

Технологии программирования

Лекция 8 | Дизайн-код и рефакторинг

Что для вас чистый код?



Чистый код – это код, который...

Лаконичный
и структурированный

Легкий
и понятный

Легко читается
любым
разработчиком

Легко
модифицируется
любым
разработчиком

Делает то, что от него
ожидается

Чистый код – это код, о котором заботится его автор

Чистый код необходим для минимизации человеческих усилий, необходимых для создания и сопровождения системы

«Быстрый и грязный» код

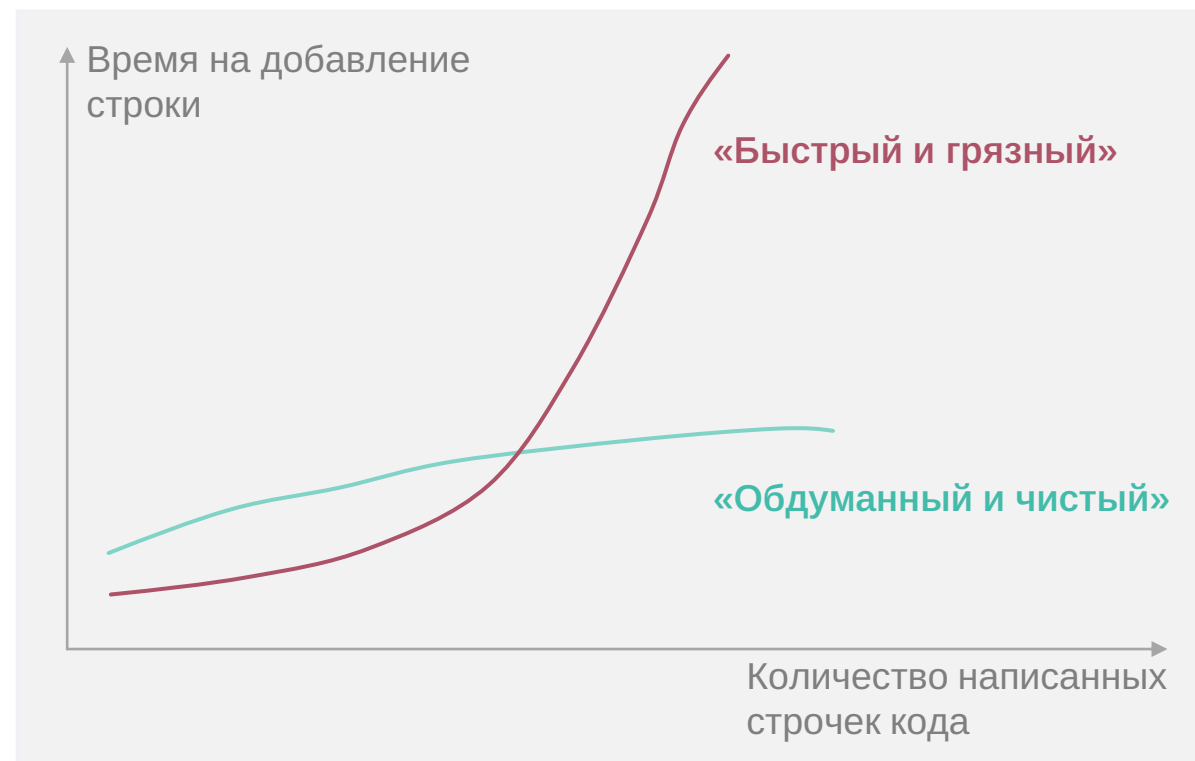
Когда вы пишете код быстро, без тщательного планирования и структурирования

Сначала может показаться, что это экономит время, но по мере увеличения количества строк код становится все сложнее понимать и исправлять

«Обдуманнный и чистый» код

Когда вы пишете код внимательно, делая его понятным и простым для изменений

В начале это может занять чуть больше времени, но со временем код остается удобным в работе





Чистый и качественный код

Одно и то же? Code style и утилиты

Показатели плохого дизайн-кода

Жесткость, неподвижность, вязкость, хрупкость

Принципы хорошего дизайн-кода

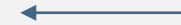
SOLID, связанность, паттерны, тесты

Code Smells?

Что это значит, основные антипаттерны при разработке

Рефакторинг

Определение понятия и подходы к улучшению кода



Мы здесь



Чистый vs Качественный код

Чистый код – это код, написанный читаемо и структурированно:
с понятными названиями переменных, минимальной сложностью, соблюдением стиля
(например, PEP8 для Python)

Его легко поддерживать и модифицировать

vs

Качественный код – это код, который эффективно решает задачу:
он надежен, производителен, безопасен, покрыт тестами и масштабируем

Почему они не всегда совпадают?

Чистота != эффективность

Красивый код может быть медленным (например, из-за избыточных абстракций)

Пример: Реализация алгоритма с читаемыми вложенными циклами вместо оптимизированной, но менее очевидной версии

Качество требует компромиссов

Иногда для скорости или безопасности приходится жертвовать читаемостью, т.е. чистотой

Пример: Низкоуровневые оптимизации в C++

Параметры чистого кода могут мешать архитектуре

Слепое следование Code Style может привести к избыточности, т.е. снижению качества

Пример: Дробление логики на микрофункции ради «чистоты»

Может различаться контекст задачи

Для прототипа важнее быстрота разработки (качество), а для legacy-системы – читаемость (чистота)

Code Style как основа чистоты кода

Стиль кода (code style) – это набор правил, как писать код в проекте

Они могут включать прямые рекомендации, примеры кода, ссылки на лучшие практики, «рецепты», что делать в спорных ситуациях

Примеры популярных кодстайлов:

- Google Code Style
- Mozilla Code Style
- GNU
- LLVM

При этом внутри каждой команды могут быть свои договоренности и параметры по стилю кода

Code Style как основа чистоты кода

```
const ThisIsExample=42
class
someclass{constructor(a,b,anotherArgument){this.a=a;
this.b= b;
this.c =anotherArgument}
} function
another_style_name(butArguments,ForSomeReason,
are_written_alternatively){
// отступов нет
let a = butArguments
  - are_written_alternatively;
}
```



```
const thisIsExample = 42

class SomeClass {
  constructor(mass, acceleration, force) {
    this.mass = mass
    this.acceleration = acceleration
    this.force = force
  }
}

function anotherStyleName(
  allTheArguments,
  forThisTime,
  areWrittenCorrectly
){
  let a = allTheArguments - areWrittenCorrectly
}
```



Google Code style | Цели руководства

! Правила должны стоить изменений

Преимущества от использования единого стиля должны перевешивать недовольство инженеров по запоминанию и использованию правил

Преимущество оценивается по сравнению с кодовой базой без применения правил, поэтому если ваши люди всё равно не будут применять правила, то выгода будет очень небольшой

Этот принцип объясняет почему некоторые правила отсутствуют:

Например, goto нарушает многие принципы, однако он практически не используется, поэтому Руководство это не описывает

Оптимизировано для чтения, не для написания



Официальная документация
от Google

Google Code style | Причины

Наша кодовая база (и большинство отдельных компонентов из неё) будет использоваться продолжительное время

Поэтому, на чтение этого кода будет тратиться существенно больше времени, чем на написание

Мы явно заботимся чтобы нашим инженерам было легко читать, поддерживать, отлаживать код

«Оставляй отладочный или логирующий код» – одно из следствий: когда кусок кода работает «странно» (например, при передаче владения указателем), наличие текстовых подсказок может быть очень полезным (`std::unique_ptr` явно показывает передачу владения)



Официальная документация
от Google

! Пиши код, похожий на существующий

Утилиты для чистого кода

Профилирование –

сбор характеристик работы программы, таких как:

- время выполнения
- число верно предсказанных условных переходов,
- число кэш-промахов

Линтеры / Статические анализаторы –

статический анализатор кода, который указывает на подозрительные участки программы и тем самым помогает программисту писать более качественный код

Форматеры / Рефакторинг –

преобразуют код согласно установленным правилам (Code Style)

Утилиты для чистого кода | Python

Профилирование:

- **cProfile** – встроенный профайлер для анализа времени выполнения функций
- **Py-Spy** – sampling-профайлер с низкими накладными расходами (работает без модификации кода)
- **memory_profiler** – профилирование потребления памяти
- **line_profiler** – построчный анализ производительности

Линтеры / Статические анализаторы:

- **Flake8** – проверка стиля (PEP8), логических ошибок и сложности кода
- **Pylint** – расширенный линтер с проверкой стиля, ошибок и возможностью кастомизации
- **MyPy** – статическая проверка типов (для Python 3.6+)
- **Bandit** – поиск security-проблем (например, небезопасных вызовов)

Форматеры / Рефакторинг:

- **Black** – автоматический форматтер с минимальными настройками
- **autopep8** – форматирование кода по PEP8
- **isort** – сортировка импортов
- **Rope** – библиотека для рефакторинга (переименование, извлечение методов)
- **PyCharm** – IDE с продвинутыми инструментами рефакторинга

Утилиты для чистого кода | Go

Профилирование:

- `pprof` – встроенный инструмент для анализа CPU, памяти и горутин
- `trace` – анализ работы горутин и системных вызовов

Линтеры / Статические анализаторы:

- `golint` – проверка стиля (совместимость с Go-community стандартами)
- `Staticcheck` – поиск ошибок, неиспользуемого кода и антипаттернов
- `revive` – расширяемый линтер с кастомными правилами
- `go vet` – встроенная проверка подозрительных конструкций

Форматеры / Рефакторинг:

- `gofmt` – официальный форматтер (стиль от Go core team)
- `goimports` – форматирование + управление импортами
- `gorename` – безопасное переименование переменных, функций и типов
- `GoLand` – IDE с поддержкой рефакторинга (извлечение методов, генерация кода)

Утилиты для чистого кода | C++

Профилирование:

- **Valgrind** – инструмент для обнаружения утечек памяти и анализа производительности
- **gperftools (Google Performance Tools)** – профилирование CPU и памяти
- **perf** – низкоуровневый анализ производительности (Linux)

Линтеры / Статические анализаторы:

- **Clang-Tidy** – анализ кода, проверка стиля и автоматические исправления
- **Cppcheck** – поиск утечек, неопределённого поведения и ошибок
- **PVS-Studio** – коммерческий анализатор для сложных уязвимостей

Форматеры / Рефакторинг:

- **Clang-Format** – форматирование кода с поддержкой стандартов (Google, LLVM)
- **Artistic Style (AStyle)** – альтернативный форматтер
- **CLion** – IDE с рефакторингом (переименование, оптимизация кода)

Утилиты для чистого кода | Rust

Профилирование:

- `perf` – низкоуровневый анализ (Linux)
- `flamegraph` – генерация флеймграфов для визуализации профилей
- `cargo-profiler` – сбор метрик производительности через Cargo

Линтеры / Статические анализаторы:

- `Clippy` – встроенный линтер для поиска антипаттернов и улучшения кода
- `rust-analyzer` – анализ кода, подсказки и диагностика ошибок (LSP-сервер)

Форматеры / Рефакторинг:

- `rustfmt` – официальный форматтер (стиль от Rust core team)
- `IntelliJ Rust` – плагин для JetBrains IDE с поддержкой рефакторинга
- `rust-analyzer` – автоматические преобразования кода (например, генерация трейтов)

Линтеры и статические анализаторы | Clang-Tidy

Анализ кода на соответствие современным стандартам (C++11/14/17/20/23), поиск антипаттернов, предложение улучшений

Особенности:

- Интеграция с Clang/LLVM
- Поддержка кастомных правил (можно писать свои проверки)
- Автоматическое исправление некоторых ошибок (-fix)

Установка:

Через пакетный менеджер
(например, `sudo apt-get install clang-tidy`)

Пример:

```
clang-tidy your_file.cpp --checks=modernize-*,readability-* -- -std=c++17 -I./include
```

Линтеры и статические анализаторы | Cppcheck

Статический анализ на утечки памяти, неопределённое поведение, неиспользуемый код

Особенности:

- Не зависит от компилятора
- Низкий уровень ложных срабатываний

Пример:

```
cppcheck --enable=all --suppress=missingIncludeSystem ./src
```

Форматирование кода | Clang-Format

Автоматическое форматирование кода в соответствии с настройками (Google, LLVM, Mozilla стили или кастомные)

Конфигурация:

Файл .clang-format в корне проекта

Пример:

```
clang-format -i --style=file ./src/*.cpp  
# Форматирует файлы на месте
```

Форматирование кода | Artistic Style (AStyle)

Альтернативный форматтер с поддержкой C, C++, Java

Конфигурация:

```
astyle --style=google --indent=spaces=4 your_file.cpp
```

ЧИСТЫЙ И КАЧЕСТВЕННЫЙ КОД

Code Style Checks | **cpplint**

Утилита для C++ для проверки кода на соответствие Google Style

Пример:

```
cpplint --filter=-build/include_subdir your_file.cpp
```

Code Style Checks | Clang-Format

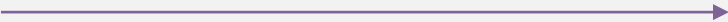
Утилита для C++ для проверки кода на соответствие Google Style

Настройка:

```
clang-format -style=google -dump-config > .clang-format
```

Custom Rules:

Настройка Clang-Tidy для проверки
именования переменных



CheckOptions:

- **key:** readability-identifier-naming.ClassCase
value: CamelCase
- **key:** readability-identifier-naming.VariableCase
value: lower_case

Чистота кода – инструмент хорошего дизайна

! Чистота кода делает дизайн видимым

Хороший дизайн проекта должен быть очевиден из структуры кода и работает как карта, которая помогает быстро понять:

- **Какие компоненты существуют** (например, модули auth, payment, analytics)
- **Как они взаимодействуют** (через интерфейсы, события или прямые вызовы)
- **Где лежат ответственности**
(например, класс OrderValidator обрабатывает проверки, а OrderProcessor – бизнес-логику)

Пример:

Если код модуля payment написан чисто, с четкими названиями методов вроде `process_refund()` или `validate_card()`, даже новый разработчик поймет, как интегрировать с ним другие компоненты

Контраст:

В запутанном коде с названиями вроде `handle()` или `doStuff()` дизайн становится «невидимым», и архитектуру приходится восстанавливать через дебаггинг.

Принципы чистого кода усиливают архитектурные паттерны

Принцип единой ответственности (SRP):

Чистый код требует, чтобы каждый класс/функция решали одну задачу

Это автоматически приводит к модульности – ключевому элементу хорошего дизайна

Пример:

Выделение **Logger**, **CacheManager** и **RequestHandler** в отдельные классы вместо смешивания их в одном God-объекте

DRY (Don't Repeat Yourself):

Устранение дублирования через абстракции (например, базовые классы или миксины) помогает создавать гибкую архитектуру, которую легко расширять

Пример:

Общий модуль **utils** с функциями для работы с датами/строками, используемый во всех компонентах

KISS (Keep It Simple):

Простой код снижает риск переусложнения дизайна ненужными абстракциями

Чистота кода как защита от архитектурного разложения

! Со временем даже продуманный дизайн может деградировать из-за хаотичных правок

Чистый код действует как «иммунитет»:

- Тесты и линтеры не дают нарушить границы компонентов
- Читаемость упрощает рефакторинг, чтобы дизайн оставался актуальным
- Стандарты именования предотвращают «расползание» ответственности (например, функция `send_email()` не будет заниматься валидацией данных)

Пример деградации:

Если разработчик добавляет логику валидации платежей прямо в класс `EmailNotifier` (из-за неочевидности дизайна), это нарушает модульность. Чистый код с четкими названиями и линтерами предотвратит такую ошибку



Чистый и качественный код

Одно и то же? Code style и утилиты

Показатели плохого дизайн-кода

Жесткость, неподвижность, вязкость, хрупкость

Принципы хорошего дизайн-кода


SOLID, связанность, паттерны, тесты

Code Smells?

Что это значит, основные антипаттерны при разработке

Рефакторинг

Определение понятия и подходы к улучшению кода



← Мы здесь



Основные показатели плохого дизайн-кода

Жесткость

трудно менять код,
так как простое изменение
затрагивает много мест

Неподвижность

сложно разделить код
на модули, которые можно
использовать в других
программах

Вязкость

изменение одного аспекта
кода связано с большими
затратами времени и усилий
(из-за модификации других
частей)

Ненужная сложность

в коде есть неиспользуемый
функционал

Ненужная повторяемость

Copy / Paste

Плохая читабельность

трудно понять что код
делает, трудно его
поддерживать

Хрупкость

легко поломать функционал
даже небольшими
изменениями

Жесткость

Одним из распространенных случаев жесткости является явное указание типов классов вместо использования абстракций (интерфейсов, базовых классов и т.п.)

```
// Пример 1: Жесткая зависимость через композицию
class B {
public:
    void DoSomething() {
        // Логика реализации
    }
};

class A {
    B _b; // Жёсткая привязка к конкретному классу
public:
    A() {} // Невозможно изменить реализацию B

    void Foo() {
        // Какая-то логика
        _b.DoSomething(); // Прямая зависимость
        // Какая-то логика
    }
};
```

```
// Пример 2: Неправильная реализация зависимостей
class IComponent {
public:
    virtual void DoSomething() = 0;
};

class A {
    IComponent* _component;
public:
    A(IComponent* component) : _component(component) {}

    ~A() {}

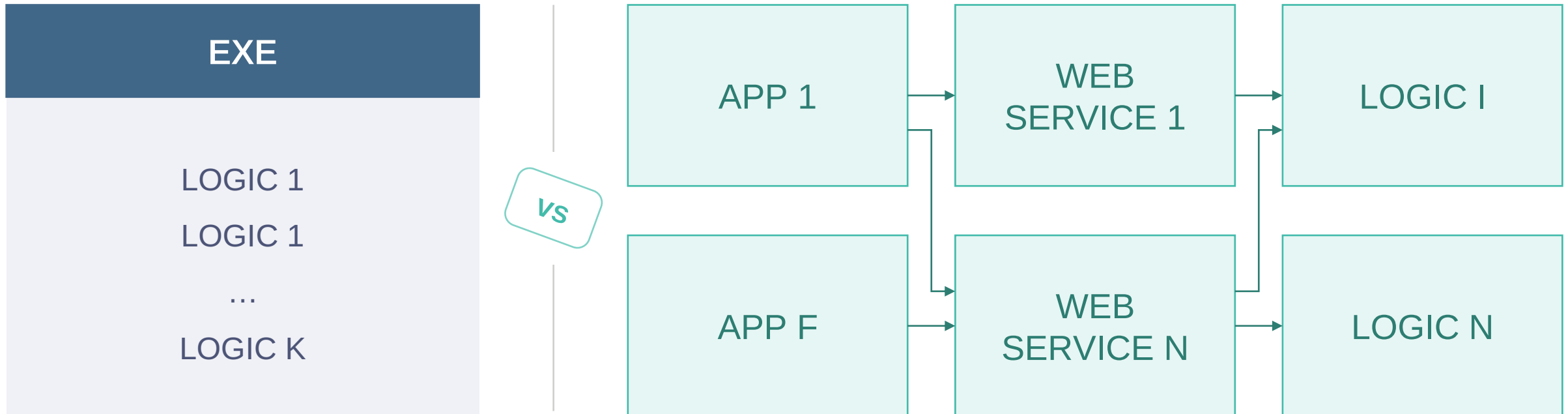
    void Foo() {
        // Какая-то логика
        _component->DoSomething()
        // Какая-то логика
    }
};
```

```
class B : public IComponent {
public:
    void DoSomething() override
    {
        // Реализация для B
    }

    // Потенциальная утечка:
    // объекты B никогда не
    // удаляются
};
```

Неподвижность

Неподвижность проявляется в сложности разделения кода на переиспользуемые модули
В результате проект может потерять способность эволюционировать и как результат перестанет быть конкурентноспособным



Вязкость

В качестве простого примера можно привести работу с константами, которые требуется помещать в отдельный модуль для использования другими компонентами

```
// constants.h - Глобальные константы  
как источник проблем  
#pragma once
```

```
// Жёсткие магические числа в  
глобальном пространстве имён  
namespace Constants {  
    constexpr float MAX_SALARY = 100.0f;  
    constexpr int MAX_PRODUCTS = 100;  
}
```

```
// marketing.h - Маркетинговый модуль  
#pragma once  
#include "constants.h" // Та же зависимость
```

```
class ProductManager {  
public:  
    void MakeOrder() {  
        for(int i = 0; i < Constants::MAX_PRODUCTS; ++i) {  
            // Логика заказа  
        }  
        if(someCondition) {  
            int tempLimit = 100; // Дублирование  
        }  
    }  
};
```

```
// finance.h - Финансовый модуль  
#pragma once  
#include "constants.h" // Прямая зависимость
```


```
class FinanceHelper {  
public:  
    static bool ApproveSalary(float salary) {  
        return salary <= Constants::MAX_SALARY; //  
        Жёсткая привязка  
    }  
  
    // Дублирование константы  
    static bool CheckBudget(float budget) {  
        return budget < 100.0f; // Магическое число  
    }  
};
```

Излишняя сложность

В этом случае дизайн имеет неиспользуемый в данный момент функционал (так сказать на будущее, а вдруг пригодится)

- Этот факт может затруднять поддержку и сопровождение программы, а также увеличивать время разработки и тестирования
- Удаляйте весь лишний код

Например, рассмотрим программу, которая требует чтения некоторых данных из базы. Для этого был написан специальный компонент DataManager, который используется в другом компоненте



Излишняя сложность

```
// Абстрактный интерфейс для "на будущее"
class IDataProvider {
public:
    virtual ~IDataProvider() = default;
    virtual std::string fetchData() const = 0;
    virtual void connectToCloud() = 0; // Не используется
    // сейчас
    virtual void cacheData(const std::string&) = 0; // Задел
    // для будущего кэширования
};

// Еще один абстрактный слой "на всякий случай"
class IDatabaseAdapter {
public:
    virtual void beginTransaction() = 0; // Не используется
    // в текущей логике
    virtual void commitTransaction() = 0; // Не используется
    // в текущей логике
};
```

```
// Конкретная реализация с избыточными функциями
class DataManager : public IDataProvider, public IDatabaseAdapter {
public:
    DataManager(const std::string& connectionStr, bool enableCache = false, int logLevel = 3)
        : m_connection(connectionStr), m_cacheEnabled(enableCache), m_logLevel(logLevel) {}

    std::string fetchData() const override {
        // Реальная логика чтения данных
        return "Sample Data";
    }

    // Реализации неиспользуемых методов
    void connectToCloud() override { /* Пустая реализация */ }
    void cacheData(const std::string&) override { /* Пустая реализация */ }
    void beginTransaction() override { /* Не требуется по ТЗ */ }
    void commitTransaction() override { /* Не требуется по ТЗ */ }

    // Дополнительные неиспользуемые функции
    void setCacheStrategy(int strategyId) { /* Задел для будущего */ }
    void validateDataConsistency() { /* Не вызывается нигде */ }

private:
    std::string m_connection;
    bool m_cacheEnabled; // Нигде не используется
    int m_logLevel;      // Нигде не используется
    std::vector<std::string> m_tempBuffer; // Резерв на будущее
};
```


Излишняя сложность

```
// Клиентский компонент, использующий только fetchData()
class DataProcessor {
public:
    DataProcessor(std::unique_ptr<IDataProvider> provider)
        : m_provider(std::move(provider)) {}

    void process() {
        auto data = m_provider->fetchData();
        std::cout << "Processing data: " << data << "\n";
    }

private:
    std::unique_ptr<IDataProvider> m_provider;
};
```

```
int main() {
    // Создание объекта с избыточными параметрами
    auto manager = std::make_unique<DataManager>(
        "DB=MyDB;User=Admin",
        false, // Кэш отключен
        5 // Уровень логирования не используется
    );

    // Инициализация неиспользуемых функций
    manager->setCacheStrategy(3); // Ни на что не влияет
    manager->validateDataConsistency(); // Пустая операция

    DataProcessor processor(std::move(manager));
    processor.process();

    return 0;
}
```

Излишняя сложность | Как можно упростить

```
class DataManager {
public:
    explicit DataManager(const std::string& connectionStr)
        : m_connection(connectionStr) {}

    std::string fetchData() const {
        return "Sample Data";
    }

private:
    std::string m_connection;
};
```

```
class DataProcessor {
public:
    explicit DataProcessor(const DataManager& manager)
        : m_manager(manager) {}

    void process() {
        auto data = m_manager.fetchData();
        std::cout << "Processing data: " << data << "\n";
    }

private:
    const DataManager& m_manager;
};

int main() {
    DataManager manager("DB=MyDB;User=Admin");
    DataProcessor processor(manager);
    processor.process();
    return 0;
}
```

Плохая читабельность

Причинами плохой читабельности может быть несоблюдение требований к оформлению кода (синтаксис, именование переменных, классов и т.д.), запутанная логика реализации и другое

Ниже приведен пример трудночитаемого кода, который реализует метод, оперирующий булевой переменной

```
void Process_true_false(string trueorfalsevalue)
{
    if (trueorfalsevalue.ToString().Length == 4) {
        // That means trueorfalsevalue is probably "true". Do something here.
    }
    else if (trueorfalsevalue.ToString().Length == 5) {
        // That means trueorfalsevalue is probably "false". Do something here.
    }
    else { throw new Exception("not true of false. that's not nice. return.") }
}
```

Хрупкость

Хрупкость программы означает простоту ее поломки в случае внесения изменений

Поломки бывают двух типов:

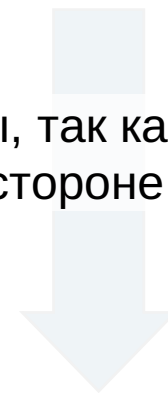
Ошибки компиляции



Оборотная сторона Жесткости

Ошибки времени выполнения

Наиболее опасны, так как часто находятся уже на стороне клиента



Являются показателем Хрупкости

Хрупкость

```
// Антипаттерн 1: Жёстко закодированная логика авторизации
class Helper {
public:
    static bool Authorize(int roleId, const string& resourceUri) {
        // Хрупкие условия с магическими числами
        if ((roleId == 1 || roleId == 10) && resourceUri == "/pictures") {
            return true;
        }

        // Смешанные условия с разной логикой
        if ((roleId == 1) || (roleId == 2 && resourceUri == "/admin")) {
            return true;
        }

        return false;

        // Проблемы:
        // 1. Сложно изменять логику условий
        // 2. Дублирование проверок roleId
        // 3. Жёстко закодированные URI
    }
};
```

Хрупкость

```
// Антипаттерн 2: Наивная реализация управления ресурсами
class Resource {
public:
    string Uri; // Публичное поле - нарушение инкапсуляции
    vector<int> roles; // Публичная реализация хранилища

    void AddRole(int roleId) {
        roles.push_back(roleId); // Нет проверки дубликатов
    }

    void RemoveRole(int roleId) {
        // Неэффективный поиск и удаление
        auto it = find(roles.begin(), roles.end(), roleId);
        if (it != roles.end()) {
            roles.erase(it); // Опасность итераторной инвалидации
        }
    }

    bool IsAvailable(int roleId) const {
        // Линейный поиск - неэффективно для больших данных
        return find(roles.begin(), roles.end(), roleId) != roles.end();
    }
};
```

Хрупкость

```
// Пример использования с проблемами
int main() {
    // Пример 1: Хрупкая авторизация
    if (Helper::Authorize(1, "/pictures")) { // Магические числа и строки
        cout << "Authorized" << endl;
    }

    // Пример 2: Небезопасное управление ресурсами
    Resource picturesResource;
    picturesResource.Uri = "/pictures"; // Прямое изменение состояния
    picturesResource.AddRole(1);        // Нет контроля за добавлением ролей

    if (picturesResource.IsAvailable(1)) {
        cout << "Authorized" << endl;
    }

    // Потенциальные проблемы:
    // 1. Изменение Uri извне: picturesResource.Uri = "/hack";
    // 2. Добавление невалидных ролей: picturesResource.AddRole(-1);
    // 3. Нет проверки уникальности ролей
    // 4. Неэффективная работа с большим количеством ролей
    return 0;
}
```

Хрупкость

```
// Хрупкая функция с множеством проблем
void Process(const char* value) {
    bool bValue;
    if (!ParseBool(const_cast<char*>(value), &bValue)) {
        throw invalid_argument(string("Invalid value: ") + value);
    }
    // Потенциальная утечка (без RAII)
    }

    // Жёстко закодированная логика без абстракций
    if (bValue) {
        cout << "TRUE PATH" << endl; // Дублирование кода
        // Логика true-ветки
    } else {
        cout << "FALSE PATH" << endl; // Дублирование кода
        // Логика false-ветки
    }

    // Потенциально опасные операции
    char* buffer = new char[100]; // Сырое управление
    памятью
    strcpy(buffer, value); // Возможное переполнение буфера
    delete[] buffer; // Возможная утечка при исключении
}
```

```
// Хрупкая проверка без учёта регистра и стандартных значений
bool ParseBool(char* value, bool* result) {
    if (strcmp(value, "1") == 0 || strcmp(value, "true") == 0) { // Не учитывает регистр
        и варианты
        *result = true;
        return true;
    }
    if (strcmp(value, "0") == 0 || strcmp(value, "false") == 0) { // Нет обработки nullptr
        *result = false;
        return true;
    }
    return false;
}

int main() {
    // Примеры опасного использования
    Process("true"); // OK
    Process("False"); // Упадёт из-за регистра
    Process(""); // Неопределённое поведение
    Process(nullptr); // Segmentation fault

    return 0;
}
```




Чистый и качественный код

Одно и то же? Code style и утилиты

Показатели плохого дизайн-кода

Жесткость, неподвижность, вязкость, хрупкость

Принципы хорошего дизайн-кода


SOLID, связанность, паттерны, тесты

Code Smells?

Что это значит, основные антипаттерны при разработке

Рефакторинг

Определение понятия и подходы к улучшению кода



← Мы здесь

Правила хорошего проектирование кода

 – пройденные принципы

1	Соблюдение принципов SOLID		5	Тестируемость	
2	Низкая связанность и высокая связность		6	Четкая модульная структура	
3	Использование паттернов проектирования		7	Документация и читаемость	
4	Управление зависимостями				

Многие принципы хорошего дизайн-кода берутся из ООП, но они относятся в любой парадигме

SOLID – принципы из ООП, но не про ООП

SOLID – акроним, введенный Майклом Фэзерсом для первых 5 основных принципов объектно-ориентированного проектирования и программирования

! Однако, принципы SOLID применимы не только к объектно-ориентированному программному коду – любая программная система имеет объединения, которые подобны «классам» ООП (например, «модули»)



- Принципы SOLID способствуют созданию системы, которую будет легко поддерживать и расширять в течение долгого времени
- Также это руководства, которые могут применяться во время работы над существующим кодом для его улучшения

SOLID – принципы из ООП, но не про ООП

S

Single responsibility principle (принцип единственной ответственности)

O

Open-closed principle (принцип открытости/закрытости)

L

Liskov substitution principle (принцип подстановки Лисков)

I

Interface segregation principle (принцип разделения интерфейса)

D

Dependency inversion principle (принцип инверсии зависимостей)

Низкая связанность (Low Coupling)

Компоненты системы должны минимально зависеть друг от друга

Основные показатели низкой связанности:

1. Интерфейсы вместо реализаций:

```
class IDataReader { // Абстракция
public:
    virtual std::string read() = 0;
};

class DatabaseReader : public IDataReader { ... };
// Конкретная реализация
```

2. Использование указателей / ссылок на абстрактные классы вместо прямого наследования


3. Dependency Injection (Внедрение зависимости):

```
class DataProcessor {
public:
    DataProcessor(IDataReader& reader) : m_reader(reader) {}
private:
    IDataReader& m_reader; // Зависимость "внедрена"
};
```

Низкая связанность упрощает модификацию, тестирование и повторное использование кода


Высокая связность (High Cohesion)

Элементы внутри модуля должны быть логически связаны и решать одну четкую задачу



```
// Класс, который делает всё: вычисления, логирование, валидацию
class GodObject {
public:
    void processData() {
        if (!validateData()) { // Валидация
            logError("Invalid data!");
            return;
        }
        calculate(); // Вычисления
        saveToFile(); // Сохранение
    }

private:
    bool validateData() { /* ... */ }
    void calculate() { /* ... */ }
    void saveToFile() { /* ... */ }
    void logError(const std::string& msg) { /* ... */ }
};
```



```
// Класс, отвечающий ТОЛЬКО за математические операции
class MathUtils {
public:
    static double calculateCircleArea(double radius) {
        return 3.1415 * radius * radius;
    }

    static int factorial(int n) {
        return (n <= 1) ? 1 : n * factorial(n - 1);
    }
};

// Класс, отвечающий ТОЛЬКО за вывод результатов
class ResultPrinter {
public:
    static void print(double value) {
        std::cout << "Result: " << value << std::endl;
    }
};
```

Высокая связность делает код более понятным, поддерживаемым и устойчивым к изменениям

Тестируемость

Классы должны быть изолированы, а юнит-тесты не требовать сложной настройки
Моки и стабы легко подменяют реальные зависимости

```
# payment.py
class PaymentGateway:
    def process_payment(self, amount: float, currency: str) -> bool:
        """Реальный сервис для обработки платежей (мы не хотим его вызывать
        в тестах)."""
        # Здесь может быть HTTP-запрос к внешнему API.
        raise NotImplementedError("Не вызывайте реальный сервис в тестах!")
```

```
class PaymentProcessor:
    def __init__(self, gateway: PaymentGateway):
        self.gateway = gateway

    def make_payment(self, amount: float, currency: str = "USD") -> str:
        if amount <= 0:
            raise ValueError("Сумма должна быть положительной.")

        success = self.gateway.process_payment(amount, currency)

        if success:
            return "Платеж успешно обработан."
        else:
            return "Ошибка платежа."
```

Высокий процент покрытия тестами – это >80%

Тестируемость

```
# test_payment.py
import unittest
from unittest.mock import Mock

class TestPaymentProcessor(unittest.TestCase):
    def test_successful_payment(self):
        # Создаем mock-объект для PaymentGateway
        mock_gateway = Mock(spec=PaymentGateway)
        # Настраиваем mock: метод process_payment возвращает True
        mock_gateway.process_payment.return_value = True

        # Внедряем mock в PaymentProcessor
        processor = PaymentProcessor(mock_gateway)
        result = processor.make_payment(100.0, "USD")

        # Проверяем, что результат корректен
        self.assertEqual(result, "Платеж успешно обработан.")

        # Проверяем, что метод process_payment был вызван с правильными аргументами
        mock_gateway.process_payment.assert_called_once_with(100.0, "USD")

if __name__ == "__main__":
    unittest.main()
```


Документация и читаемость

Читаемый код легко понять даже спустя месяцы после написания

Основные принципы читаемости:

1

Осмысленные имена



```
def calc(a, b):  
    return a * b # Что вычисляется? Непонятно
```



```
def calculate_area(width: float, height: float) ->  
float:  
    return width * height
```

Документация и читаемость

Читаемый код легко понять даже спустя месяцы после написания

Основные принципы читаемости:

2

Структурирование кода (разбивайте длинные функции, используйте отступы и пробелы)



```
// Все в одной функции  
void processData() {  
    // 50 строк кода...  
}
```



```
// Логика разделена  
void validateInput(const Data& data) { /* ... */ }  
void transformData(Data& data) { /* ... */ }  
void saveData(const Data& data) { /* ... */ }
```


Документация и читаемость

Читаемый код легко понять даже спустя месяцы после написания

Основные принципы читаемости:

3

Избегайте «магических чисел»


`if temperature > 100:
 shutdown()`


`MAX_SAFE_TEMPERATURE = 100
if temperature > MAX_SAFE_TEMPERATURE:
 shutdown()`

Документация и читаемость

Читаемый код легко понять даже спустя месяцы после написания

Основные принципы читаемости:

4

Комментарии для пояснения «почему»



```
x += 1; // Увеличиваем x на 1
```



```
// Учет смещения из-за бага в сторонней  
библиотеке (см. issue #123)
```

```
x += 1;
```



Чистый и качественный код

Одно и то же? Code style и утилиты

Показатели плохого дизайн-кода

Жесткость, неподвижность, вязкость, хрупкость

Принципы хорошего дизайн-кода

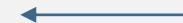

SOLID, связанность, паттерны, тесты

Code Smells?

Что это значит, основные антипаттерны при разработке

Рефакторинг

Определение понятия и подходы к улучшению кода



Мы здесь

Why the code smells?

Code Smells – очень грязный код, требующий исправления

Три основные категории причин:

Раздувальщики

Это код, методы и классы, которые раздулись до таких больших размеров, что с ними стало невозможно эффективно работать

Все эти запахи зачастую не появляются сразу, а нарастают в процессе эволюции программы

Нарушители объектного дизайна

Неполное или неправильное использование возможностей объектно-ориентированного программирования

Утяжелители изменений

Эти запахи приводят к тому, что при необходимости что-то поменять в одном месте программы, вам приходится вносить множество изменений в других местах

Это серьезно осложняет и удорожает развитие программы

Раздувальщики

Long Method

Метод содержит слишком большое число строк кода

Длина метода более десяти строк должна начинать вас беспокоить

Large Class

Класс содержит множество полей / методов / строк кода

Data Clumps

Иногда в разных частях кода встречаются одинаковые группы переменных (например, параметры подключения к базе данных)

Такие группы следует превращать в самостоятельные классы

Раздувальщики

Primitive Obsession

- Использование элементарных типов вместо маленьких объектов для небольших задач (например, валюта, диапазоны, специальные строки для телефонных номеров и т. п.)
- Использование констант для кодирования какой-то информации (например, константа `USER_ADMIN_ROLE = 1` для обозначения пользователей с ролью администратора)
- Использование строковых констант в качестве названий полей в массивах
- Long Parameter List
- Количество параметров метода больше трёх-четырёх

Нарушители объектно-ориентированного дизайна

Switch Statements

У вас есть сложный оператор switch или последовательность if-ов

Temporary Field (временные поля)

Это поля, которые нужны объекту только при определённых обстоятельствах, и только тогда они заполняются какими-то значениями, оставаясь пустыми в остальное время

Alternative Classes with Different Interfaces

Два класса выполняют одинаковые функции, но имеют разные названия методов

Refused Bequest

Если подкласс использует лишь малую часть унаследованных методов и свойств суперкласса
При этом ненужные методы могут просто не использоваться либо быть переопределёнными и выбрасывать исключения

Утяжелители изменений

Divergent Change

При внесении изменений в класс приходится изменять большое число различных методов

Например, для добавления нового вида товара вам нужно изменить методы поиска, отображения и заказа товаров

Shotgun Surgery

При выполнении любых модификаций приходится вносить множество мелких изменений в большое число классов

Parallel Inheritance Hierarchies

Всякий раз при создании подкласса какого-то класса приходится создавать ещё один подкласс для другого класса

Замусориватели

Comments

Метод содержит множество поясняющих комментариев

Duplicate Code

Два фрагмента кода выглядят почти одинаковыми

Dead Code

Переменная, параметр, поле, метод или класс больше не используются (например, устарели)

Lazy Class

На понимание и поддержку классов всегда требуются затраты времени и денег

А потому, если класс не делает достаточно много, чтобы уделять ему достаточно внимания, он должен быть уничтожен

Замусориватели

Speculative Generality

Класс, метод, поле или параметр не используются

Data Class (классы данных)

Это классы, которые содержат только поля и простейшие методы для доступа к ним (геттеры и сеттеры)

Это просто контейнеры для данных, используемые другими классами

Эти классы не содержат никакой дополнительной функциональности и не могут самостоятельно работать с данными, которыми владеют

Опутыватели связями

Feature Envy

Метод обращается к данным другого объекта чаще, чем к собственным данным

Inappropriate Intimacy

Один класс использует служебные поля и методы другого класса

Message Chains

Вы видите в коде цепочки вызовов вроде такой `$a->b()->c()->d()`

Middle Man

Если класс выполняет одно действие – делегирует работу другому классу – стоит задуматься, зачем он вообще существует



Чистый и качественный код

Одно и то же? Code style и утилиты

Показатели плохого дизайн-кода

Жесткость, неподвижность, вязкость, хрупкость

Принципы хорошего дизайн-кода


SOLID, связанность, паттерны, тесты

Code Smells?

Что это значит, основные антипаттерны при разработке

Рефакторинг

Определение понятия и подходы к улучшению кода



← Мы здесь

Что такое рефакторинг?

Рефакторинг – изменение во внутренней структуре ПО, имеющее целью облегчить понимание его работы и упростить модификацию, не затрагивая наблюдаемого поведения

Это естественный процесс разработки, призванный улучшать написанный код

Также используется для уменьшения технического долга* и способствует более плавному и эффективному процессу разработки

Писать «грязный» код как первый этап решения задачи – нормально
Если вовремя провести его рефакторинг



* Технический долг – это накопленные проблемы и недостатки в коде, которые могут замедлять разработку и усложнять поддержку продукта

Первопричины для рефакторинга

Тестовый код (MVP)

Недоработки архитектуры проекта

Недостаточная тестируемость

Отсутствие покрытия всех краевых случаев может привести к проблемам при запуске

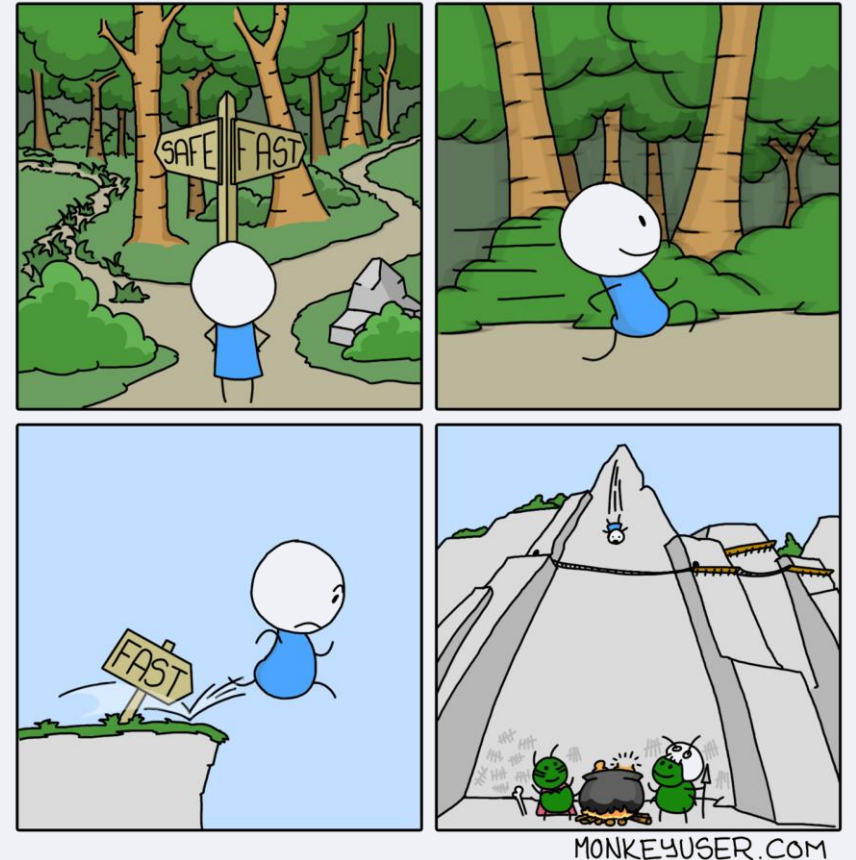
Изменение требований

Когда требования к разрабатываемой системе меняются, то это нередко приводит к изменениям в коде и архитектуре программы

Отсутствие договоренностей по кодстайлу и условиям качества

Отсутствие договоренностей в команде может привести к фундаментальным проблемам, в результате которых разработчики будут вынуждены провести объемный рефакторинг

КОМПРОМИСС



Когда рефакторить?

Правило трёх

Делая что-то в первый раз, вы просто это делаете

Делая что-то аналогичное во второй раз, может быть неприятно от необходимости повторения, но все-таки вы повторяете то же самое

Делая что-то похожее в третий раз, вы начинаете рефакторинг

Когда исправляете баги

Ошибки любят жить в темных и запутанных местах кода

Если навести порядок в коде, то ошибки могут найтись сами собой

Когда делаете новую фичу

Рефакторинг помогает понять чужой код и позволяет сделать его очевидней для вас и для того, кто будет работать с ним в будущем

Также после рефакторинга добавление новой фичи происходит значительно более гладко и занимает меньше времени

Во время код-ревью

Когда вы делаете ревью новой фичи, возможно, это последний шанс почистить код перед тем, как он окажется доступным публично

Лучше всего проводить такое ревью вместе с автором кода

Как рефакторить?

Рефакторинг следует проводить серией небольших изменений, каждое из которых делает существующий код чуть лучше, оставляя программу в рабочем состоянии



Как рефакторить?

Чистить можно всё, но в первую очередь найдите эти проблемы:

1

Мертвый код

Переменная, параметр, метод или класс больше не используются: требования к программе изменились, но код не почистили
Мёртвый код может встретиться и в сложной условной конструкции, где какая-то ветка никогда не исполняется из-за ошибки или изменения требований

2

Дублирование

Один и тот же код выполняет одно и то же действие в нескольких местах программы

3

Имена переменных, функций или классов не передают их назначение

Имена должны сообщать, почему элемент кода существует, что он делает и как используется

4

Слишком длинные функции и методы

Оптимальный размер этих элементов – 30-40 строк

Если получается больше, разделите функцию на несколько маленьких и добавьте одну общую

Как рефакторить?

Чистить можно всё, но в первую очередь найдите эти проблемы:

5

Слишком длинные классы

Оптимальная длина класса – 30-40 строк

6

Слишком длинный список параметров функции или метода

Вынесите параметры в отдельную структуру или класс с понятным именем, а в функцию передайте ссылку на него

7

Много комментариев

Как правило, плохой код покрывается обилием излишних комментариев

Необходимо найти правильный баланс в их написании

В целом к рефакторингу можно приступить на любом этапе разработке

Пример | Замена «Магических чисел» на константы

БЫЛО

```
double calculateCircleArea(double radius) {  
    return 3.1415926535 * radius * radius; // Что это  
    за число?  
}  
  
void process() {  
    if (temperature > 100) { // 100 — магическое число  
        shutdownSystem();  
    }  
}
```

СТАЛО

```
constexpr double PI = 3.1415926535;  
constexpr int CRITICAL_TEMPERATURE = 100;  
  
double calculateCircleArea(double radius) {  
    return PI * radius * radius;  
}  
  
void process() {  
    if (temperature > CRITICAL_TEMPERATURE) {  
        shutdownSystem();  
    }  
}
```

Пример | Избавление от «Длинного метода»

БЫЛО

```
void processData() {  
    // Валидация  
    if (data.empty()) {  
        cerr << "Error: Empty data!" << endl;  
        return;  
    }  
  
    // Обработка  
    for (auto& item : data) {  
        item = item * 2 + 5; // Непонятная логика  
    }  
  
    // Логирование  
    cout << "Processed " << data.size() << " items" << endl;  
}
```

СТАЛО

```
bool validateData(const  
vector<int>& data) {  
    return !data.empty();  
}  
  
void transformData(vector<int>&  
data) {  
    for (auto& item : data) {  
        item = item * 2 + 5;  
    }  
}  
  
void logResult(const  
vector<int>& data) {  
    cout << "Processed " <<  
data.size() << " items" << endl;  
}  
  
// Основной метод  
void processData() {  
    if (!validateData(data)) {  
        cerr << "Error: Empty data!"  
        << endl;  
        return;  
    }  
  
    transformData(data);  
    logResult(data);  
}
```

Пример | Замена «Примитивной obsession» на объекты

БЫЛО

```
// Хранение даты в виде трёх чисел
int day = 12;
int month = 3;
int year = 2024;

void printDate(int d, int m, int y) {
    cout << d << "/" << m << "/" << y << endl;
}
```

СТАЛО

```
class Date {
public:
    Date(int d, int m, int y) : day(d), month(m), year(y) {
        validate();
    }
    void print() const {
        cout << day << "/" << month << "/" << year << endl;
    }

private:
    void validate() {
        if (month < 1 || month > 12) throw invalid_argument("Month is invalid");
        // ... другие проверки
    }
    int day, month, year;
};

// Использование
Date date(12, 3, 2024);
date.print();
```

Спасибо за внимание!
Ваши вопросы?

