выхода из итерации:

```
// Хорошо [✓]
foreach ($orders as $order) {
    if (! $order->isPaid()) {
        continue;
    }

    foreach ($order->items as $item) {
        if (! $item->isInStock()) {
            continue;
        }
        // обработка
    }
}
```

Избегайте использования `else`

Отдельного упоминания требует оператор `else`. Сам по себе он не является злом, но часто его использование указывает на неудачную структуру кода и отличным решением будет минимизировать его использование, заменяя его ранним выходом. Тогда вместо громоздких условных блоков будут простые и прямолинейные конструкции.

Рассмотрим метод проверки доступа:

```
// Плохо [X]
public function hasAccess(User $user): bool {
    if (!$user->isBanned()) {
        if ($user->isAdmin()) {
            // Доступ разрешён: админ.
            return true;
        } else {
            if ($user->isGranted(GRANT::EDIT)) {
                // Доступ разрешён: может редактировать.
                return true;
            } else {
                // Доступ запрещён: нет нужных прав.
                return false;
            }
        }

        return false;
    }
    // Доступ запрещён: пользователь заблокирован.
    return false;
}
```


Вариант без `else` — чище и проще:

```
// Хорошо [✓]
public function hasAccess(User $user): bool
{
    if ($user->isBanned()) {
        // Пользователь заблокирован
        return false;
    }

    if ($user->isAdmin()) {
        // Пользователь является администратором
        return true;
    }

    // Пользователь имеет разрешение на редактирование
    return $user->isGranted(GRANT::EDIT);
}
```

Хороший код, как хороший рассказ, не уводит читателя в сторону. Ранний выход — один из самых эффективных приёмов. Применяйте его — и ваши функции станут яснее, проще и приятнее для чтения.



Управляющие конструкции

Хороший код строится из простых и понятных блоков. Мы уже научились избегать лишней вложенности и выходить из метода как можно раньше. Но есть ещё одна область, где может скрываться неявная сложность, способная незаметно разрастаться — это **управляющие конструкции**.

В живом проекте требования неизменно растут. Это нормально: бизнес двигается, пользователи чего-то хотят, а нам приходится подкручивать код, чтобы он всё это выдержал. Но вместе с требованиями растёт и сложность. И одна из самых тихих, но опасных зон роста — это управляющие конструкции.

Они сначала выглядят безобидно. Один `if`, один `while`, пара сравнений. Всё понятно. Но потом приходит ещё один параметр. Потом фильтр. Потом проверка статуса, даты, порогового значения. И вроде бы всё ещё нормально — но код уже не читается.

Посмотрим на конкретный пример:

```
while (File::where('status', '=', File::STATUS_NEW)->count()) {  
    // ...  
}
```

На первый взгляд всё просто и понятно: пока есть новые файлы — продолжаем обработку. Всё логично.

Через пару недель появляется задача обработать только файлы, связанные с определённым событием:

```
while (File::where('event_guid', '=', $event->document_id)->where('status', '=', File::STATUS_NEW)->count()) {  
    // ...  
}
```

А ещё через день кто-то убирает лишние `=` из условий. Или добавляет:

```
- while (File::query()->where('event_guid', '=', $event->document_id)->where('status', '=', File::STATUS_NEW)->count()) {  
+ while (File::query()->where('event_guid', $event->document_id)->where('status', File::STATUS_NEW)->count()) {  
    // ...  
}
```

Такой `git diff` неудобно читать: сложно сразу заметить, что именно поменялось. Вам не хочется разбираться, что конкретно изменилось, только побыстрее пройти мимо.

Но и более развернутый вариант с переносами строк — лишь чуть улучшает `diff`, но не облегчает жизнь при отладке:

```
while (  
    File::where('event_guid', $event->document_id)  
        ->where('status', File::STATUS_NEW)  
        ->count()  
) {  
    // ...  
}
```

Допустим, мы видим эту часть кода, как нам узнать, сколько записей вернулось? Придётся скопировать весь запрос, передать его в `dd()` или функцию логирования.

А если условие будет сложнее, чем простая проверка на ненулевое значение, например нужно сравнить количество с порогом, возвращаемым из другого метода? Тогда нам придётся копировать уже дважды:

```
$count = File::where('event_guid', $event->document_id)
->where('status', File::STATUS_NEW)
->count();

dd([
    'count' => $count,
    'secret' => $secret,
]);

while (
    File::query()
        ->where('event_guid', $event->document_id)
        ->where('status', File::STATUS_NEW)
        ->count()
        >= $secret
) {
    // ...
}
```

Вместо этого воспользуемся ранним выходом, с которым мы познакомились ранее, и вынесем условие:

```
while (true) {  
    $count = File::where('event_guid', $event->document_id)  
        ->where('status', File::STATUS_NEW)  
        ->count();  
  
    if ($count <= $secret) {  
        break;  
    }  
  
    // ...  
}
```

А ещё лучше — спрятать проверку, как только условие перестаёт быть тривиальным — вынести его в отдельный метод с говорящим именем:

```
// Хорошо [✓]  
while ($this->hasTooManyNewFiles()) {  
    // ...  
}
```

И где-нибудь в коде:

```
private function newFilesQuery()
{
    return File::where('event_guid', $this->event->document_id)
        ->where('status', File::STATUS_NEW);
}

private function hasTooManyNewFiles(): bool
{
    return $this->newFilesQuery()->count()
        > $this->threshold();
}
```

Избегай «мудрёных» решений

Сложные условия не всегда находятся только в if или while. Иногда они маскируются под «краткость» — особенно в тернарных или null coalescing-выражениях.

```
// Плохо [X]
return $cache ?: ($computed ?: $default);
```

Трудно сходу сказать, что вернётся в каждой ситуации. Краткость не самоцель — важнее, чтобы код был понятен с первого взгляда:

```
// Хорошо [✓]
if ($cache) {
    return $cache;
}

if ($computed) {
    return $computed;
}

return $default;
```

Присваивания в условиях

Иногда хочется быть «умным». Сделать красиво, коротко, выразительно. Одной строкой. Как в старых учебниках:

```
// Плохо [X]
if ($user = $this->getUser()) {
    // ...
}
```

Метод вызвали, результат проверили, в переменную записали — три в одном. Сэкономили строку. Но только не время других разработчиков. Потому что наш смысл уехал в сторону, мы создали дополнительную когнитивную нагрузку. Продолжим пример:


```
// Плохо [X]
if ($user = $this->getUser()) {
    $this->sendNotification($user);
}
```

На беглом просмотре кажется, что мы сравниваем `$user` с чем-то. Только приглядевшись, понимаешь — ага, тут присваивание, и оно возвращает значение. Но это нужно замечать.

А если переменная `$user` объявлена где-то выше? Нужно держать в голове, что здесь происходит **переопределение**, и это влияет на остальной код.

Уместно будет не создавать неявных сложностей и всегда разделять: сначала присвоение, потом условие.

```
// Хорошо [✓]
$user = $this->getUser();

if ($user !== null) {
    $this->sendNotification($user);
}
```

Отрицания в условиях

Иногда нужно проверить, что **что-то не произошло**. Пользователь не авторизован, не найден, не соответствует. Самый очевидный путь — добавление знака отрицания **!** в условие:

```
// Плохо [X]
if (!$user->isActive()) {
    // ...
}
```

С технической точки зрения это работает, но с точки зрения читаемости — это проблема. Приходится мысленно инвертировать название метода, а это **добавляет когнитивную нагрузку**. Гораздо понятнее, когда условие говорит само за себя, без отрицаний:

```
// Хорошо [✓]
if ($user->isInactive()) {
    // ...
}
```

Конечно, может быть, такого метода изначально нет. Но если такое условие встречается часто, стоит добавить его — даже если он просто возвращает отрицание другого метода:

```
public function isInactive(): bool
{
    return !$this->isActive();
}
```

То же самое касается переменных:

```
// Плохо [X]
if (!$hasErrors) {
    // ...
}
```

В голове снова приходится прокручивать: **«если не есть ошибки, тогда продолжить»**. Лучше сделать переменную с положительным смыслом:

```
// Хорошо [✓]
if ($isValid) {
    // ...
}
```



Аргументы

Чем меньше, тем лучше

Методы с большим числом аргументов сложнее читать, тестировать и использовать. **Правило трёх:** метод не должен принимать больше **трёх параметров**. Если больше — разделите.

```
// Плохо [X]
$fileSystem->write(
    '/path/to/file.txt', // Путь до файла
    true,                // Перезапись файла
    'Пример данных',     // Содержимое
    'UTF-8',             // Кодировка
    true                 // Включаем логирование
);
```

Если у метода четыре, пять, а то и шесть параметров — становится сложно понять, что есть что, в каком порядке это передавать и как вообще это использовать. Особенно это усугубляется, когда имена аргументов очень похожи.

Даже если вы напишете великолепный комментарий перед методом, человек читающий код будет вынужден каждый раз к нему возвращаться.

Необязательные аргументы — в конец

При проектировании методов порядок аргументов имеет значение. Один из самых простых и эффективных способов сделать его чище — располагать необязательные параметры в конце.

Рассмотрим пример:

```
// Плохо [X]
$fileSystem->write(
    '/path/to/file.txt', // Путь до файла
    null,                // Перезапись файла
    'Пример данных',     // Содержимое
);
```

Чтобы просто записать файл, нам приходится явно указывать `null` — значение, которое на самом деле нам не нужно.

Куда лучше такой вариант:

```
// Хорошо [✓]
$fileSystem->write(
    '/path/to/file.txt', // Путь до файла
    'Пример данных',     // Содержимое
);
```

А если нужно изменить поведение по умолчанию, мы просто добавим третий параметр:

```
// Плохо [X]
$fileSystem->write(
    '/path/to/file.txt', // Путь до файла
    'Пример данных',     // Содержимое
    true,                // Перезапись файла
);
```

В этом случае метод будет принимать только обязательные параметры, а необязательные будут в конце. Это делает код чище и понятнее. Так как их можно не указывать, если они не нужны.

Что делать, если аргументов много

Иногда метод требует не один-два, а сразу пять или больше параметров. Передавать всё списком в строго заданном порядке — не лучшая идея. Легко перепутать аргументы, особенно если они одного типа. К тому же вызов такого метода выглядит пугающе и плохо читается.

Первое, что приходит на ум это использование ассоциативного массива:

```
// Плохо [X]
$fileSystem->write(
    '/path/to/file.txt',
    'Пример данных',
    [
        'encoding' => 'UTF-8',
        'overwrite' => true,
        'debug' => true,
    ]
);
```

Это отвратительный способ. Такой подход не даёт информации о том, какие параметры действительно ожидаются, и не позволяет IDE подсказывать возможные опции. Более того, здесь нет проверки типов — любые ошибки проявятся только во время выполнения. Это усложняет отладку и увеличивает вероятность багов.

Другая популярная попытка — создать объект, инкапсулирующий значения. Например:

```
$config = new Config($encoding, $overwrite, $debug);

// Пример использования
$fileSystem->write(
    '/path/to/file.txt', // Путь до файла
    null,                // Перезапись файла
    $config,             // Объект с метаданными
);
```

Это лишь видимость решения. Мы создали объект, который сам по себе бессмысленен: он не содержит поведения и не добавляет никакой бизнес-логики. Фактически, это тот же массив, только завернутый в класс. Польза от него — разве что автодополнение в IDE. Но теперь мы должны создавать или таскать этот объект везде, где вызываем метод `write`, что только усложняет код.

Если язык поддерживает именованные аргументы и их количество очень-очень ограничено, стоит использовать их:

```
$fileSystem->write(
    '/path/to/file.txt',
    'Пример данных',
    debug: true, // Именованный параметр
);
```

Это уже лучше: вызов становится самодокументируемым, и порядок аргументов не имеет значения.

Но есть гораздо более выразительный и управляемый способ — **fluent-интерфейс**. Это объект, методы которого возвращают самого себя, позволяя вызывать их цепочкой:

```
// Хорошо [✓]
$fileSystem
  ->path('/path/to/file.txt')
  ->encoding('UTF-8')
  ->overwrite(true)
  ->debug(true)
  ->write('Пример данных');
```

В этом подходе сразу видно, что происходит. Каждый шаг отделён, названия методов описывают действия, и вся цепочка читается как связный набор настроек. Такой стиль легко расширяется, хорошо покрывается тестами и открывает дорогу к более гибкой архитектуре.

Булевы аргументы

Стоит отдельно упомянуть, что многие разработчики и известные авторы, например, Роберт Мартин — автор «Чистого кода», считают использование булевых аргументов признаком плохого тона. И предлагают создавать отдельный метод вместо передачи булева значения.

Например, вместо:

```
$fileSystem->write(  
    '/path/to/file.txt', // Путь до файла  
    'Пример данных',    // Содержимое  
    true                 // перезаписать файл  
);
```

Предпочтительнее сделать:

```
$fileSystem->reWrite(  
    '/path/to/file.txt', // Путь до файла  
    'Пример данных',    // Содержимое  
);
```

Однако я бы поспорил с этой категоричной рекомендацией. В ряде случаев булевый параметр — вполне удобный и компактный способ управления поведением метода, особенно если код остаётся понятным.

Но если булевый аргумент **существенно меняет поведение метода**, превращая его фактически в две разные функции — тогда действительно стоит рассмотреть разделение.

Предпочитайте объекты

Строки, булевы, числа очень удобны в начале разработки, но с течением времени логика усложняется, и эти простые значения не справляются.

Что раньше было флагом `true`, теперь требует дополнительных условий: *если админ, если включён режим отладки, если пользователь подтвердил e-mail*.

Скалярные значения не умеют расти. Они не подстраиваются под новые требования. А объект — может. Он расширяется методами, валидирует себя, хранит контекст и смысл.

Рассмотрим пример списка исключений. Вместо того чтобы передавать набор строк, лучше использовать объект, который сам знает, как представлять себя:

```
// Плохо [X]
class ExcludeList
{
    public function add(
        string $itemName,
        string $itemIdentityName,
        string $itemIdentityValue
    ): void
    {
        // ...
    }

    public function has(
        string $itemName,
        string $itemIdentityName,
        string $itemIdentityValue
    ): bool
    {
        // ...
    }
}
```

Вместо того чтобы передавать несколько строковых значений, можно использовать уже существующий объект или создать новый, который сам решит, как обработать добавление и поиск элемента:

```
// Хорошо [✓]
class ExcludeList
{
    public function add(Model $model): static
    {
        $key = $model->getKey();
        // ...
    }

    public function has(Model $model): bool
    {
        $key = $model->getKey();
        // ...
    }
}
```

Теперь метод `add` и метод `has` работают с объектами, а не с простыми значениями. Это упрощает добавление новых параметров и изменений в модель, не затрагивая логику работы методов, а также облегчает тестирование.



Обработка ошибок

Каждый раз, когда вы пишете код, вы должны помнить о том, что он может сломаться. Ошибки могут возникать по самым разным причинам: от неправильного ввода данных до сбоев в работе внешних сервисов. Поэтому важно правильно обрабатывать ошибки, чтобы ваш код не падал и не оставлял пользователей в недоумении.

Исключения

Как правило, исключения, которые вы ожидаете и планируете обрабатывать заранее, должны наследоваться от `Exception`. Те же исключения, которые вы создаёте, но обработка которых необязательна или не предусмотрена, — лучше наследовать от `RuntimeException`.

Например, это ошибки, возникающие из-за неверного состояния программы, неправильного использования API или логических ошибок.

Это связано с понятием **unchecked exceptions** — исключений, которые не требуют обязательной обработки. Они не могут быть предсказаны заранее и проявляются только во время выполнения программы.

В PHP, в отличие от некоторых других языков, нет строгого разделения на checked и unchecked исключения, но по смыслу `RuntimeException` относится именно к категории unchecked.

Если ваш класс наследуется от `Exception` и не обработан, инструменты вроде PhpStorm и статические анализаторы могут обратить на это внимание, предупреждая, что исключение не перехвачено, в то время как исключения, наследуемые от `RuntimeException`, в этом плане рассматриваются иначе.

Скрытое замалчивание

Абсолютно ни в коем случае нельзя «поймать и забыть» исключение. Пустой catch или молчаливое подавление ошибок:

```
// Плохо [X]
try {
    $this->calculate($data);
} catch (\Throwable $throwable) {
    // ничего не делаем
}
```

Пример — «смертельно» опасен: проблема произошла, но никто об этом не узнает. Главное — не потерять факт ошибки. Так делать нельзя: ошибка уходит в тень, вы теряете информацию.

В некоторых случаях, чтобы избежать падения приложения, можно вернуть резервный результат:

```
// Плохо [X]
try {
    $message = $this->greeting($time);
} catch (ExternalApiException $exception) {
    $message = 'Добро пожаловать!';
}
```

Иногда ошибку можно безопасно обработать без прерывания выполнения, но её обязательно нужно залогировать для последующего анализа:

```
// Хорошо [✓]
try {
    $message = $this->greeting($time);
} catch (ExternalApiException $exception) {
    Log::warning('Не удалось получить приветствие', [
        'arguments' => ['time' => $time],
        'exception' => $exception,
    ]);

    $message = 'Добро пожаловать!';
}
```

Ясные ошибки

Сообщения об ошибках должны быть максимально информативными и конкретными. Вместо абстрактного «RuntimeException» нужно описывать, что именно произошло и где:

```
// Плохо [X]
foreach ($users => $user) {
    if ($user->isActive()) {
        throw new Exception(
            'Пользователь должен быть неактивен для удаления'
        );
    }
}
```

Невозможно понять, какой именно пользователь вызвал ошибку. Чтобы исправить, нужно добавить контекст:


```
foreach ($users as $user) {
    if ($user->isActive()) {
        throw new RuntimeException(sprintf(
            'Нельзя удалить активного пользователя: ID=%d',
            $user->id,
        ));
    }
}
```

Следи за логами

Чтобы не пропускать серьёзные проблемы, все необработанные исключения нужно обязательно фиксировать в логах. Логи — это ваша единственная возможность узнать, что пошло не так, когда программа работает в производственной среде.

Обычно ошибки записывают в файл на диске и/или отправляют во внешний сервис. Но важно это делать последовательно. Посмотрите на пример:

```
try {
    // ...
} catch (Throwable $e) {
    file_put_contents('log-process.txt', 'Ошибка #5');
}
```

В другом месте:

```
try {  
    // ...  
} catch (Throwable $e) {  
    file_put_contents('log-user.txt', json_encode([  
        'line' => $e->getLine(),  
        'status' => 500,  
    ]));  
}
```

В этом примере исключения обработаны, но их структура абсолютно разная: в одном — формат JSON, а в другом — строка. Это не позволит автоматически проанализировать их и создаст лишнюю когнитивную нагрузку.

Вместо этого используйте единую точку для логирования:

```
function logException(Throwable $e): void  
{  
    $message = sprintf(  
        "[%s] %s in %s on line %d\nStack trace:\n%s\n\n",  
        date('Y-m-d H:i:s'),  
        $e->getMessage(),  
        $e->getFile(),  
        $e->getLine(),  
        $e->getTraceAsString()  
    );  
  
    //  
}
```

А ещё лучше — использовать современный подход с PSR-3-совместимыми логгерами, такими как [Monolog](#). Он поддерживает уровни логирования, форматирование, хендлеры и интеграцию с внешними системами.

Важно также не писать всё бесконтрольно в один файл, так как его содержание будет расти. Это может вылиться в то, что любимая IDE или редактор не смогут открыть файл на 30 ГБ, не говоря уже о том, чтобы осуществить эффективный поиск по нему. Чтобы такого не произошло, можно настроить ротацию логов — например, чтобы каждый новый день старый лог переименовывался и архивировался, а спустя некоторое время удалялся. Так у вас будет меньше мусорных данных, которые не нужны в контексте.

Но просто собирать логи мало. Нужно, чтобы вы сразу узнавали о проблемах. Для этого подключают централизованные системы мониторинга: Sentry, Graylog, ELK или другие. Они собирают все ошибки в одном месте, позволяют группировать и отслеживать повторяющиеся баги, а при необходимости — шлют оповещения на почту или в мессенджеры.

Отладка

Среди разработчиков распространено мнение, что пошаговая отладка, например, с помощью Xdebug, — признак хорошего специалиста. Выглядит это следующим образом: разработчик ставит точку остановки, затем начинает «шагать» по коду, наблюдает за значениями переменных, отслеживает условные переходы — словно читает чужие мысли. Якобы именно так приходит понимание, «как всё устроено».

Но позвольте — если вам нужно так делать, значит, что-то пошло не так!

Это не норма. Значит, ваш код неочевиден, сложен и плохо структурирован.

Настоящая причина, по которой вам нужно пошагово проходить каждую строчку, в том, что вы не понимаете, что происходит в системе. И не потому что вы недостаточно умны, а потому что код запутан. В нём всё связано со всем, всё влияет на всё, и даже чтобы просто проверить, вам приходится запускать сервер, кликать через интерфейс и ставить точку остановки где-то внутри `shouldGoOutside()`.

Представим, что система должна вернуть рекомендацию пользователю — выходить ли на улицу:

```

// Плохо [X]
function shouldGoOutside(array $weather): bool
{
    return ! (function () use (&$weather, $check) {
        try {
            extract($weather, EXTR_SKIP);
            if ($this->check($temp, -10, 35, function ($t) {
                return $this->isExtremeTemperature($t);
            }) {
                return false;
            }

            // ...

            foreach ($alerts as $carry) {
                try {
                    return $this->shouldPanic($carry);
                } catch (Throwable) {}
            }
        } catch (Throwable $throwable) {
            Log::error($throwable);
            return false;
        }
    })();
}

```

Здесь есть и вложенные блоки `try-catch`, и `if` внутри `if`, и цикл с ловлей исключений «внутри» функции. Без отладки трудно понять, за какой именно шаг «цепи» падает — внешняя обработка или внутренняя.

А теперь посмотрите, как это должно быть устроено:

```
final class DecisionEngine
{
    /**
     * @param WeatherRule[] $rules
     */
    public function __construct(
        private array $rules
    ) {}

    public function shouldGoOutside(array $weather): bool
    {
        foreach ($this->rules as $rule) {
            if (!$rule->passes($weather)) {
                return false;
            }
        }

        return true;
    }
}
```

Всё. Логика выделена, изолирована, читается за секунду.

Вы можете протестировать её без всякого дебаггера в тестах:

```
$engine = new DecisionEngine([
    new TemperatureRule(),
    new WindRule(),
    new PrecipitationRule(),
    new NoSevereStormAlertRule(),
]);

$engine->shouldGoOutside([
    'temperature' => 20,
    'wind' => 5,
    'precipitation' => 10,
    'alerts' => [],
]);
```

А если нужно что-то проверить, то мы можем легко добавить тест:

```
$rule = new WindRule();

$this->assertTrue($rule->passes(['wind' => 10]));
$this->assertFalse($rule->passes(['wind' => 25]));
$this->assertTrue($rule->passes([])); // Ветра нет — нормально
```

Отладка нужна, когда вы не можете локализовать поведение. В хорошо структурированном коде вместо Xdebug — вы пишете `->assertTrue(...)`.



Комментарии

Комментарий — это способ передачи мысли не компьютеру, а человеку, который в будущем будет его читать и работать с ним.

Полезный комментарий объясняет, почему написан тот или иной код, а не дублирует очевидное. Комментарии должны раскрывать контекст и мотив, а не пересказывать строку кода.


```
// Плохо [X]
// Устанавливаем переменную в 5
$counter = 5;

// Добавить 1 к счётчику
$counter++;
```

Здесь комментарии не добавляют никакой ценности — код и так очевиден. Они просто повторяют написанное и засоряют пространство, вместо того чтобы объяснить логику или замысел.

```
// Хорошо [✓]
// Начинаем с 5, потому что первые 5 пользователей
// уже были зарегистрированы вручную
$counter = 5;

// Увеличиваем счётчик, потому что обработан новый заказ
$counter++;
```

Здесь комментарии оправдывают поведение. Они отвечают на вопрос: «Почему 5?» и «Почему инкремент?» Это контекст, который не видно из самого кода. Именно в этом — ценность комментария.

Иногда комментарий вообще не нужен — достаточно хорошо подобранного имени переменной:

```
// Плохо [X]
// date for yesterday
$date = date('Y-m-d', strtotime('yesterday'));
```

Имя переменной — это уже встроенный комментарий. Используйте силу языка. Не пиши объяснений в воздухе — дай имя понятию.

```
// Хорошо [✓]
$yesterday = date('Y-m-d', strtotime('yesterday'));
```

Это намного лучше любого комментария.

Обновляйте комментарии вместе с кодом

Если вы меняете логику, обязательно обновите комментарий. Иначе он превращается в ложь, а ложь хуже полного отсутствия информации.

```
// Плохо [X]
// Этот метод работает с MySQL 5.7
// поэтому не использует JSON-функции
public function processData($data)
{
    // Код, который уже использует JSON-функции
}
```

Комментарий устарел, но код остался. Теперь он вводит в заблуждение. Либо обновите комментарий, либо удалите его.

Также при выполнении срочной задачи мы заранее понимаем, что код может быть неидеальным или потребовать доработок, и тем самым как бы заранее оправдываем это через:

```
// TODO: временное решение, нужно переписать  
// FIXME: костыль, но работает  
// HACK: не очень красиво, но быстро
```

Вместо комментария-оправдания исправьте код. Если времени нет — создайте задачу в трекере. Или настройте инструмент автоматизации, который по ключевым словам будет создавать новые задачи.

Многострочные комментарии

Многострочные комментарии — это особый жанр. Они должны быть не просто информативными, но и визуально удобными. Чем легче комментарий воспринимается глазом, тем быстрее программист понимает код.

Часто можно встретить такой вариант:

```
// Плохо [X]  
  
/**  
 * Сохраняем пользователя сразу,  
 * чтобы избежать потери данных, ведь последующие  
 * действия могут вызвать исключение.  
 * Ранее делали событие, но было ненадёжно.  
 */
```

Проблема здесь в том, что строки разбиты случайно: одна слишком короткая, другая — длинная. В итоге комментарий выглядит тяжело и рвано.

Гораздо лучше, чтобы каждая последующая строка была такой же длины или чуть компактнее предыдущей. Комментарий тогда превратится в лестницу, по которой глаз легко скользит вниз.

Такой приём превращает комментарий в ступеньки, по которым взгляд движется естественно:

```
// Хорошо [✓]

/**
 * Сохраняем пользователя сразу, чтобы избежать потери данных,
 * ведь последующие действия могут вызвать исключение.
 * Ранее делали событие, но было ненадёжно.
 */
```

Комментарий с примером

Комментарии в конфигурационных файлах часто страдают от излишней абстрактности. Например, разработчику нужно указать, где находятся SVG-иконки приложения:

```

/*
|-----
| Icons Path
|-----
|
| Provide the path from your app to your SVG icons directory.
|
*/

'icons' => [],

```

Возникают вопросы: каков формат массива, путь полный или относительный? Без конкретного примера разработчик вынужден угадывать или спрашивать у коллег, что приводит к дополнительному времени и несамостоятельности. Чтобы исправить ситуацию, нужно добавить пример использования:

```

/*
|-----
| Icons Path
|-----
|
| Provide the path from your app to your SVG icons directory.
|
| Example: ['fa' => storage_path('app/fontawesome')]
|
*/

'icons' => [],

```

Теперь комментарий не только объясняет идею, но и даёт готовую основу для настройки. Абстрактная фраза превратилась в конкретное знание, и разработчику не нужно гадать.



Не бойся удалять код

Удаление кода так же важно, как и его написание. Это акт заботы о проекте. Вы следите за его чистотой, избавляетесь от ненужного, делаете структуру понятнее и помогаете другим быстрее разобраться, как всё устроено. Это ни в коем случае не поражение, это не «зря писал». Это развитие. Это значит, что вы переросли старое решение, нашли лучшее или поняли, что оно больше не нужно.

Последовательность важнее

Если вы внимательно следовали предыдущим главам, вы уже начали менять свой код: упрощать, переименовывать, разделять ответственность. И вот тут появляется важный момент: последовательность. Когда код меняется, названия устаревают. Логика перерастает своё оформление — нужно адаптироваться как можно быстрее и без страха.

Когда-то метод `validate()` проверял логин и пароль. Сегодня он уже проверяет два токена, куки и капчу. А называется всё ещё `validate()`. И это вводит в заблуждение. Сегодняшнему разработчику нужно читать реализацию, потому что по названию уже ничего не понятно.

Код меняется, и ему нужно давать возможность изменяться — в том числе через переименование и удаление. Не стоит оставлять старое имя «ради совместимости» или из лени. Это путь к путанице. Пусть код выглядит так, как он работает сейчас, а не когда-то давно.

Избавляйся от закомментированного кода

Бывало ли у вас: «Пока закомментирую, может, потом пригодится»? Или: «Это временно, на выходных уберу»?

Проходит неделя. Потом две. Потом месяц. И вот закомментированный кусок кода перекачивается из релиза в релиз — как будто это часть системы. Только на деле — это мусор, который мы сами оставили.

```

// Плохо [X]
public function generateAccessToken(): string
{
    $userId = $this->user->getKey(),
    // Log::info("Generating token for user: $userId");

    $payload = [
        // 'role' => 'user',
        // 'aud' => 'my-app-client',
        'sub' => $userId,
        'exp' => $this->calculateExpiration(),
    ];

    // Старый способ (оставлен на всякий случай)
    // $token = base64_encode(json_encode($payload));

    $token = $this->signToken($payload);

    // echo "Generated token: $token";
    return $token;

    // Всё, что ниже — никогда не выполнится
    $this->logTokenGeneration($userId, $token);
    // $refreshToken = $this->generateRefreshToken($userId);
    // $this->storeRefreshToken($userId, $refreshToken);
}

```

Во-первых, шум. Когда открываешь файл и видишь кучу закомментированного кода, начинаешь гадать: это ещё работает? Это было важно? А может, наоборот, это старая логика, которую уже заменили? Такой шум мешает сосредоточиться и понять, какой код сейчас «правильный».

Во-вторых, портится дисциплина. Стоит один раз оставить закомментированный фрагмент — и понеслось. Один, второй... В проекте появляются простыни закомментированных строк, которые никто не трогает, но все боятся удалить. «А вдруг кто-то оставил это не просто так?..» Так команда привыкает к мысли: захламлять код — это нормально.

Нет, это не нормально. Вот как должно быть:

```
// Хорошо [✓]
public function generateAccessToken(): string
{
    return $this->signToken([
        'sub' => $this->user->getKey(),
        'exp' => $this->calculateExpiration(),
    ]);
}
```

Коротко. Просто. Чисто. Только то, что нужно. А всё остальное уже сохранено в [Git](#), и если понадобится — всегда можно восстановить.

Без тестов страшно

Когда в проекте нет автоматических тестов, появляется страх. Страх, что твои изменения могут сломать что-то важное, о чём ты даже не вспомнишь, пока не увидишь ошибку в продакшене или не получишь жалобу от пользователя.

Этот страх подталкивает к самому худшему — к FDD (Fear Driven Development), когда ты боишься что-то менять, боишься удалять устаревший код, боишься переименовывать методы и даже боишься добавить новую фичу.

Без тестов ты вынужден постоянно держать в голове кучу деталей — как работает старый функционал, какие побочные эффекты могут быть у твоих изменений, где спрятаны зависимости и что сломается, если изменить этот кусок.

Из-за этого возникает нервозность, замедляется работа, а код превращается в хаос, потому что проще не трогать, чем проверить.

Поэтому инвестируй в своё спокойствие — пиши тесты. Они освободят тебя от страха, позволят спокойно менять и улучшать код без риска всё сломать.



Тесты

Слово **тестирование** испортили и изуродовали. Потому что часто под этим подразумевают ручную проверку, что задача, которую делал разработчик, работает как задумано после изменений. Это не то, что нужно разработчику — нам нужен контроль качества.

Многие до сих пор думают, что тестированием занимается кто-то другой: тестировщик, QA-инженер, автоматизатор или великий господин начальник. Это не так.

Ты — разработчик, и именно ты становишься первым тестирующим своим кодом. Ты запускаешь приложение, проверяешь, что оно работает, затем вносишь изменения и снова убеждаешься, что всё работает как прежде. А тест — это первый клиент твоего кода.

Больше юнит-тестов, меньше всего остального

В мире тестирования есть много терминов: интеграционные, e2e, smoke, UI, acceptance, snapshot, regression... Становится страшно даже начинать перечислять. Но непосредственно на качество кода влияет только один вид тестов — юнит-тесты.

Почему? Потому что они:

- Проверяют маленькие части кода, например функции, методы, классы.
- Работают быстро.
- Легко читаются и поддерживаются.
- Заставляют твой код быть тестируемым, а значит — аккуратным и логичным.

О чём я говорю? Как это связано с разработкой?

Допустим, у нас есть endpoint, который должен вернуть фазу Луны на определённую дату. Мы можем написать feature-тест, который проверит, что функция, вычисляющая фазу Луны, работает корректно. В котором мы обратимся по url-адресу [/api/moon?date=2025-06-01](#), получим ответ и проверим, что он соответствует ожидаемому значению.

```

public function test_returns_moon_phase_data(): void
{
    $response = $this->get('/api/moon', [
        'date' => '2025-06-01',
    ])
    ->assertOk()
    ->assertJsonStructure([
        'age',
        'phase',
        'distance',
        'nextNewMoon'
    ])
    ->json();

    [$age, $phase] = $response;

    $this->assertEquals(13.8, round($age, 1));
    $this->assertEqualsWithDelta(0.47, $phase, 0.01);
}

```

Это хороший тест, который проверяет, что API возвращает правильные данные для известной даты. Он проверяет, что ответ содержит нужные поля и что значения в них соответствуют ожидаемым.

Но как именно работает функция, вычисляющая фазу Луны? Как она получает данные о Луне? Как она обрабатывает дату? И вроде бы всё хорошо, но это обманка. Такой тест ничего не говорит о логике внутри. Он — витрина. Он проверяет фасад, но не фундамент. Он проверяет только конечный результат, это должно быть как вишенка на торте, а не основа.

Мы можем написать прямо в контроллере и добавить туда с десяток функций, которые будут вызывать другие функции, и в итоге получим правильный ответ. Но это не лучший подход.

Вместо этого лучше сосредоточиться на написании как можно большего числа тестов, которые проверяют поведение отдельных компонентов в изоляции.

А для этого вам потребуется использовать объекты:

```
// Хорошо [✓]
public function test_moon_phase_for_known_date(): void
{
    $date = new DateTimeImmutable('2025-06-01');
    $moon = new MoonPhase($date);

    // Проверяем округлённые значения
    $this->assertEquals(13.8, round($moon->age, 1));
    $this->assertEqualsWithDelta(0.47, $moon->phase, 0.01);
}
```

Такой подход заставляет вас писать код, который легко проверить и переиспользовать. Вы отделяете логику расчёта (в классе `MoonPhase`) от внешних интерфейсов (контроллеров, команд, CLI, API), и это делает код переносимым и модульным.

И самое важное: теперь никто не сможет просто так «вставить» бизнес-логику в контроллер — просто потому что она **уже вынесена в объект**, и её поведение **зафиксировано тестами**.

Тесты как средство симметрии и архитектурной дисциплины

Тесты — это не только проверка правильности работы кода, но и инструмент, который помогает поддерживать симметрию и согласованность между компонентами, особенно когда они тесно связаны.

Например, в сервисе погоды могут быть два класса — экспортер и импортер исторических данных:

```
$exporter = new WeatherHistoryExporter();
$exporter->export('/tmp/weather.zip');

$importer = new WeatherHistoryImporter();
$importer->import([
    'devices' => '/tmp/weather/devices.xml',
    'locations' => '/tmp/weather/locations.xml',
    'readings' => '/tmp/weather/readings.xml',
]);
```

В обычном коде вызовы этих классов часто разбросаны по разным частям приложения, и никто не замечает, что результат экспорта не подходит для импорта.

Если же написать тест, объединяющий эти сценарии в единый процесс, сразу становится очевидно, что экспорт и импорт — два конца одного процесса, и между ними должна быть полная совместимость.

Такой тест заставляет думать не только о том, что делает каждый отдельный компонент, но и о том, как классы на разных точках входа взаимодействуют, какой контракт между ними.

Это помогает сделать архитектуру цельной, логичной и поддерживаемой.

Тесты перестают быть просто проверкой — они становятся инструментом поддержания архитектурной дисциплины. Когда это понимаешь, начинаешь писать код иначе — так, чтобы все части системы были симметричны и идеально подходили друг к другу.

Arrange-Act-Assert (AAA)

Разделяйте тест на три логических фазы:

- Arrange. Подготовьте данные, объекты и окружение.
- Act. Выполните единственное действие — метод, который тестируете.
- Assert. Убедитесь, что результат совпадает с ожиданием (одно утверждение = одно тестовое поведение).

```
public function test_something(): void
{
    // Arrange: подготовка данных
    $obj = new MyClass(...);

    // Act: выполнение действия
    $result = $obj->doWork();

    // Assert: проверка результата
    $this->assertTrue($result->isSuccessful());
}
```


Если с выполнением действия и проверкой результата всё понятно, то с подготовкой данных могут быть нюансы. Подготовка данных — это главная часть теста, и она должна быть максимально простой и понятной.

Например, если вы тестируете метод, который работает с базой данных, то вам нужно создать необходимые записи в базе. Но не нужно создавать всю базу целиком, достаточно только тех записей, которые нужны для теста.

Есть несколько способов, как организовать подготовку данных. Например, определить заранее записи в базе данных, которые бы записывались перед исполнением теста.

```
users:
  - id: 1
    name: Иван Иванов
    email: ivan.ivanov@example.com
    password: '$2y$10$e0NRDUE8...'
    created_at: 2024-05-01 10:00:00
    updated_at: 2024-05-01 10:00:00

  - id: 2
    name: Мария Петрова
    email: maria.petrova@example.com
    password: '$2y$10$Fjs98JDk...'
    created_at: 2024-05-02 12:30:00
    updated_at: 2024-05-02 12:30:00
```

Это заставляет нас каждый раз возвращаться к этому файлу и обновлять его, когда мы добавляем новые поля в модель. При написании теста нам нужно сначала создать эти записи, а потом уже использовать их в тестах. Мы не знаем, а точно ли используется пользователь #2 в проекте или мы просто забыли удалить его из этого файла.

Тогда в тесте будет выглядеть примерно так:

```
// Плохо [X]
public function test_something(): void
{
    $user = User::find(2);
}
```

Кроме того, мы не видим никаких подробностей пользователя, которого мы используем в тесте. Мы не знаем, что это за пользователь, какие у него данные и зачем он нужен.

Лучше всего, чтобы подготовка была максимально близка к тестируемому коду. Например, если мы тестируем метод, который работает с пользователем, то лучше всего создать пользователя прямо в тесте:

```
// Хорошо [✓]
public function test_something(): void
{
    $user = User::factory()
        ->withPassword('password123')
        ->create();
}
```

Рекомендации по организации тестов

Тесты должны зеркалировать структуру вашего приложения. Это поможет вам быстро находить нужные тесты и понимать, что они проверяют.

```
tests/
├─ Unit/
│   └─ MoonPhaseTest.php
└─ Feature/
    └─ MoonPhaseTest.php
```

Независимость тестов

При работе с тестами иногда возникает неприятная ситуация: один тест проходит только в том случае, если он выполняется сразу после другого. Если поменять порядок запуска — тест ломается. Это явный признак того, что тесты зависят друг от друга.

Надёжные тесты должны быть независимы — их результат не должен зависеть ни от порядка выполнения, ни от состояния, оставленного другими тестами. Другими словами, каждый тест должен запускаться «с чистого листа», независимо от остальных.

Отличный способ обнаружить скрытые зависимости — запускать тесты в случайном порядке. Если при таком запуске какой-то тест начинает падать, значит, он опирается на предыдущие тесты, и с этим необходимо разобраться.

PHPUnit и Laravel поддерживают специальный флаг для случайного порядка **--order-by=random**

```
# Для Laravel
php artisan test --order-by=random

# Для Laravel Dusk
php artisan dusk --order-by=random

# Для PHPUnit
vendor/bin/phpunit --order-by=random
```

Попробуйте запустить свои тесты в случайном порядке и посмотрите, есть ли у вас зависимые тесты.

Ещё лучше добавьте атрибут `executionOrder` в конфигурационный файл, чтобы запуск тестов в случайном порядке был по умолчанию.

```
<?xml version="1.0" encoding="UTF-8"?>
<phpunit
    executionOrder="random"
>
```

Проверяй покрытие

Тесты могут успешно выполняться и радовать зелёным цветом. Но есть важная деталь — ничего не забыть. Допустим, в классе есть метод с условием:

```
if ($user->isPremium()) {
    // ...
}
```

На него уже написан тест, и он проходит. Но что, если `isPremium()` в тесте всегда возвращает `false`? Код внутри условия никогда не выполняется — и вы об этом даже не узнаете.

Чтобы понять, какие строки действительно исполнялись, нужна статистика покрытия. Обычно её можно получить с помощью тех же инструментов, например:

```
php vendor/bin/phpunit --coverage-html coverage/
```

Открыв [coverage/index.html](https://coverage.codacy.com/public/view/00000000000000000000000000000000/index.html), видно, какие строки проекта покрыты тестами.

- Зелёные — хорошо, код выполнен.
- Красные — плохо, код вообще не исполнялся.
- Жёлтые — частично, например, сработала только одна ветка `if`.

Покрытие не говорит, насколько хороши тесты, но сразу показывает, где их точно нет.

Убери `sleep()`

Использование `sleep()` в тестах — это признак отсутствия контроля над поведением системы. Вместо того чтобы управлять процессом и делать тесты предсказуемыми, ты просто надеешься, что всё «само как-нибудь успеет». Это не разработка, а угадайка.

Рассмотрим пример:

```
// Плохо [X]
public function test_email_is_sent(): void
{
    $this->dispatch(new SendEmailJob($user));

    sleep(3); // надеемся, что задача обработается

    $this->assertDatabaseHas('emails', [
        'user_id' => $user->id
    ]);
}
```

Что здесь происходит? Вместо того чтобы изолировать логику, использовать моки или запустить код синхронно, тест просто делает паузу и надеется на удачу. Это создаёт ложное ощущение стабильности, а на деле скрывает нестабильность и делает тесты хрупкими.

Такие тесты зависят от внешних факторов: загрузки системы, скорости обработки очередей, состояния базы данных. Результат становится непредсказуемым — сегодня тест успешен, завтра падает без очевидной причины. Кроме того, `sleep()` замедляет весь процесс автоматической проверки, увеличивая время запуска тестов.

Гораздо лучше — замочать очередь (или шину команд) и проверить, что нужная задача была отправлена:

```
// Лучше [✓]
public function test_email_is_dispatched(): void
{
    Bus::fake();

    $this->dispatch(new SendEmailJob($user));

    Bus::assertDispatched(
        SendEmailJob::class,
        fn($job) => $job->user->id === $user->id
    );
}
```

Такой мок не только для очередей работает. Аналогично можно мокать HTTP-запросы, внешние сервисы, события — везде, где важно проверить, что вызов произошёл.

Если обязательно нужно проверить побочный эффект, не жди фиксированное время — жди **по условию**. Например, для асинхронного обновления:

```
// Плохо [X]
public function test_user_status_updated(): void
{
    $this->externalApi()->newUser($user);

    sleep(5); // надеемся, что данные обработаются за это время

    $this->assertEquals(
        'processed',
        $this->externalApi()->status($user->id)
    );
}
```

Гораздо лучше реализовать проверку с повторным опросом, которая ждёт изменения состояния в течение заданного таймаута:

```
// Лучше [✓]
public function test_user_status_updated(): void
{
    $this->externalApi()->newUser($user);

    $this->waitUntil(function () {
        return $this->externalApi()
            ->status($user->id) === 'processed';
    }, 10);
}
```

Где `waitUntil` — метод, который опрашивает условие с интервалом, пока оно не станет истинным или не выйдет таймаут.



Играй по правилам

Программисты часто упрямы. Мы гордимся тем, что умеем абстрагироваться, строить свои слои и границы, моделировать сложные бизнес-процессы. Мы читаем книги, впитываем принципы SOLID, обсуждаем DDD на митапах и конференциях. Мы хотим, чтобы наш код жил дольше, чем фреймворк, на котором он написан.

Это выглядит как зрелость, но в действительности это просто страх зависимости. Мы боимся, что инструмент сделает нас менее универсальными. Что мы «завязнем» в платформе. Что не сможем мигрировать. Что нас будут называть «разработчик на XXX», а не просто «разработчик».

Но это — иллюзия.

Брюс Ли говорил: не бойся того, кто знает тысячу приёмов. Бойся того, кто отточил один приём тысячу раз.

Прими фреймворк

Когда ты добавляешь фреймворк в `composer.json`, ты подписываешь негласный договор. Ты не просто берёшь инструмент — ты принимаешь архитектурный стиль, соглашения, ритм разработки. Ты говоришь: «Эта платформа решает мои задачи. Я готов работать по её правилам».

Но что часто делают разработчики после этого? Начинаем сопротивляться. Гнём платформу под себя. Строим абстракции поверх уже готовых механизмов. Добавляем лишние слои. Изобретаем велосипед, чтобы чувствовать контроль.

Вот конкретный случай. В Laravel у нас есть Eloquent ORM. У модели `User` есть уже готовый интерфейс к данным: `User::query()`, `User::find()`, `User::where(...)`. Эти методы уже **инкапсулируют доступ к данным**.

Но некоторым разработчикам кажется, что этого недостаточно. Они строят поверх этого `UserRepositoryInterface`, `EloquentUserRepository`, `CachedUserRepository`, внедряют их в сервисы, пишут фабрики. Почему?

Потому что где-то они прочитали, что «работа с базой данных должна быть скрыта за интерфейсом». Но этот принцип вырван из контекста. Он применим в условиях, где инфраструктура сложна и разнообразна: файловые БД, распределённые хранилища, переключаемые бэкенды. Laravel же, как и в большинстве современных фреймворков, сам фреймворк уже является мощным адаптером.

В результате мы не получаем ни гибкости, ни производительности. Только архитектурный шум.

Принять фреймворк — значит использовать его силу, выразительность и экосистему. Это не поражение. Это зрелость. Играй по правилам — и ты удивишься, насколько всё может быть просто.

Не смешивай

В один момент тебе покажется, что ты стал умнее фреймворков. Что можешь выжать максимум из каждого. Взять миграции из CakePHP, FormRequest из Laravel, консольные команды из Symfony, что-нибудь из Yii. Ведь ты же архитектор. Ты знаешь, что делаешь. Правда?

Нет. Ты просто устроил себе проводной ад.

С виду это кажется гибкостью. Мол, ты не привязываешься ни к чему. Но на практике — ты просто собрал чемодан, полный переходников. Всё греется, шумит, не влезает в рюкзак, требует постоянной настройки.

Любой фреймворк — это не просто набор библиотек. Это договор на стиль разработки. Он решает проблемы целиком. У него есть ритм, философия, экосистема. Когда ты от него откусываешь кусками, это уже не Symfony, не Laravel и не Yii. Это что-то, что не будет полноценно поддерживаться ни сообществом, ни документацией, ни будущими разработчиками.

Мне не нравится

Важно понимать, что не все разработчики сразу открыты к использованию новых инструментов или технологий. Навязывание определённого фреймворка или подхода без учёта мнения и опыта команды часто вызывает сопротивление, снижение мотивации и даже саботаж.

Если вы столкнулись с сопротивлением со стороны вашей команды, необходимо провести открытый и честный разговор, чтобы по возможности уладить их опасения. Поиск совета у опытного профессионала может быть полезным для нахождения взаимоприемлемого решения.

Но если ты чувствуешь, что инструмент тебе не близок, — это нормально. Не пытайся «переделать» его под себя. Не стоит быть как наивная девушка, надеющаяся, что парень изменится — и всё станет как в сказке. Так не работает. Это не значит, что ты плохой разработчик. И не значит, что инструмент плохой. Просто вы не совпали.

Слишком часто мы боимся это признать — и начинаем «улучшать». Переписывать, извращать, подгонять. Вместо этого — просто не используй его вовсе. Выбирай те инструменты, которые соответствуют твоей философии и задачам. Не строй CQRS там, где у тебя обычный CRUD. Не впихивай микросервисы в монолит. Не применяй DDD, если у тебя нет сложной предметной области.

Ты либо принимаешь инструмент целиком и используешь его силу. Либо честно отказываешься — и идёшь другим путём. Половинчатое принятие — не компромисс, а архитектурное лицемерие. Борьба с инструментом — всегда путь в хаос.



Не отказывайтесь от будущего

Часто можно услышать от разработчиков фразу «**пока работает — не трогай**». Это звучит как здравый смысл. Но это в корне неверно. Очень плохая практика, которая является на самом деле самообманом. Как бы аккуратно ни был написан код — он не живёт в вакууме. Фреймворки развиваются, стандарты обновляются, экосистема не стоит на месте.

Игнорировать это — значит сознательно идти к деградации: медленно, мучительно и дорого. И с каждым днём этот выбор всё сильнее бьёт по эффективности команды.

Каждое «потом» становится «никогда». А каждое «не сейчас» — будущим блокером. И чем дальше откладываешь — тем больнее возвращаться.

Давайте разберёмся, почему откладывание обновлений — это отказ от будущего.

Временные решения всегда становятся постоянными

Разработчики вынуждены тратить время на создание временных решений и костылей для работы с устаревшими компонентами, вместо использования стандартных средств и функциональности, доступных в новых версиях.

Пример:

Разработчики, использующие Laravel 5.0, разрабатывали собственную проверку, чтобы клиент мог просматривать только свои заказы в интернет-магазине. Однако менее чем через полгода в версии 5.1 были представлены Policies, ставшие стандартом. Вместо обновления как можно скорее, увеличение кодовой базы лишь увеличивало время на последующее устранение технического долга.

Усложнение процесса обновления

Большие разрывы в обновлениях создают снежный ком, который требует значительных усилий и ресурсов для обновления проекта. Это может привести к тому, что если вы захотите обновиться, придётся потратить на это несколько месяцев без внедрения какого-либо нового функционала.

Пример:

Разработчики «Яндекс. Еда» пропустили три мажорных релиза, и полное обновление заняло целый год. За время накопления технического долга поддержка фреймворка, пакетов и самого РНР изменилась.

Актуальность — это и про найм тоже

Использование поддерживаемых версий также облегчает процесс найма новых разработчиков. Большинство опытных разработчиков предпочитают работать с современными и актуальными технологиями. Кроме того, знание того, что проект использует поддерживаемые версии, может быть важным моментом для кандидатов при принятии решения о присоединении к команде. Это также позволяет обеспечить более плавный процесс адаптации новых членов команды, так как они уже знакомы с особенностями работы с актуальными версиями.



Второй пилот — не капитан

В начале книги уже было отмечено: код читают гораздо чаще, чем пишут. Сегодня у разработчика появился второй пилот — только теперь он не сидит рядом, а встроен прямо в редактор. ChatGPT, Copilot и другие LLM-инструменты. Быстрые, дружелюбные, полезные. Но не стоит обманываться: это всё ещё второй пилот, а не капитан.

Обычно код не меняют просто так. Чаще всего причина — баг или новая функциональность, и задача для инструмента формулируется примерно так:

«Выступи в роли X. Иногда этот код выбрасывает исключение XXX. Исправь, пожалуйста».

Но именно в таких формулировках и кроется проблема.

Если передать ему метод на 800 строк, полных противоречий, логических дыр и нелепых зависимостей — не надейтесь, что он сделает из этого конфетку. Он будет **продолжать**. Механически, без разбора. Потому что его задача — **продолжить твой стиль**, а не **исправить твой хаос**. Старое правило **GIGO — *Garbage In, Garbage Out***. никуда не делось.

Рассмотрим простой пример: функция, которая записывает email в базу и отправляет приветственное письмо. Попросим второго пилота добавить логирование:

```
// >_ Добавь логирование адресов почты

function store()
{
    $email = $_POST['email'];

    $conn = new mysqli('localhost', 'user', 'pass', 'db');
    $conn->query("INSERT INTO users (email) VALUES ('$email')");

    mail($email, 'Hi', 'Welcome!');

+   file_put_contents(
+       'log.txt',
+       "User $email\n",
+       FILE_APPEND
+   );

    echo 'Done';
}
```

Он чертовски хорошо выполнил свою задачу, но проблема не в его работе, а в том, что переданный код выглядит очень плохо, потому что в нём:

- Нет возврата осмысленного результата.
- SQL-инъекция (строка вставляется напрямую).
- Нет обработки ошибок.

Если разработчик передаёт мусор — метод, который и швец, и жнец, и на дуде игрец — не стоит удивляться, что в ответ получит ещё и трубача.

LLM не откажет. Он скажет: «Да, командир!» — и продолжит работу.

На сайтах вроде Stack Overflow вопросы задают и решают живые люди. Часто можно встретить ответы, отмеченные как «решение», но при этом получившие минусы — в комментариях объясняют: да, это работает, но в долгосрочной перспективе приведёт к серьёзным проблемам.

«Второй пилот» — помощник другого типа.

Он не будет указывать на архитектурные проблемы или спорные решения, пока его об этом не попросите. Его задача — помогать вам как можно быстрее двигаться к результату, решать бизнес-проблемы здесь и сейчас. Но и большинство разработчиков настроены точно так же: задача должна быть закрыта, сроки — вчера, и мало кто будет тратить время, чтобы просить помощника подумать над более выразительными именами переменных, архитектурой или стилем.

Поэтому пилот продолжает ехать по плохой дороге. А это самое страшное — мы просто раздуваем хаос. И не просто так, а ещё и автоматизируем его рост.

Вот почему важно начать с хорошей базы: чистого кода, понятной архитектуры, простых методов и имён.

Контекст

У LLM-инструментов есть ограничение, про которое редко говорят — окно контекста. Оно невидимо, его нельзя контролировать напрямую. Но оно **всегда рядом**. И иногда критически важные части вашей системы просто выпадают из внимания модели.

Появляется асимметрия. Поверхность кажется правильной, но под капотом — несовместимость.

Мы уже несколько раз в книге говорили о симметрии, но давайте повторим. И посмотрим на небольшой пример импортера исторических данных о погоде:

```
$importer = new WeatherHistoryImporter();  
$importer->import([  
    'devices'    => '/tmp/weather/devices.xml',  
    'locations' => '/tmp/weather/locations.xml',  
    'readings'  => '/tmp/weather/readings.xml',  
]);
```

Его мог бы написать любой, и выглядит вполне разумно. Но этот класс — часть процесса импорта/экспорта. LLM может не знать о том, что класс экспорта выдаёт не набор XML-структур, а zip-архив:

```
$exporter = new WeatherHistoryExporter();  
$exporter->export('/tmp/weather.zip');
```

И мы снова получим асимметрию, хотя между экспортом и импортом должна быть полная совместимость. Результат экспорта по идее должен сразу и без искажений становиться аргументом для импорта — чтобы данные шли по замкнутому кругу без потерь и несоответствий.

И что? Они просто не стыкуются. LLM не «ошибся». Он просто не увидел целиком весь процесс.

Вот почему важно проверять границы. Убедиться, что один этап действительно продолжает другой. Что данные не просто «появились», а дошли по цепочке. Что код — не набор изолированных кусочков, а единое движение данных.

Именно контекст определяет смысл. И если ты его теряешь — модель его точно не найдёт.

Другой пример такой асимметрии — если мы попросим LLM, например, внести проверку на уникальность в базе данных:

```
class TokenService
{
    public function generate(): string
    {
        return bin2hex(random_bytes(16));
    }
}
```

В ответ получишь:

```
public function generate(): string
{
    do {
        $token = bin2hex(random_bytes(16));
    } while ($this->canTokenExists($token));

    return $token;
}

private function canTokenExists(string $token): bool
{
    return DB::table('tokens')
        ->where('value', $token)
        ->exists();
}
```

Он так же отлично справился с задачей. Но вот что он не знает: в другом месте системы у тебя уже есть **TokenRepository**, и именно он отвечает за сохранение, валидацию, поиск, всё, что связано с токенами. И теперь у тебя два источника правды: один в **TokenService**, второй в **TokenRepository**.

Почему это произошло? Всё по той же причине — потому что пилот не видит всей картины. Он сфокусирован на том куске, который ему показали. И всё остальное — выпадает.

Контекст — это не «удобно держать в голове». Это то, без чего невозможно принимать архитектурные решения. LLM не знает, как устроена твоя система. Он просто не видит границ. А разработчик — **видит**. Или, по крайней мере, должен видеть.

Не твой код

Есть и ещё одна неочевидная причина, по которой работа второго пилота может вызывать раздражение или разочарование, даже если он выполнил задачу точно.

Для очень многих разработчиков код — это не просто способ что-то реализовать. Это форма мышления, контроля, развития. Это способ выразить себя через структуру, стиль, архитектурные решения.

Когда множество программного кода рождается не в процессе размышлений, а просто появляется по запросу — остаётся ощущение отчуждённости. Да, задача решена. Но путь к решению пройден не тобой. Ты не выбирал между подходами, не ошибался, не искал компромисс. А значит — не чувствуется и результат.

Сгенерированный код может выглядеть и работать правильно, но разработчик становится не уверен в его деталях, не понимает всех нюансов и не чувствует связи с ним. Он становится чем-то внешним — как инструмент, который решил задачу, но не передал опыт.

В результате со временем может очень сильно снизиться мотивация и вовлечённость разработчика. Чтобы такого не допускать, нужно не просто «принимать» результат от LLM, а делать его «своим».

- Понимать, откуда он взялся.
- Переписывать под собственный стиль.
- Встраивать в архитектуру осознанно.

AI — это помощник. Он может ускорить работу. Но он не заменит твой выбор, твой стиль, твоё мышление.

Второй пилот не заменит твоё мышление — он просто помогает писать. А хороший код начинается с тебя!



Послесловие

Эта книга показала: аккуратный, читаемый код — это не недо-стижимый идеал, а повседневная реальность. Это та работа, кото-рую можно и нужно выполнять каждый день.

В реальных проектах побеждает не самый «умный» код, а са-мый понятный. Не самый модный подход, а самый предсказуе-мый. Не тот, что поражает архитектурой, а тот, что не оставляет вопросов.

Если после прочтения вы начали обращать внимание на име-на переменных и длину методов — значит, всё было не зря. На-смотренность и чувство стиля придут быстрее, чем вы думаете.