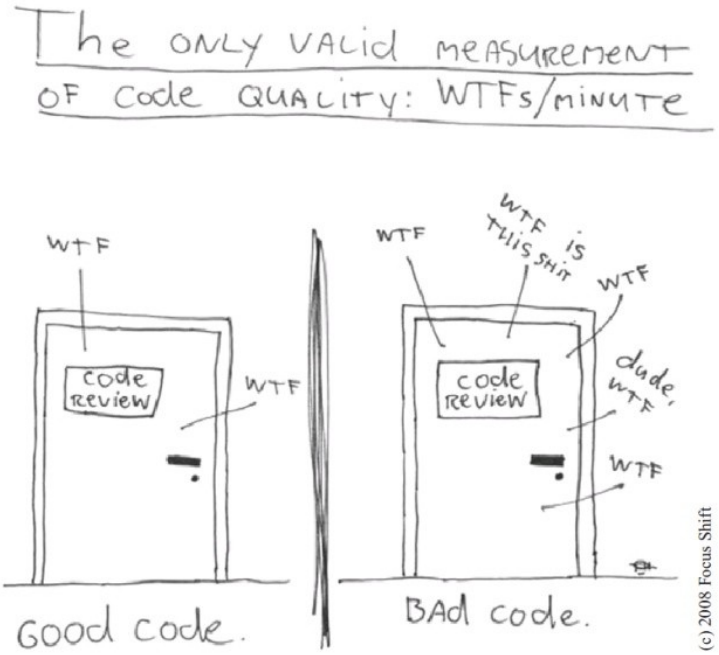
Clean Code – A Handbook of Agile Software Craftmanship
Robert C.Martin
Robert C. Martin Series - Prentice Hall - 2008x



Introduction

"Which door represents your code? Which door represents your team or your company? Why are we in that room? Is this just normal code review or have found a stream of horrible problems shortly after going live? Are we debugging in a panic, poring over code that we thought worked? Are customers leaving in droves and managers breathing down ours necks? How can we make sure we wind up behind the right door when the going gets tough? The aswer is: ***craftmanship***."

"Learning to write clean code is hard work. It requires more than just the knowledge of principles and patterns. You must sweat over it. You must practice it yourself, and watch yourself fail. You must watch others practice it and fail. You must see them stumble and retrace theirs steps. You must see them agonize over decisions and see the price they pay for making those decisions the wrong way."

Chapter 1: Clean Code

Por que não escrever código ruim?

- Empresas acabam se apressando para liberar algum software no mercado sem dar muita atenção para a qualidade do seu código. Ao adicionar novas funcionalidades e consertar bugs, o código muitas vezes fica cada vez mais complexo e com "remendos" se tornando cada vez mais difícil de ser mantido.
- Uncle Bob cita que muitas vezes pegamos um código que simplesmente não vemos sentido. Gastamos muito tempo tentando entender seu funcionamento, mas acabamos sempre encontrando mais código sem aparente sentido. Isso faz com que seja dificil adicionar qualquer nova funcionalidade.
- Enquanto no começo pareciamos estar avançando com grande velocidade, no longo prazo, podemos estar andando muito vagarosamente se escrevemos código ruim. Pequenas mudanças em um canto podem quebrar muitas outras partes do sistema, já que nenhuma mudança é trivial. Quando os problemas começam a surgir, os "managers" do projeto podem resolver contratar mais gente para aumentar novamente a produtividade. Nesse momento, os recém-contratados não sabem como é o design do software e não sabem o que é melhorar o design e o que é piorá-lo. Além disso, existe uma pressão para o aumento da produtividade e a qualidade do código cai ainda mais.

Por que escrevemos códigos ruins?

- Porque temos pressa. Temos que entregar logo muitas funcionalidades, módulos ou pedaços do programa, fazendo com que muitas vezes a gente queira simplesmente resolver de qualquer maneira.

- Porque ficamos cansados. Às vezes o dia foi muito longo ou simplesmente queremos partir para outro programa/software e acabar logo com o atual.
- Porque deixamos para o dia seguinte. Sempre falamos que podemos fazer a limpeza “depois” e geralmente “depois” quer dizer “nunca” (LeBlanc's Law).
- Porque ficamos satisfeitos que ele está fazendo o trabalho que precisavamos fazer.

O que devemos fazer como profissionais?

- “[...] You will not make the deadline by making the mess. Indeed, the mess will slow you down instantly, and will force you to miss the deadline. The only way to make the deadline – the only way to go fast – is to keep the code as clean as possible at all times.”

Sobre a arte de escrever código limpo:

- “[...] being able to recognize good art from bad does not mean that we know how to paint. So too being able to recognize clean code from dirty code does not mean that we know how to write clean code!”

O que é “Clean Code”?

- Bjarne Stroustroup, criador de C++ e escritor do livro “The C++ Programming Language”: *“I like my code to be **elegant** and **efficient**. The logic should be **straighforward** to make it hard for bugs to hide, the **dependencies minimal to ease maintenance**, **error handling complete** according to an articulated strategy, and **performance close to optimal** so as not to tempt people to make the code messy with unprincipled optimizations. **Clean code does one thing well.**”*
- *Grady Booch, autor de “Object Oriented Analysis and Design with Applications”: “Clean code is **simple** and **direct**. Clean code **reads like well-written prose**. Clean code never obscures the designer's intent but rather is full of crisp[clearly defined] abstractions and **straighforward** lines of control”.*
- Dave Thomas, fundador da OTI, pai do Eclipse, autor de Pragmatic Programmer: *“**Clean code can be read**, and enhanced by a developer other than its original author. **It has unit and acceptance tests.** It has **meaningful names**. It provides **one way** rather than many ways for doing one thing. It has **minimal dependencies**, which are explicitly defined, and provides a clear and **minimal API**. Code should be **literate** since depending on the language, not all necessary information can be expressed clearly in code alone.”*
- Michael Feathers, autor de “Working Effectively with Legacy Code”: “*I could list all of the qualities that I notice in clean code, but ther is one overarching quality that leads to all of them. **Clean code always looks it was written by someone who cares**. **There is nothing obvious that you can do to make it better**. All of those things were thought about by the code's author, and **if you try to imagine improvements, you're led back to where you are**, sitting in appreciation of the code someone left for you – code left by someone who cares deeply about the craft.*”
- Ron Jeffries, autor de “Extreme Programming Installed” e “Extreme Programming Adventures in C#”: *“In recent years I begin, and nearly end, with Beck's rules of simple code. In priority order, simple code: **runs all tests**; contains **no duplication**; **expresses all the design ideas** that are in the system; **minimizes the number of entities** such as classes, methods, functions, and the like. […]. I also look at **wheter an object or method is doing more than one thing**.[...]”*
- Ward Cunningham, inventor do Wiki, inventor do Fit, coinventor de XP. Força motivo por trás de Design Patterns. Líder sobre pensamentos quanto a Smalltalk e OO. O pai de todos aqueles que se importam com código: *“You know you are working on clean code when **each routine you read turns out to be pretty much what you expected**. You can call it beatiful code when the codes also **makes it look like the language was made for the problem**.”*

Resumindo o que eles acham. Clean code é:

- Simples
- Direto
- Eficiente
- Aquele que passa nos testes
- Legível e literal
- Sem duplicações
- Faz e é o que você esperava
- Minimal
- Cheio de significado
- Feito por alguém que se importa
- Não existem melhoras óbvias

"The next time you write a line of code, remember you are an author, writing for readers who will judge your effort". Uncle Bob.

"Indeed, the ratio of time spent reading vs writting is well over 10:1. We are constantly reading old code as part of the effor to write new code. Because this ratio is so high, we want the reading of the code to be easy, even if it makes the writing harder. Of course there's no way to write code without reading it, so making it easy to read actually makes it easier to write", Uncle Bob.

"Leave the campground cleaner that you found it". The Boy Scouts of America.

Chapter 2 – Meaningful Names

Use Intention-Revealing Names:

- Escolher bons nomes consome tempo, mas salva muito mais tempo para leitura.
- Sempre que encontrar um nome que você demora para entender, troque-o
- Deve contar porque a coisa existe, o que faz e como é usada.
- **Se depois de um nome vem um comentário, o nome talvez devesse ser o comentário**
- **Para listas**: que tipo de itens ela contém?
- **Em uma lista x, usamos x[0]**: qual é a importância da posição 0?
- **Para constantes:** qual é o significado dessa constante? Um nome explicito poderia ajudar a entender.

Avoid Desinformation:

- Temos que tomar cuidado com os nomes para não dar impressão que uma coisa é uma, mas na verdade é outra bem diferente
- Ex: devemos usar accountList somente se realmente for uma lista.
- Cuidado com nomes parecidos na **forma**: XYZControllerForEfficientHandlingOfString é muito parecido na forma com XYZControllerForEfficientStorageOfStrings.
- **Formas e ortografia são informações**. Ajuda bastante se colocarmos nomes parecidos para ordenar alfabeticamente (auto-complete do Eclipse) coisas que façam coisas parecidas, mas a diferença entre elas deve ser óbvia.

Make meaningful distictions:

- Às vezes colocamos nomes somente para agradar o compilador.
- Sabemos que não vai compilar se declarmos coisas com mesmo nome, então criamos uma diferença sutil entre os nomes para diferenciá-las para o compilador/interpretador, mas não para humanos.

- Colocamos um erro de ortografia, um número no final, etc.
- Uma função copyChars(a1[], a2[]) poderia ser muito mais clara como copyChars(source[], destination[]).
- A idéia é sempre usar nomes que se distinguem claramente uns dos outros. Qual é a diferença entre accounts e accountsInfo? E de theMessage para message? money para moneyAmount?

Use Pronounceable Names:
- Se você não pode pronunciar o nome de algo, fica bem mais difícil de discutir com alguém
- Por que usar pszqint ao invés de recordId? Ou genymdhms ao invés de generationTimestamps?

Use Searchable Names:
- Hoje em dia procuramos muito por uma variável, constante ou método.
- Se o nome não for fácil de procurar, isso pode consumir tempo.
- Constantes são difíceis de encontrar: 7 pode ser qualquer coisa em qualquer lugar, enquanto que MAX_CLASSES_PER_STUDENT é bem fácil de encontrar.
- Para o Uncle Bob, nomes com uma letra só deveriam ser usado **somente** como variáveis locais em métodos pequenos.
- **Heurística: *"The lenght of a name should correspond to the size of its scope"***
- Se uma váriavel será acessada de vários lugares, ela deve ser fácil de ser encontrada.

Avoid Mental Mapping:
- Leitores não deveriam ter que traduzir os nomes para outros nomes mentalmente
- Os nomes deveriam espelhar os nomes do domínio da solução e do domínio do sistema
- Se você sabe como mapear um nome para outro, talvez o nome deveria ser esse outro
- Você pode saber mapear, mas um leitor vai ter que aprender a mapear ainda.

Class Names:
- Nomes de classes deveriam ter substantivos
- Evite nomes como Manager, Processor, Data e Info
- O nome de uma classe não deve ser um verbo.

Method Names:
- Deveriam ser verbos ou uma frase com verbos como postPayment, deletePage, save...
- Nomes de acessors deveriam sempre seguir o padrão da linguagem (get e set para Java e nome e nome= para Ruby)
- Nomes de methods booleanos deveriam ter *is* para java ou *?* para Ruby
- Quando usamos sobrecarga de construtores, talvez seja uma melhor idéia criar métodos de Classe (static) para cada um dos tipos. Ex: Complex fulcrumPoint = new Complex(23.0) poderia ser substituido por Complex fulcrumPoint = Complex.FromRealNumber(23.0). Nesse caso talvez seja uma boa idéia colocar o construtor como private para forçar que usem esse tipo de método.

"Say what you mean. Mean what you say". Uncle Bob.

Pick One Word per Concept/Don't Pun:
- Escolha um único nome para uma abstração e mantenha-o em todo lugar.
- Ex: qual a diferença entre fetch, retrieve e get em classes diferentes? Para qual classe eu uso qual?
- O mesmo vale no outro sentido. Não se deve usar um nome para uma coisa que não pertence

à aquela abstração.

Use **Solution Domain** Names:

- O código será lido por programadores
- Não tenha medo de usar termos comuns para programadores com nomes de algoritmos, nomes de padrões, matemática, etc
- Não temos que ter todos os nomes no domínio do problema. Os programadores podem não ser totalmente familiarizados com isso e podem acabar tendo que perguntar para o cliente por exemplo.

Use **Problem Domain** Names:

- Quando não há uma palavra técnica para o que você está definindo, use um termo do domínio do problema.
- Desse modo, pelo menos o programador pode descobrir do que se trata mais facilmente.

Add Meaningful Context

- Alguns nomes tem significado completo sozinhos. A grande maioria não!
- È sempre uma boa idéia colocar os nomes em contextos claros com classes, funções e namespaces bem nomeados.
- Caso isso não for possível, talvez seja bom adicionar no nome a que contexto ele pertence. Por exemplo, se você ver *state* por ai pode não saber que é parte de um endereço, então uma boa idéia pode ser colcocar *addrState*. (logicamente a melhor solução seria Adress ser uma classe).

“Short names are generally better than longer ones, so long as they are clear. Add no more context to a name than is necessary”

Chapter 3 – Functions

Small!

- Primeira regra: Funções deveriam ser pequenas!
- **Segunda regra: Funções deveriar ser menores do que pequenas!**
- Não existe nenhum estudo que comprove que é melhor fazer funções pequenas
- Funções deveriam ter poucas linhas; todas óbvias e que contam bem o que está acontecendo.

Blocks and Identing:

- Statements como if, while e else deveriam todos ter apenas uma linha.
- **Provavelmente a expressão do condicional devesse ser uma chamada de função.**
- Essa idéia não só deixa a função pequena e simples, mas a deixa mais documentada e descritiva.
- **Assim sendo, funções não deveriam ser grandes o suficiente para conter estruturas encadeadas.**

One thing

- **Functions should do one thing. They should do it well. They should do it only.**
- Como podemos saber se uma função está fazendo somente uma coisa? A idéia é que criamos funções para descer um nível de abstração.
- Exemplo do TO (como são definidas as funções no LOGO):
  - *TO renderPageWithSetupsAndTeardowns, we check to see wheter the page is a test page and if so, we include the setups and teardowns. In either case we render the page in HTML.*

    - Se uma função faz apenas esses passos que estão a apenas um nível abaixo do nome da função, então a função faz apenas uma coisa.
- **Outro maneira é ver se é possível extrair uma função dessa função com um nome que não seja meramente uma repetição da primeira função.**

Sections within Functions:

- Uma função que pode ser dividida em seções (do tipo declarations, initializations e sieve), provavelmente fazem mais do que uma coisa

One Level of Abstractions per Function:

- Para ter certeza que a função está fazendo uma coisa, temos que assegurar que as operações da nossa função estão todas no mesmo nível de abstração.
- Misturar níveis de abstrações é sempre confuso.

The Stepdown Rule:

- Queremos que o nosso programa seja organizado de uma maneira em que vamos descendo os níveis de abstrações lentamente. Isso pode ser exemplificados com parágrafos com TOs.

> *To include the setups and teardowns, we include setups, then we include the test page content, and then we include the teardowns.*
>
> *To include the setups, we include the suite setup if this is a suite, then we include the regular setup.*
>
> *To include the suite setup, we search the parent hierarchy for the "SuiteSetUp" page and add an include statement with the path of that page.*
>
> *To search the parent. . .*

Switch Statements

- Se um método tem switch (ou ifs encadeados), muito provavelmente faz mais do que uma coisa
- A idéia é enterrá-los no nível mais baixo possível do programa, ficando escondidos.
- Uma boa idéia é criar uma Factory de objetos, que dado uma entrada devolve o objeto apropriado que podemos usar polimorficamente.

Use Descriptive Names:

- Para seguir o que Ward Cunningham disse ("*You know you are working on clean code when each routine you read turns out to be pretty much what you expected")*, precisamos escolher nomes para nossas funções para que o leitor leia seu nome e saiba o que ela faz claramente. Afinal, ela de fato faz o que ela diz fazer se colocarmos um bom nome.
- **Quanto menor e mais focada uma função, mais fácil é para encontrar um bom nome para ela. Se ela tem um bom nome, o leitor sabe o que a função faz.**
- Não devemos ter medo de colocar um nome grande, porque:
    - É melhor do que um nome curtinho e enigmático
    - É melhor do que uma documentação longa
- Se você usar nos nomes, uma palavra consistente e que expressa bem um conceito, a legibilidade será simples e o leitor pode até intuir outros métodos parecidos.
    - Ex: includeSetupAndTeardownPages, includeSetupPages, includeSuiteSetupPage,... o leitor irá pensar que provavelmente existe includeTeardownPages e includeSuiteTeardownPage.

Function Arguments:

- **O ideal seria uma função não ter nenhum argumento; seguido por um, dois e três.**
- 4 já começa a ser bizarro e só deveria acontecer se extremamente necessário. E mesmo assim, você provavelmente encontraria uma melhoria para ser feita.

- Se quebramos nossas funções em muitas pequenas e passamos um argumento para função de mais alto nível, é bem provável que esse argumento não faça sentido em alguma das funções. Ele estará lá simplesmente porque deve ser passado adiante.
- Provavelmente os argumentos poderia ser variáveis de instância da classe do método.
- Testar com argumentos é mais chato e demorado (temos que pensar em várias possibilidades) do que um método que simplesmente faz alguma coisa. Com um argumento existem algumas possibilidades; com dois já temos combinações de possibilidades; com três...
- Argumentos que acabam sendo a saída de um método são estranhos para o leitor.

Common Monadic Forms

- O nome do método deve deixar claro que tipo de método será e o que fará com o argumento.
- Existem três tipos comuns de funções com um parâmetro:
  - Aqueles que fazem uma pergunta sobre o argumento. Ex: boolean fileExists(“MyFile”)
  - Aqueles que transformam o argumento em outro objeto. Ex: InputStream fileOpen(“MyFile”)
  - Aqueles que chamam um evento. Geralmente são métodos void que simplesmente mudam o estado do objeto em que foi chamado.
- Métodos fora dessas formas devem ser usados com cuidado.
  - Um argumento que é a saída de um método é confuso.
  - Se um objeto será “transformado”, ele deve aparecer como o valor de retorno.

Flag Arguments

- **Argumentos Flags são feios. É uma péssima prática passar um booleano como argumento.**
- Claramente mostram que a função faz mais do que uma coisa e que o nome não é bom.
- Provavelmente o melhor a se fazer é quebrar em dois métodos!

Argument Objects:

- Quando uma função precisa de 2 ou 3 argumentos, muito provavelmete um objeto representaria melhor esses argumentos que está passando.
  - Ex: makeCircle(double x, double y, double radius) para makeCircle(Point center, double radius).

Argument Lists

- Passar uma lista como argumento talvez seja uma boa ideia quando:
  - queremos passar um número variável de argumentos,
  - os argumentos são tratados da mesma maneira pelo método

Verbs and Keywords:

- **Bons nomes de funções dizem qual é sua intenção e quais são os argumentos que aceita**
- Funções Monades devem ter nomes que combinam com sua funcionalidade e seu argumento
  - Deveria ser uma pequena oração como verbo e substantivo
  - Ex: write(name). Sabemos que name está sendo escrito.
- Podemos usar keywords para deixar mais claro a ordem e quais são os argumentos.u
  - Ex: writeField(name); temos mais uma informação sobre o que é o name.
  - Ex: assertExpectedEqualsActual(expected, actual); agora sabemos a ordem!

Have no side effects

- Sua função diz que vai fazer alguma coisa, mas ela também faz uma coisa “escondida”
- Geralmente resultam em acoplamentos temporários (temporary coupling) e dependência na ordem de execução (se rodar uma função depois a outra, podem ocorrer erros se rodar a

outra depois a uma).
- Geralmente são aonde o leitor se perde e diz “O que raios é isso?”.
- Geralmente, quando alguém precisa fazer uma alteração, não sabe encontrar aonde está acontecendo algo já que isso está escondido em algum lugar.
- Alguém que acredita no que a função faz através do nome, pode ter efeitos colaterais bastante ruins
- Claramente não fazem seguem “One thing only”

Output arguments:
- **Em geral, output arguments deveriam ser evitados!**
- Se vai criar uma função que altera o estado de alguma coisa, que essa alteração seja chamada como um método na classe a ser alterada.
  - Ex: void appendFooter(StringBuffer report) poderia ser um método de StringBuffer em que fazemos report.appendFooter().
- É bom lembrar que argumentos geralmente são inputs!

Command Query Separation:
- **Ou a função altera o estado do objeto sobre a qual é chamada, ou devolve alguma informação sobre aquele objeto. Fazer as duas coisas é bastante confuso!**
- Métodos que fazem alguma coisa e devolvem um booleano se deu certo ou não são bastante estranhos algumas vezes.
  - Ex: boolean set(String attribute, String value). Quando usamos em if (set(“username”, “joaomm”), não sabemos muito bem o que está acontencendo. Será que estamos checando se o username está “setado” para “joaomm”? Parece que o autor da função pensou em set como um verbo, mas o leitor pensa primeiro em um adjetivo quando lido dentro do if.
  - Uma boa solução seria fazer a checagem do atributo em um método e só depois de fato setar. Algo como if (attributeExists(“username”)) setAttribute(“username”, “joaomm”).

Prefer Exceptions to Returning Error Codes
- Retornar um código de erro é bem parecido com retornar um boolean.
- Primeiro que vai ter um problema de leitura como com booleans
- Segundo e pior do que primeiro, provavelmente geraremos estruturas encadeadas para lidar com esse erro. O método que chama terá que lidar com o erro imediatamente.
- A solução é usar exceções tratadas com try/catch.

Extract Try/Catch blocks
- A contrução do try/catch mistura tratamento de erro com o código “normal”
- Uma boa idéia é sempre extrair o que está dentro do try para um método e o do catch para outro.

Erro Handling Is One Thing
- Obviamente o tratamento de erro é “One thing”. Logo, uma função que tem um try/catch deveria só conter o try/catch; sem nada antes e nada depois.

The Error.java Dependency Magnet
- Geralmente quando fazemos códigos de retornos, criamos um enum Erro com todos os erros.
- Isso gera uma grande dependência dentro do sistema, porque todas as classes que podem soltar ou tratar um erro, vão ter que incluir esse Error.java e teremos que recompilar e buildar tudo.
- Algumas vezes as pessoas reaproveitarão (acochambrarão) os erros para não ter que recompilar

- A solução é usar exceptions!

Don't Repeat Youself

- A raíz do mal em um software é a duplicação de código!
- Geralmente é possível fazer uma refatoração para extrair o código duplicado

How Do We Write Functions Like This?

- Quase nunca escreveremos funções seguindo essas idéias de primeira.
- Escrevemos uma função cheia de problemas, mas que passa nos teste!
- Depois aos poucos vamos refatorando para chegar em funções bonitos e agradaveis.

Chapter 4 – Comments

"*Nothing can be quite so helpful as a well-placed comment. Nothing can clutter up a module more than frivolous dogmatic comments. Nothing can be quite so damaging as an old crufty comment that propagates lies and misinformation.*

Comments are not like Schindler's List. They are not "pure good." Indeed, comments are, at best, a necessary evil. If our programming languages were expressive enough, or if we had the talent to subtly wield those languages to express our intent, we would not need comments very much—perhaps not at all.

The proper use of comments is to compensate for our failure to express ourself in code. Note that I used the word failure. I meant it. Comments are always failures . […].

[…] Every time you express yourself in code, you should pat yourself on the back. Every time you write a comment, you should grimace and feel the failure of your ability of expression. "

O problema com comentários:

- Geralmente eles dizem mentiras. Não intencionalmente, mas porque é difícil mantê-los.
- O código muda muito. Pedaços são trocados o tempo todo, mas comentários não.
- Nós poderiamos usar do nosso tempo mantendo os comentários corretos e atualizado, mas poderiamos usar esse tempo para melhorar o código.
- Comentários desatualizados é bem pior do que não ter comentários. Eles podem nos dar informações falsas que nos desorientam.

Explain yourself in code

```
// Check to see if the employee is eligible for full benefits
if ((employee.flags & HOURLY_FLAG) &&
    (employee.age > 65))
```

OU

```
if (employee.isEligibleForFullBenefits())
```

Good Comments

- Comentários para explicar/exemplificar expressões regulares
- Comentários que não explicam o que está acontencendo no código, mas sim uma decisão que foi tomada de porque fazer aquele código. Por exemplo: explicar porque aquele teste é interessante.
- Comentários sobre uma biblioteca padrão ou em coisas que não podemos alterar o código. Talvez seja bom colocar um comentário para expressar o que ocorre já que a biblioteca não faz.
- Comentários para deixar avisos. Exemplo: testes longos ou que algo não é thread safe.
- Comentários de TODOs.
- Comentários que amplificam a importância de algo.
- JavaDocs se você estiver fazendo uma API pública (cuidado para não ficar desatualizados)

Bad Comments:

- Todo comentário que te força a olhar em outro módulo para entender seu sentido.
- Comentários que “narram” o código e acabam sendo redundantes.
- Comentários que não são precisos. Alguém pode confiar nele!
- Comentários que servem de log da classe. Ficam cada vez maiores e hoje temos controle de versão!
- Comentários que são apenas ruídos...não dizem nada de importante!
- Código comentado: eles são importantes? Porque estão ali? Será colocado de volta? Alguém esqueceu? Não sabemos o que quer dizer...
- Comentários html
- Comentários que não são relacionados ao local onde foram colocados.
- Comentários enormes!
- Comentários que precisam de uma explicação!

Chapter 5 – Formatting

“*We want them to perceive that professionals have been at work. If instead they see a scrambled mass of code that looks like it was written by a bevy of drunken sailors, then they are likely to conclude that the same inattention to detail pervades every other aspect of the project.* ”

The Newspaper Metaphor

- Queremos que nossos código sejam como reportagens em um jornal. O nome do arquivo/classe nos mostra se é isso que queremos ler; logo no inicio, temos os conceitos e algoritmos de alto níve; e conforme vamos descendo, o nível de detalhes vai aumentando, até encontrarmos as funções de baixo nível e de maior detalhe.

Vertical Openness Between Concepts

- Cada linha é uma operação/expressão; cada grupo de linhas representa um pensamento completo; assim, cada pensamento deveria ser separado por linhas vazias.
- **Então queremos que os imports, packages, atributos e métodos fiquem separados.**

Vertical Density

- Enquanto a “Openness” cuida da separação entre os conceitos, a “Density” cuida da proximidade das associações.
- Idéias que estão intimamente relacionadas tem que estar próximas.
- Comentários inúteis atrapalha a associação entre variáveis, por exemplo.

Vertical Distance

- **“*Concepts that are closely related should be kept vertically close to each other.*”**
- Não querermos ter que perder tempo procurando aonde está algo; queremos usar nosso tempo para entender rapidamente o que queremos para implementar novas funcionalidades.
- A separação entre os conceitos deveria espelhar a separação dos conceitos.

Declarações de Variáveis

- Variáveis deveriam ser declaradas perto da onde são usadas
- Variáveis relacionadas com loops deveriam ser declaradas no loop

Variáveis de Instância

- Deveriam ser declaradas no topo da classe
- Isso não aumenta a distância vertical da onde são usadas porque, teoricamente, a grande maioria dos métodos deveriam usar essa variável.

Funcões dependentes

- Se uma função chama a outra, elas deveriam estar verticalmente perto.
- Se possível, a função que chama deveria estar acima da chamada.

Horizontal Formatting

- **Devemos deixar as linhas curtas**
- **O limite do Uncle Bob é 120**

Horizontal Openness and Density

- Usamos espaços em branco para desassociar ou associar elementos em uma linha.
- Atribuições tem que ter espaços em branco pois claramente temos duas partes (a = b)
- A função e seus argumentos são intimamente ligados. Cada argumento é separado um do outro.

Team Rules

- Obviamente, quando trabalhamos em equipe, devemos estabelecer regras comuns de formatação
- A formatação é um importante aspecto da **unidade** de um projeto

Chapter 6 – Objects and Data Structures

Data Abstraction

- Por que usamos variáveis privadas e métodos de acesso?
- “Hiding implementation is not just a matter of putting a layer of functions between the variables. Hiding implementation is about abstractions!”
- Classes devem expor uma interface abstrata para um cliente manipular a essência sem conhecer a implementação.
- Ao criar a interface de uma classe, temos que pensar cuidadosamente. A abstração criada será muito importante para um cliente não se preocupar com a implementação.
- Getters e Setters nos dão uma idéia do que está acontencendo por baixo dos panos. Abstrações bem escolhidas encobrem o que está por baixo.
    - Exemplo: Ponto Cartesiano. Uma abtração pode nos dar os métodos como getY, getX, getR e getTheta. Não sabemos como o ponto está representado na implementação.

Data/Object Anti-Symmetry

- **Qual a diferença entre Objetos e Estruturas de Dados? Elas são complementares!**
- Objetos escondem os seus dados usando abstrações, provendo funções para operar com eles.
- Estruturas de Dados expõe seus dados e não tem funções significativas.
- “Procedural code makes it hard to add new data structures because all the functions must change. OO code makes it hard to add new functions because all classes must change.”

Exemplo: Formas Geométricas e seu calculo de área

- “Procedural”:
    - Cada forma é uma Estrutura de Dados com as informações específicas
    - A classe Geometry seria responsavel pelo calculo de área (tem um if para cada forma).
    - Se adicionarmos a função perimetro, nenhuma forma seria alterada e a única mudança seria na classe Geometry.
    - Se uma nova forma for adicionada, teriamos que mexer em todos os métodos de Geometry.
- “Solução polimorfica”

    - Cada forma é uma Classe que herda de Shape
    - Se adicionarmos uma figura, nenhuma outra é afetada.
    - Se queremos adicionar a função perimetro, todas as classes devem ser alteradas.

The Law of Demeter

- Modulos não deveriam conhecer a “parte interna” (the innards) dos objectos que manipula
- **Um método f de uma classe C deveria chamar métodos:**
    - **Da própria classe C;**
    - **De um objeto criado por f;**
    - **De m objeto passado como argumento para f.**
    - **De um objeto guardado em uma variável de instância.**
- Dessa forma, um método não deveria chamar métodos de objetos retornados por um dos permitidos. “Talk to friends, not to strangers”.

Train Wrecks

- São chamadas como *outputDir = ctxt.getOptions().getScratchDir().getAbsolutePath();*
- Geralmente é um pouco melhor não fazer essas chamadas inline (passar para variaveis temporárias)
- Se eles quebram ou não a Law of Demeter é uma questão se são objetos ou estruturas de dados.

Hiding Structures

- Mas qual seria a solução para *outputDir = ctxt.getOptions().getScratchDir().getAbsolutePath()*?
- Não queremos ficar fazendo perguntas sobre a estrutura interna dos objetos, mas sim falando para eles fazerem alguma coisa! Provavelmente não é responsabilidade dessa classe se ela precisa ficar fazendo tantas perguntas para conseguir o que precisa.
- Para o que essa class precisa do AbsolutePath do ScratchDir? O autor queria criar um Scratch file usando um caminho dado.
- Passamos a responsabilidade para o ctxt com ctxt.createScratchFileStream(classFileName)!

Conclusion

- Devemos saber bem a diferença entre Objetos e Estruturas de Dados.
- O objetivo é ter grande flexibilidade: as vezes queremos mais tipos de dados e usamos objetos; as vezes queremos adicionar comportamentos simplesmente.

Chapter 7 – Error Handling

“Error handling is important, but if it obscures logic, it's wrong”

Use Exceptions Rather Than Return Codes

- Usar códigos de retorno obrigam o cliente a ficar checando todos os erros possíveis. Além disso, são bastante fáceis de serem esquecidos.
- Levantar erros são bons nesse sentido. O código cliente é mais limpo e sua lógica não é obscurecida pelo tratamento de infinitos códigos de erros.
- O código fica mais limpo porque existe uma separação clara entre a lógica e o tratamento de erros.

Write Your Try-Catch-Finally Statement First

- O primeiro caso de teste que deveria ser criado é aquele que verifica que uma exceção é lançada.
- Uma vez feito isso, você não se preocupa mais com o erro e garante que seu comportamento

seja adicionado com um código limpo.

Define a Normal Flow

- Não deixe o tratamento de erros definir o fluxo e a lógica!
- Faça refatorações como a Special Case Pattern para fazer o cliente trabalhar em um fluxo normal.

Don't Return Null and Don't Pass Null

- Se nossos métodos retornam null então temos que ficar verificando isso o tempo todo.
- Se você está tentado a retornar null, lance uma exceção ou faça um Special Case.
- Se um método devolve uma lista, devolva uma lista vazia ao invés de null.

Chapter 8 – Boundaries

Using Third-Party Code

- Geralmente os códigos de terceiros tendem a ter muito mais do que precisamos. Eles devem ser aplicáveis em várias situações e por isso devem ser ultra flexiveis.
- O problema são coisas como Sensor s = (Sensor)sensors.get(sensorId). Temos um cast porque sensors é um HashMap de Object. Isso é feio!
- Mesmo que usassemos os generics, a interface pode mudar (como já aconteceu) e teremos que mudar em vários trechos do código.
- Uma boa solução é fazer um wrapper que prove somente os métodos que queremos para a nossa aplicação e deixamos detalhes de implementação (como generics ou casts) escondidos.
    - Minimiza suas dependencias, tratamento de erros encapsulados e nós definimos os erros.
    - Fácil de fazer mocks e você não depende tanto das escolhas de desing da library.

Chapter 9 – Unit Tests

The Three Laws of TDD

1. You may not write production code until you have a failing unit test
2. You may not write more of a unit test than is sufficient to fail, and not compiling is fail
3. You may not write more production code than is sufficient to pass the currently failing test

Keeping Tests Clean

- O código de teste também deve ser limpo, pois também evoluem com o tempo.
- Sem os testes não há como ter a segurança que tudo está funcionando a cada mudança.
- Sem a segurança fica cada vez mais complicado querer limpar o código.
- “Test code is just as important as production code. (…) It requires thought, design and care. It must be kept clean as production code.”

  “If you don’t keep your tests clean, you will lose them. And without them, you lose the very thing that keeps your production code flexible. Yes, you read that correctly. It is unit tests that keep our code flexible, maintainable, and reusable. The reason is simple. If you have tests, you do not fear making changes to the code! Without tests every change is a possible bug. No matter how flexible your architecture is, no matter how nicely partitioned your design, without tests you will be reluctant to make changes because of the fear that you will introduce undetected bugs. ”

Clean Tests

- O que faz um código de teste ser limpo? “Readability, readability and readability!”
- Testes devem ser curtos, direto ao ponto e usar somente o objetos que realmente importam. Qualquer que ler sabe o que está acontecendo sem ser preocupar com detalhes.

Domain-Specific Testing Language

- Construir um conjunto de funções e utilitários que deixam os detalhes escondidos e fazem com que os testes fiquem simples de serem implementados e de serem lidos.
- Essa API de testes é construida aos poucos através de refatorações do seu código

One Assert Per Test

- Existe uma escola de pensamento que um teste JUnit deveria ter somente um assert
- De fato isso ajuda na legibilidade, mas as vezes é necessário fazer muita coisa para evitar duplicação do código.
- A regra é interessante como um guia. Seria bom que cada teste tivesse um nome bem claro e o assert seria muito natural. Para isso tentamos usar boas DSTL.

Single Concept Per Test

- Cada teste deveria lidar com apenas um conceito!

FIRST

- Testes deveriam seguir as seguintes regras para serem limpos
  - **Fast**: testes deveriam executar rapidamente. Se forem lentos você não os rodará com tanta frequencia e você não terá a segurança necessária para limpar o código.
  - **Independent**: um teste não pode depender de outro e podemos executá-los em qualquer ordem. Um teste não pode preparar uma algo para o outro.
  - **Repeatable**: testes deveriam ser repetitiveis em qualquer ambiente.
  - **Self-Validating**: testes devem ter uma resposta booleana para serem automatizados.
  - **Timely**: deveriam ser criados de maneira conveniente. Faça testes antes do código de produção.

Chapter 10 - Classes

Class Organization

- Segundo o padrão Java deveriamos organizar a classe da seguinte maneira
  - Lista de Variáveis: public static constants, private static variables e private instance variables
  - Métodos: publicos primeiro e os métodos privados que usa logo abaixo

Classes Should be Small!

- A primeira regra é que classes deveriam ser pequenas.
- A segunda é que as classes deveriam ser menores do que isso ainda.
- Com funções contamos linhas. **Com classes contamos responsabilidades**!
- Isso não é o mesmo que contar métodos! A classe pode ter poucos métodos mas cada um faz coisa demais.
- O nome da classe é um bom jeito de pensar se ela tem várias responsabilidades.
  - Se não conseguimos dar um nome conciso para a classe, provavelmente está muito inchada.
  - Quanto mais ambiguo é seu nome, provavelmente ela tem muitas responsabilidades.
  - Palavras como Processor, Manager ou Super são indicativos de um acumulo.
- Deveriamos conseguir resumir uma classe em poucas palavras sem usar If, And, Or, But.

The Single Responsability

- O principio diz que uma classe deveria ter somente uma razão para mudar.
- **Ter uma responsabilidade = Ter somente uma razão para mudar**
- Encontrar as responsabilidades nos ajuda a criar e reconhecer mais informações

> “Do you want your tools organized into toolboxes with many small drawers each containing well-defined and well-labeled components? Or do you want a few drawers that you just toss everything into? ”

Cohesion

- **Classes deveriam ter um número pequeno de variáveis de instância.**
- Cada método da classe deveria manipular uma ou mais destas variáveis.
- **Quanto mais variáveis um método usa, mais esse método é coeso dentro da classe.**
- “A class in which each variable is used by each method is maximally cohesive”.
- “When cohesion is high, it means that the methods and variables of the class are co-dependent and hang together as a logical whole”.

> “ The strategy of keeping functions small and keeping parameter lists short can sometimes lead to a proliferation of instance variables that are used by a subset of methods. When this happens, it almost always means that there is at least one other class trying to get out of the larger class. You should try to separate the variables and methods into two or more classes such that the new classes are more cohesive. ”

Maintaining Cohesion Results in Many Small Classes

- Quebrar funções grandes em funções pequenas já causam uma proliferação de classes
- Suponha que você queira extrair uma função para uma outra separada e ela usa 4 variaveis declaradas na função original. Devemos passar para a função os 4 parametros?
  - Não! Se essas variaveis fosse variaveis de instância não precisariamos passar.
  - Porém isso também significa que nossa classe perde coesão!
  - Mas pera! Se existem algumas funções que querem compartilhar variáveis, essas funções são uma classe propriamente ditas! “When classes lose cohesion, split them!”

Organizing for Change

- **Open-Closed Principle**: “Classes should be open for extension but closed for modification.”
- “In an ideal system, we incorporate new features by extending the system, not by making modifications to the existing code”

Isolating from Change

- “Needs will change, therefore code will change”
- Clientes que dependem de classes concretas estão muito sucetiveis a eventuais mudanças
- É mais esperto fazer ela receber um objeto de um tipo abstrato no seu construtor
  - Fica mais fácil de testar e cliente não ficar tão sucetivel as mudanças.
- A falta de Acoplamento significa que os elementos do nosso sistema são melhor isolados entre si e as mudanças serão isoladas.
- **Dependency Inverstion Principle**: “Our classes should depend upon abstractions, no on concrete details”

Chapter 11 – Systems

Separation of Main

- Um modo de separar construção e uso é mover os aspectos de construção para a classe main
- O resto do sistema não pensará na construção e assumirá que está feita
- As “flechas de dependencia” saem do main e vão para o resto.

Factories

- As vezes a aplicação deve saber Quando algo tem que ser construído.
- Podemos usar Abstract Factories, já que dão o poder de alguém chamar a construção de um objeto, mas escondem como a construção é feita.

Dependency Injection

- Dependency Injection: “application of Inversion of Control to dependency management”
- Inversion of Control: “Control moves secondary responsibilities from an object to ohter objects that are dedicated to the purpose”
- “In the context of dependency management, an object should not take responsibility for instantiating dependencies itself“.

  “An optimal system architecture consists of modularized domains of concern, each of which is implemented with Plain Old Java (or other) Objects. The different domains are integrated together with minimally invasive Aspects or Aspect-like tools. This architecture can be test-driven, just like the code. ”

  “The agility provided by a POJO system with modularized concerns allows us to make optimal, just-in-time decisions, based on the most recent knowledge. The complexity of these decisions is also reduced. ”

Chapter 12 – Emergence

Kent Becks Simple Design: a design is simple if it follows these rules (na ordem de importancia):

- Run all tests
- Contains no Duplications
- Expresses the intent of the programmer
- Minimizes the number of classes and methods

Simple Design Rule 1: Run all Tests

“The fact that we have these tests eliminates the fear that cleaning up the code will break it! ”

- O primeiro objetivo é fazer um sistema que faça o que foi feito para fazer
- Testable System: “A system that is comprehensively tested and passes all its tests all of the time”
- “Systems that aren't testable aren't verifiable.
- Para ter fazer um Testable System precisamos fazer classes pequenas e com um único propósito.
- Alto acoplamento torna os testes dificeis.
- Quanto mais testes criamos mais abstrações, interfaces e Dependency Injection usamos
- Com testes nós temos a segurança e poder para fazer melhorias no código!

Simple Design Rule 2 – 4: Refactoring

- Melhoramos nosso código fazendo refatorações incrementalmente.
  “ During this refactoring step, we can apply anything from the entire body of knowledge about good software design. We can increase cohesion, decrease coupling, separate concerns, modularize system concerns, shrink our functions and classes, choose better names,

and so on. This is also where we apply the final three rules of simple design: Eliminate duplication, ensure expressiveness, and minimize the number of classes and methods. ”

No Duplication

- Duplicações é o primeiro maior inimigo de um sistema com bom design.
- *“Undestanding how to achieve reuse in the small is essential to achieving reuse in the large”*
- O Template Method é uma refatoração comum de duplicações de mais alto nível

Expressive

- Podemos nos expressar com bons nomes, mantendo nossas funções e classes pequenas, usar uma nomenclatura padrão (como os Design Patterns) e ter testes expressivos.

  “ So take a little pride in your workmanship. Spend a little time with each of your functions and classes. Choose better names, split large functions into smaller functions, and generally just take care of what you’ve created. Care is a precious resource. ”

Minimal Classes and Methods

“ Our goal is to keep our overall system small while we are also keeping our functions and classes small. Remember, however, that this rule is the lowest priority of the four rules of Simple Design. So, although it’s important to keep class and function count low, it’s more important to have tests, eliminate duplication, and express yourself. ”

Chapter 13 – Concurrency

Why Concurrency?

- “Concurrency is a decoupling strategy. It helps us decouple what gets done from when it gets done

Concurrency Defense Principles

Single Responsibility Principle

- “The SRP stats that a five method/class/component should have a single reason to change.
- Design de concorrência é complex e é uma razão para se mudar um código. Logo, a recomendação é “Keep your concurrency-related code separated from other code”.

Corollary: Limit The Scope of Data

- Usamos synchronized para protejar as sessões criticas que usa o objeto compartilhado
- É importante minimizar o número de sessões como essas,
- Recomendação: “Take data encapsulation to heart; severely limit the access of any data tha may be shared.”

Corollary: Use Copies of Data

- Um bom jeito de evitar problemas com dados compartilhados é não ter dados compartilhados
- Uma boa ideia é fazer copias dos dados e tratá-los como “Read-only”.
- Outra ideia é distribuir copias capazes de armazenar as açoes feitas e só depois acontece um merge

Corollary: Threads Should Be as Independent as Possible

- Uma boa ideia é fazer cada thread existir em um mundo separado, sem compartilhar nada
- Recomendação: “Attempt to partition data into independent subsets that can be operated on by independent threadas, possibly in different processors”

Know Your Library

- Já existe muita coisa thread-safe em Java
- Recomendação: conhecer bem a java.util.concurrent

Beware Dependencies Between Synchronized Methods

- Recomendação: “Avoid using more than on method on a shared object”
- Existem horas que isso não é possivel. Então usamos um desses três jeitos:
  - Client-Based Locking: o cliente locka quem prove os métodos e garante que o lock segura os outros métodos também
  - Server-Based Locking: dentro de quem prove os métodos crie um método que o locka, chama os métodos e depois unlocka. O cliente chama o novo método.
  - Adapted Server: crie um intermediário que cria o lock.

Keep Synchronized Sections Small

- Locks são caros porque criam atrasos e adicionam overhead.
- Temos que fazer o design do nosso código ter o minimo de sessões críticas.

Writing Correct Shut-Down Code is Hard

- Recomendação: “Think about shut-down early and get it working early. It's going to take longar than you expect. Review existing algorithms because this is probably harder than you think.”

Testing Threaded Code

- Recomendações:
  - Treat spurious failures as candidate threading issues.
    - “Do not ignore system failues as one-offs”
  - Get your nonthreaded code working first.
    - Não tente procurar bugs relacionados as threads e não threads ao mesmo tempo.
    - Tenha certeza que seu código funciona sem as threads
  - Make your threaded code pluggable.
    - Faça seu código para facilitar testes de diferentes configurações
  - Make your threaded code tunable.
    - Considere permitir mudança do número de threads em tempo de execução
    - Considere permitir auto-tuning
  - Run with more threads than processors.
  - Run on different platforms.
  - Instrument your code to try and force failures.
    - Hand-coded: inserimos comandos para gerar outras histórias
    - Automated: ferramentas com Aspect-Oriented são capazes de fazer instrumentações automaticamente.

Conclusion

- Siga o Single Responsability Principle
- Quebre o sistema em “thread-aware” e “thread-ignorant codes”
- Garanta que você está testando cada tipo separadamente
- Código thread-aware deve ser pequeno e focado
- Aprenda a encontrar regiões critica e que precisam de locks e faça locks.
- “Change designs of the objects with shared data to accommodate clients rather than forcing clients to manage shared state.”