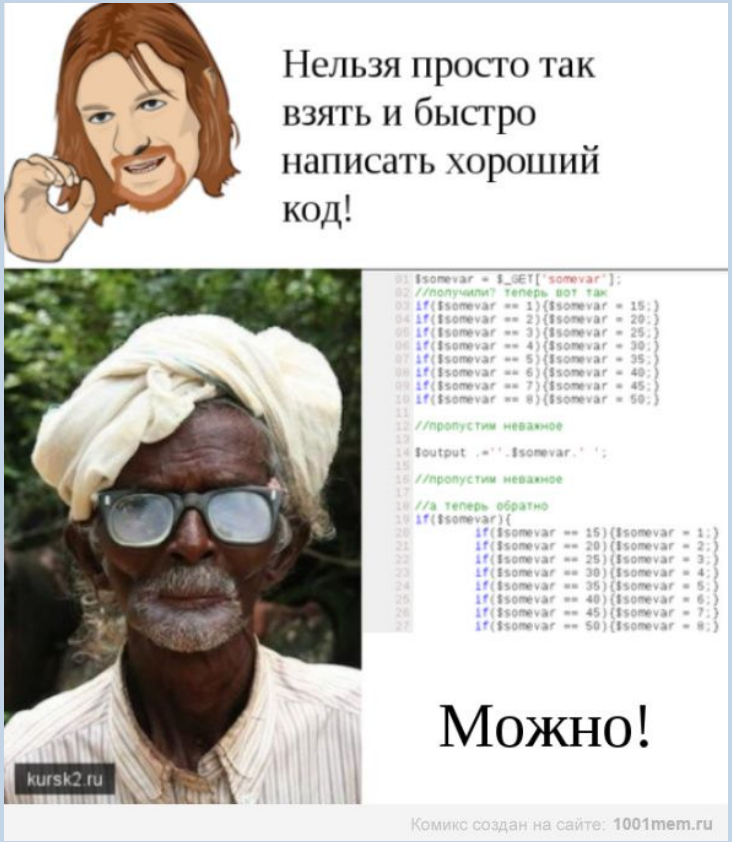
Практические навыки промышленной разработки

Практические навыки промышленной разработки

Практические навыки промышленной разработки




```php
$somevar = $_GET['somevar'];
//получили? теперь вот так
if($somevar == 1){$somevar = 15;}
if($somevar == 2){$somevar = 20;}
if($somevar == 3){$somevar = 25;}
if($somevar == 4){$somevar = 30;}
if($somevar == 5){$somevar = 35;}
if($somevar == 6){$somevar = 40;}
if($somevar == 7){$somevar = 45;}
if($somevar == 8){$somevar = 50;}

//пропустим неважное

$output .='<b>'.$somevar.'</b> ';

//пропустим неважное

//а теперь обратно
if($somevar){
        if($somevar == 15){$somevar = 1;}
        if($somevar == 20){$somevar = 2;}
        if($somevar == 25){$somevar = 3;}
        if($somevar == 30){$somevar = 4;}
        if($somevar == 35){$somevar = 5;}
        if($somevar == 40){$somevar = 6;}
        if($somevar == 45){$somevar = 7;}
        if($somevar == 50){$somevar = 8;}
```

Практические навыки промышленной разработки

“Разрыв между самыми лучшими и средними методиками разработки ПО очень широк — вероятно, шире, чем в любой другой инженерной дисциплине. Средство распространения информации о хороших методиках сыграло бы весьма важную роль.”

Фред Брукс

“Для построения метровой башни требуется твердая рука, ровная поверхность и 10 пивных банок, для башни же в 100 раз более высокой недостаточно иметь в 100 раз больше пивных банок. Такой проект требует совершенно иного планирования и конструирования.”

Стив Макконнелл

План

1. Проектирование. Методики. ООП.
2. Защитное программирование. Обработка ошибок.
3. Кодстайл и его важность. Принципы определения кодстайла.
4. Методики оценки качества программ
5. Тестирование силами разработчика. Отладка.
6. Рефакторинг. Стратегии и предпосылки.
7. Оптимизация кода. Оценка и подходы.
8. Общесистемная разработка. Интеграция отдельных частей.
9. Взаимодействие внутри команды. Личность разработчика.

Проектирование. Методики. ООП

Проектирование - “грязная” проблема: проблему нужно «решить» один раз, чтобы получить ее ясное определение, а затем еще раз для создания работоспособного решения.

Проектирование - недетерминированный процесс. Спроектировать компьютерную программу можно десятками разных способов.

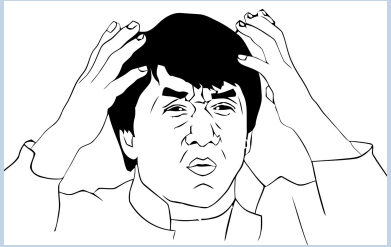
Проектирование. Методики. ООП

Главный технический принцип разработки ПО?

Проектирование. Методики. ООП

Управление сложностью!

Если работа над проектом достигает момента, после которого уже никто не может полностью понять, как изменение одного фрагмента программы повлияет на другие фрагменты, прогресс прекращается.



*“Компьютерные технологии - единственная отрасль, заставляющая человеческий разум охватывать диапазон, простирающийся от отдельных битов до нескольких сотен мегабайт информации, что соответствует отношению 1 к 10^9, или разнице в девять порядков”**

Эдсгер Дейкстра

* сейчас уже более 15 порядков

Проектирование. Методики. ООП

Уровни проектирования:

1. Программная система: что должна делать система в целом?
2. Разделение системы на подсистемы/пакеты
   a. Примеры: Business rules, DB, GUI, OS interaction
   b. Чем меньше взаимодействий между подсистемами - тем лучше!
3. Разделение пакетов на классы
   a. Определяем основные сущности и детали их взаимодействия
4. Разделение классов на данные и методы
5. Проектирование методов

Эвристические принципы проектирования

aka правила выполнения проб при использовании метода проб и ошибок
aka2 принципы, характеризующие то, каким должен быть завершенный проект приложения

1. **Итеративный подход, пока не найдем оптимальное решение:**
2. **Определение объектов реального мира:**
   a. Выделить объекты и их атрибуты
   b. Действия над объектом
   c. Действия объекта над другими объектами
   d. Видимые и скрытые части объекта, его интерфейс
3. **Определение согласованных абстракций:**
   a. Игнорируем нерелевантные детали
   b. Если бы, открывая или закрывая дверь, вы должны были иметь дело с отдельными волокнами древесины, молекулами лака и стали, вы вряд ли смогли бы войти в дом
   c. Если программисты не создают более общие абстракции, разработка системы может завершиться неудачей.

Эвристические принципы проектирования

4. **Инкапсуляция деталей реализации:**
    a. Блокируется доступ к деталям, к соответствующей им сложности.
    b. Инкапсуляция позволяет смотреть на дом, но не дает подойти достаточно близко, чтобы узнать, из чего сделана дверь

5. **Сокрытие информации:**
    a. Это наиболее простой и эффективный способ снижения сложности
    b. Интерфейс класса должен как можно меньше сообщать о своей внутренней работе (что делает, а не как делает)
    c. Вопрос-тест: “Что мне скрыть?”

Эвристические принципы проектирования

6. **Определение областей вероятных изменений:**

   Самое сложное в проектировании - обеспечить легкость внесения изменений.

   Очень полезно заранее определить и локализовать элементы, изменение которых наиболее вероятно. Такие элементы стоит изолировать в отдельных классах и пакетах.

   Наиболее часто меняющиеся элементы:
   a. Бизнес-правила
   b. Зависимость от оборудования
   c. Ввод-вывод
   d. Нестандартные возможности языка
   e. Сложные аспекты проектирования
   f. Переменные статуса
   g. Размеры структур данных

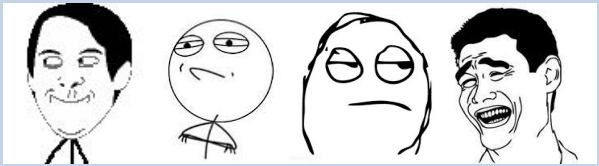
Эвристические принципы проектирования

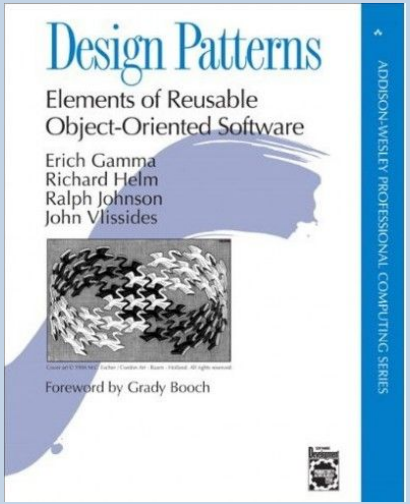
7. **Использование шаблонов проектирования**

   Примеры: Singleton, Facade, Observer, Factory, Factory method, Builder, Adapter etc.

   Профит:

   a. снижение сложности при помощи применения готовых абстракций
   b. стандартизация деталей популярных решений
   c. указывают на возможные варианты решения при анализе проблемы
   d. общий язык общения разработчиков





Эвристические методики проектирования

aka что же конкретно делать?

1. **Итеративный подход**

   Детали, полученные при низкоуровневом проектировании, помогут создать прочный фундамент для принятия высокоуровневых решений. И наоборот.

   Способность переключаться между высокоуровневыми и низкоуровневыми точками зрения - очень важное условие эффективного проектирования!

2. **Разделяй и властвуй**

   Проектируйте разные части программы по-отдельности.
   Если в какой-то компоненте вы зашли в тупик - оглянитесь назад и вспомните про итерацию.

Эвристические методики проектирования

3. **Нисходящий и восходящий подходы к проектированию**

   a. Аргументы в пользу нисходящего:
   Простота. Плавное и контролируемое увеличение количества деталей.
   Несколько попыток разного декомпозирования позволят найти оптимальный вариант.
   Когда заканчивать проектирование? Когда уже проще закодить, чем описывать.
   Если что-то неясно, то надо продолжать, ибо для других это будет головоломка.

   b. Аргументы в пользу восходящего:
   Если нисходящий подход слишком абстрактен и непонятно, с чего начинать.
   Вопросы: Какие функции должна выполнять эта подсистема? Какие в ней есть объекты и сферы ответственности?
   Низкоуровневые факторы (взаимодействие с оборудованием или внешними системами) могут так же влиять на основные параметры системы.

Восходящий подход, как правило, комбинируется с нисходящим.
Необходимо контролировать, чтобы восходящие наверх сложности не потопили всю систему.

Эвристические методики проектирования

4. **Экспериментальное прототипирование**
   Следствие “грязности” проблемы проектирования: нельзя полностью определить проблему, не решив ее хотя бы частично.

   Примеры: производительность хранилища, скорость обработки, насколько легко система X позволит реализовать требования Y.

   Основной инструмент: написание абсолютно минимального кода, отвечающего на поставленный вопрос и подлежащего выбрасыванию.

   Основные требования для хорошей работы методики: минимальность кода, конкретно сформулированная задача, настрой на выбрасывание кода. Иначе проектирование получается еще более “грязным”.

5. **Совместное проектирование** (одна голова - хорошо, а две - лучше)
   Варианты: помучать коллегу, парное программирование, мини-собрание, формальный обзор, возвращение через неделю, специализированные форумы. Комбинировать по вкусу.

Эвристические методики проектирования

6. **Документирование проектировочных решений**

> *Я никогда не встречал человека, желающего читать 17 000 страниц документации, а если бы встретил, то убил бы его, чтобы он не портил генофонд.*
>
> Джозеф Костелло (Joseph Costello)

Правило Парето: 80% усилий на разработку и анализ вариантов проектирования и 20% на документирование.

Варианты документирования:

- прямо в коде
- решения и протоколы дискуссий в wiki или почте
- цифровой фотоаппарат для фиксации схем и чертежей
- диаграммы UML с уместным уровнем детальности

Классы

Что такое Абстрактный тип данных?

Хорошая абстракция - это:

- Согласованность - реализация только одного АТД, а не убер-класс с мегафункционалом.
- Четкое понимание, какой абстракции соответствует класс. Нет лишних методов других или более низкоуровневых абстракций.
- Минимум семантических зависимостей между методами. Все требования должны быть очевидными и явными, максимально проверяться компилятором. На худой край - assert-ами.
- Хорошая абстракция согласуется с хорошей связностью (cohesion).
- Наличие противоположных методов: добавление и удаление, включение и выключение.

Классы

Хорошая инкапсуляция - это:

- Минимум доступных классов и членов: по умолчанию всё закрываем.
- Доступ к внутренним переменным - только через методы.
- В интерфейсе нет закрытых деталей реализации
- Минимум (в идеале - нет вообще) дружественных классов
- Минимум семантических зависимостей между методами. Все требования должны быть очевидными и явными, максимально проверяться компилятором.
- Минимальное сопряжение (coupling) между абстракциями

Классы

Примеры семантических зависимостей, ломающих инкапсуляцию:

- Не вызывать `A.init()`, потому что `A.doOperation()` его делает самостоятельно
- Не вызывать `DB.connect()` перед вызовом `A.get(DB)`, потому что знаем, что `DB` самостоятельно это сделает при отсутствии соединения
- Не вызывать `A.close()`, потому что знаем, что `A.completeOperation()` уже его вызвал
- Использовать константу `B.MAX_VALUE` вместо константы `A.MAX_VALUE`, потому что знаем, что они равны

Все эти примеры показывают зависимость от реализации, а не от абстракции.

Хорошая абстракция и хорошая инкапсуляция тесно связаны. Не бывает одного без другого.

Классы. Наследование

Общие правила игры:

- Отношение включения - отношение “содержит”
- Отношение наследования - отношения “является”
- Проектируем с учетом возможности наследования или запрещаем его
- Данные делаем private, а не protected.
- Принцип подстановки Лисков: производный класс должен являться более специализированной версией базового класса. То есть, все методы должны иметь то же самое значение (пример про счет и знак процентов)
- Наследуем только то, что хотим наследовать. Если в довесок получаем еще и пачку других методов, то правильней делать “включение”
- Не дублируем имена приватных методов.

Классы. Наследование

Общие правила игры (продолжение):

- Не дублируем имена приватных методов.
- Общие методы выносим как можно выше (пока это согласуется с абстракцией)
- Базовый класс с только одним производным классом? Запашок преждевременого проектирования.
- Переопределенный метод имеет пустую реализацию (в отличие от исходного)? Запашок нарушения контракта базового класса и начала чрезмерного усложнения (свойство присуще только части классов из всей иерархии).
- Глубина наследования от 3 до 6 уровней? Или даже больше 6? При таком проектировании наследование становится трудноуправляемым.
- Много однотипных блоков case или if-else? Возможно, реализация должна быть более объектно-ориентированной.

Классы. Множественное наследование

“В некотором смысле наследование похоже на цепную пилу: при соблюдении мер предосторожности оно может быть невероятно полезным, но при неумелом обращении последствия могут оказаться очень и очень серьезными.

Если наследование — цепная пила, то множественное наследование — это старинная цепная пила повышенной мощности с барахлящим мотором, не имеющая предохранителей и не поддерживающая автоматического отключения. Иногда такой инструмент может пригодиться, но большую часть времени его лучше хранить в гараже под замком.”

Стив Макконнелл, Совершенный код

Главным образом, множественное наследование полезно при создании “миксинов” - примесей отдельных небольших поведений: Comparable, Serializable, Displayable etc., ибо они по-настоящему независимы друг от друга.

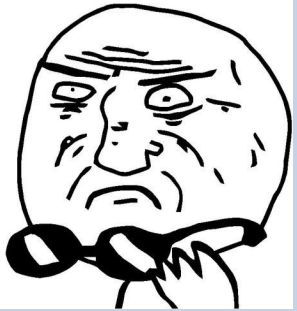
Классы

Разумные причины создания классов:

- Моделирование объектов реального мира - причина №1 во всех хит-парадах
- Моделирование абстрактных объектов: Shape = {Circle, Square}. Нахождение таких объектов - одна из главных проблем ООП.
- Снижение сложности: спрятать сложность, выведя наверх максимально простые “ручки”
- Изоляция сложности: ограничение влияния таких сложных вещей, как запутанные алгоритмы, структуры данных, протоколы и пр.
- Сокрытие деталей реализации: конкретный способ подключения к БД, представление данных
- Ограничение влияния изменений: бизнес-правила, генерация отчетов, ввод-вывод
- Облегчение повторного использования кода
- Группировка родственных операций: статистические функции, работа со строками и пр.

Высококачественные методы

```
def HandleStuff(inputRec: CorpData, crntQtr: Int, empRec: EmpData, estimRevenue: AtomicInteger,
ytdRevenue: Double, screenX: Int, screenY: Int, newColor: AtomicReference[ColorType],
prevColor: AtomicReference[ColorType], status: AtomicReference[StatusType], expenseType: Int )
{
var i: Int = 0
for ( i <- 0 until 100 ) {
  inputRec.revenue(i) = 0
  inputRec.expense(i) = corpExpense(crntQtr)(i)
  }
UpdateCorpDatabase( empRec )
estimRevenue.set((ytdRevenue * 4.0 / crntQtr).toInt); newColor.set(prevColor.get)
status.set(SUCCESS)
if ( expenseType == 1 ) {
  for ( i <- 0 until 12 )
    profit(i) = revenue(i) - expense.type1(i)
  }
else if ( expenseType == 2 ) {
  profit(i) = revenue(i) - expense.type2(i)
} else if ( expenseType == 3 )
  profit(i) = revenue(i) - expense.type3(i)
}
```



Высококачественные методы

Метод - самый простой и самый эффективный способ уменьшения сложности программ.

Разумные причины создания методов:

- Снижение сложности: скрываем информацию, минимизируем код, уменьшаем вложенность блоков (циклов, условий)
- Формирование понятной промежуточной абстракции. Например, название `resolveDestination()` уже, фактически, создает такую абстракцию и автоматически документирует ее
- Предотвращение дублирования кода
- Поддержка наследования: чем проще метод в базовом классе, тем проще его переопределять
- Упрощение сложных булевых проверок: автодокументирование и сокрытие деталей
- Повышение быстродействия: оптимизировали метод - ускорили все места, где он вызывается

Высококачественные методы

Удачные имена методов:

- Имя описывает все, что метод выполняет
- Имя содержит глагол-действие + (опционально) объект, к которому это действие относится
- Имя не содержит невыразительные/неоднозначные глаголы: `handleMessage`, `executeService`, `processInput`.
- Имена не отличаются только цифрами: `doPart1`, `doPart2`
- Имена часто используемых операций соответствуют конвенциям: `find` / `get` / `load` / `search`

Высококачественные методы

Оптимальный размер метода (основано на исследованиях):

- Оптимально: 1-2 экрана
- Сложный алгоритм: 100-200 непустых строк. Критерии разбиения:
  - Связность метода
  - Глубина вложенности
  - Число переменных
  - Число ветвлений (точек принятия решения)
  - Необходимое число комментариев
- Более 200 строк - неизбежно уменьшение понятности

Домашнее задание

Отрефакторить lectures.oop.FatUglyController

Защитное программирование. Обработка ошибок

Программы содержали, содержат и будут содержать ошибки.

Их, конечно, можно и нужно исправлять. Но и надо быть готовым с этим жить.

Идея защитного программирования похожа на идею внимательного вождения: это безопасная поездка с учетом непредсказуемости остальных участников дорожного движения.

Защитное программирование. Обработка ошибок

Стандартный подход: мусор на входе - мусор на выходе.

Ответственный подход (промышленное ПО): мусор на входе - четкая ошибка на выходе.

Ключевые правила:

1. Проверять все данные из внешних источников
2. Проверять все входные параметры метода
3. Решить, как обрабатывать неправильные входные данные

Защитное программирование. Обработка ошибок

Утверждения (assert) - нужны для документирования допущений, на которые опирается код:

1. значение параметра находится в ожидаемом интервале
2. файл/поток открыт/закрыт в начале/конце метода
3. указатель в файле в начале/конце
4. значение не null (Java etc)

Утверждения, как правило, не попадают в продовскую сборку. Отсюда их проверка не должна содержать выполняемый не отладочный код.

Не надо путать ассерты с ожидаемой обработкой ошибок.

Актуально для особенно высоконадежных программ.

Защитное программирование. Обработка ошибок

Возможные реакции на ошибки:

- Игнорировать/вернуть нейтральное/предыдущее/ближайшее корректное значение
- Залогировать ошибку/ворнинг
- Кинуть ошибку выше по стеку
- Вызвать процедуру-обработчик
- Прекратить выполнение программы

Две крайности:

1. Игнорировать (устойчивость)
2. Прекратить выполнение программы (корректность)

Общий принцип при реализации обработки ошибок:
выбрать единый подход и соблюдать его везде в проекте.

Защитное программирование. Изоляция

Чтобы не проверять всегда и везде все входные параметры, программу можно разделить на 2 зоны: недоверенную (буферную) и доверенную.

В недоверенной (буферной) зоне все входные данные считаются небезопасными и всегда очищаются/проверяются перед обработкой. Здесь же рекомендуется преобразовать их к нужному типу (строка - к дате и пр.).

В доверенной зоне все входные данные считаются “чистыми”. Поэтому к ним очистка не применяется, зато могут (и должны) применяться assert-ы.

Защитное программирование. Исключения

Основная задача исключений - оповещение других частей программы об ошибках, которые нельзя игнорировать.

Отсюда несколько правил по работе с исключениями:

- Используйте только для действительно исключительных ситуаций
- Генерируйте исключения в соответствии с уровнем абстракции
- В описании исключения должны быть четко указаны все причины
- Избегайте пустых блоков catch или комментируйте, почему они пустые
- Стандартизуйте использование исключений в проекте (в т.ч. создайте общий для всех предок)
- При работе со сторонней библиотекой разберитесь, какие исключения она может кидать

Защитное программирование. Тест и Прод

При разработке и тестировании:

- Периодическая самодиагностика, проверка всех ошибок
- Fail fast всегда
- Отображение ошибок пользователю

На бою:

- Проверка только серьезных ошибок (вычисления и пр.)
- Fail fast только в случае критической ошибки (порча данных), аккуратное завершение работы (сохранение пользовательских данных)
- Логирование ошибок
- Отображение только критических ошибок с инструкцией, что делать дальше и к кому обратиться за помощью

Кодстайл и его важность. Принципы определения

Форматирование не влияет на скорость выполнения, объем памяти и прочие эксплуатационные характеристики приложения на бою. Однако, от него зависит, насколько легко код можно будет прочитать и исправить, а также насколько сложно в нем будет разобраться новому программисту.

Хорошее визуальное форматирование показывает логическую структуру программы.

При выборе между красотой кода и логической структурой кода предпочтение должно отдаваться логической структуре. Отсюда также будет хорошо виден нелогичный код.

Кодстайл и его важность. Принципы определения

Опыт показывает, что **детали конкретного кодстайла гораздо менее важны, чем факт наличия и повсеместного использования этого кодстайла**.

Принципы хорошей схемы форматирования:

- Точное представление логической структуры кода
- Единообразие представления логической структуры кода
- Улучшение читаемости
- Минимальные изменения других строк при внесении правок

Кодстайл и его важность. Принципы определения

```
def someMethod[T](input: T): Unit = { // do not write expression right after "="

  // Multiline expressions should end with operator call to denote multilineness
  val name = "Some long value that is so long that it should not fit into one line " +
    "and it should continue on the second line"

  // Group code with empty lines
  for (i <- 1 to 100) {
    try {
      doSomeImportantStuff(i)
    } catch {
      case _ => // Do nothing -- place comments like that to denote that the body should be empty
    }
  }

  // Chaining function calls place on separate lines
  generateArray
    .filter(_ % 2 == 0)
    .take(100)
    .foreach(doSomeImportantStuff)

  // Do not align values with spaces/tabs
  val TYPE_ONE = 1
  val TYPE_THREE = 3
  val TYPE_FORTY_TWO = 42
}
```

Кодстайл и его важность. Принципы определения

```
/*
 * Default order of methods:
 * - constructors
 * - public methods
 * - protected methods
 * - private methods
 * - public attributes
 * - protected attributes
 * - private attributes
 */
class Greeter(message: String, quantity: Int) {

  def this(message: String) = this(message, 1)

  def sayHi() = doSayHi()

  protected def overridableSayHi() = doSayHi()

  private def doSayHi() = println(message * quantity)

  val publicValue = 42

  protected val protectedValue = 43

  private val privateValue = 44
}
```

Самодокументирующийся код

Виды документации:

- Внешняя документация:
  - Общая постановка задачи
  - Детальное описание проектирования и его результатов
- Внутренняя документация:
  - Включается в код, наиболее подробная и актуальная

Основа внутренней документации - не комментарии к коду,
а **хороший стиль программирования**:

- Грамотная структура программы
- Простые и понятные подходы, шаблоны
- Подходящие имена классов, переменных и методов
- Именованные константы
- Ясное форматирование
- Минимизация сложности потока управления и структур данных

Самодокументирующийся код

Хорошо написанный код - это уже
большая часть детальной документации!

```
for ( i = 2; i <= num; i++ ) {
meetsCriteria[ i ] = true;
}
for ( i = 2; i <= num / 2; i++ ) { j = i
+ i;

while ( j <= num ) {
meetsCriteria[ j ] = false;
j = j + i;
}
}
for ( i = 1; i <= num; i++ ) {
if ( meetsCriteria[ i ] ) {
System.out.println ( i + " meets
criteria." ); }
}
```

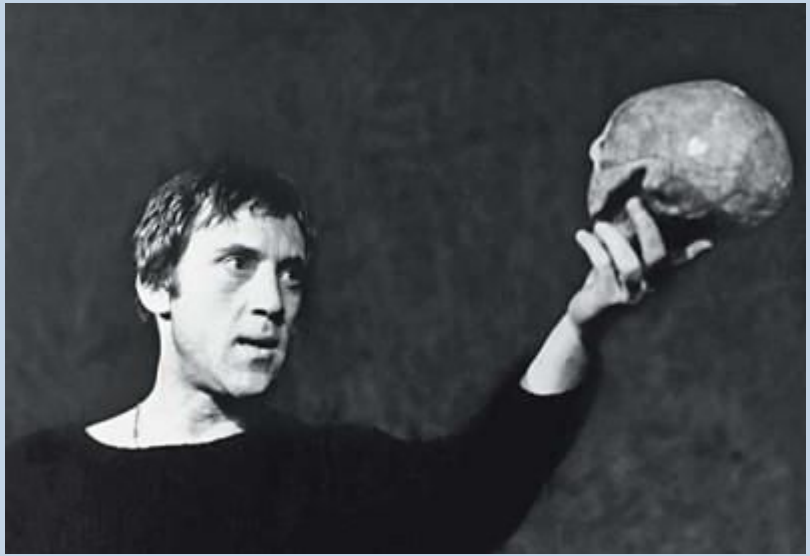
```
for ( primeCandidate = 2; primeCandidate <= num; primeCandidate++ ) {
  isPrime[ primeCandidate ] = true;
}

for ( int factor = 2; factor < ( num / 2 ); factor++ ) {
  int factorableNumber = factor + factor;
  while ( factorableNumber <= num ) {
    isPrime[ factorableNumber ] = false;
    factorableNumber = factorableNumber + factor;
  }
}

for ( primeCandidate = 1; primeCandidate <= num; primeCandidate++ ) {
  if ( isPrime[ primeCandidate ] ) {
    System.out.println( primeCandidate + " is prime." );
  }
}
```

Самодокументирующийся код

Комментировать или не комментировать?



Самодокументирующийся код

Хорошие комментарии не повторяют код и не объясняют его. Они поясняют его цель. Комментарии должны объяснять намерения программиста на более высоком уровне абстракции, чем код.

Шестимесячное исследование, проведенное в IBM, показало, что программисты, отвечавшие за сопровождение программы, «чаще всего говорили, что труднее всего было понять цель автора кода» (Fjelstad and Hamlen, 1979).

Самодокументирующийся код

Виды комментариев:

1. Повторение кода. Самые бесполезные.
2. Объяснение кода. Лучше писать код понятно. Но сюрпризы надо пояснять!
3. Маркер в коде. Должны быть стандартизованы.
4. Резюме кода. Как повторение кода, но на более высоком уровне. Обычно полезны.
5. Описание цели кода. Самые полезные.
6. Информация, не выражаемая в форме кода: копирайт, внешние ссылки. Тоже полезно.

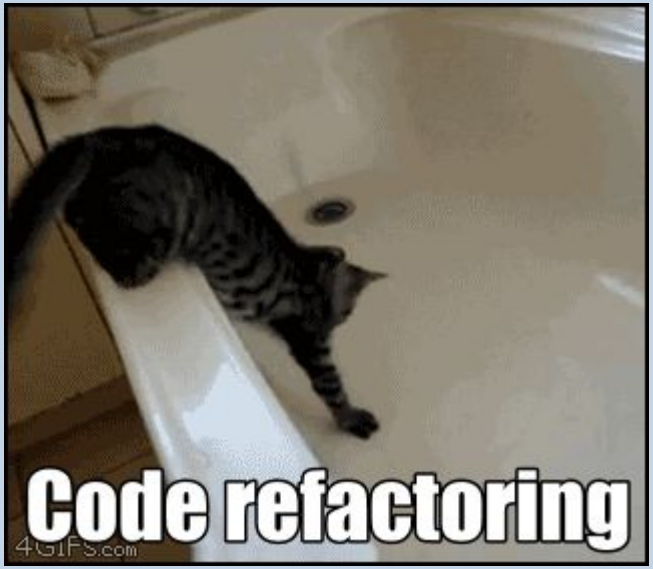
Хитрый код = плохой код. Его лучше переписать, чем комментировать.

Самодокументирующийся код

Полезные советы:

1. Документируйте в интерфейсе его контракт (неявные предположения)
2. Комментируйте ограничения методов
3. Если меняется глобальное состояние - так же надо зто описать
4. Ссылайтесь на источники используемых алгоритмов

Рефакторинг



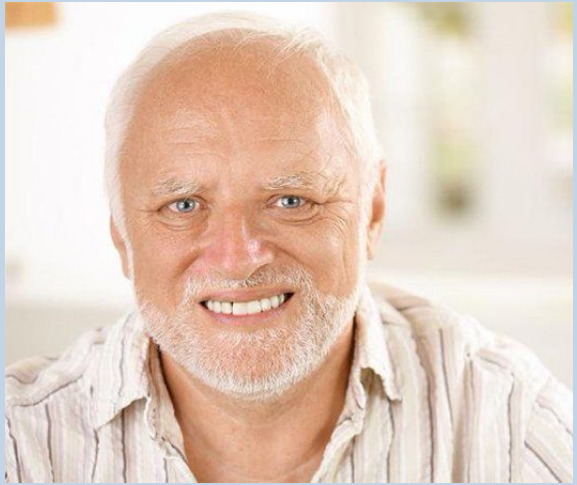
Рефакторинг

Программы меняются.

Всегда.

Даже те, которые меняться не должны.

Надо научиться с этим жить.

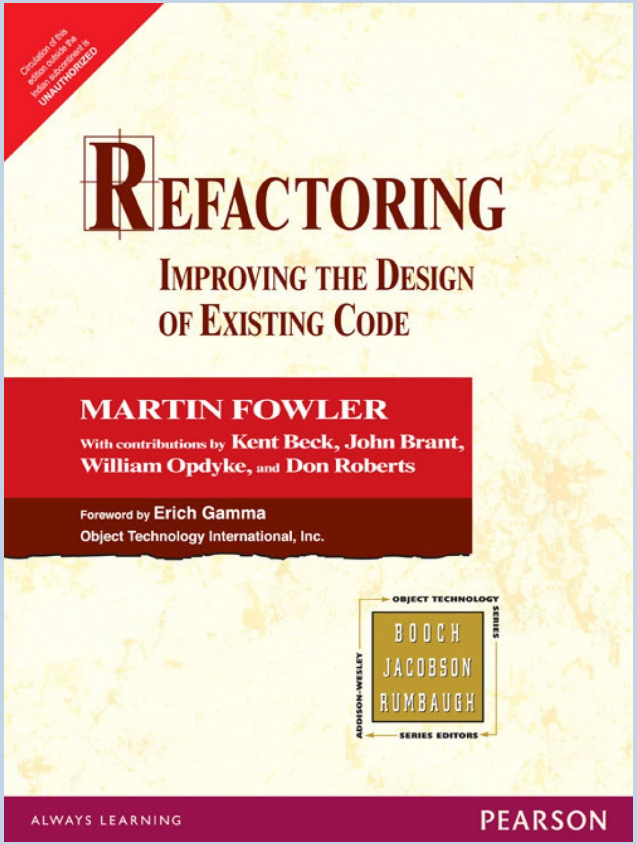


Рефакторинг

Но гораздо печальней тот факт,
что если этим процессом не управлять,
то программы обязательно скатываются в УГ.

Рефакторинг. Стратегии и предпосылки

Рефакторинг - это изменение внутренней структуры программы без изменения наблюдаемого поведения, призванное облегчить ее понимание и удешевить модификацию.



Рефакторинг. Виды запашков

1. Код повторяется - принцип DRY
2. Метод слишком велик - в ООП редко методы вырастают за пределы одного экрана. Возможно, стоит выделить отдельный класс.
3. Цикл слишком велик или глубоко вложен в другие циклы - преобразовать в метод.
4. Класс имеет плохую связность - разбить на несколько классов
5. Интерфейс класса не формирует согласованную абстракцию
6. Метод принимает слишком много параметров - вероятно, абстракция, формируемая интерфейсом метода, выбрана неудачно
7. Отдельные части класса изменяются независимо от других его частей
8. При изменении программы требуется одновременно изменять несколько классов

Рефакторинг. Виды запашков

9. Приходится одновременно изменять несколько иерархий наследования
10. Приходится одновременно изменять несколько блоков if/case
11. Метод использует больше данных из чужого класса, чем из своего собственного - возможно, его стоит перенести в этот чужой класс
12. Элементарный тип данных перегружен (на него наложено много ограничений), возможно стоит вынести его в отделный класс (тип Money)
13. Класс имеет слишком ограниченную функциональность - возможно, имеет смысл его удалить, распределив функционал по другим классам
14. Вызов метода только передает данные другому методу - возможно, абстракция плохо согласована с этим методом
15. Один класс знает слишком много о другом - нарушение инкапсуляции
16. Метод имеет неудачное имя

Рефакторинг. Виды запашков

17. Данные-члены являются открытыми
18. Подкласс использует только малую часть методов своих предков - возможно, наследование надо поменять на включение
19. Сложный код объясняется при помощи комментариев - перепишите код
20. Код содержит глобальные переменные
21. Код “на будущее”, “впрок”:
    - Требование к такому коду хуже специфицировано
    - Не учтены специфичные этим требованиям тонкости
    - Другие программисты полагаются на то, что этот код достаточно отлажен и протестирован
    - Дополнительная сложность при разработке и рефакторинге с нулевой пользой
    - Гораздо эффективней писать текущий код просто и понятно так, чтобы потом можно было легко его дополнять и исправлять

Рефакторинг. Топ видов рефакторинга

1. Замена магического числа на именованную константу
2. Присвоение переменной более ясного имени
3. Замена выражения на вызов метода
4. Введение/удаление промежуточной переменной
5. Преобразование многоцелевой переменной в одноцелевые
6. Выделение набора констант в Enum
7. Замена if/case на вызов полиморфного метода
8. Преобразование объемного метода в класс с несколькими методами
9. Отделение операции запроса данных от операции их изменения
10. Передача в метод целого объекта вместо набора полей
11. Передача в метод набора полей вместо целого объекта

Рефакторинг. Когда и как его делать

1. Правило 80/20: тратьте время на 20% видов рефакторинга, обеспечивающих 80% выгоды.
    - 80% ошибок происходят в 20% наиболее сложных модулей.
2. При внесении изменений в код улучшайте те места, к которым прикасаетесь
3. При работе над легаси-системой полезно выстроить промежуточный интерфейс между старой и новой системами и постепенно переводить функционал с одной стороны на другую

Оптимизация кода. Оценка и подходы

Во имя эффективности — причем достигается она далеко не всегда — совершается больше компьютерных грехов, чем по любой другой причине, включая банальную глупость.

У. А. Вульф (W. A. Wulf)

Опять принцип Парето: на 20% методов программы приходится 80% времени ее выполнения (на практике 4% кода выполняются 50% времени).

Следствие для интерпретируемых языков: напишите все на более медленном языке и перепишите наиболее проблемные места на быстром языке типа C/C++.

Оптимизация кода. Типичные заблуждения

1. Чем меньше строк высокоуровневого языка, там быстрее
   a. Контрпример: разворачивание циклов
2. Одни операции, вероятно, выполняются быстрее других
   a. Нужны измерения производительности для данной версии языка, компилятора, библиотек, среды исполнения (ЦПУ, память и пр.)
3. Оптимизацию следует выполнять по мере написания кода
   a. Программисты очень плохо угадывают наиболее узкие места (те самые 4%)
   b. Уменьшается акцент на понятности и легкости дорабатывания основного кода)
4. Быстродействие программы не менее важно, чем ее корректность
   a. Некорректные, но быстрые программы никому не нужны. Сначала всегда корректность!

Оптимизация кода. Алгоритм

1. Напишите хороший и понятный легко изменяемый код
2. Если его надо ускорить:
   a. Сохраните рабочую версию, чтобы потом можно было к ней вернуться
   b. Выполните профилирование с целью найти узкие места
   c. Выясните причины низкого быстродействия: архитектурная ошибка, алгоритм, типы данных и др. Если оптимизация кода неуместна - возвращаемся к п.1
   d. Оптимизируйте найденное узкое место
   e. Оцените каждое улучшение (по одному за раз!)
   f. Если оптимизация не привела к улучшению - откатитесь
3. Повторить пункт 2

Оптимизация кода. Методики

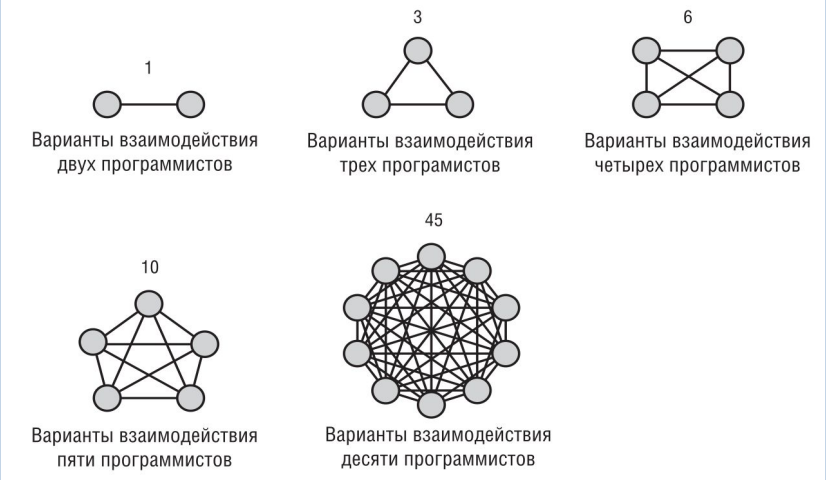
Конкретные методики оптимизации, как правило, сильно зависят от используемого языка и архитектуры.

Общее понимание подходов можно посмотреть в 26-й главе книги “Совершенный код”.

Для нашего с вами случая наиболее эффективными являются подходы:

1. Минимум внешних вызовов
2. Асинхронные взаимодействия и правильные таймауты
3. Кеширование
4. Оптимизация запросов в БД, создание нужных индексов

Общесистемная разработка



С ростом размера проекта появляется необходимость поддерживать взаимодействие. Большинство методологий разработки сводится к уменьшению проблем взаимодействия (отсюда основной принцип выбора “лучшей” методологии - облегчение взаимодействия).

Общесистемная разработка

Размер проекта (число строк кода)	Число строк кода на человека в год
1 К	2.5 К - 25 К
10 К	2.0 К - 25 К
100 К	1.0 К - 20 К
1 000 К	0.7 К - 10 К
10 000 К	0.3 К - 5 К

Общесистемная разработка

При прочих равных, производительность в больших проектах будет ниже, чем в маленьких.

При прочих равных большие проекты будут содержать больше ошибок на 1000 строк кода, чем маленькие.

Деятельность, которая в малых проектах сама собой разумеется, в больших проектах должна быть тщательно спланирована.

С ростом размера проекта непосредственно программирование занимает все меньшую ее часть.

Общесистемная разработка

В программировании многие исследования показывают разницу на порядки в качестве написанных программ, их размерах и производительности программистов. Вот цифры для лучших и худших программистов с 7-летнем стажем:

- Время первоначальной разработки: 20 к 1
- Время отладки: 25 к 1
- Размер программы: 5 к 1
- Скорость выполнения: 10 к 1
- Нет четкой взаимосвязи между опытом и качеством кода.
- Хорошие программисты стремятся к объединению. Плохие - тоже :)

=> Польза первых 10% разработчиков намного превышает пользу

Личность разработчика

Важнейшие качества разработчика:

1. Скромность - необходимо понимание, что наши возможности ограничены и эти ограничения надо компенсировать (снижение нагрузки на мозг)
2. Любопытство к постоянно меняющимся техническим вопросам
    a. Изучение процесса разработки
    b. Экспериментирование
    c. Изучение чужих способов решения проблем (проще прочитать, чем самим решить)
    d. Анализ и планирование перед действием
    e. Изучайте другие проекты, ищите удачные и неудачные решения
    f. Показывайте свои проекты другим, собирайте мнения других
    g. Постоянное профессиональное развитие

Личность разработчика

Важнейшие качества разработчика (продолжение):

3. Профессиональная честность
   a. Охотное признание собственных ошибок. Бесполезно их скрывать.
   b. Четкое понимание того, что происходит в программе
   c. Предоставление реалистичных отчетов о сроках и статусе проекта
4. Творчество и дисциплина: чем больше проект, тем более упорядоченным он должен быть. Конвенции - для второстепенного.
5. “Просвещенная” лень: решать неприятные проблемы, тратя на них как можно меньше времени. Голова здесь - лучший помощник.

Личность разработчика

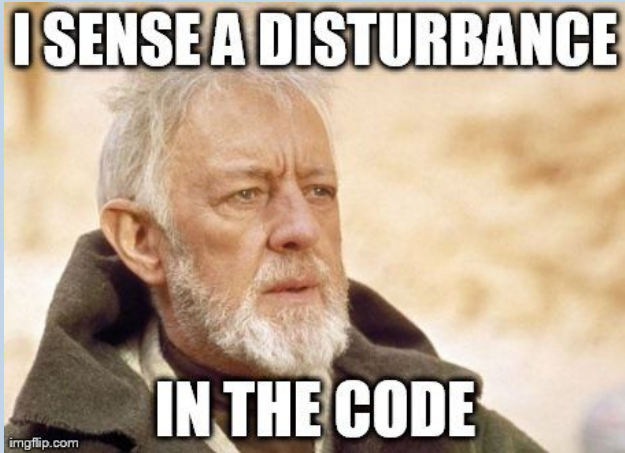
Сомнительные качества разработчика:

1. Настойчивость: если нет прогресса какое-то время, то отложите проблему и дайте подсознанию над ней поработать. Осознанно прерывайте себя.
2. Опыт - каждые 5-10 лет технологии значительно меняются
3. Ночные бдения и овертаймы - уменьшают эффективность и угнетают интеллект

Если вы посвятите небольшую долю своего времени чтению книг и изучению программирования (например, одну книгу за 1-2 месяца), через полгода вы будете намного превосходить большинство своих коллег.

Бонус: основы мастерства

- Главная цель программирования - борьба со сложностью
- Процесс программирования оказывает большое влияние на результат
- Групповое программирование - это больше общение с людьми, а не компьютером; код пишется для людей
- Программирование в терминах бизнес-области помогает управлять сложностью
- Итерация в любом процессе помогает улучшить результат: проектирование, оценка, разработка, ревью, оптимизация.
- Религия и догматы - неуместны. Лучше всего комбинировать различные методики и подходы. Единого лучшего решения - нет.



Бонус 2: Литература

- Junior:
  - Programming in Scala, 3rd Edition
  - The Pragmatic Programmer
  - Design Patterns
- Middle:
  - Code Complete 2
  - Refactoring
  - Clean Code
  - The Clean Coder
- Senior:
  - Java Concurrency In Practice
  - Peopleware
  - Functional Programming in Scala
  - Patterns of Enterprise Application Architecture

12 Most Influential Books Every Software Engineer Needs to Read