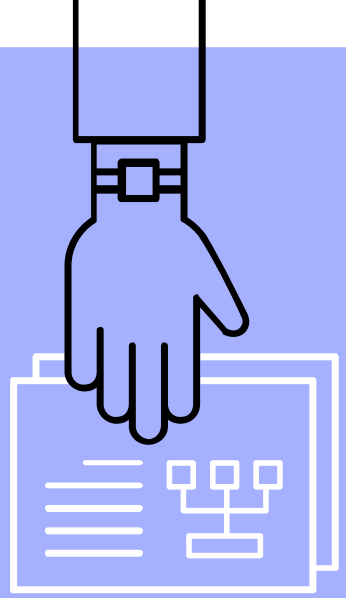
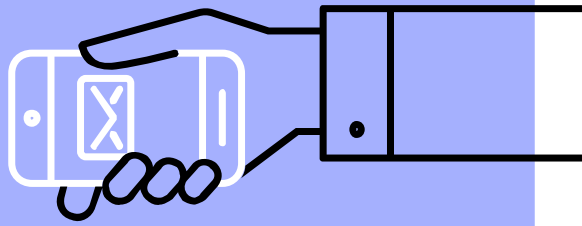
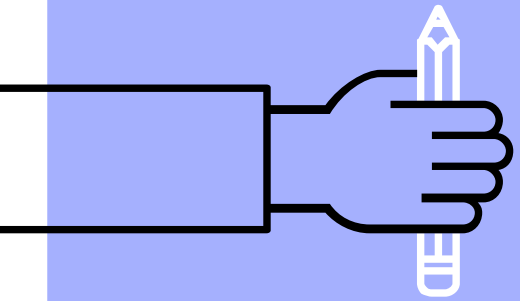
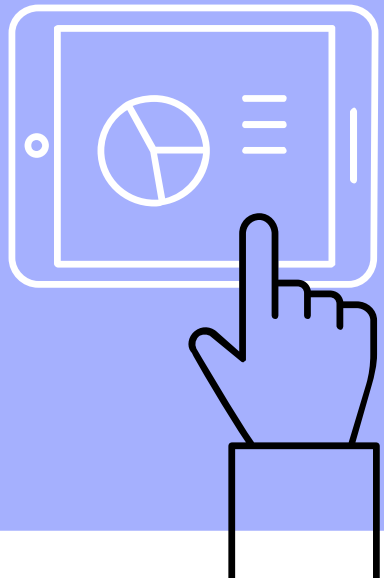




Princípios de Design de Software (DRY, YAGNI, KISS)

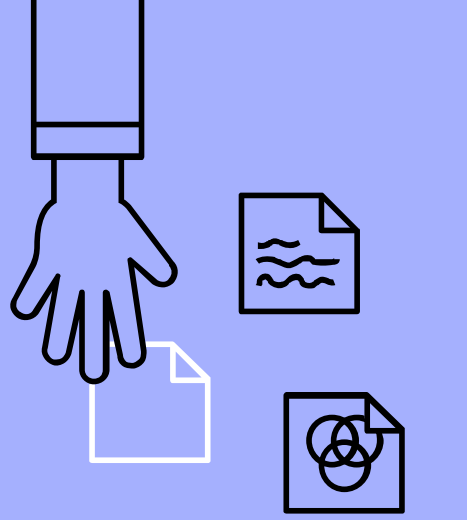


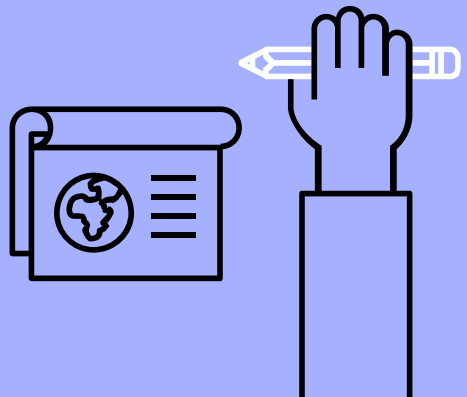


Olá!

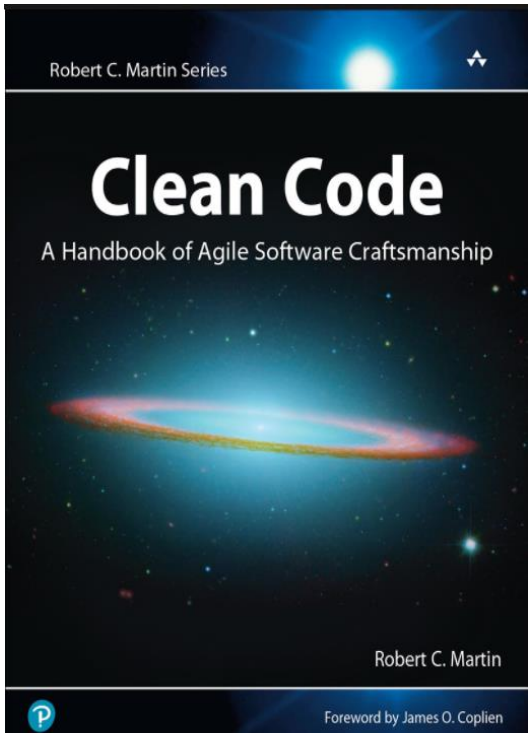
Eu sou Willian Brito

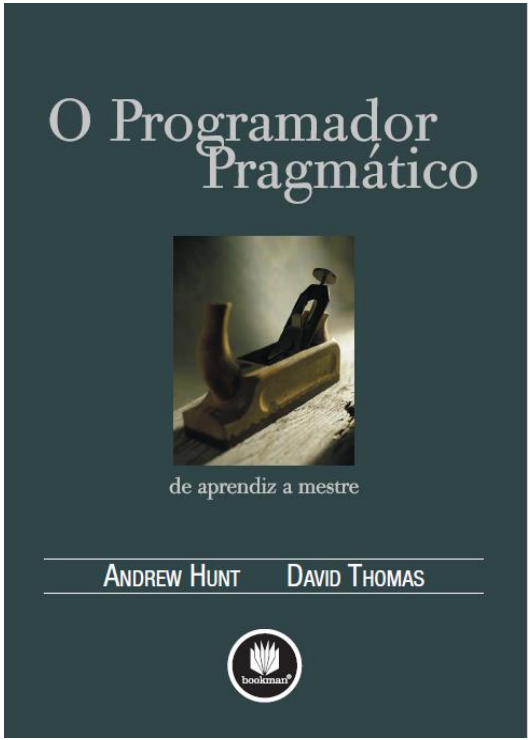
- Desenvolvedor FullStack na Msystem Software
- Formado em Analise e Desenvolvimento de Sistemas.
- Pós Graduado em Segurança Cibernética.
- Certificação SYCP (Solyd Certified Pentester) v2018





Este conteúdo é baseado nessas obras:





“

DRY, KISS e YAGNI são três princípios que todo desenvolvedor, **não importa o nível que esteja**, deveria ao menos conhecer..

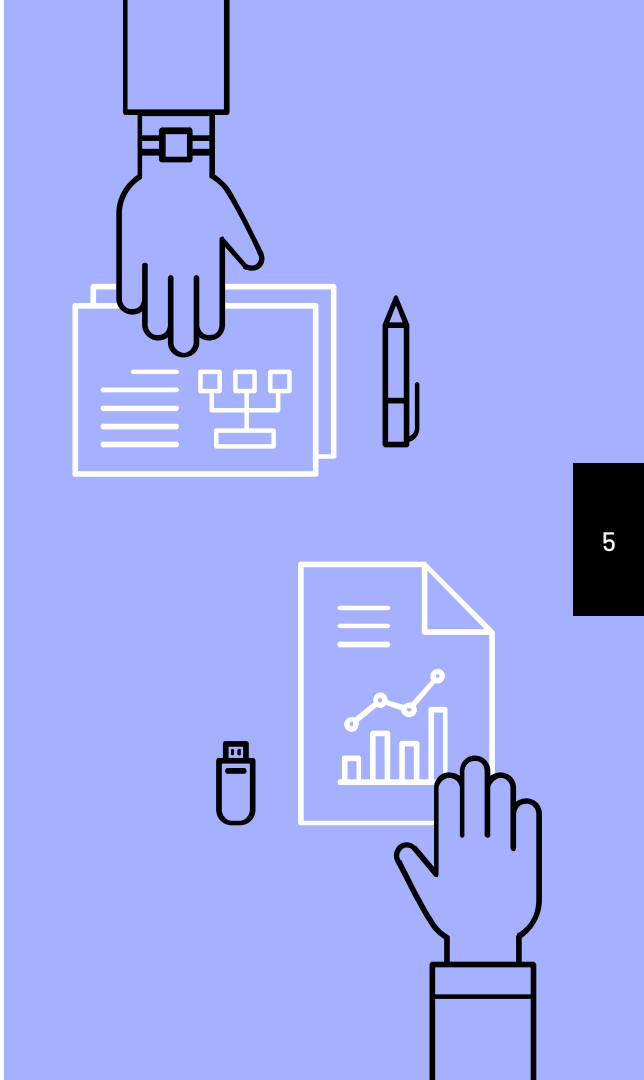
Princípios de Design de Software.

Desenvolvedores têm desafios todos os dias e, é sabido que independente da solução e linguagem utilizada, é possível ter o mesmo resultado de muitas formas diferentes.

É comprovado que juntar vários programadores diferentes e pedir para que eles escrevam algo esperando determinado resultado, certamente teremos soluções diversas.

Indiferente dos requisitos necessários para a solução, o importante para nós desenvolvedores mantermos nosso código escalável é escrevê-lo de forma que outros desenvolvedores consigam entender com facilidade e, evolui-lo se necessário.

Passamos boa parte do tempo lendo código e, é fundamental escrevê-lo de forma que qualquer outro desenvolvedor consiga entendê-lo de forma rápida.

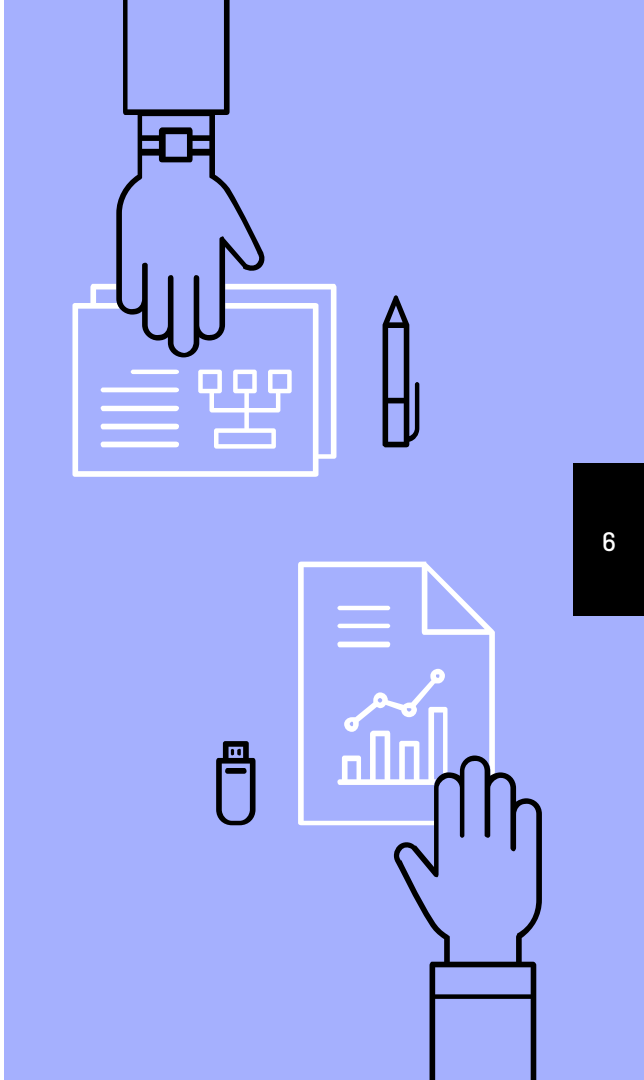


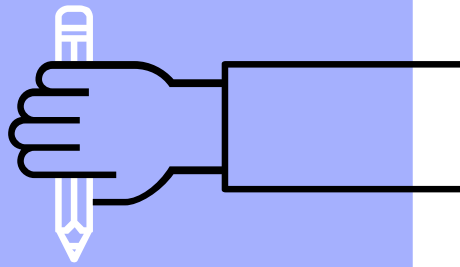
Princípios de Design de Software.

Com o propósito de ter sistemas mais escaláveis e de fácil manutenção, ao passar dos anos, alguns gurus da programação definiram alguns padrões e boas práticas para dar uma direção à outros desenvolvedores.

Resolvi falar de três princípios básicos de Design de Software (DRY, YAGNI, KISS) que muitas pessoas já utilizam sem saber ao certo que eles são usados como princípios para qualquer linguagem de programação que você utilize.

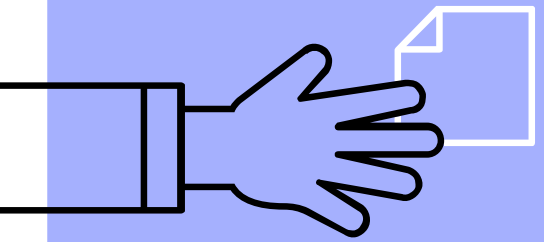
Caso você não conheça, comece a pensar com carinho na utilização deles.





1. DRY (Don’t Repeat Yourself)

Não se Repita

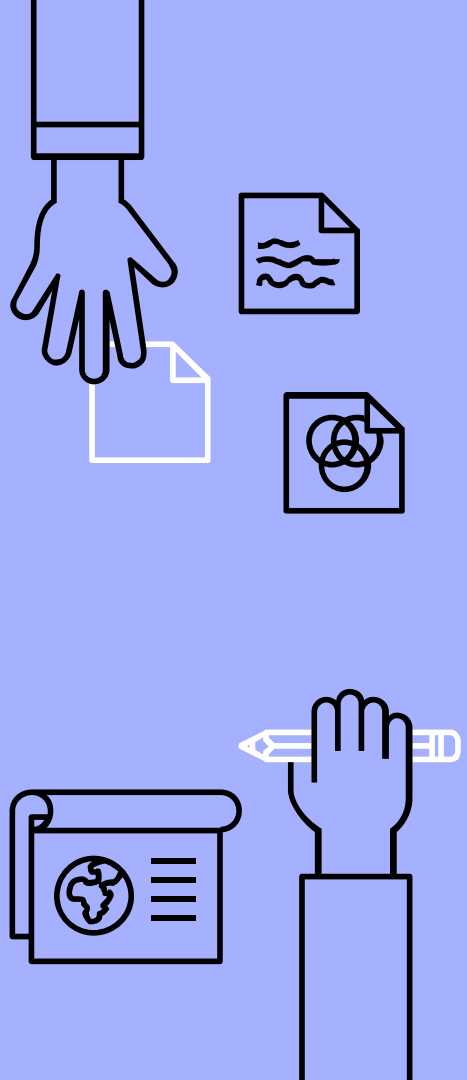


DRY – Não se Repita

O **DRY** é um conceito presente na programação computacional que propõe que a cada porção de conhecimento em um software deve possuir uma representação única, livre de ambiguidades em todo o software . Esta expressão foi apresentada pelos autores **Andy Hunt** e **Dave Thomas** no seu livro **The Pragmatic Programmer**.

Você já deve ter se deparado com códigos idênticos espalhados pelo mesmo sistema e se precisarmos fazer a correção de uma função que, por algum motivo, se repete em diversos lugares, vamos ter que alterar o mesmo código N vezes, afim de manter sua compatibilidade. Isso é horrível quando pensamos em manutenção de código, mais um motivo para evitarmos a repetição desnecessária de código ou regra de negócio!

Este conceito implica em não repetir códigos que tenham a mesma funcionalidade, ou seja, se a regra é responsável por resolver o mesmo problema, o DRY orienta a arrumar uma forma de manter o comportamento em um único local, uma ótima maneira de fazer isso é utilizar **abstração** que é um dos pilares na **Orientação a Objetos**. O objetivo principal aqui é de ajudar na manutenção e evolução do software.

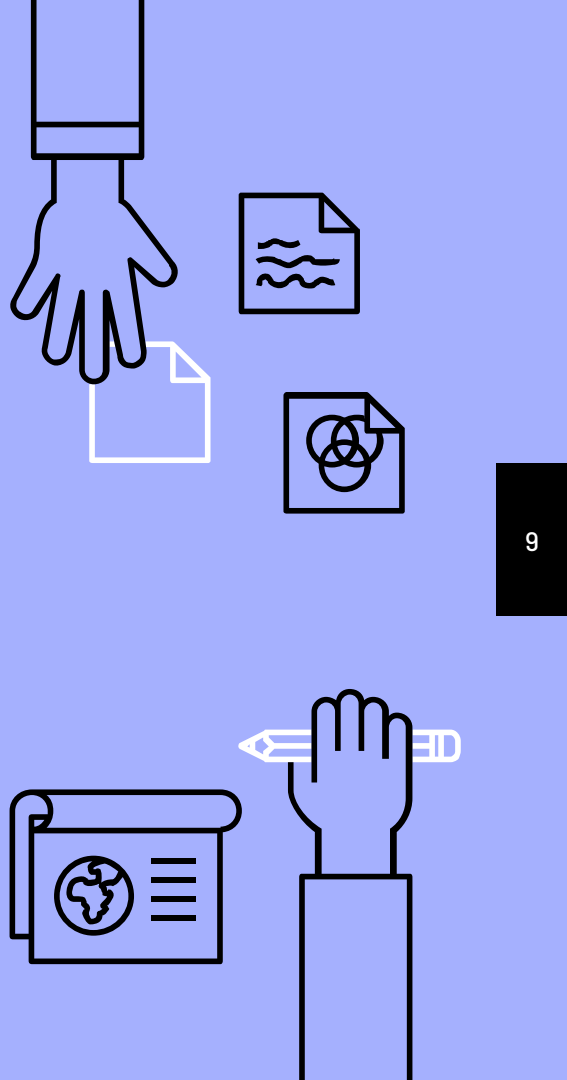


DRY – Não se Repita

Agora, iremos entrar um pouco mais no contexto da duplicação. Você já parou pra pensar, por exemplo, o motivo que leva um desenvolvedor a duplicar o código?

Quando fazemos a modelagem das classes, utilizamos bastante o conceito de **abstração** da Orientação a Objetos. Através dela, é possível identificar a hierarquia de classes, heranças, dependências e os métodos que podem ser polimórficos. Logo, se um determinado código está duplicado no software, pode-se afirmar que houve uma falha na modelagem, ou melhor, na abstração do projeto.

Muitas vezes, tudo se inicia com o tradicional "Copiar e Colar". Ao implementar uma nova funcionalidade que converte um relatório para PDF, por exemplo, o desenvolvedor pensa consigo: "*Oras, se eu preciso converter um relatório para PDF e já existe um método pronto pra isso, vou usá-lo!*". Porém, ao invés de utilizar o mesmo método, o cidadão cria **outro** método e copia o código!!!

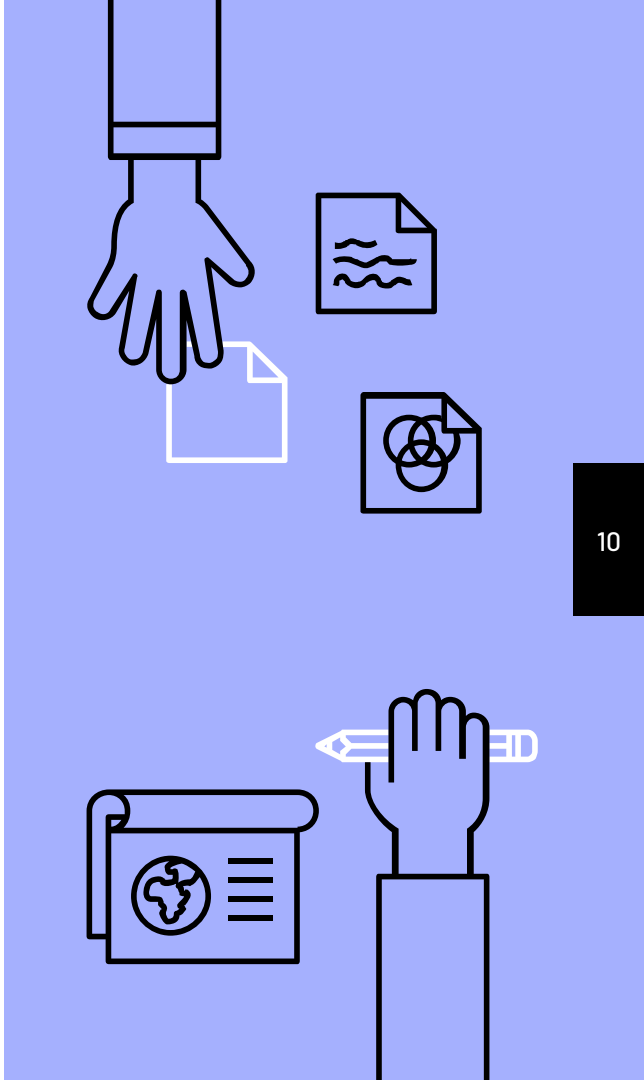


DRY – Não se Repita

E por que os desenvolvedores fazem isso?

Há vários motivos. Talvez porque seja mais rápido, embora o desenvolvedor não pense na dificuldade de manutenção que isso irá resultar.

Outro motivo é a "separação de preocupações": vamos supor que o método de conversão para PDF esteja na classe "RelatorioPedido" e seja necessário utilizar essa mesma funcionalidade na emissão de orçamentos. Se fizermos referência da classe "RelatorioPedido" dentro da classe "EmissaoOrcamentos", vai ficar sem sentido, concorda? São escopos diferentes e não devem ser confundidos. Além disso, estaríamos causando um efeito conhecido como **Dependência Magnética**.

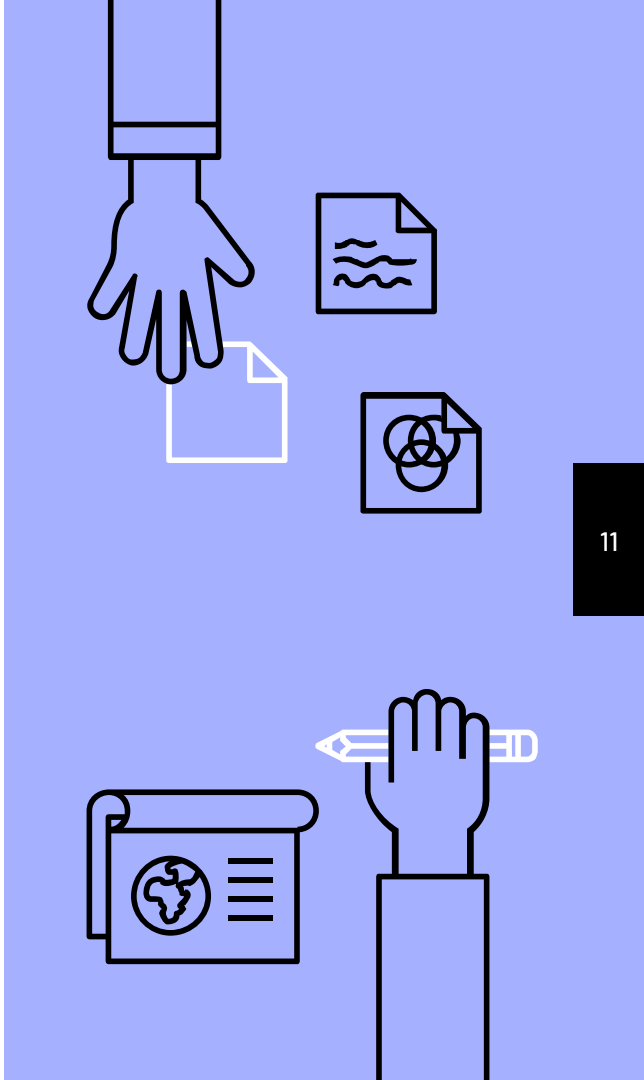


DRY – Não se Repita

Dependência Magnética?

Sim. Continuando o exemplo, imagine que o desenvolvedor efetue uma modificação no método de conversão da classe "RelatorioPedido". Ao compilar o código, a alteração irá afetar indiretamente a classe de orçamentos, já que ela também utiliza esse método. Se houver particularidades nas classes, é possível que a emissão de orçamentos pare de funcionar. E então, o desenvolvedor se pergunta: *"Mas eu alterei só a classe de pedidos... por que o orçamento não está mais funcionando?"*.

Isso é o que chamamos de Dependência Magnética. Uma classe, quando alterada, afeta várias outras classes que estão impropriamente vinculadas à ela. Na prática, é como se fosse um imã que puxasse objetos indevidos. Este tipo de dependência deve ser evitado a todo custo para não prejudicar a arquitetura do software.



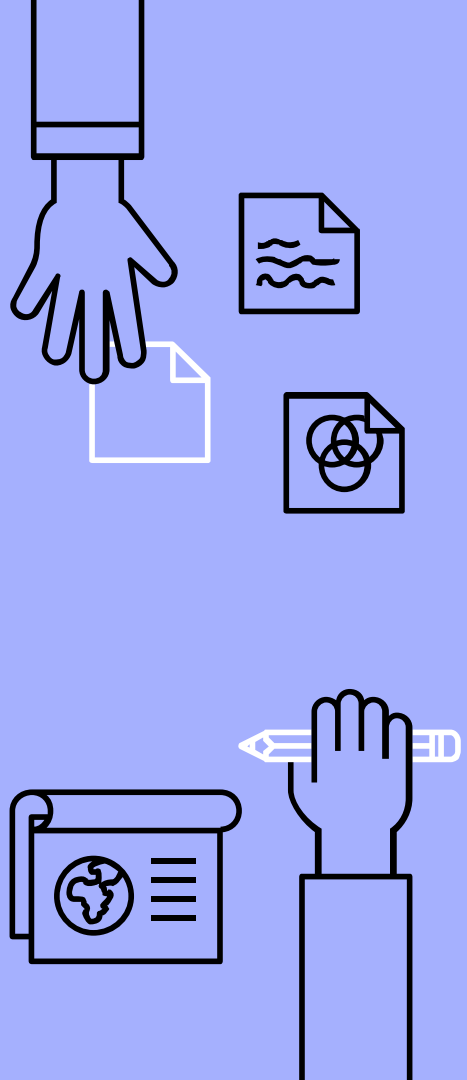
DRY – Não se Repita

Certo, e o que é recomendável neste caso?

É aqui que o princípio **DRY** e a abstração entram na história. Antes de copiar o código, se o desenvolvedor visualizasse o projeto a um nível mais abstrato, saberia que o correto seria criar uma classe exclusiva para essa finalidade como, por exemplo, "ConversorRelatorio". A função principal dessa classe seria converter relatórios para diversos formatos e ser utilizada pelas classes de pedidos e orçamentos de forma explícita.

Sendo assim, saberíamos que qualquer modificação neste conversor afetaria os relatórios de pedidos e orçamentos. Além disso, teríamos a certeza de que, ao alterar a classe de pedidos, a classe de orçamento ficaria intacta, já que o "magnetismo" entre elas deixaria de existir. Em outras palavras, deixaríamos o código menos propenso a erros e mais organizado simplesmente pelo fato de elevar a abstração.

No âmbito da programação, essa ação de extrair um método para uma classe independente compõe o conceito de **refatoração**. Logo, quando alguém lhe disser para refatorar ou "subir" um método (termo informal), significa que este método será utilizado em mais de um local e deve ser adequadamente estruturado para isso.

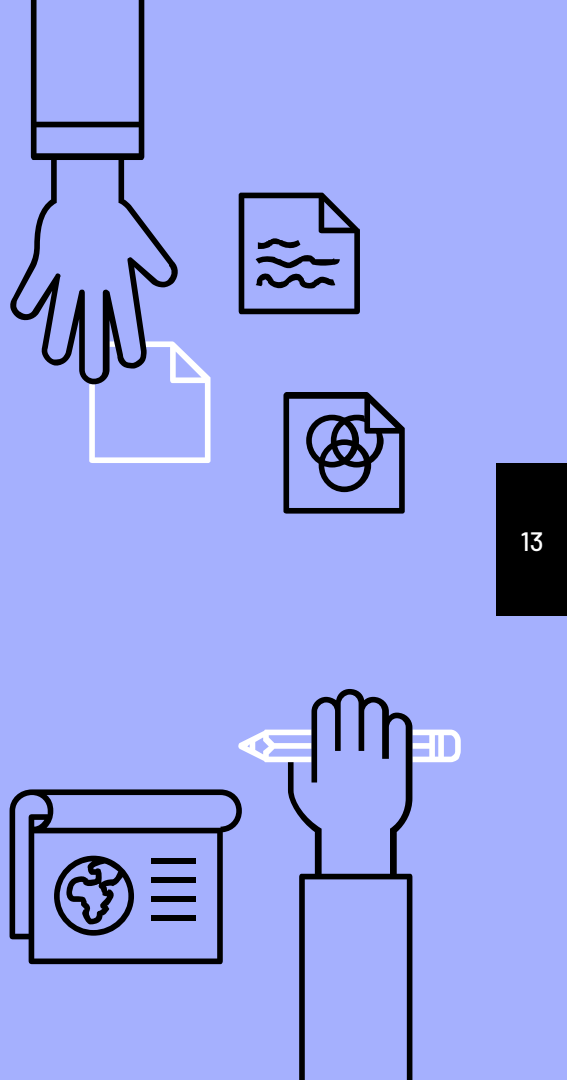


DRY – Não se Repita

Programe para uma interface e não para uma implementação.

Programar orientado a objetos é pensar em abstrações. Um ótimo exercício é tentar ver abstrações em tudo, por exemplo: "Gato, cachorro e pássaro", logo se pensa em "Animal", outro exemplo: "IOF, IPI e ICMS", o que vem na cabeça é "Imposto". Esses tipos de exercícios é excelente para praticarmos á sempre pensar em abstrações, e isso é programar orientado a objetos. É pensar primeiro na abstração, e depois, na implementação.

Essa é uma mudança de pensamento difícil para quem programa no paradigma procedural. Porque no mundo procedural, você está muito preocupado com a implementação. E é natural. Na programação Orientada a Objetos, você tem que inverter, a sua preocupação maior, tem que ser com a abstração ao projetar suas classes.



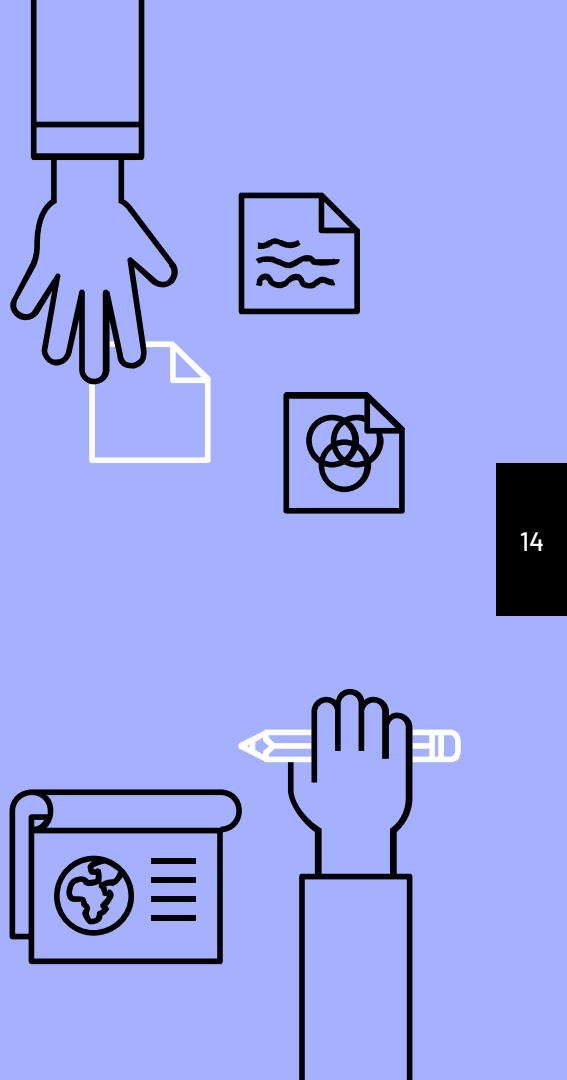
DRY – Não se Repita

Os primeiros passos são:

- Procurar por todo código fonte do seu sistema se a solução que você está tentando implementar já não existe;
- Se precisou duplicar/copiar alguma coisa: pense, porque não tornar este trecho de código único? e, então, simplesmente chamá-lo nos lugares desejados;
- Não esqueça de criar testes unitários para garantir que os próximos desenvolvedores não quebrem o seu trecho de código tão bem implementado.

E para completar as grandes vantagens do DRY:

- Economia de tempo e esforço;
- Facilidade de manutenção do código;
- Redução de possíveis erros.

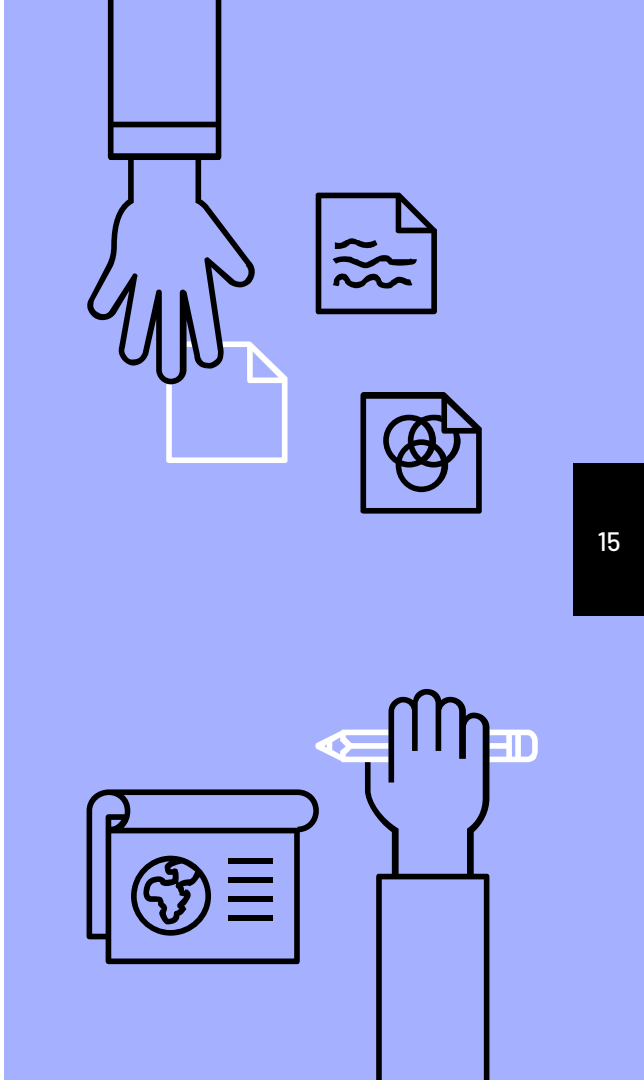


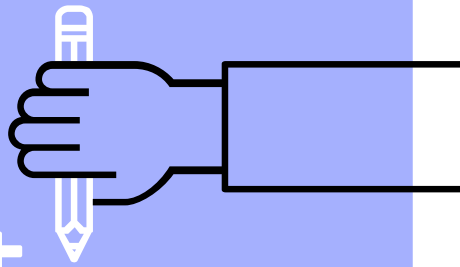
DRY – Não se Repita

Conclusão

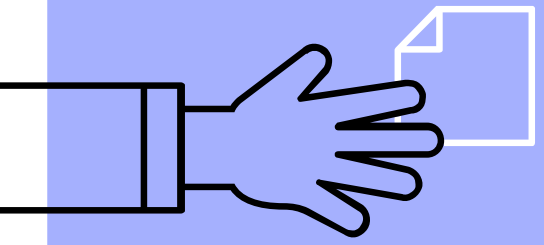
Duplicação de funções ou tarefas podem aparecer na arquitetura, em requisitos de negócio, no desenvolvimento de código e até mesmo na documentação do software. Isto pode causar falhas na implementação e até confusão para os desenvolvedores a respeito do que cada trecho de código realmente faz, podendo ocasionar erros que afetam o projeto inteiro.

A utilização do DRY evita diversos problemas como manutenção de código desnecessária, aumento de bugs por conta da duplicidade de responsabilidades e até mesmo desempenho da aplicação.





2. YAGNI (You Aren’t Gonna Need It)



Você não vai precisar disto.

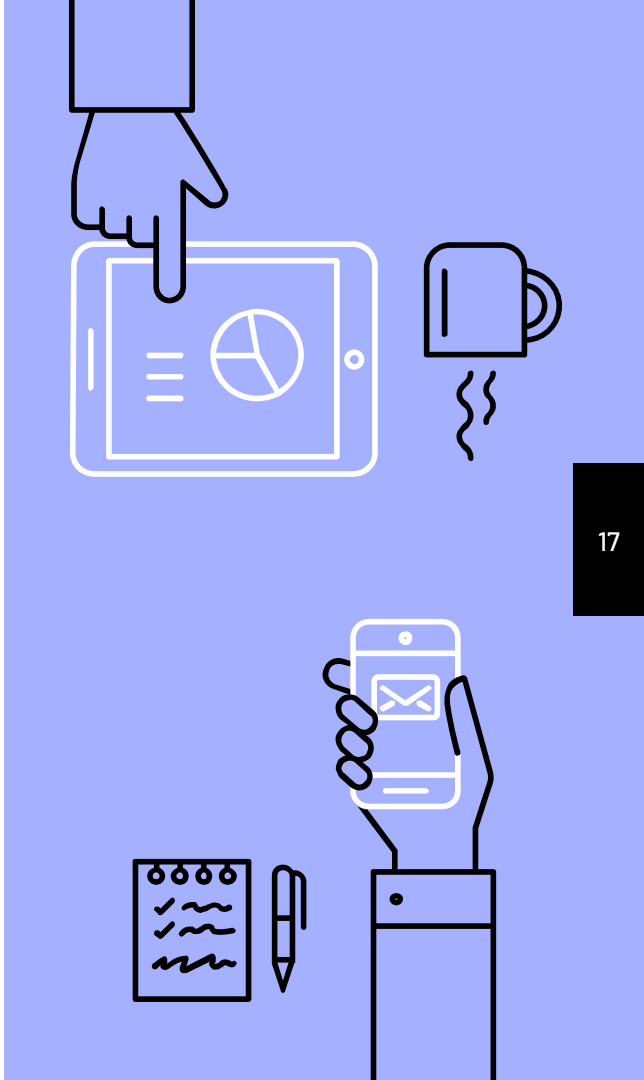
YAGNI - Você não vai precisar disto

Você já se pegou alguma vez pensando se realmente iria existir a necessidade de alguma funcionalidade no futuro que justifique uma implementação técnica no presente? Ou se um dia o cliente iria precisar de uma funcionalidade X? Pois é, ás vezes nos pegamos preocupados com o futuro e em determinados momentos gastamos tempo, esforço e custo para algo que sequer chegará a ser usado ou necessário por alguém.

O princípio **YAGNI** vem justamente para ajudar momentos assim, ou seja, *You Aren't Gonna Need it* (**Você não vai precisar disso**), o mesmo trata-se de um princípio da metodologia XP (*Extreme Programming*).

Esse princípio diz que você não deveria criar funcionalidades que não é realmente necessária, ou seja, apenas crie e se preocupe com o que é necessário e tem a sua necessidade conhecida. A ideia é remover lógica e funcionalidades desnecessárias. **Ron Jeffries (cofundador do XP), dizia:**

*"Sempre implemente funcionalidades quando você realmente **precisar delas**, e nunca quando você **prever** que vai precisar delas".*

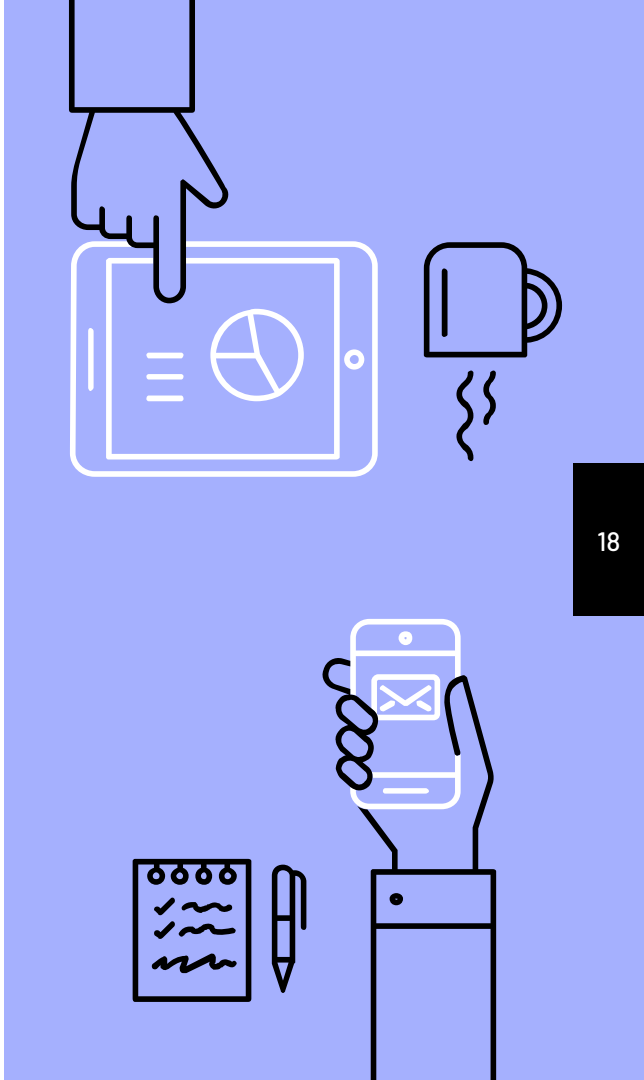


YAGNI - Você não vai precisar disto

Isso quer dizer que você não deveria implementar uma funcionalidade apenas porque você acha que pode precisar dela algum dia (em algum momento), mas, implemente aquilo quando você realmente já precisa e tenha uma necessidade real. Isso irá evitar gastar tempo com implementações que sequer são necessárias, então caso você **não tenha um bom motivo** para adicionar ao projeto, não adicione!

Aprenda **complexidade desnecessária é custo.**

Logo se você evitar tais implementações desnecessárias, estará economizando tempo, pois não será preciso escrever, refatorar ou debuggar código que não se faz útil no momento, mantendo um código limpo, objetivo e organizado.



YAGNI - Você não vai precisar disto

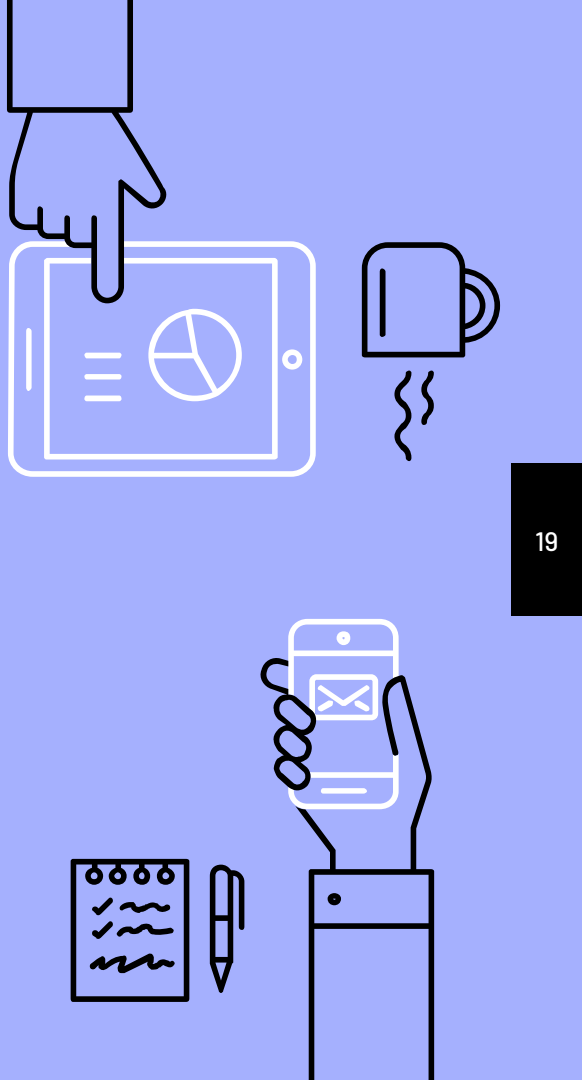
Conclusão

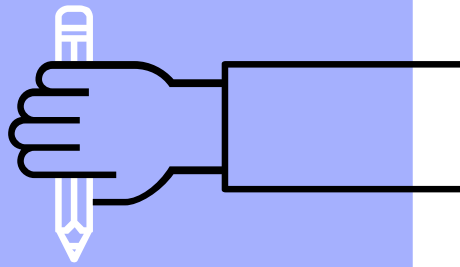
O princípio do desenvolvimento de *software* YAGNI, foca nas necessidades reais e já conhecidas. Economizando tempo, esforço e custos para apenas aquilo que de fato será utilizado e irá retornar algum valor para empresa ou para o cliente.

Se você precisar implementar agora, e se isso já foi definido, então apenas implemente o que é necessário **agora**.

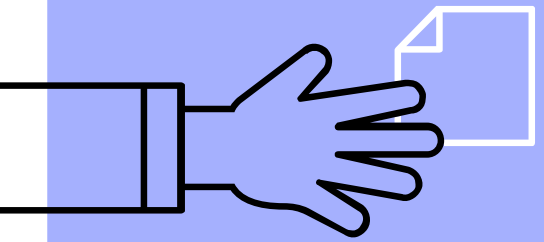
Se for necessário no futuro, então **no futuro** você implementa.

Resista á tentação de codificar agora o que você não precisa, afinal **"You Aren't Gonna Need it".**





3. KISS (Keep It Simple, Stupid)



Mantenha Estupidamente Simples

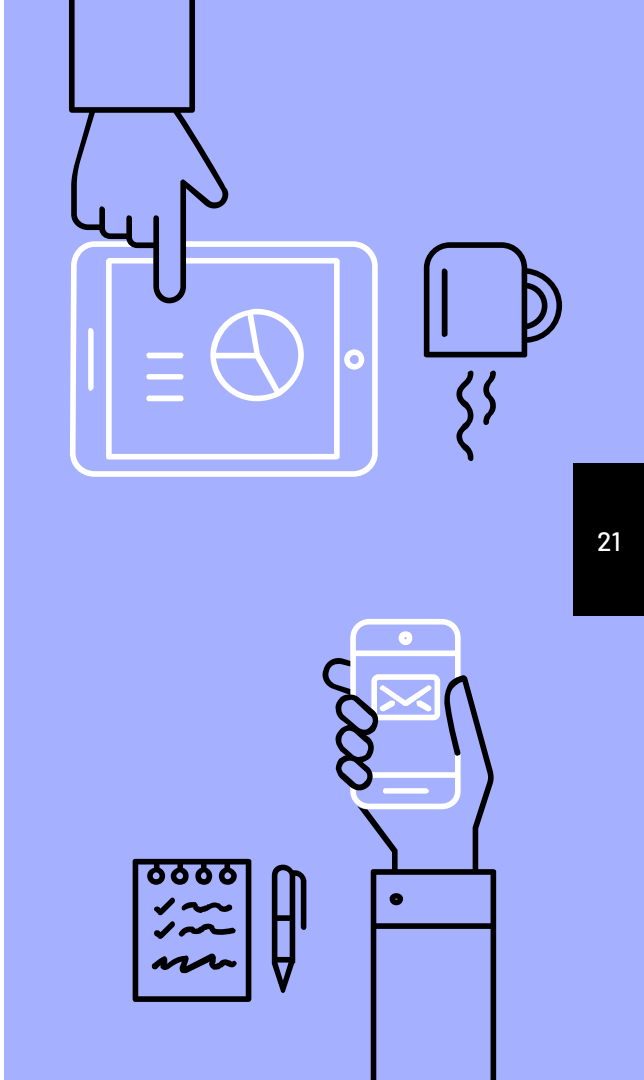
KISS - Mantenha as coisas simples

Keep It Simple, Stupid em tradução livre seria algo como "Mantenha Isso Estupidamente Simples" ou "Mantenha Isso Simples, Estupido".

O termo foi cunhado na década de 60 quando um engenheiro da marinha americana, Kelly Johnson, explicou aos seus subordinados que os aviões deveriam ser simples a ponto de que se algum problema ocorresse em campo de batalha, qualquer mecânico mediano fosse capaz de concerta-lo.

Muitos desenvolvedores (principalmente no começo da carreira) se impressionam e se sentem realmente bons quando desenvolvem um algoritmo consideravelmente complexo, mas com baixo número de linhas.

É... não queria estragar sua felicidade, mas se este é seu caso pare imediatamente!



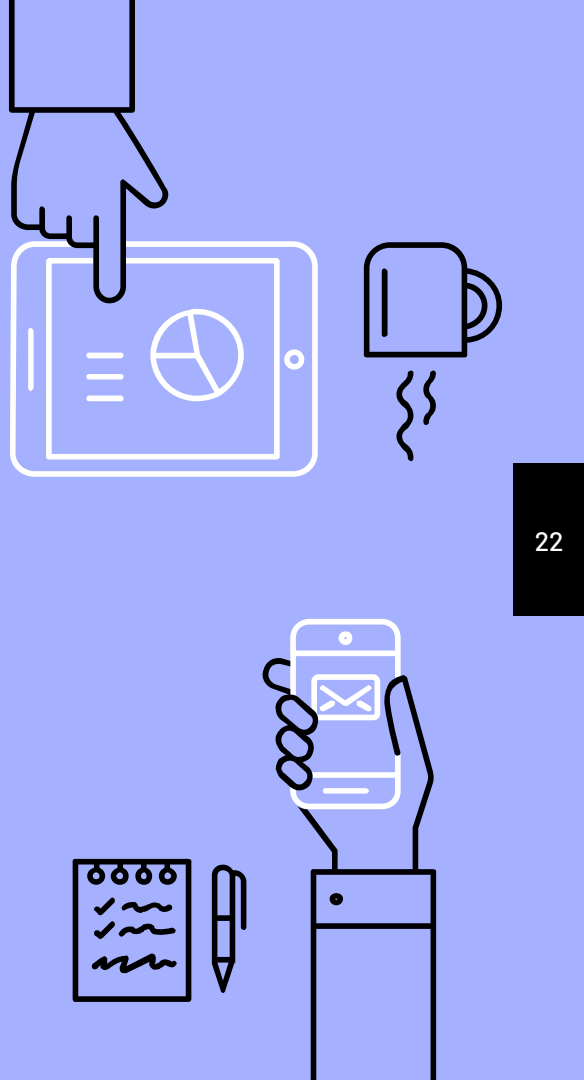
KISS - Mantenha as coisas simples

Quando estamos trabalhando em equipe, precisamos sempre ter em mente que outras pessoas de diferentes contextos e níveis de senioridade estarão lendo nosso código. Então é de extrema importância deixar as coisas literalmente estúpidas, para que qualquer um possa ler e entender sem perder muito tempo.

Quanto mais simples é seu código, mais simples será para dar manutenção nele depois.

Acho a frase escrita por *Martin Fowler* no clássico livro *Refactoring – Improving the Design of Existing Code* muito pertinente para isto:

"Qualquer tolo consegue escrever código que um computador entenda. Bons programadores escrevem código que humanos possam entender".



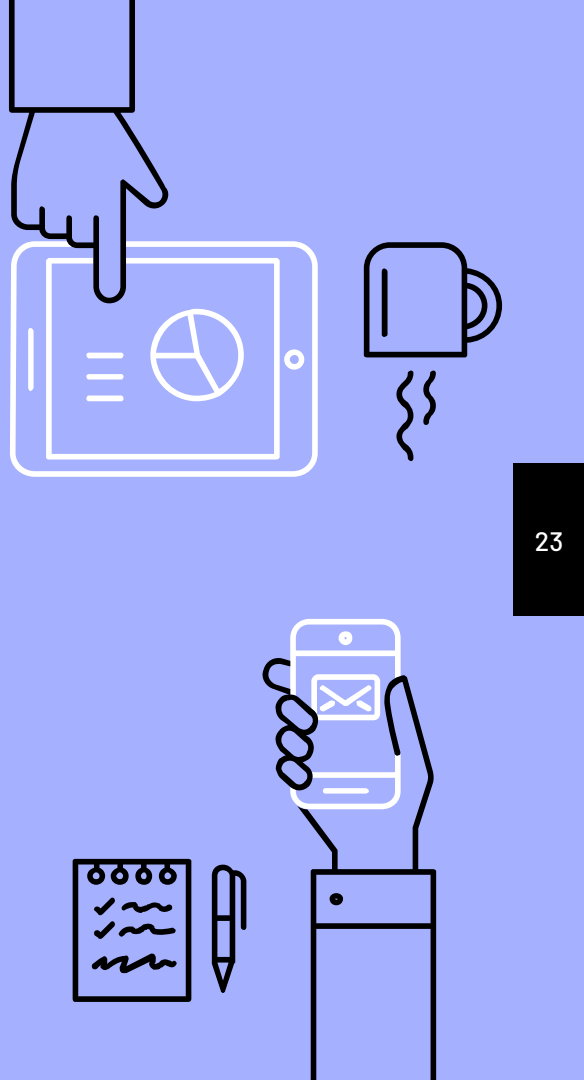
KISS - Mantenha as coisas simples

Como devo aplicar este principio ?

O termo KISS tem como objetivo, **manter a simplicidade e descartar a complexidade**. No contexto do desenvolvimento de software o *"descartar a complexidade"* seria algo como: Quebrar aquele algoritmo complexo em "N" partes simples onde cada parte teria uma única responsabilidade.

Exemplos de violação do principio:

- Classes com 1000-2000 linhas de código provavelmente têm mais de uma responsabilidade do domínio. Isso é algo que fere ao menos as boas práticas de desenvolvimento orientado a objetos. Consequentemente, a facilidade de leitura do código é severamente atingida. Pode chegar um momento que nem mesmo o criador da classe vai conseguir trabalhar nela.
- Comunicação direta entre camadas que não deveriam se comunicar diretamente. Isso utilizando APIs nada simples que exigem configurações globais no projeto. Isso, no médio e longo prazo, simplesmente destrói a possibilidade de evolução saudável do projeto de software.

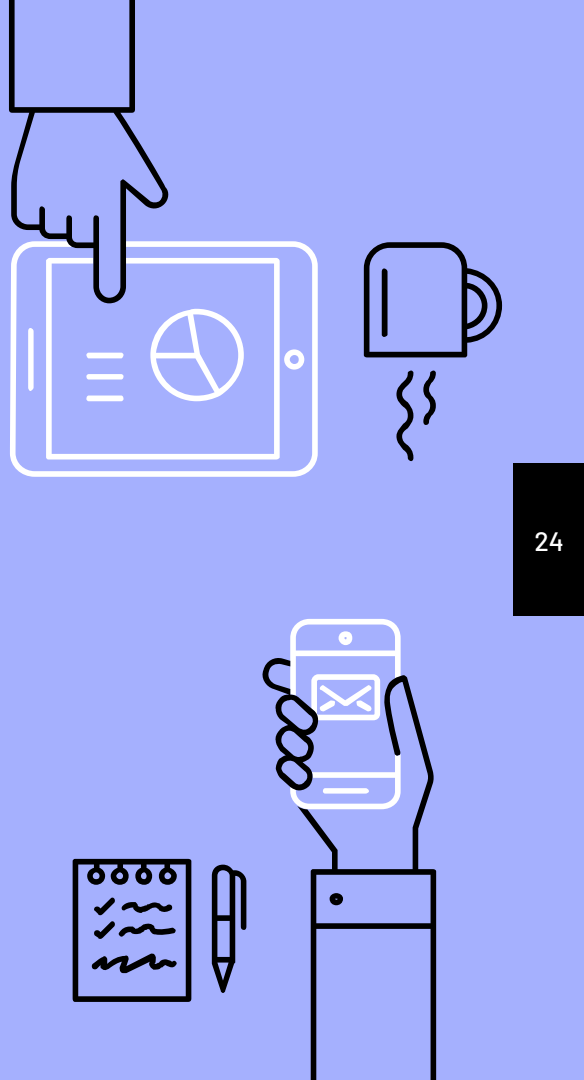


KISS - Mantenha as coisas simples

- Não utilizar variáveis autoexplicativas, que não indique o seu propósito;
- Condicionais em uma linha muito grande de difícil compreensão.

Benefícios ao aplicar esse principio:

- Você será capaz de resolver mais problemas com ainda mais eficiência;
- Seus códigos terão mais qualidade, principalmente em termos de leitura;
- Sua base de código será mais flexível. Fácil de estender, modificar ou refatorar;
- Você será capaz de trabalhar em grandes grupos de desenvolvimento e em grandes projetos desde que todos mantenham os códigos estupidamente simples.



KISS - Mantenha as coisas simples

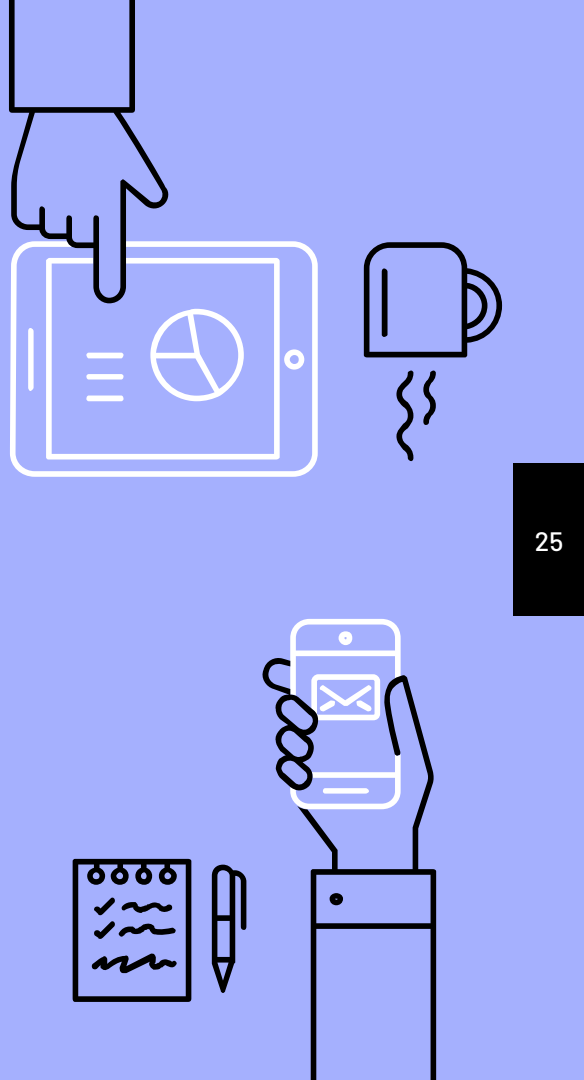
A aplicação deste princípio é um processo que mesmo sendo simples exige paciência. Onde a maior parte dessa paciência será com você mesmo.

1º passo

- Seja humilde. Não pense que você é um super gênio. Aliás, esse costuma ser o primeiro erro.
- Seu código pode ser muito simples e você não precisa ser gênio para trabalhar dessa forma.
- A humildade vai permitir que você continue melhorando, refatorando, o código.

2º passo

- Quebre as tarefas em subtarefas. O famoso: dividir para conquistar.
- Isso sempre funciona como um dos melhores caminhos para resolver os problemas de codificação.



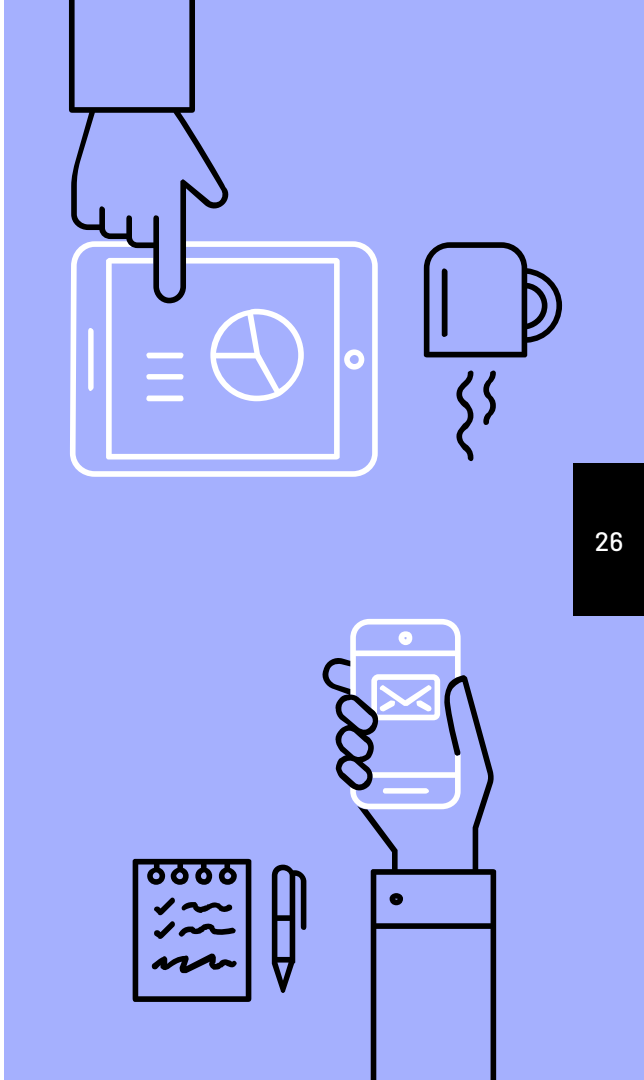
KISS - Mantenha as coisas simples

3º passo

- Quebre os problemas em problemas ainda menores onde cada problema tem que ser passível de ser resolvido em uma ou poucas classes.
- Esse passo é bem similar ao passo anterior, porém o anterior é com ênfase no tempo e esse é com ênfase no código fonte.

4º passo

- Mantenha os métodos pequenos.
- Cada método não pode ter mais do que 30-40 linhas de código. Cada método deve resolver apenas um pequeno problema.
- Se você tem vários blocos condicionais em seu método, transforme esses blocos em outros métodos menores.
- Esses rearranjos não somente deixarão seus códigos fáceis de ler e manter como também será bem mais tranquilo e rápido encontrar e corrigir bugs.



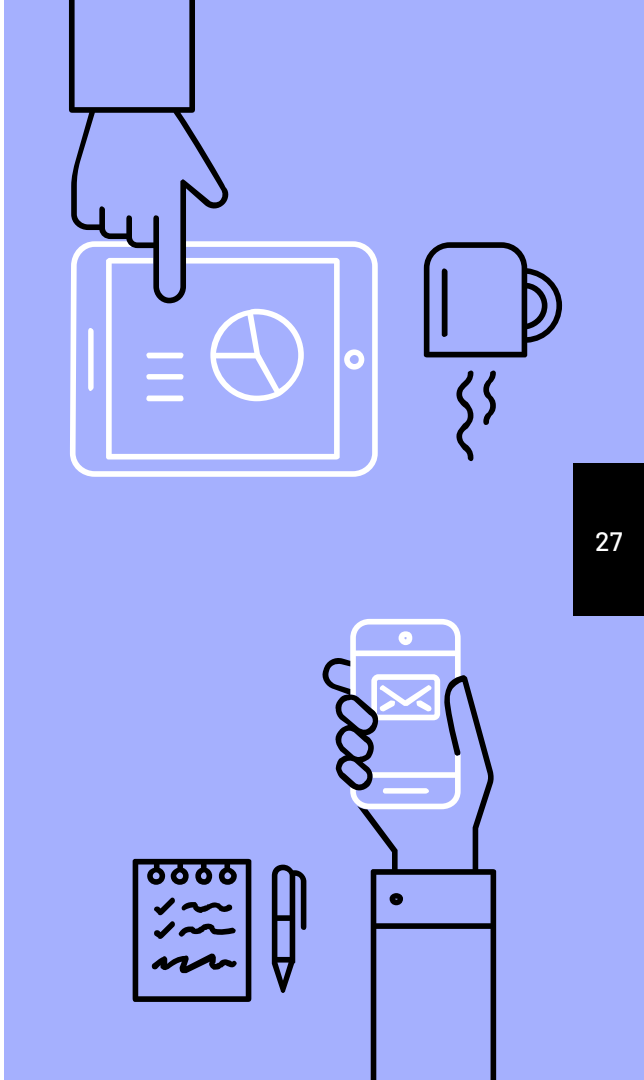
KISS - Mantenha as coisas simples

5º passo

- Mantenha pequenas as classes em seu domínio.
- Se sua classe tem dezenas, quiçá centenas de métodos. Então provavelmente ela contém mais de uma responsabilidade do domínio.

6º passo

- Primeiro resolva o problema. Pode ser no papel. Logo depois codifique.
- Muitos desenvolvedores resolvem os problemas enquanto estão codificando. E normalmente esse não é o melhor caminho para resolver problemas com algoritmos. Além de pensar na solução você também terá que pensar sobre os recursos da linguagem. Se ela fornece, por exemplo, uma API que facilita a escrita de uma possível solução.



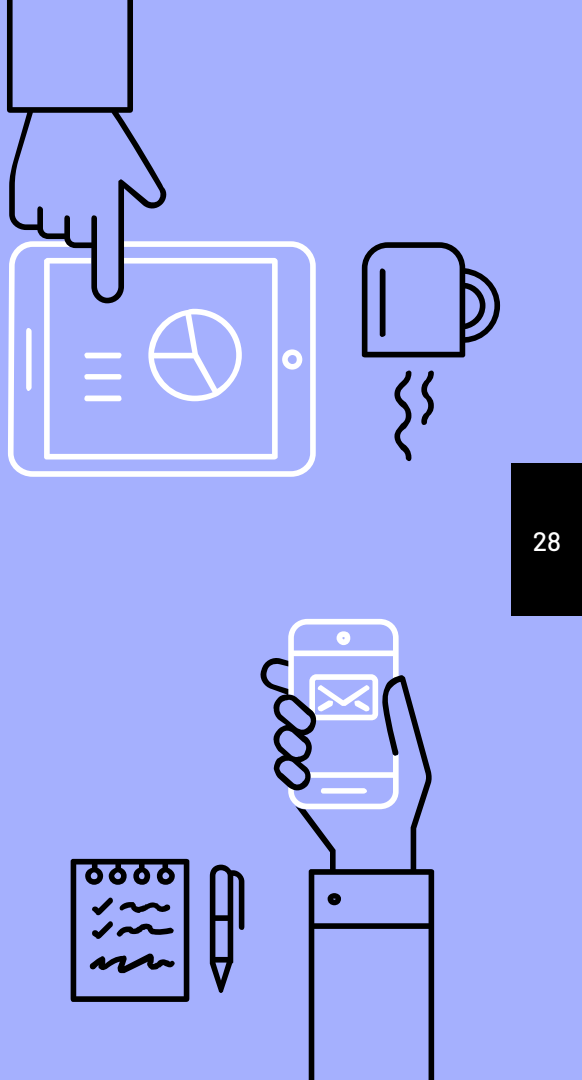
KISS - Mantenha as coisas simples

7º passo

- Não tenha receio em deletar parte do código fonte.
- Refatorar e recodificar são duas importantes tarefas no desenvolvimento de software.
- Se chegarem a você pedidos de funcionalidades que ainda não existem ou que você não foi avisado sobre a possível implementação na época em que estava codificando o projeto e se você seguiu todos os passos anteriores, então será bem mais simples de apagar parte do código fonte sem danos críticos a outras partes do aplicativo. E assim reescrever uma solução melhor (ou nova) para as funcionalidades.

8º passo

- Para todos os cenários: sempre tente colocar o código o mais simples possível.
- Esse é o passo mais complicado de se aplicar. Mas depois de aplicado você possivelmente perceberá que estava programando de maneira menos eficiente antes dos princípios do KISS.



KISS - Mantenha as coisas simples

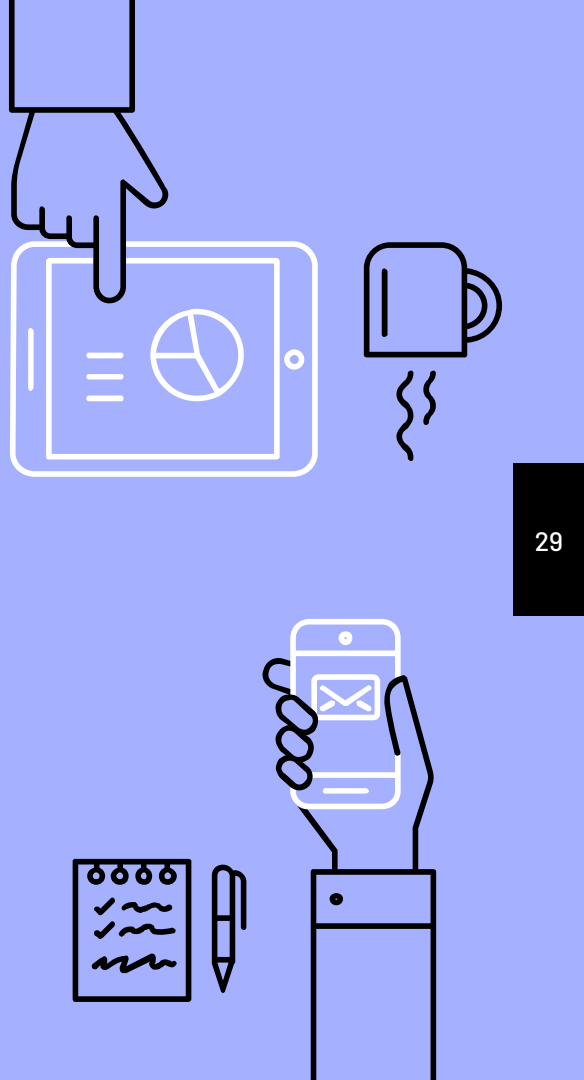
Se você já é veterano de guerra, muitos anos como profissional de desenvolvimento. Então é provável que este conteúdo não tenha lhe falado muito do que você já não sabia.

Porém mesmo assim é válido ressaltar que até mesmo os mais veteranos, às vezes, na correria de entregar tarefas, passamos por cima de coisas simples.

Coisas que muitas vezes não exigem nem mesmo um pensamento lógico de nossa parte, é apenas trabalho mecânico.

Trabalho como:

- Reduzir a responsabilidade de métodos quebrando eles em métodos ainda menores;
- Trabalhar a ideia de código auto comentado, onde as propriedades e métodos têm nomes explicativos ao que eles armazenam e processam;
- Utilizar variáveis de explicação para blocos condicionais complexos;



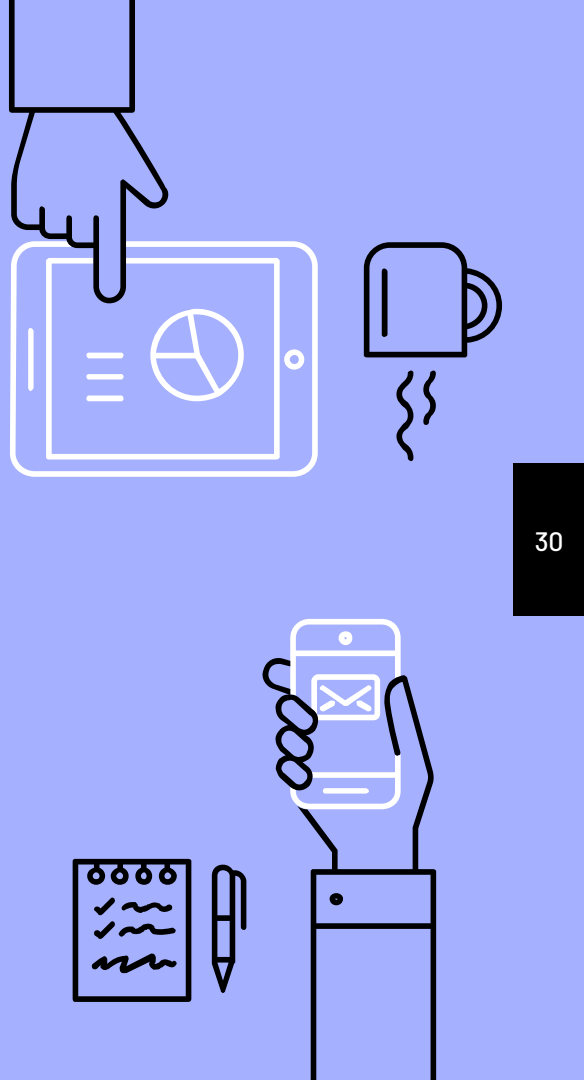
KISS - Mantenha as coisas simples

Verdade seja dita... não só para iniciantes no desenvolvimento de software, mas para todo o cenário de TI. Se manter ciente, estudar as boas práticas em como entregar o melhor código possível, em um tempo aceitável. Isso nunca se torna algo antigo ou "já estudado o suficiente".

É simplesmente uma questão de bom senso. Consumir cada vez mais e praticar. Os resultados sempre vêm.

Robert C. Martin (Uncle Bob) já dizia:

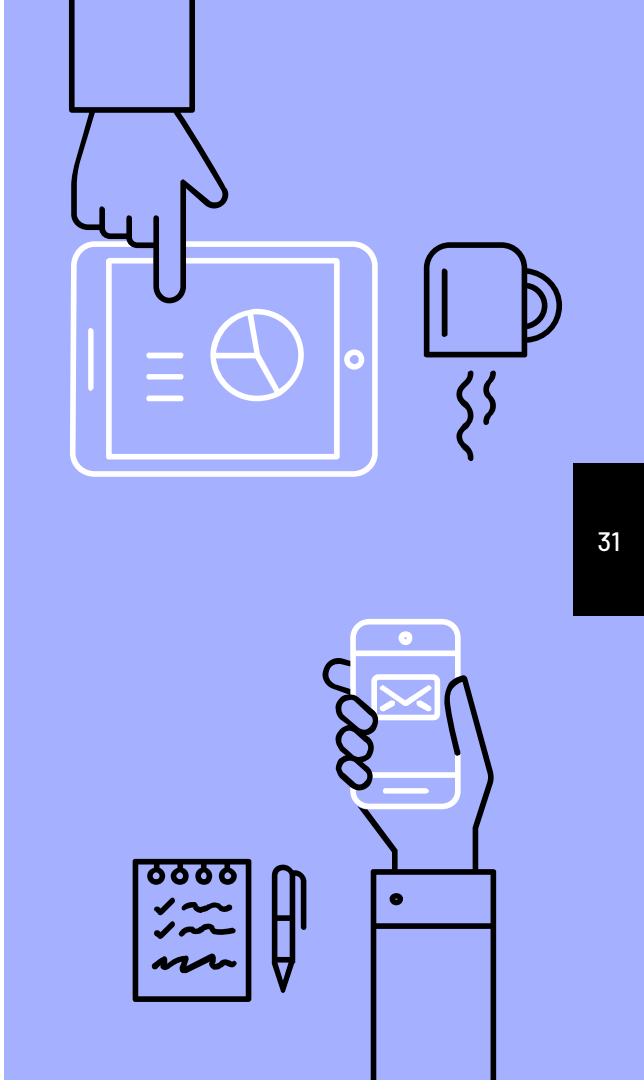
"A proporção de tempo gasto lendo código versus a escrita dele é bem mais do que 10 para 1... portanto tornando-o fácil de ler faz com que seja mais fácil de escrever [refatorar]."



KISS - Mantenha as coisas simples

Conclusão

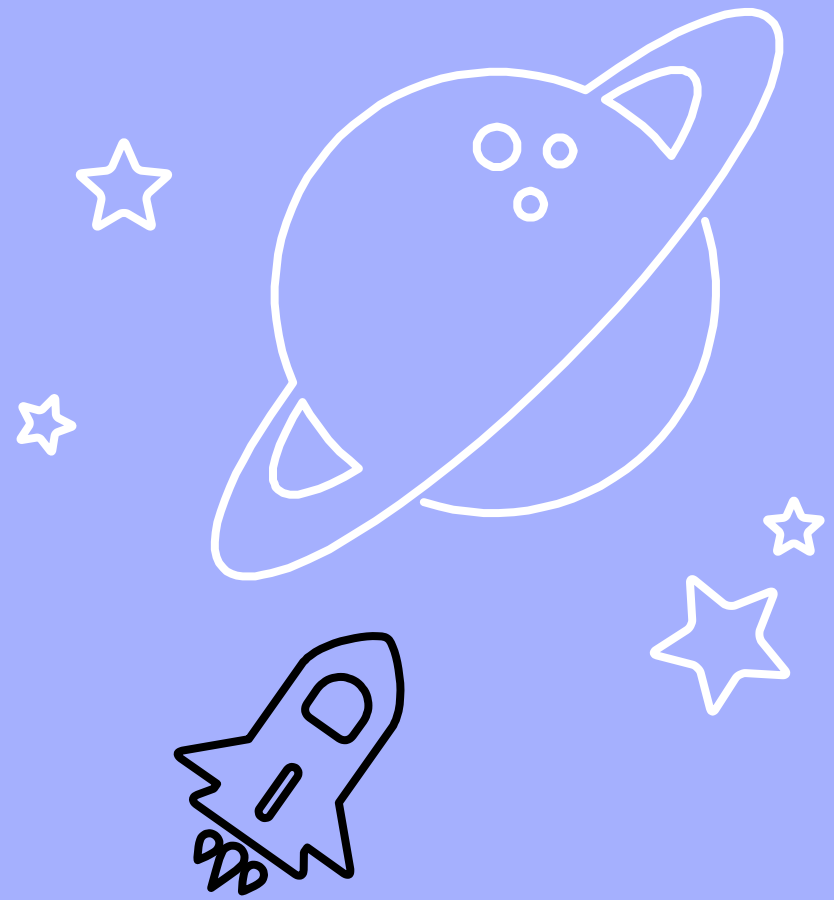
Um código simples de se entender tem como as principais vantagens a evolução do sistema e até economia de tempo, tanto para quem está lendo quando para quem precisa evoluir o código.



Conclusão Final

Os princípios foram criados para dar uma direção para nós desenvolvedores com o objetivo de sempre melhorar o código e o software que estamos trabalhando.

Portanto, pense com carinho nesses princípios antes de sair escrevendo código por ai.



Obrigado!

Alguma pergunta?

Você pode me encontrar em:

- willianbrito05@gmail.com
- www.linkedin.com/in/willian-ferreira-brito
- github.com/willian-brito