DISCIPLINA DE
QUALIDADE DO CÓDIGO FONTE (CLEAN CODE)

Sumário

Sobre o autor

Prof. Esp. Wagner Mendes Voltz

Olá, este é o livro de **Clean Code - Codificação limpa**, que foi elaborado especialmente para você conhecer ou aprimorar a arte de codificar uma solução. O conteúdo deste livro se aplica diretamente a pessoas desenvolvedoras, mas todos outros envolvidos na produção de um software, dentre eles gerentes de projetos, implantadores, analistas, testadores e donos de produtos, podem se beneficiar da leitura deste material. .

Meu nome é Wagner Mendes Voltz e sou o autor deste livro. Minha formação é em Tecnologia em Informática pela Universidade Federal do Paraná (UFPR) e o ano de conclusão desta graduação foi em 2005.

Antes mesmo de estar formado, já participava de programas de estágio na área de desenvolvimento de sistemas, utilizando a linguagem de programação PHP.

Em 2006, já atuando como analista de sistema, percebi a dificuldade que eu tinha em gerenciar equipes de desenvolvimento e para aprender mais sobre gestão de pessoas, ingressei numa especialização oferecida por uma escola de negócios (FAE Business School). Conclui este curso em 2007, com o título de Especialista em Administração e Gestão da Informação.

Creio que esta especialização trouxe uma nova visão de como trabalhar com pessoas e computadores e o desejo de lecionar já era existente em 2007.

Em 2008, comecei a desenvolver software na linguagem de programação Delphi, utilizando banco de dados Oracle e em 2010, dediquei todos os meus esforços para investir no aprendizado e desenvolvimento de aplicativos em Java, que atualmente é a minha especialidade.

Com o foco na linguagem Java e em desenvolvimento de sistemas, pude participar de diversos projetos e desafios, os quais me levaram a aprofundar o conhecimento de orientação a objetos e testes unitários. Mas alguma coisa ainda não ia bem, pois estávamos fazendo software, mas de forma descontrolada e desorganizada. Mesmo usando modelos tradicionais da engenharia de software, parecia que não saíamos do lugar.

Em 2011, tive o meu primeiro contato com agilidade. Uma consultoria, que visitou o lugar onde eu trabalhava, nos apresentou o Scrum. Gostei tanto que investi tempo esforço para me certificar neste *framework*, com isto me tornei um CSM (*Certified Scrum Master*) pela Scrum Alliance.

Com o passar dos anos fui percebendo pelo dia a dia que era necessário aprender outros “sabores” da agilidade. Aprendi sobre XP (*eXtreme Programming*), Kanban, métricas, *clean code* (código limpo), entre outros. A partir destas experiências com outros “sabores” de agilidade, hoje tenho a certificação KMP I (Kanban Management Professional) e sigo escrevendo e estudando sobre agilidade e codificação de soluções.

Nesta caminhada, tive oportunidades de palestrar no Agile Tour de Maringá-PR, Agile Brasil, Scrum Gathering Rio, CapiConf, Agile Curitiba e no TDC (*The Developers Conference*). Além disso, fiz vários amigos que estão fazendo uma grande mudança na produção de software nacional, gerando valor para o cliente.

Hoje estudo de tudo um pouco e amplio a minha caixa de ferramentas e tento identificar qual é o momento para usar cada um dos aprendizados.

Espero que este livro possa lhe ajudar a conhecer um pouco mais deste assunto.

Caso queira me acompanhar, me adicione em alguma rede social para trocarmos figurinhas:

- Twitter: @tiofusca
- Blog: http://medium.com/@wagnerfusca/
- Linkedin: https://www.linkedin.com/in/wagnerfusca/

E aí, está preparado? Vamos nessa??!!

O Código Limpo

O código e a justificativa deste material

Todo software é escrito numa determinada linguagem de programação que posteriormente
pode ser compilada, interpretada e executada pelo sistema operacional ou hardware. Além
destas características, cada linguagem de programação é classificada em um paradigma
baseado em suas funcionalidades. Dos paradigmas mais conhecidos temos:

- programação estruturada
- programação orientada a objetos

O paradigma de programação fornece a visão que o desenvolvedor possui sobre a estruturação e execução do programa.

Independente da forma como a linguagem trabalha (paradigma e compilação), independente
de qual sistema operacional é utilizada pela pessoa desenvolvedora, independente da
versão da linguagem de programação, sempre é possível ter código fonte organizado e que
facilite a manutenção e o entendimento.

Este material irá falar de código fonte e de como deixá-los mais profissionais. Irá também
falar de como você pode ser uma pessoa desenvolvedora de software melhor. Afinal, ainda
estamos aprendendo a fazer software nós mesmos e ensinando outros a fazê-lo melhor
(ver [MANIFESTO-AGIL]).

Além da frase acima citada logo no início do manifesto ágil, temos alguns princípios da agilidade que justificam o porquê é importante abordarmos o tema de codificação limpa. O manifesto ágil diz:

- ***Entregar software funcionando com freqüencia, na escala de semanas até meses, com preferência aos períodos mais curtos.***
    - Muitos de vocês já precisaram dar manutenção em software e sabem o quanto uma má escrita de código pode atrasar entregas, afinal entender o que o outro desenvolvedor fez não é algo tão fácil, caso este tenha feito sem usar boas práticas de programação.

- ***Construir projetos ao redor de indivíduos motivados. Dando a eles o ambiente e suporte necessário, e confiar que farão seu trabalho.***

    - Qual é a melhor pessoa dentro de um time de software que pode dizer se o código está bom ou não? Quem deve responder isto é a pessoa desenvolvedora. E caso não esteja boa a codificação, devemos incentivar e dar autonomia para que esta pessoa desenvolvedora consiga fazer daquele código fonte algo profissional. Indivíduos com estas motivações e desafios irão produzir software com primazia.
- ***O Método mais eficiente e eficaz de transmitir informações para, e por dentro de um time de desenvolvimento, é através de uma conversa cara a cara.***
    - Uma conversa cara a cara sempre traz resultados quase incalculáveis. Um código fonte bem escrito fala por si só e quando você for usar de uma conversa cara a cara, você irá focar no problema a ser resolvido e não ficará reclamando do código fonte do projeto.
- ***Simplicidade: a arte de maximizar a quantidade de trabalho que não precisou ser feito.***
    - Como é difícil manter a simplicidade no código fonte. Clean code vem para ajudar nisto e ser o nosso trilho para que todos possam caminhar em uniformidade (independente de linguagem ou projeto).
- ***Contínua atenção à excelência técnica e bom design, aumenta a agilidade.***
    - seu código fonte pode ser melhor hoje do que era ontem. Como você programa pode ser melhor hoje. Sempre busque a excelência técnica. Clean code é um dos passos que você precisa caminhar.

Pensando na justificativa acima, dividi o livro em 5 tópicos::

- Como identificar um código bom e ruim baseados em princípios e regras?
- Ter um código limpo depende mais de você do que das técnicas!
- Como escrever um código limpo?
- 9 regras para que seu código fonte seja melhor!
- Como analisar o seu código fonte a partir de validações dos padrões de mercado?

Ao ler cada um dos tópicos, você poderá refletir como tem sido o código fonte que você produz. Em alguns momentos serei bem provocativo e a resposta que você der a cada provocação irá ter tornar uma pessoa desenvolvedora mais profissional ou não. A provocação não é nada pessoal e caso você não concorde com algo, me procure no Linkedin. Terei o prazer em ouvi-lo e conversar contigo.

Por mais que eu seja provocativo, vou deixar algumas dicas para você começar amanhã uma mudança. A cada tópico você será desafiado a preencher novos compromissos que você pode adotar para ter uma codificação mais limpa.

E aí, preparado para o desafio?

Como identificar um código bom e ruim baseados em princípios e regras?

O código ruim

Todo algoritmo é uma sequência de passos para a solução de um problema. Como você escreve este algoritmo é que faz toda a diferença.
Um código ruim é fácil de identificar. Os próprios desenvolvedores não querem mexer naquela funcionalidade e se realmente precisam mexer, irão ter uma reação natural de frustração, como a imagem abaixo:

Figura 2 - Como é fácil identificar um código ruim

O código ruim pode ser identificado pelas seguinte características:

- classes muito longas
- métodos muito longas
- nomes de variáveis seguindo um padrão próprio e não sendo da forma mais legível.
- ausência de testes unitários
- métodos e atributos não agrupados e espalhados pelo classe ou arquivo
- entre outras tantas maneiras existente

Pensando em todas estas características, os especialistas no tema costumam categorizar o código fonte ruim através de antipadrões. Um anti padrão é uma solução aplicada pelo

desenvolvedor e que não representa a melhor decisão. Um determinado trecho de código pode representar um ou vários anti padrões. A soma destas más práticas representam o termo dívida técnica.

Podemos identificar os seguintes anti padrões:

- **Grande bola de lama (Big ball of mud):** Um sistema sem uma estrutura reconhecível

Figura 3 - A grande bola de lama

- **Âncora do barco (Boat anchor):** Manter uma parte de um sistema que não tem mais uso
- **Programando por exceção (Coding by exception):** Adicionar código novo para lidar com cada caso especial quando esse é reconhecido
- **Culto de programação (Cargo cult programming):** Usar padrões sem saber o motivo
- **Fluxo de lava (Lava flow):** Manter código indesejável (redundante ou de baixa qualidade) porque removê-lo é caro ou tem conseqüências imprevisíveis
- **Números mágicos(Magic number):** Incluir números inexplicáveis em algoritmos
- **Strings mágicas (Magic Strings):** Incluir literais no código para comparações inexplicáveis
- **Loop-switch sequence:** Codificar uma seqüência de passos usando switch com loop
- **Código espaguete (Spaghetti code):** Programas que têm a estrutura pouco compreensível, especialmente por mal uso das estruturas de código

- **Código lasanha (Lasagna code):**Programas cuja estrutura é composta por muitas camadas

Para conhecer mais sobre anti padrões acesse: http://antipatterns.com/

O código limpo

Gosto de imaginar que um código limpo é aquele que outras pessoas conseguem lê-lo sem dificuldades e entendem rapidamente para qual propósito cada linha de código foi implementada.
Um código limpo não me gera atraso de interpretação, gerando assim oportunidade para pensar nas regras e no valor a ser entregue.
Um código limpo é o resultado de muitas investidas e da aplicação de muitos conceitos que serão listados daqui para baixo.
Lembrando que você além de identificar o que é um código limpo ou não, você precisa desenvolver a habilidade de fazer o código limpo. Para isto, gostaria de apresentar algumas definições sobre a arte de programar.

Desenvolver um software é uma arte e que não é possível de ser repetida. Não é um modelo fabril o qual todo dia eu produzo código fonte de forma exatamente igual, semelhante uma linha de produção de indústria. A produção de software está mais próxima do artesanato de software. Toda pessoas desenvolvedora que se considera profissional, entende isto e considera a codificação um artesanato.

Figura 4 - O artesão

Agora chegamos num momento que precisamos refletir:

E aí, como você se considera atualmente? Mais um artesão de software ou um modelo repetitivo e sem trabalhar com boas práticas de programação?

E até quando você esperar para se tornar um artesão?

Sua empresa não busca e valoriza isto? Mas você só programa pensando na sua empresa ou você está preocupado em desenvolver este dom para atender os clientes da melhor maneira?

A culpa de algumas coisas estarem acontecendo é que não temos um comportamento de artesão de software e só estamos aceitando mais e mais demandas, sem tempo para evolução técnica.

O resultado do artesanato nunca é igual ao outro, mas os princípios aplicados sempre são os mesmos. Seguindo os princípios, o resultado da sua arte será apreciada por todos.

Princípios para um código limpo

A regra do Escoteiro

Figura 5 - O Escoteiro

Sempre mantenha o código limpo. Se comprometa não somente em escrever o código bom, mas mantenha ele limpo, mesmo se não tenha sido você que fez a sujeira. A regra do

escoteiro vem do lema da maior organização de jovens escoteiros dos EUA (*Boy Scouts of America*). O lema desta organização é:

Deixe a área do acampamento mais limpa do que como você a encontrou
Se seguirmos este lema, todo código novo estará limpo e a cada mudança numa classe existente, o código irá sempre melhorar. Assim evitamos a depreciação e deterioramento do código fonte.

E a pergunta vem em mente:
Quando você pode começar a praticar a regra dos escoteiros? Precisa de autorização formal para seguir este lema? Que tal começar agora?

Keep It Simple Stupid (KISS): Mantenha Isto Estupidamente Simples

Figura 6

O princípio por trás do KISS é ter código fonte simples e organizado sem a existência de:

- complexidade adicional desnecessária
- variáveis globais não usadas
- constantes globais não usadas
- nomes de classes, métodos e variáveis que realmente explicam o que que elas vão fazer
- estruturas de repetição (for, switch e while) que façam somente a repetição e sem encadeamento (uma estrutura dentro da outra)
- estruturas de condição pequenas evitando assim muitos caminhos de probabilidade

Don’t Repeat Yourself (DRY): Não Repita Você Mesmo

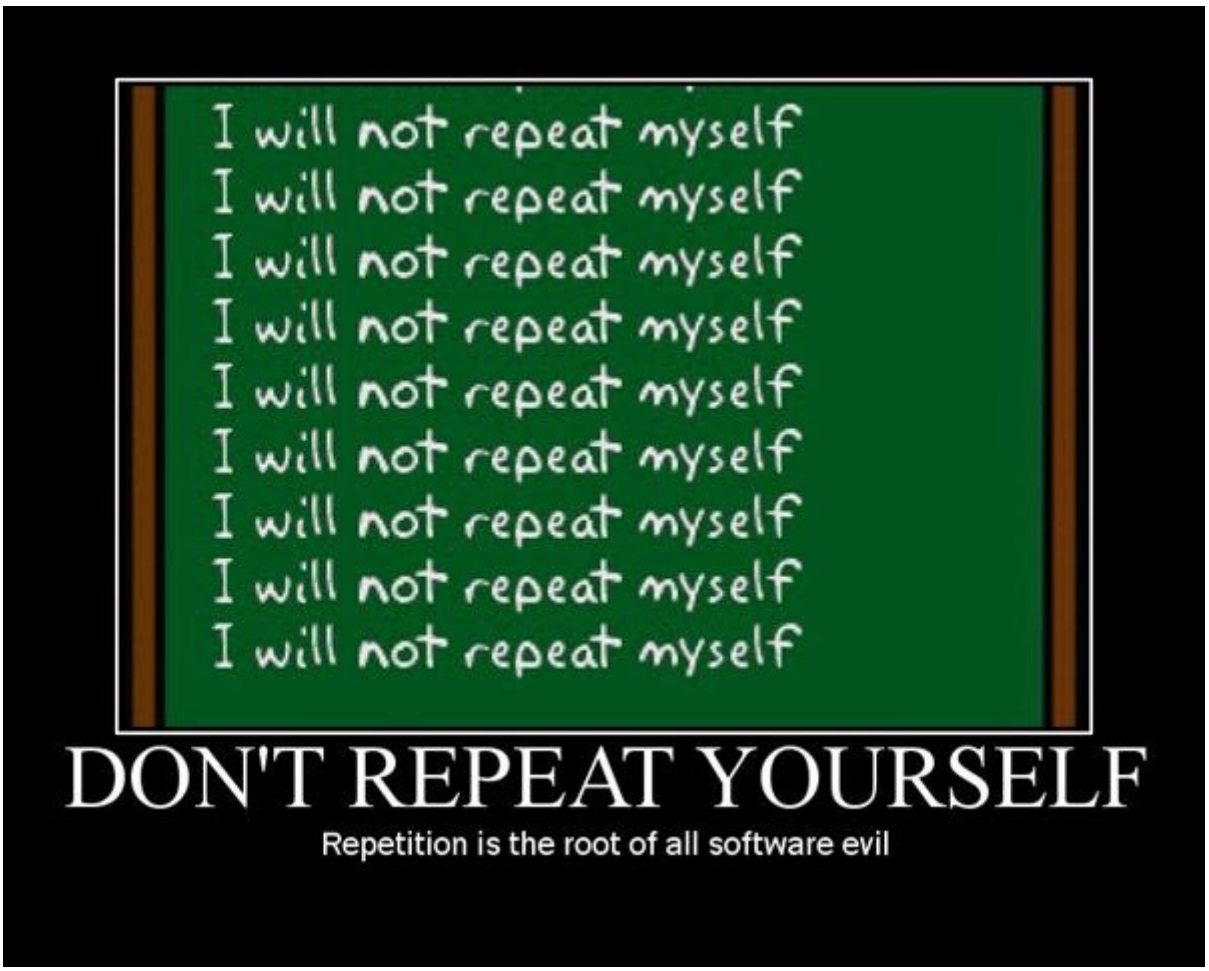

Figura 7

Duplicação de código deve ser evitada em qualquer paradigma de programação, em especial na orientação a objetos. Se isto estiver acontecendo, é sinal que você não sabe usar orientação a objetos a seu favor. Toda duplicação pode ser unificada num único método ou classe e as demais classes invocar esta nova classe contendo o código centralizado. Além disso, use o polimorfismo para manter seu código aberto a implementações e fechado a mudanças.

Duplicação gera dificuldade na manutenção do software e cria particularidades de negócios e validações que tendência a criação de bugs e comportamentos não definidos nos requisitos.

You Ain’t Gonna Need It (YAGNI): Você Não Vai precisar Disso

Figura 8

Conforme a imagem deste tópico mostra, uma pessoa no deserto não precisa de uma bóia. Chega a ser ridículo isto, mas e no nosso software. Quanto código que está lá e a gente mantém e não remove?

São linhas de código comentadas, classes depreciadas e tantas outras coisas que estão lá e ninguém teve coragem de tirar.

Já ouvir questionamentos do tipo que não era para eliminar pois era a documentação do sistema. Ou então não remover pois um dia ia precisar.

Se um dia vai precisar, por que não usar o controlador de versão (como git)?

Remova tudo aquilo que você identificar não ser necessário. Comece por códigos comentados que não trazem valor algum.

Separation Of Concerns (SoC): Separação de Responsabilidades

Figura 9

Será que numa geladeira deveríamos guardar produtos de limpeza? Desentupidores? Veneno para mosquito? E tudo isto ao lado de água e frutas prontas para consumo?

Um tanto quanto estranho isso né?!

Em código fonte, fazemos isso muitas das vezes. Temos uma classe chamado Pessoa que faz tudo. Ela sabe sobre os dados de documentos da pessoa, endereço, telefone e tantas outras informações. Uma única classe fazendo isto.

Não seria melhor separar as responsabilidades?

Que tal se tivéssemos uma classe Endereco que soubesse tudo sobre logradouro, número, CEP, cidade. E tendo esta classe, variamos um vínculo entre Pessoa e a classe Endereco? Assim melhoramos a responsabilidade.

O mesmo para telefones. Que tal ter uma classe Telefone e que tenha a responsabilidade de saber como é o formato do telefone, com DDD ou DDI. Se é celular ou não. Estas informações não são de responsabilidade de Pessoa e sim de Telefone.

No nosso exemplo, podemos refatorar a classe Pessoa e ter muitas outras classes como:

- Endereco
- Cidade
- Uf
- Telefone
- CPF
- entre outras.

Suas classes devem ser pequenas (COESÃO) e fazer pequenas coisas (MÉTODOS COESOS). Quanto menor forem elas, mais reaproveitamento você terá. Mais testes unitários você terá. Maior organização você terá.

Separe corretamente cada parte do código fonte e você irá manter os demais princípios ativos, afinal separando as responsabilidades vocÊ irá manter o código simples (KISS) e irá identificar o que não precisa (YAGNI).

Fechamento

Pois bem, vimos sobre o que é um código bom e ruim e os princípios que regem um bom código. Responda e se comprometa com as atividades abaixo:

1) Identifique quais anti padrões você costuma realizar. Selecione 3 e preencha abaixo:

_____

_____

_____

_____

2) Destes 3 anti padrões, escolha um que você vai começar a ter atenção já no seu próximo dia de trabalho. Como você fazer para não ter mais novos códigos com este anti padrão?

_____

_____

_____

_____

3) Dentre os princípios de bom código qual deles você pode começar a seguinte já amanhã? Responda o por que selecionou este princípio.

_____

_____

_____

_____

4) o que é a regra do escoteiro? Além do código fonte, é possível aplicá-la no seu dia a dia? Como?

Ter um código limpo depende mais de você do que das técnicas!

O comportamento

Quando buscamos um serviço, sempre buscamos profissionais qualificados e que resolvam o nosso problema. Quando sabemos que o profissional faz algo com qualidade escolhemos esperar um pouco mais para ser atendido e até pagamos mais pelo serviço prestado.

Fazemos isto quando levamos o carro a um mecânico, afinal sabemos que a manutenção correta no carro pode gerar segurança numa viagem e até prolongar a durabilidade do carro. Quando precisamos de um atendimento médico, desejamos especialistas que nos atendam corretamente e saibam identificar os sintomas das nossas doenças e a partir disto consigam nos medicar, trazendo mais tempo de vida. Quando algum destes profissionais é negligente, algo de pior acontece.

Quando o mecânico realiza um serviço desprezível, você precisará ir em outra oficina e refazer todo o serviço. Quando um médico não segue as orientações de higiene e prescrição ética da profissão, vida são prejudicadas.

O que quero dizer com esta introdução é que você é um desenvolvedor e você presta um serviço muito delicado. Seu código fonte fala muito de quem você é do que você valoriza. Quem escolhe fazer um código ruim ou bom é você. Esta escolha é algo intrínseco. É preciso ter coragem pra admitir que você precisa ter um código melhor. É preciso aceitar isso caso você queira ser um melhor programador.

Este capítulo vai falar um pouco do que é ser um codificador limpo. Vamos trabalhar um pouco valores e princípios que você precisa entender e viver. Você é responsável por algo muito valioso e não pode ter atitudes amadoras. Você é um produtor de tecnologia e não um mero consumidor. E como produtor, você precisa sempre melhorar a sua forma de programação.

Tudo começa com a coragem

Primeiro parabéns pela coragem de estar no mundo da tecnologia. É algo fascinante, mas ao mesmo tempo muito acelerado. Você já enfrentou muitos desafios quanto a desenvolvedor software e foi preciso coragem para encarar novas linguagens de programação e várias disciplinas técnicas. Hoje você está num curso de pós graduação e quando você decidiu em cursar esta especialização, foi necessário coragem para encarar os desafios de estudar aos finais de semana ou a distância.

Coragem fala disto. De tomarmos uma posição e irmos firmes no que acreditamos. Quando você tem coragem, você avança e faz acontecer.
Não sou psicólogo ou a melhor pessoa para trazer a definição de coragem, mas posso te dizer que os momentos que tive mais coragem, me fizeram crescer na carreira.

Tenha coragem de seguir naquilo que acredita. Tenha coragem em buscar ser um profissional melhor dia após dia. Tenha coragem de fazer alterações em código fonte de outras pessoas. Tenha coragem de aprender novas linguagens de programação. Tenha coragem para trabalhar em par de trabalho (*pair programming*). Tenha coragem para fazer software de forma incremental e contínua, sempre gerando valor. Tenha coragem para manter o software simples. Tenha coragem de fazer testes automatizados. Tenha coragem de expor o seu código fonte a todos os membros do time. Tenha coragem de abrir mão de documentação que servem como defesa. Tenha coragem de propor mudanças na forma como sua empresa produz software.

É isto que se espera de um profissional, que ele busque melhorar sempre. Este comportamento vai fazer você ter códigos limpos.

Transparência

Admita, você não produz o melhor código fonte do mundo.

Figura 10 - o mito grego Narciso, que gerou o termo narcisista

A imagem acima, é uma representação do mito grego Narciso, que era um jovem de boa aparência que ao ver seu rosto refletido na água, acabou se apaixonado pela própria imagem refletida.

Algumas pessoas desenvolvedoras são assim, acabam chamando de seu o código fonte de um produto, chegando ao ponto de amar demais aquele código sem considerar que este pode estar não nas melhores maneiras da programação.

Figura 11 - Ame o problema, não a solução - Consultoria K21

Precisamos tomar cuidado com isto. Devemos mais amar os problemas do que a solução. O seu código fonte é a solução e talvez ela pode ser melhorada através de refatoração e programação pareada. Entenda que existem problemas que precisam de mais de uma cabeça pensando, pois são problemas complexos. Entenda também que você está aprendendo a fazer software melhor diariamente.

Entender o como você está atualmente de forma sincera e transparente é uma atitude profissional. Eu não sei muitas coisas, mas caso precise trabalhar com uma nova linguagem, irei dizer ao líder o que eu domino e o que eu preciso aprender e qual o tempo eu preciso para isto. Eu não sei codificar com determinados padrões de projeto, então eu procuro pessoas que podem me ajudar nisto. Esta atitude que se espera de uma pessoa profissional.

Talvez a frase que mais represente alguém ser transparente, seja a frase do filósofo Sócrates

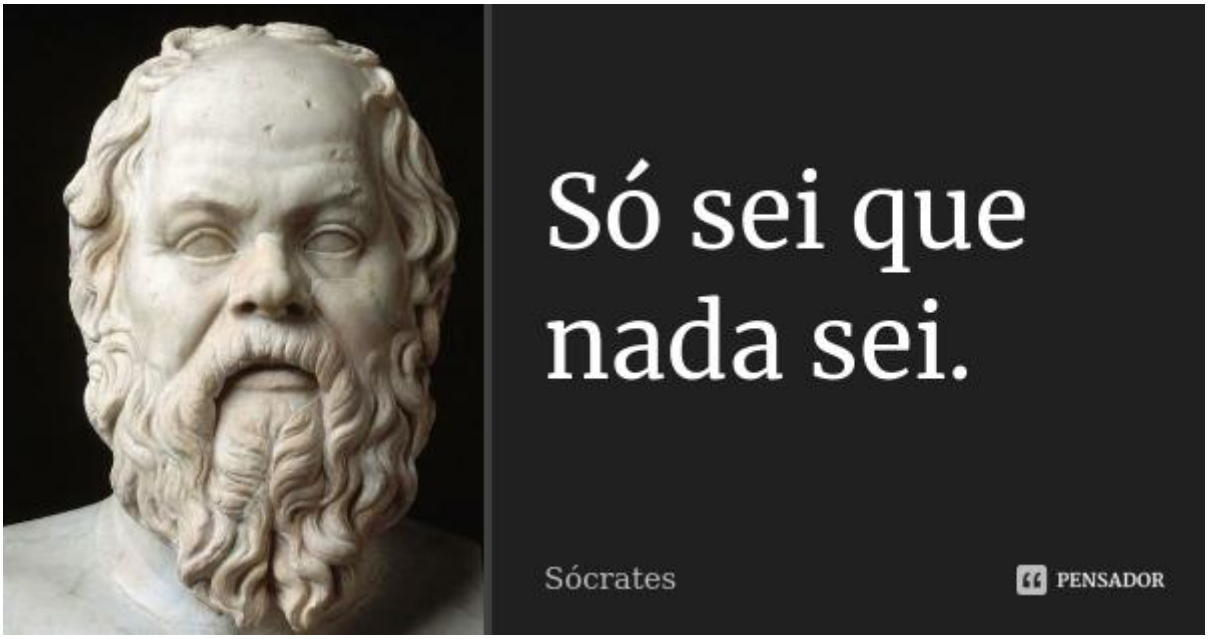

Figura 12

Adaptação

A partir da transparência e do entendimento de como estamos, precisamos traçar direções corretas para melhorar. Este é um outro ponto de um profissional. Ee consegue se adaptar. Eles conseguem melhorar as práticas de desenvolvimento, tanto individual como de time.
Adaptar-se a mudanças num mundo de constantes alterações é vital para a sua continuidade em seu trabalho .

Até o momento falamos de três comportamentos que se todo desenvolvedor tivesse, teríamos códigos muito melhores. São eles:

- coragem
- transparência
- adaptação

Além destes itens, o livro [Martin(2012)] conta sobre outros comportamentos esperados para um profissional. São eles:

- Assumir responsabilidades
- Dizer Não
- Dizer Sim
- Colaboração
- ensino, aprendizagem e habilidade

Assumir responsabilidades

Você aprende a assumir responsabilidades quando você encontra as consequências por não ter assumido antes. Ou seja,quando você não faz um teste unitário ou automatizado

você está entregando alterações no repositório, você está está assumindo uma responsabilidade que pode ser automatizada..

Dizer Não

Num time composto por analistas, gerentes de projetos, testadores e desenvolvedores, espera-se que o papel de desenvolvedor seja o que tem maior conhecimento técnico para dizer sobre como será implementada uma demanda. Você, como desenvolvedor, sabe em maiores detalhes quanto difícil será a implementação de um determinado problema. Você sabe que determinadas histórias de usuário não podem ser liberadas antes de um prazo. Compete a você dizer NÃO, quando algo não está bom no desenvolvimento. Compete a você dizer NÃO para codificar de forma suja. Compete a você dizer não às pressões de prazos impossíveis de serem cumpridos. Ao invés de aceitar toda a demanda, por que não tentar entregar partes pequenas e a cada entregar verificar o que foi entregue e de maneira incremental ter um novo software?

Compete a você dizer não para anti profissionais!

Figura 13

Dizer Sim

Aquele que diz não para tudo acaba sendo alguém difícil de lidar. O que diz não para tudo acaba sendo pessimista e negativo. Você como profissional precisa dizer sim e se comprometer com as boas práticas. E quando tiver que se comprometer com uma entrega, negocie com os interessados, sempre buscando o ganha-ganha.

Sim, toda demanda é possível de ser negociada, afinal somos profissionais e podemos ajudar todo o projeto a que este tenha um ritmo sustentável e de qualidade. Isto é ser profissional.

Quando você só diz sim, você irá perder, pois assumirá mais do que consegue fazer. Quando você só diz não, você ganha pois não irá fazer algo que não concorda, mas a outra parte interessada também perde, pois não haverá software.

Busque a cooperação. Negocie as demandas e ajude seus clientes a terem software com excelência.

Figura 14

Uma boa negociação irá te ajudar a você manter a disciplina de excelência técnica. Em prazos muito apertados, desenvolvedores tendem a não realizar práticas como testes e a produção de code smells (más práticas) aumenta consideravelmente.

Por isso, negocie e chegue a um acordo bom para ambas as partes.

Colaboração

Da mesma forma que você percebeu que precisa ser transparente, que tal receber ajudar outros para você ter um código mais limpo? E que tal ajudar outros? Afinal, todos nós estamos aprendendo a fazer software melhor e ensinando outros.
O código fonte de um projeto é colaborativo. Não é só de um único desenvolvedor. O código não pode ficar somente na máquina de alguém. Ele deve estar num versionador de código (como o Git) e com isto, todos podem colaborar.

Use práticas como programação pareada para que você aprenda e ajude outros na programação. Use também revisão de código e *pull requests*. Nada melhor do que uma pessoa olhando seu código e dando feedback para você melhorar.

Figura 15

Ensino, aprendizagem e habilidade

Em 2012 comecei a lecionar em cursos de graduação. Eram três disciplinas distintas:

- sistemas operacionais
- padrões de projeto (design pattern)
- desenvolvimento mobile (com Android)

Com exceção da disciplina de Android, as demais eu acreditava que sabia muita coisa e poderia lecionar de forma tranquila. Lembro perfeitamente de ter chegado na primeira aula de padrões de projeto. Cheguei com todo o tema preparado (orientação a objetos). Eram duas aulas de 50 minutos. Mas em menos de 25 minutos tudo o que eu tinha para falar já tinha acabado. E acabei adiantando o assunto. Quando eu percebi, eu tinha falado tudo o que sabia em menos de 100 minutos.

E agora??? o que fazer para a próxima aula?

Foi aí que eu aprendi que quanto mais você compartilha e ensina, mais você aprende. Quanto mais você ajuda alguém, isto vai se tornando uma habilidade e você fica com maior conhecimento daquele tema. Hoje ainda não sou um expert em padrões de projeto, mas posso dizer que cada dia que compartilho sobre o tema, fixo mais o aprendizado e este tem virado habilidade.

Para aplicar código limpo, você precisa aprender, mas também ensinar. E estas duas ações em conjunto efetuadas de forma frequente irão gerar em você uma habilidade. Com o passar do tempo será tão natural que você nem lembrará como escrevia código anteriormente.

Fechamento

Pois bem, vimos que comportamentos profissionais são importantes. Sem eles, a codificação limpa, poderá não existir. Responda e se comprometa com as atividades abaixo:

1) No seu dia a dia de trabalho, uma carga grande de coragem é necessária para que você desempenhe diversas atividades. Qual práticas da programação de software ou linguagens você precisa colocar em prática? Seja transparente

_____

_____

_____

_____

2) Baseado no que você anotou acima, como você pode ser adaptar e começar amanhã a fazer?

_____

_____

_____

_____

3) Você tem conteúdo, mas às vezes não se vê como alguém que pode ensinar. Elenque alguns conteúdos que você gosta e que poderia ensinar. Não fique preocupado na profundidade dos temas. Elenque o que você gosta de tecnologia e poderia escrever um post para um blog.

_____

_____

_____

_____

4) Na sua carreira profissional, muitas pessoas colaboraram com o seu dia a dia. Provavelmente você já tenha usado algum software ou projeto do GitHub que lhe foi muito útil Você já pensou como seria importante você participar e colaborar de algum projeto? Vamos refletir sobre isto? Que tal você começar a colaborar com algum projeto open source? Pense num repositório do Git Hub e tente ajudar o projeto cumprindo algumas *issues* (tarefas). Escreva abaixo o nome do repositório para você ajudar.

Como escrever um código limpo?

Neste capítulo você terá um resumo das boas práticas referente a codificação limpa. Para maior aprofundamento no tema, sugiro fortemente a leitura do livro Código Limpo ([Martin(2011)]). Além da apresentação das práticas, vou apresentar alguns códigos fontes. Caso você não esteja familiarizado com isto, concentre-se somente no que cada tópico diz. Vamos lá? agora é hora de aprender ferramentas poderosas e colocá-las em prática ainda hoje.

A última dica é comece com algum dos sub-tópicos abaixo. Após praticar bem um dos itens, avance para a próxima prática. A sugestão é a cada dois dias bem aplicados um tópico, avance para o próximo.

Nomes significativos

Alguns anos atrás me tornei pai. Eu e minha esposa nunca tivemos problemas em escolher o nome caso fosse menina, mas para menino até hoje temos divergência. Pois bem, venho uma menina e tudo ficou tranquilo.
Escolher um nome para uma classe, arquivo, método, função ou variável é algo que deve ser bem pensando. Leva tempo para escolher um bom nome e um bom nome economiza muito mais tempo de outra pessoa.
A melhor maneira de você validar se escolheu bem o nome de algo é pedir para outra pessoa desenvolvedora verificar se o seu código está entendível. Esta validação rápida por outra pessoa é o feedback mais efetivo. Use ela com frequência e caso esteja trabalhando com programação pareada (pair programming), aplique isto a todo o momento do pareamento.

Mas você já pensou em ter uma listagem de itens que seja possível checar se o nome que você escolheu atende uma codificação segura? Pois bem, é possível. Um nome significativo deve:

- revelar o propósito do que aquilo faz ou é.
- evitar informações erradas
- usar nomes pronunciáveis por seres humanos, evitando códigos não claros
- usar nomes passíveis de busca
- ser isentos de piadas ou gírias, pois nem todos as pessoas poderão entender

Para classes e objetos, a sugestão é usar nomes com substantivos. Não devemos usar verbo em classes e objetos.

Para métodos, a sugestão é usar verbos.

Funções e métodos

As funções e métodos são a primeira linha de organização que podemos ter em código fonte. Por isso, mantenha elas

- **PEQUENAS!!!!!**
- todos num mesmo nível de abstração e indentação.
- seguindo a regra de leitura de cima para baixo, ou seja, métodos mais importantes no topo
- com poucos parâmetros (no máximo 3).

Vamos aos detalhamentos.

Funções com mais de 100 linhas são ficam complicadas de entender. Algo entre 20 e 30 linhas por função ou método torna a leitura mais agradável do código fonte e também ajuda na elaboração de cenários de testes unitários, pois você estará testando métodos que fazem poucas coisas mas as fazem muito bem feitas. Fazer uma única coisa nos remete a COESÃO (que é o último subtópico) . Quanto temos algo coeso, é possível realizamos testes unitários eficientes e teremos reaproveitamento do código em todo o sistema.

Uma função pequena visa manter toda as linhas daquele método num mesmo nível de abstração e indentação. Este item também faz parte da regra que será vista no próximo capítulo (object calisthenics). Para mantermos no mesmo nível, precisamos encapsular em métodos as estruturas de condição (if) e de repetição (while, for). Vamos ao exemplo

Código ruim:

```
public void exibirExemploDeFuncaoPequenaComCondicional(){
        Date dataNascimento = 19/10/2000
        Integer idade = new Date().getAno() - data.getAno();
        if (new Date().getMes() == data.getMes() &&
                new Date().getDia() > data.getDia()){
                idade = idade + 1;
        }

        return idade;
```

Código limpo

```
public void exibirExemploDeFuncaoPequenaComCondicional(){
        Date dataNascimento = 19/10/2000
        Integer idade = calcularIdade(dataNascimento);

}
private Integer calcularIdade(Date data){
        Integer quantidadeAnos = retornaQuantidadeAnos(data);
        return validarIdadePorMesEDia(quantidadeAnos, data);
}

private Integer retornaQuantidadeAnos(Date data){
        return new Date().getAno() - data.getAno();
}

private Integer validarIdadePorMesEDia(Integer quantidadeAnos, Date data){
```

```
        if (new Date().getMes() == data.getMes() &&
                new Date().getDia() > data.getDia()){
                return quantidadeAnos + 1;
        }

        return quantidadeAnos;
}
```

Organize o seu código para que ele seja possível de ler de cima para baixo. Ou seja, os métodos principais estarão no topo da classe. Os métodos privados devem estar mais abaixo.

Por último, cuidado com os parâmetros de funções. O ideal é que não haja parâmetro e que você evite mais que dois parâmetros. Use o poder da orientação a objetos e encapsule os parâmetros em objetos com nomes significativos e bem coesos.

Comentários

Comentários são um mal necessários nas linguagens de programação. O ideal é que não houvesse comentários no código fonte, pois isto só explicita o quanto o seu código não está passando a mensagem que deveria passar de forma clara e objetiva. Por isto eu sugiro que nunca use comentários no seu código fonte. Pode ser polêmico mas pense bem, qual foi a mensagem do último comentário que você leu num código fonte? Era algo do tipo “não mexa nesta linha” ou era uma explicação básica de como o método funcionava?
Comentários mentem! Eles foram escritos para demonstrar um sentimento que o software não tem. O código cresce e evolui e estas mensagens fixas de comentário ficam fora de contexto. Por isto afirmo, antes de colocar um comentário, revise se você não pode transformar aquele trecho num método com nome significativo, que tal?

E nunca comente um código fonte achando que um dia vai precisar dele num futuro. Remova o código que não será usado. Reduza o peso do software. Você usa algum versionador de código (como o Git), por isso confie a este a responsabilidade de cuidar do histórico do código fonte. Caso ainda não esteja usando, providencie um versionador o mais rápido possível (como o GitHub, GitLab e outros que são gratuitos para uso público).

Formatação e code styles

Figura 16

Você já viu algo como a figura acima? Pesquisando na internet sobre “imagens de toc” você encontrará muitas referências semelhantes a esta imagem. E qual é o sentimento que você tem ao ver a foto? Dá vontade de ir lá arrumar? Ou nada acontece?

Quando eu vejo algo assim, fico perguntando quanto tempo a mais levaria para o prestador deste serviço ter deixado a tampa do bueiro como era para estar. Imagino que seja algo menor que 1 minuto. E quanto tempo leva para deixarmos o código fonte formatado e organizado?

Será que estamos preocupado em deixar o código melhor do que estava antes ?(*ver lema do escoteiro no capítulo 1*).

Pequenos cuidados com a formatação do código irão facilitar a vida de outros desenvolvedores quando estes tiverem que dar manutenção no código.

O objetivo da formatação é facilitar a comunicação entre o time. Ninguém no time deve atrapalhar o outro e sim buscar ajudar. Tendo um código formatado você irá ser um excelente ajudador no time.

Comece tendo classes de até 200 linhas. Isto irá ajudar em manter a coesão e você não percorrerá um grande arquivo para saber o que ele faz (lembrando que o nome significativo irá poupar tempo de abrir a classe para entender o que ela faz).

Uma classe deve ser semelhante a uma redação para o vestibular. Você o lê de cima para baixo. O título da redação deve passar a mensagem sobre o que será a redação. A seguir teremos a introdução, depois o desenvolvimento e por último a conclusão.

No seu código fonte, o título é o nome da classe. A introdução são declarações de constante, as variáveis globais e construtores. Logo em seguida devemos ter os métodos principais e após eles os métodos privados. Seguindo esta lógica, a leitura do código ficará bem organizada e você só precisará ler os métodos privados da classe caso realmente precise dar manutenção em um deles. É normal num código bem organizado só lermos os principais métodos pois eles já mostram a sequência de passos para algo.

```
public void imprimirRelatorio(){
        verificarImpressora();
        imprimir();
        contabilizarImpressoesPorUsuario("wagner@email.com,br")
}

private Boolean verificarImpressora(){
        -
        -
        -
        -

}
private  void imprimir(){
        -
        -
        -
        -

}

private  void contabilizarImpressoesPorUsuario(String nome){
        -
        -
        -
        -

}
```

Além da disposição e organização dos métodos numa classe, devemos ter atenção a como está a disposição de cada código no arquivo (indentação). Geralmente as IDE (softwares que usamos codificar), já tem seus próprio guia de estilo (style guide) e é importante que todas as pessoas desenvolvedoras sigam este guia para que não haja divergência na comunicação.

O Google mantém um projeto no GitHub chamado Style Guide no qual são compartilhadas as convenções para diversas linguagens de programação, dentre elas:

- C++ ,
- Objective-C ,
- Java ,
- Python ,
- R ,
- Shell ,
- HTML/CSS ,
- JavaScript ,
- AngularJS ,
- Common Lisp .

As convenções são referente a como devem ser os nomes de classes e métodos, uso de caracteres especiais, uso de blocos com chaves {, uso de espaço ou tabs spaces além de diversas outras formatações. Para saber mais deste guia de estilo acesse https://github.com/google/styleguide

Tratamento de erros

Tratar erros e exceções garantem que você ainda mantém o controle da aplicação quando algo inesperado acontece. Como exemplo podemos citar o retorno nulo quando não era esperado isto. Ou então uma divisão por zero, a qual não pode existir. Tratar estas situações e retornar uma mensagem amigável ao usuário é uma função de um programador profissional.

Em linguagens modernas é possível utilizar de recursos como blocos de exceção (algo semelhante ao try - catch - finally do Java). Entenda o propósito de cada bloco de exceção e use-os de maneira organizada.

Sempre que puder, crie uma exceção personalizada e com mensagem clara do erro, de preferência falando do contexto do erro. Não dependa unicamente do que a linguagem de programação fornece para você.

Coesão e acoplamento

Ao programar, você deve buscar produzir códigos altamente coesos e pouco acoplados.

Mas o que isto quer dizer?

Coesão fala de algo que:
- DEVE FAZER UMA ÚNICA COISA
- DEVE FAZÊ-LA BEM
- DEVE FAZER APENAS ELA

Um método chamado validarCpfValido, deve únicamente fazer a verificação dos dígitos verificadores do CPF e informar se são válidos ou não. Este método não é para validar outros documentos. Este método deve fazer uma única coisa é bem feito.
Quando temos a coesão de classes e métodos, provavelmente teremos baixo acoplamento. O acoplamento é o quanto uma classe se relaciona com outras. Quando temos uma classe que todas as outras invocam ela, temos alto acoplamento e isso é ruim, já visto que uma manutenção nesta classe irá afetar todas as outras que estão acopladas a ela.

Busque sempre fazer códigos coesos e que se relacionem com outras classes que realmente precisam de acoplamento. **Busque alta coesão e baixo acoplamento**.

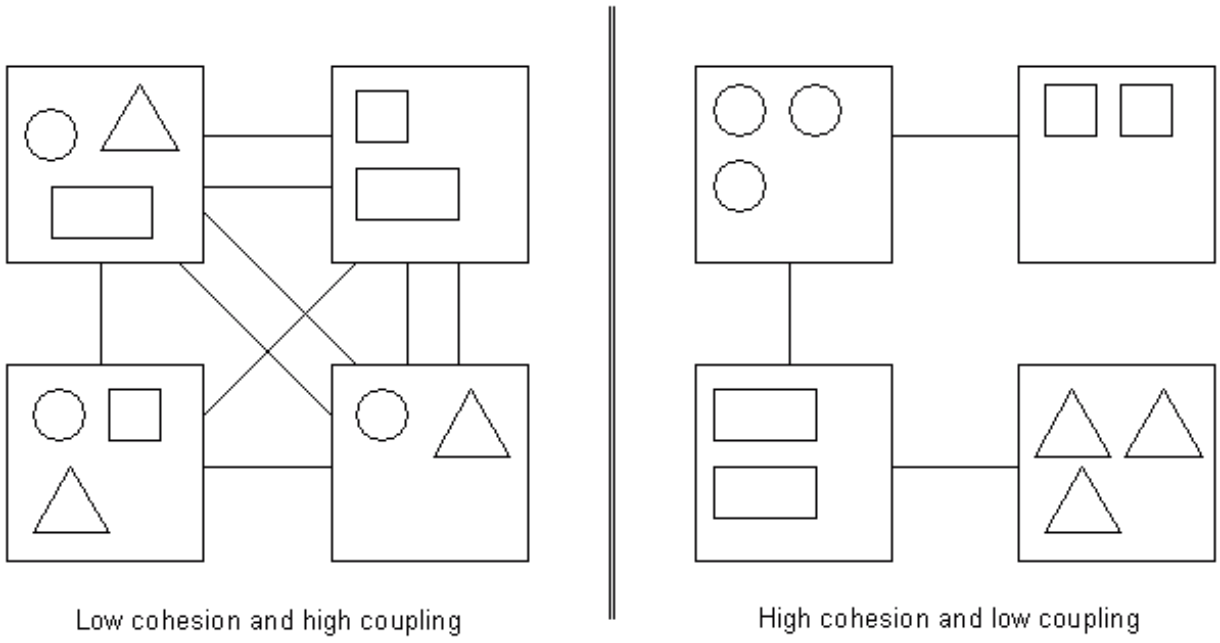

Figura 17

Fechamento

Pois bem, vimos sobre o que é um código bom e ruim e os princípios que regem um bom código. Responda e se comprometa com as atividades abaixo:

1) Quais práticas o time que você está atuando já está fazendo? Pode consultar os itens acima

_____

_____,

2) Quais práticas você pode começar a adotar (on boarding)?

9 regras para que seu código fonte seja melhor!

Figura 18

Object Calisthenics

Em toda olimpíada, a última competição esportiva é a maratona que é uma homenagem à antiga lenda grega de um soldado que percorreu a distância de aproximadamente 40 km

entre as cidades de Maratona até Atenas para informar sobre a vitória dos exércitos gregos. Nos dias atuais, uma maratona propõe aos seus participantes o percurso de 42km e 195m e percorrer esta distância exige preparo, treino e dedicação. Exige o hábito de se exercitar com frequência e que começou com distâncias bem menores que 42 km.

Será que poderia existir algo assim na programação? Algo que podemos nos exercitar pensando em se tornar melhores profissionais? Sim, existe e se chamam object calisthenics.

A palavra calisthenics tem suas raízes do grego e remete a exercícios. Estes exercícios foram definidos por Jeff Bay no capítulo 6 do livro The ThoughtWorks Anthology: Essays on Software Technology and Innovation (Bay(2008)).

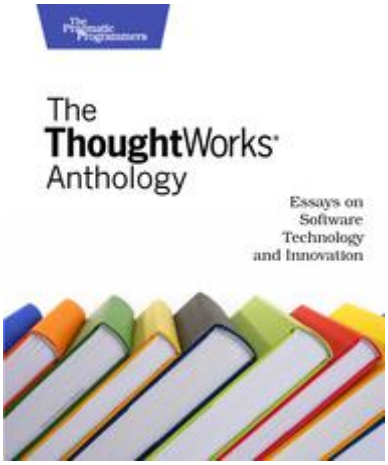

Figura 19

Vale a pena lembrar que este capítulo deve ser usado junto com o anterior. Ambos irão tornar você um programador profissional e com código mais limpo.

Vamos as nove regras definidas por Jeff Bay:

- Um único nível de indentação por método
- Nunca use a palavra reservada ELSE
- Encapsule todos os tipos primitivos e Strings
- Encapsule suas coleções de dados
- Um ponto por linha
- Nunca use abreviação
- Mantenha todas as entidades pequenas
- Nenhuma classe deve ter mais de duas variáveis de instâncias
- Não use getters/setters/properties

Um único nível de indentação por método

Nos anos 90, um jogo ganhou fama em vários fliperamas e em consoles 32 bits. Era um jogo de luta de rua, denominado Street Fighter e em especial dois lutadores tinham a preferência dos jogadores. Eram eles o jogador Ken e Ryu. Ambos tinham golpes semelhantes e um deles era o *hadouken*, que era uma espécie de bola e poder que saia das mãos deles. Esta bola ao encontrar um obstáculo causava destruição e se fosse um adversário causava um dano razoável. Mas o que tem a ver esta contextualização com esta regra. A imagem abaixo vai explicar:

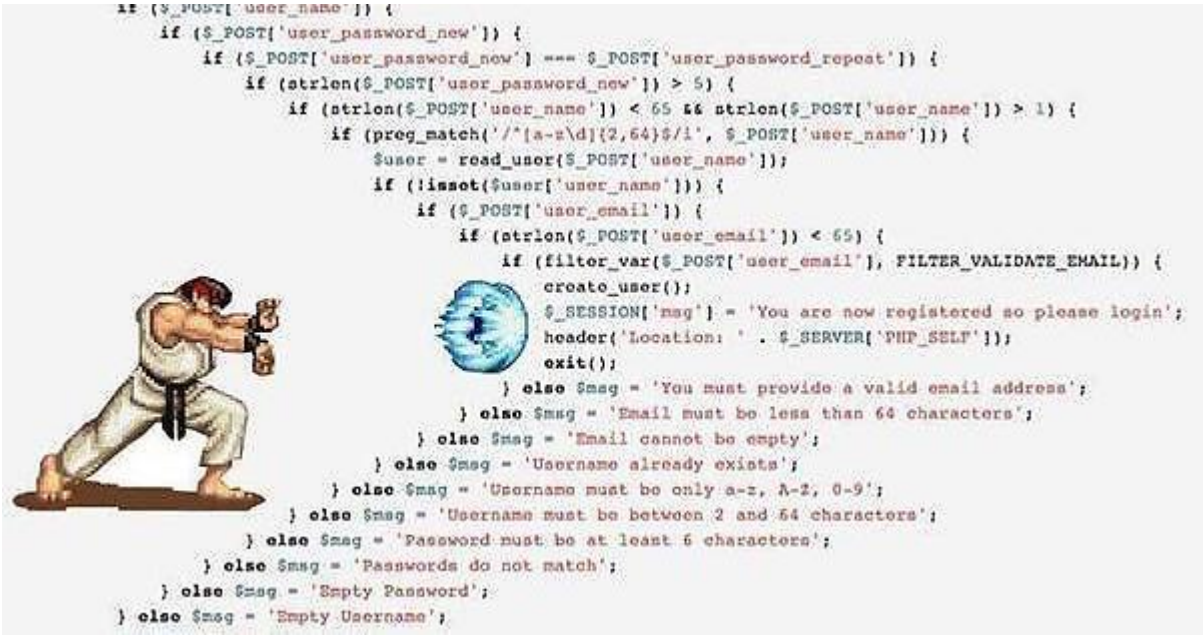

Figura 20

Em nosso código fonte, é comum termos situações condicionais que vão nos levando a outras condições. Mas se programarmos da forma como a imagem acima retrata, estaremos afetando a qualidade do código fonte em :

- Legibilidade, afinal é muito difícil entender o que este conjunto de código faz
- Testes, pois o cenário de probabilidade é tão grande que é quase impossível um teste unitário conseguir garantir que todos os cenários possíveis sejam cobertos e validados.
- Complexidade ciclomática: este conceito refere-se a quantas níveis de código um código precisa passar. Quanto maior a indentação, maior a complexidade do código.
- Por isto não tenha mais que um nível de indentação no código fonte. Use métodos pequenos e coeso. Refatore seu código com frequência e use os benefícios das IDE (programas que você utiliza para escrever seu código fonte) que já podem criar extrair códigos fontes em pequenos métodos.

Nunca use a palavra reservada ELSE

Sim! É possível programar sem a palavra ELSE. Toda vez que você usa ELSE, você adiciona uma complexidade ao código que poderia ser evitada. Fomos ensinados a sempre criar a estrutura de condição (IF) para a validação que ser positiva, então que tal mudar este pensamento e logo no ínicio do código fonte, validar o que realmente precisa ser validado e

já retornar para o trecho de código anterior? Chamamos esta regra de RETURN FIRST (retorne primeiro). O exemplo abaixo irá evidenciar o benefício desta prática:

Figura 21

Encapsule todos os tipos primitivos e Strings

Tipos como Integer, String, int, Double, Float, char e outros, são comuns nas diversas linguagens de programação e não há nada de errado em usa-los. Mas que tal tornar o código mais legível usando o benefício da orientação a objetos a seu favor. Por isto encapsule todos os tipos primitivos e String.
Quando falo encapsular, estou dizendo que iremos criar uma classe com um nome significativo e que dentro desta classe iremos colocar o que representa aquele conjunto de dados. Por exemplo:

```
public class Pessoa(){
        private Integer ddd;
        private Integer numeroTelefone;
}
```

Que tal mudarmos a classe para:

```
public class Pessoa(){
        private Telefone telefone;
}

public class Telefone(){
        private Integer ddd;
        private Integer numero;
}
```

Com isto manteremos o nosso código legível e reaproveitado, pois outras classes poderão usar telefone, reduzindo a duplicidade de código fonte.

Encapsule suas coleções de dados

A mesma regra acima de encapsulamento serve para coleções de dados como arrays, vetores, matrizes e collection. Se você encapsular suas coleções, a leitura do código fonte será otimizada.

```
public class Pessoa(){
        private List<Endereco> endereco;
}

public class Endereco(){
        private String tipoLogradouro;
        private String logradouro;
        private Cep cep;
        private Cidade cidade;
}
```

Que tal:

```
public class Pessoa(){
        private EnderecosPessoas enderecos;
}

public class EnderecosPessoas(){
        private List<Endereco> endereco;
}

public class Endereco(){
        private String tipoLogradouro;
        private String logradouro;
        private Cep cep;
        private Cidade cidade;
}
```

Um ponto por linha

Existe um princípio de qualidade em código fonte que diz que seu código deve falar e não perguntar (tell, dont ask). E este princípio está ligado diretamente a esta regra. Toda vez que encontro um uma chamada com vários pontos é sinal que estou ferindo este princípio e esta regra. Mas não fique preocupado que é só você quem comete este erro, muitas das linguagens de programação tem disto. Um exemplo é o próprio Java que tem o famoso System.out.println(“mensagem”). O correto era termos um System.exibirTexto(“mensagem”).

Nunca use abreviação

Este item é semelhante a item nomes significativos do capítulo anterior. Mas não custa reforçar que não devemos:

- Ter abreviações em métodos, classes e variáveis
- Ter nomenclaturas que não sejam comuns a todos
- Ter variáveis que tenham uma única letra.

Na duvida, peça para alguem ler o nome que você deu ao método, variável ou classe e busque feedback se aquilo é ou não uma abreviação e que possa atrapalhar a legibilidade do código fonte.

Mantenha todas as entidades pequenas

No capítulo anterior lemos sobre coesão e acoplamento. A coesão garante que faremos pequenas coisas e que elas serão muito bem implementadas, gerando assim facilidade para testes unitários, aumento da legibilidade, baixa complexidade ciclomática e pouca dificuldade em eventuais manutenções. As linguagens de programação possui uma convenção de qual é a quantidade de linhas para dentro de um método e quantas linhas uma classe pode ter. São elas:

Em Java:

- Número de linhas de um método:75
- Número de linhas de uma classe:200

Em .NET:

- Número de linhas de um método:80
- Número de linhas de uma classe:1000

Em PHP:

- Número de linhas de um método:150
- Número de linhas de uma classe:1000

Em Kotlin:

- Número de linhas de um método:100
- Número de linhas de uma classe:1000

Em Swift:

- Número de linhas de um método:100
- Número de linhas de uma classe: 1000

Em Javascript:

- Número de linhas de um método:200
- Número de linhas de uma classe:1000

Nenhuma classe deve ter mais de duas variáveis de instâncias

Este é com certeza a regra mais difícil de ser implementada, pois seguindo ela você poderá ter muitas classes no código fonte sendo todas coesas e fazendo poucas coisas mas todas bem feitas. Ao mesmo tempo que tudo está coeso, você terá muitas classes no sistema, gerando dificuldade em criar nomes e achar o que cada um faz. Por isto use esta regra conforme a maturidade do seu time cresce. Não adianta você seguir estas regras se o seu time não conhece-as. Influencie eles a entender que classes menores serão mais fáceis de serem testadas e reaproveitadas.

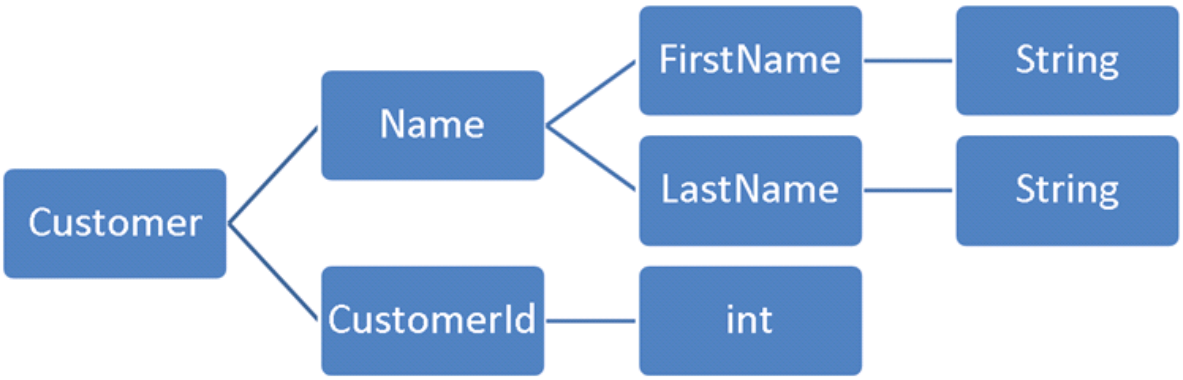

Figura 22 - Lidando com encapsulamento para a regra de classes com mais de uma instância de variavel.

Perceba que a classe Customer é composta de dois atributos (Name que é uma classe) e CustomerID que é um sequence do banco.
Na classe Name, ela possui outros dois atributos que representam duas refeições (*fisrt name e lastname*). O objetivo desta regra é manter classes extremamente coesas

Não use getters/setters/properties

Por fim, temos a regra que diz para não usarmos métodos do tipo get e set. Para quem programa em Java, já deve ter ouvido falar de classes POJO (Plain object Java ore.as.as.as.) . Estas classes POJO são arquivos com os atributos da classe de forma privada e seus métodos assessores (get e set public). Mas quem disse que todos os atributos deveriam ter métodos assessores? E quem garante que estes métodos getters e setters estão legíveis a partir de seus nomes?
Esta regra diz que você deve criar os métodos a medida que realmente usar. Isto não irá expor coisas da classe de forma irresponsável e desnecessária. Existe um artigo que fala muito bem desta regra e pode ser conferido no link abaixo:

SITE CAELUM – como NÃO USAR GETTER E SETTER - http://blog.caelum.com.br/nao-aprender-oo-getters-e-setters/
Para saber mais do assunto, consulte o material abaixo:

Dica de vídeos e leitura sobre o tema:

● Matheus Gontijo - https://www.infoq.com/br/presentations/codigo-limpo-e-object-calisthenics
● Thoughtworks Anthology - https://pragprog.com/titles/twa/thoughtworks-anthology
● Jeff Bay - https://bolcom.github.io/student-dojo/legacy-code/DevelopersAnonymous-ObjectCalisthenics.pdf
● Guilherme Blanco - https://www.youtube.com/watch?time_continue=1&v=u-w4eULRrr0

Fechamento

Responda as perguntas abaixo sobre o nosso aprendizado

1) Numa organização quem são as pessoas/cargos que precisam mudar? Justifique a sua resposta

_____

_____,

_____

_____

2) Qual das regras você consegue colocar de prática amanhã?

_____

_____,

_____

_____

Como analisar o seu código fonte a partir de validações dos padrões de mercado?

Dívida Técnica

O que é

Um dia, Ward Cunningham , que é um dos criados da plataforma Wiki, cunhou este termo dívida técnica (debt technical em inglês), quando mencionou toda e qualquer escolha que é feita em código fonte e que não era a melhor forma de ser implementada. Diferente do bug (que é um erro no código), um código fonte com dívida técnica funciona, mas não usando a melhor maneira disponível.

Ward usou este termo (dívida) pois ele faz uma metáfora com as questões financeiras. Quando você faz uma compra num cartão de crédito (por exemplo), você está consumindo algo agora e pagando em até 45 dias a partir da data de hoje. E caso você não faça o pagamento da fatura, juros abusivos irão começar a ser acrescidos da dívida inicial. Chegando ao ponto de você acumular tanta dívida que é quase impossível fazer o pagamento da mesma

Confira o vídeo do Ward em: https://www.youtube.com/watch?v=Jp5japiHAs4

Em software, este pagamento da dívida técnica é algo semelhante. Existem projetos que possuem tantas dívidas que ficam quase impossíveis de serem pagas, gerando assim a necessidade da escrita de um novo sistema. Abaixo podemos ver um gráfico de como a dívida técnica se comporta durante o período o projeto de software:

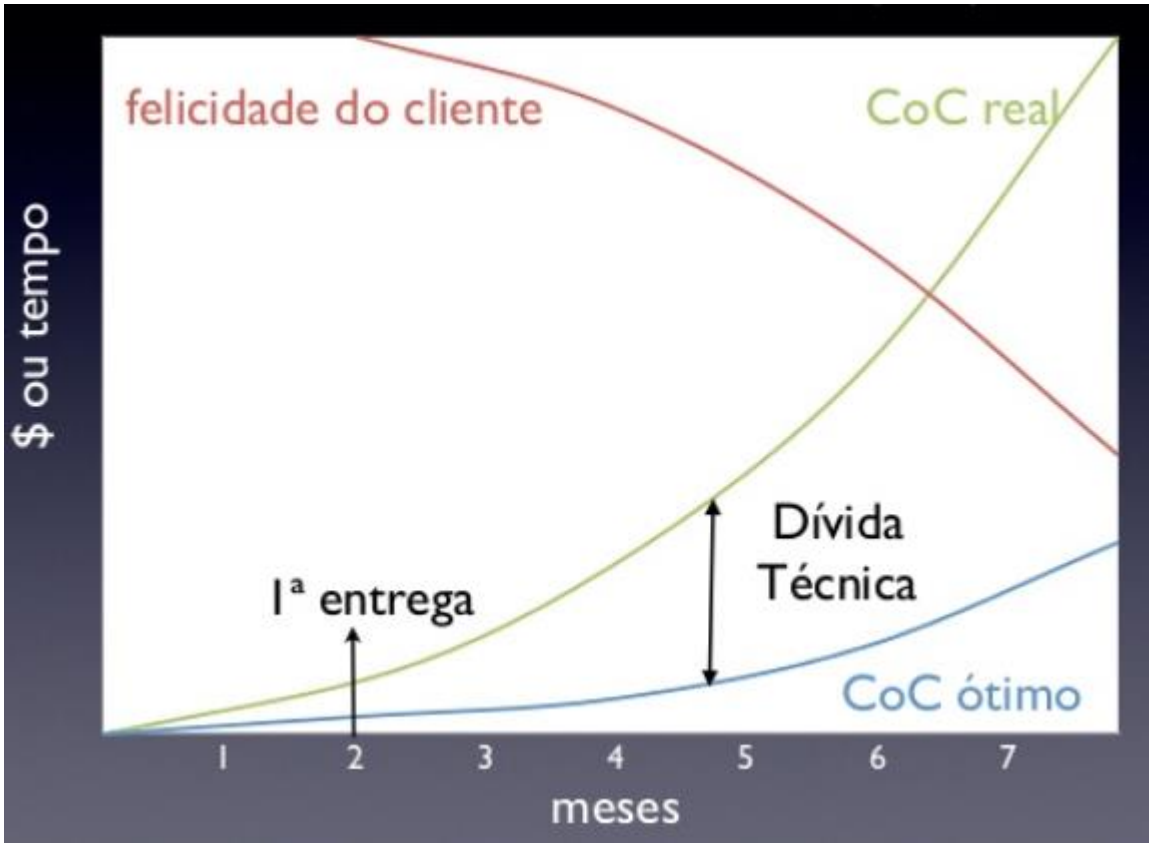

Figura 23 - o custo da dívida técnica

Numa primeira entrega de software a dívida técnica, tende a ser bem baixa, mas se ela existir e não for sendo paga o mais rápido possível, será difícil retomar o pagamento nas demais entregas.

Todo projeto de software deveria começar já tendo uma maneira de medir a dívida técnica do projeto. Vejamos abaixo os sintomas que podem ser um sinal que seu projeto possui dívida técnica e você deveria dar atenção a isto:

Sintomas

- Projetos sem cobertura de testes unitários
- Projetos que são difíceis de darem manutenção
- Projetos onde ninguém do time se sente confortável com atualizações
- Projetos engessados onde cada alteração é bem custosa, tanto de tempo como de pessoas dispostas ao desafio
- Lentidão para executar cenário de testes
- Lentidão para gerar o build
- Entre outras

Para medirmos tudo isto,utilizamos de softwares para este cálculo. O software mais difundido atualmente é o SonarQube. Iremos falar mais dele no próximo tópico que é sobre análise estática de código.

E como pagar a dívida

A primeira coisa a ser feita é usar uma ferramenta de análise estática de código fonte (ver abaixo). Ela irá contabilizar qual é o tamanho real da sua dívida e a partir disto você poderá tomar planos de ação; Da mesma forma que você foi contraindo dívida dia após dia, o pagamento desta deverá ser feito com disciplina e empenho, mas sem atrapalhar as tarefas do dia a dia. Sou contra termos um momento (sprint de uma semana, por exemplo) para pagar dívida técnica. Ela deveria ser feita em todo o momento que eu estiver fazendo uma alteração na classes.
Projetos sem dívida técnica beiram a utopia, por isso, cuidado com o desejo de ter 0 (zero) de dívida técnica,mas se conseguir é ótimo. Logo que você identificou a quantidade de dívida técnica, a cada nova versão, não pode ser realizado o acréscimo de divididas em nova s linhas de código. Com esta pequena atitude, você vai transformando o seu código fonte em algo melhor. Em poucos meses é possível ver melhorias e logo o código ficará legível e a dívida técnica reduzindo de forma grandiosa

Análise estática de código

Todo código fonte pode (e deveria) ser submetido a uma análise estática, que consiste em um software especifico ficar analisando o código fonte a partir de um conjunto de regras pré estabelecidas e que seguem uma convenção definida.
O software mais conhecido para esta análise estática é o Sonarqube que consegue verificar mais de 20 linguagens diferentes de programação.
O Sonarqube, consegue a partir de suas regras previamente programadas analisar possíveis más práticas (*code smells*),vulnerabilidades (itens que falam de segurança da aplicação) e eventuais códigos maus programados e que causarão bug a medida que forem executados estes trechos de código. Além disso ele consegue evidenciar duplicidade de código fonte.

O conjunto das más práticas irá trazer o indicador de dívida técnica do projeto. O *sonarQube*, já cria até as tarefas (caso configurado) para que as cada má prática seja corrigida. Por último o *sonarQube* faz a análise da cobertura de teste que é:

Cobertura de testes

Todo software que se preze deve ter o uso de testes unitários; Um desenvolvedor profissional faz os testes unitários como uma cláusula de garantia para si, pois o teste unitário dá feedback instantâneo de com o código fonte está se comportando. O testes unitário garante coesão e simplicidade.
Caso não haja testes unitários, o software fica a margem de uma possibilidade de estar com bugs e incentiva o uso de más práticas, ferindo todo este livro está pregando quanto a qualidade de código fonte.
Faz testes pode retardar uma entrega, mas não retarda um bug voltar .
Por isto faça testes e saiba que as ferramentas analistas de código fonte irão ajudar você a olha para qual é a classe mais importante do software e como você deve fazer para ter testes unitários;

Fechamento

Responda as perguntas abaixo sobre o nosso aprendizado

1) Qual a importância de ter uma etapa de análise estática de código fonte?

_____

_____,

_____

_____

2) Explique o por que não é débito técnico e sim dívida técnica?

_____

_____,

_____

_____

3) Como pagar dívida técnica?

_____

_____,

_____

Referências Imagens:

CAPA - https://www.pexels.com/photo/desk-flatlay-items-keyboard-399160/
Figura 2 : https://www.pexels.com/photo/adult-alone-anxious-black-and-white-568027/
Figura 3 :https://www.synthesis.co.za/wp-content/uploads/bfi_thumb/AgileArchitecture1-1rl3zo5ig9008zq8zgwie6cn9vnbl7um76y52b72jvbo.jpeg
Figura 4: https://www.pexels.com/photo/person-making-clay-pot-162574/
Figura 5: https://meritbadge.org/wiki/images/3/3f/Boy_Scout_Sign.jpg
Figura 6 - https://keepitsimplestupid.dk/wp-content/uploads/2017/08/KISS_logo_C.png
Figura 7 - https://ardalis.com/wp-content/files/media/image/WindowsLiveWriter/DRYDontRepeatYourselfMotivator_BA85/dontrepeatyourself_motivator_2.jpg
Figura 8 - http://farm8.staticflickr.com/7135/7021453805_b2b981d7c0.jpg
Figura 9 - https://deviq.com/wp-content/uploads/2014/11/Separation-of-Concerns-Feb-2013.png
Figura 10 - https://upload.wikimedia.org/wikipedia/commons/d/de/Michelangelo_Caravaggio_065.jpg
Figura 11 - https://www.obo65.com.br/product-page/caneca-ame-o-problema
Figura 12 - https://cdn.pensador.com/img/frase/so/cr/socrates_so_sei_que_nada_sei_lk1nd82.jpg
Figura 13 - https://i.pinimg.com/236x/c2/d0/56/c2d05618cdb02520fd37e7b996376542--grumpy-cat-meme-cat-memes.jpg
Figura 14 - https://jaimemarlonsilva.files.wordpress.com/2011/08/negociacao61.jpg
Figura 15 - http://www.luiztools.com.br/wp-content/uploads/2016/09/pair.jpg
Figura 16 - https://www.psicologiamsn.com/wp-content/uploads/2014/04/toc-5.jpg
Figura 17 - http://sce2.umkc.edu/BIT/burrise/pl/design/cohesion-coupling-abstract.gif
Figura 18 - https://www.pexels.com/photo/man-running-on-black-asphalt-road-1555354/
Figura 19 - https://imagery.pragprog.com/products/105/twa_xlargecover.jpg?1298589765
Figura 20 - https://i.imgur.com/FT1vYUY.png?fb
Figura 21 - https://image.slidesharecdn.com/object-calisthenics-160423152436/95/object-calisthenics-13-638.jpg?cb=1465051437
Figura 22 - https://williamdurand.fr/2013/06/03/object-calisthenics/
Figura 23 - http://www.slideshare.net/alexandrefreire/divida-tecnica

Referências Bibliográficas:

[MANIFESTO-AGIL] - http://www.manifestoagil.com.br/

[Martin(2011)] MARTIN, Robert C. Código Limpo: Habilidades práticas do Agile Software, Alta Books, 2011

[Martin(2012)] MARTIN, Robert C. O Codificador Limpo: um código de conduta para programadores profissionais, Alta Books, 2012

TELES, Vinícius Manhães. Extreme Programming: aprenda como encantar seus usuários desenvolvendo software com agilidade e alta qualidade. 1. Ed. São Paulo: Novatec, 2004.

[Bay(2008)] BAY, J. 2008. The ThoughtWorks Anthology: Essays on Software Technology and Innovation, Chapter Chapter 6: Object calisthenics. O'Reilly Series. Dallas, TX: Pragmatic Bookshelf.