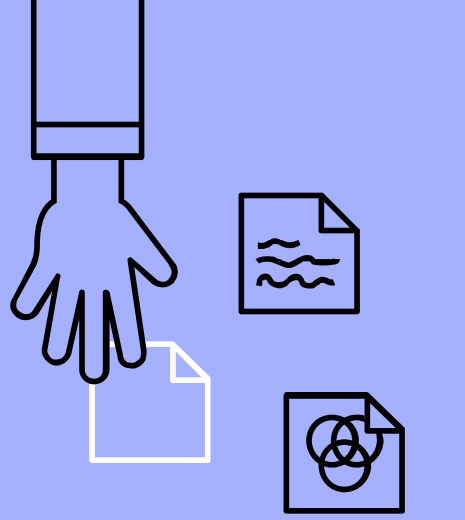
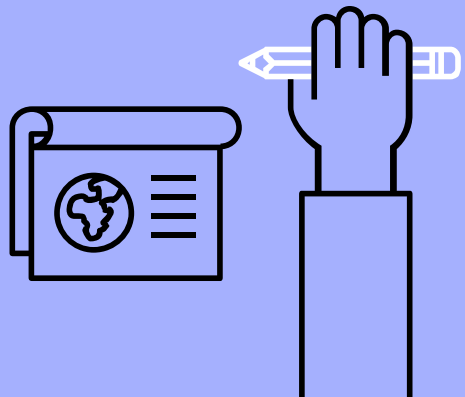

Princípios de Design
de Software

Olá!

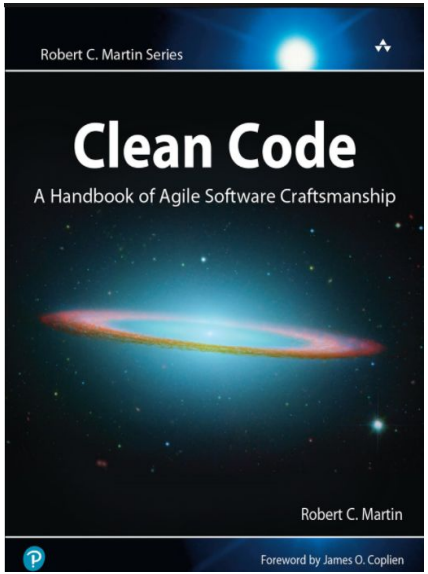
Eu sou Willian Brito

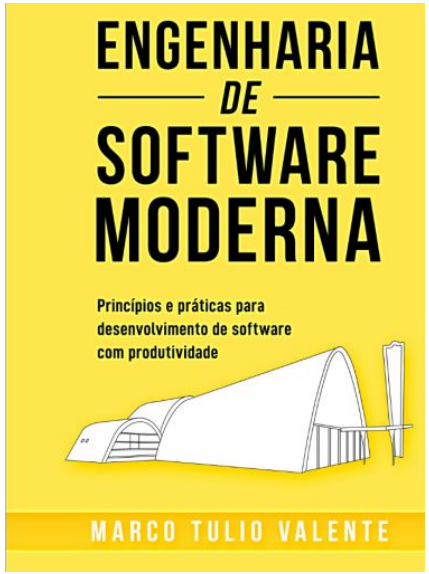
- Desenvolvedor FullStack na Msystem Software
- Formado em Analise e Desenvolvimento de Sistemas.
- Pós Graduado em Segurança Cibernética.
- Certificação SYCP (Solyd Certified Pentester) v2018

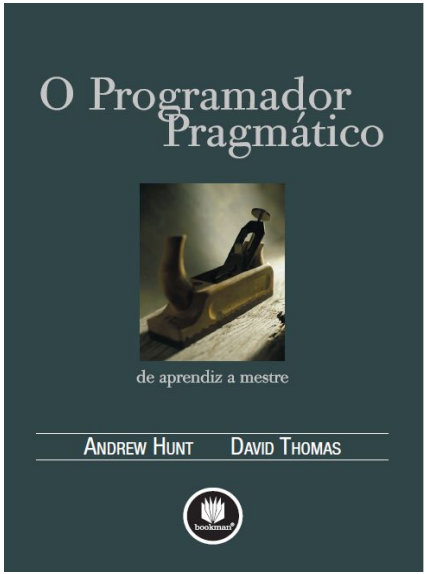




Este conteúdo é baseado nessas obras:







“

Princípios de **Design** são **conceitos** que todo desenvolvedor, **não importa o nível que esteja**, deveria ao menos conhecer..

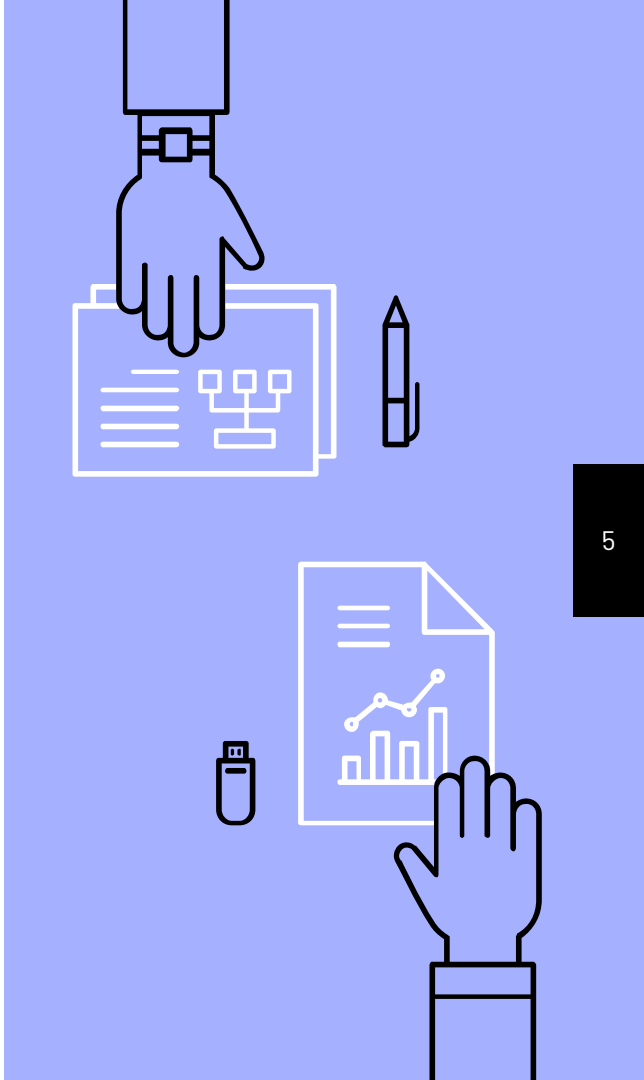
Princípios de Design de Software.

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.

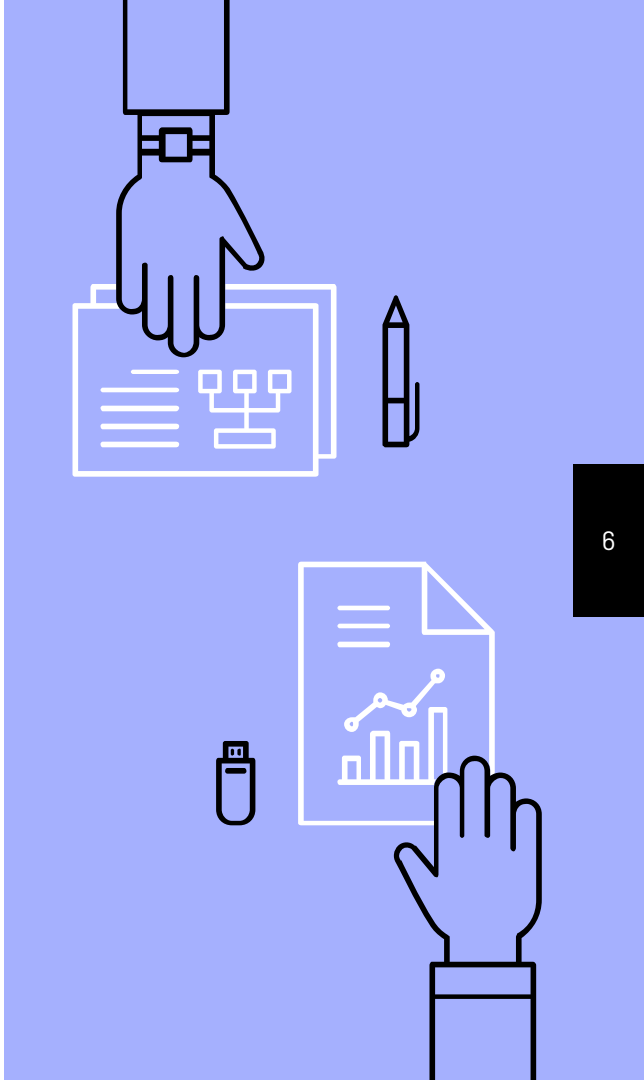


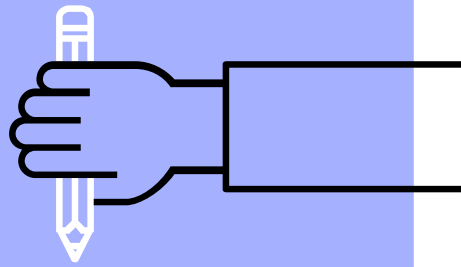
Princípios de Design de Software.

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.

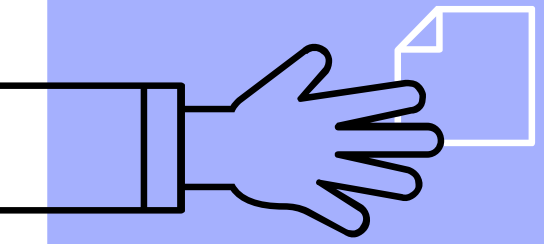
Resolvi falar destes princípios básicos de Design de Software que muitas pessoas já utilizam sem saber ao certo que eles são usados como princípios para qualquer linguagem de programação que você utilize.

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





0. Propriedades de Projetos



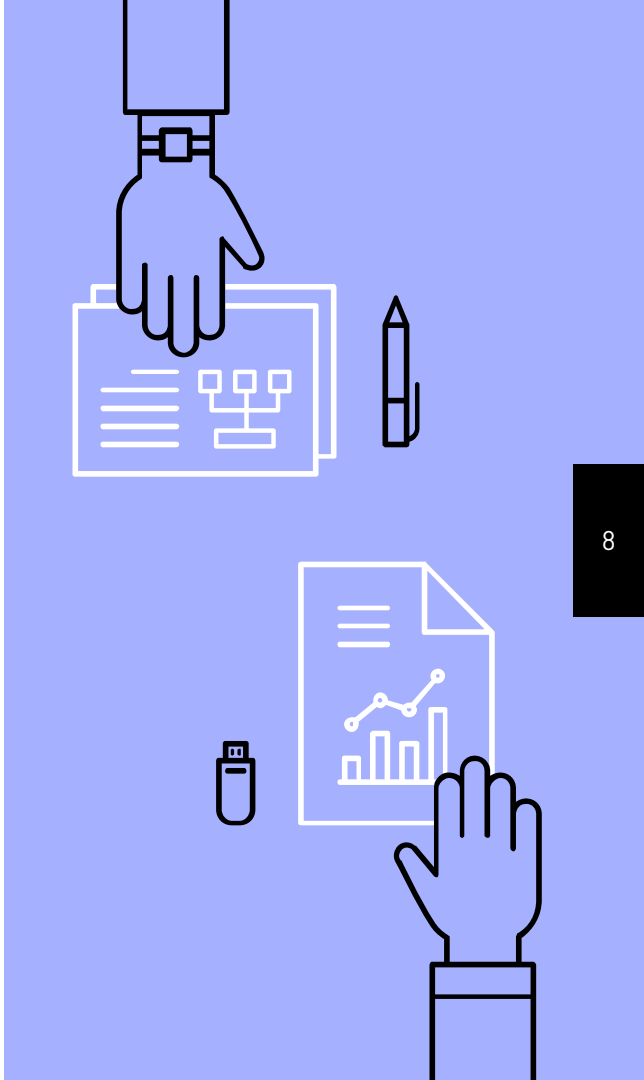
Propriedades de Projetos

É verdade que o projeto de sistemas de software depende de **experiência** e, em alguma medida, também de **talento** e **criatividade**. No entanto, existem algumas propriedades importantes no projeto de sistemas. Por isso, estudar e conhecer essas **propriedades de projeto** pode ajudar na concepção de sistemas com maior qualidade.

Vamos estudar as seguintes propriedades de projetos de software:

- Integridade Conceitual.
- Ocultamento de Informação.
- Coesão.
- Acoplamento.

Para tornar o estudo mais prático, iremos, em seguida, enunciar alguns **princípios de projeto**, os quais representam diretrizes para se garantir que um projeto atende a determinadas propriedades.



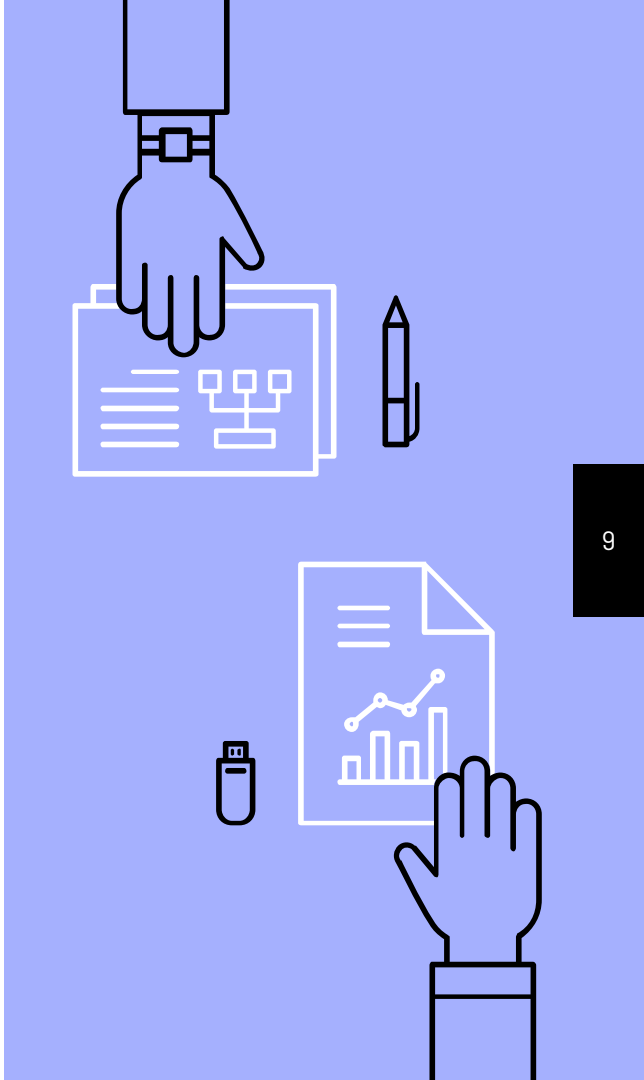
Propriedades de Projetos

Integridade Conceitual

Integridade conceitual é uma propriedade de projeto proposta por **Frederick Brooks**. O princípio foi enunciado em 1975, na primeira edição do livro **The Mythical Man-Month**. Brooks defende que um sistema não pode ser um amontoado de funcionalidades, sem coerência e coesão entre elas.

Integridade conceitual é importante porque facilita o uso e entendimento de um sistema por parte de seus usuários. Por exemplo, com integridade conceitual, o usuário acostumado a usar uma parte de um sistema se sente confortável a usar uma outra parte, pois as funcionalidades e a interface implementadas ao longo do produto são sempre consistentes.

Para citar um contra-exemplo, isto é, um caso de ausência de integridade conceitual, vamos assumir um sistema que usa tabelas para apresentar seus resultados. Dependendo da tela do sistema na qual são usadas, essas tabelas possuem leiautes diferentes, em termos de tamanho de fontes, uso de negrito, espaçamento entre linhas, etc.



Propriedades de Projetos

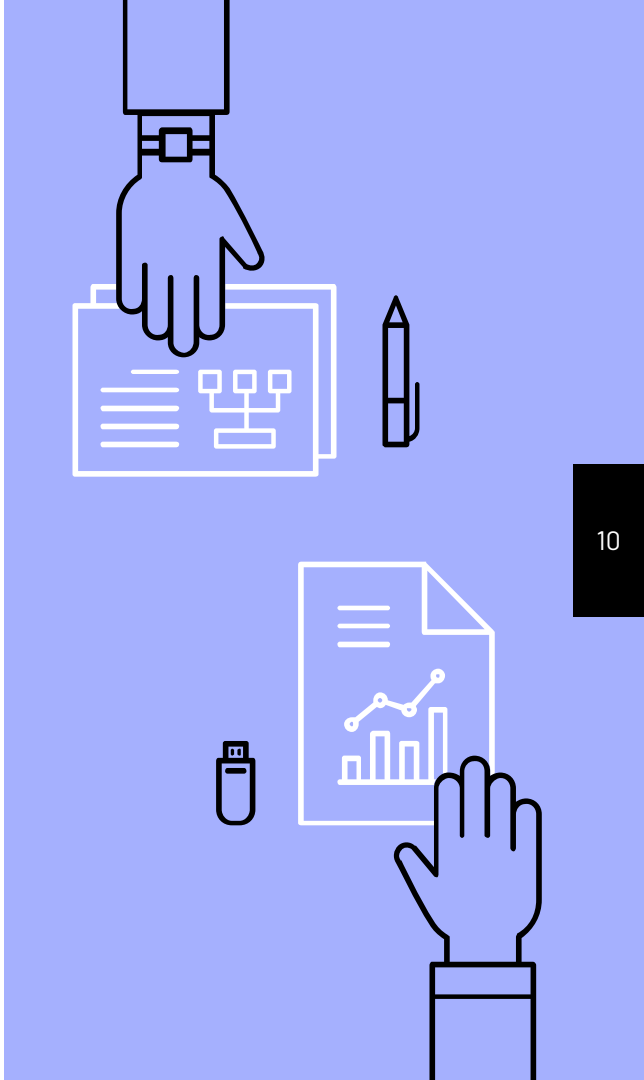
Integridade Conceitual

Além disso, em algumas tabelas pode-se ordenar os dados clicando-se no título das colunas, mas em outras tabelas essa funcionalidade não está disponível. Por fim, os valores são mostrados em moedas distintas. Em algumas tabelas, os valores referem-se a reais; em outras tabelas, eles referem-se a dólares. Essa falta de padronização é um sinal de falta de integridade conceitual e, como afirmamos, ela adiciona complexidade acidental no uso e entendimento do sistema.

Na primeira edição do seu livro, Brooks faz uma defesa enfática do princípio, afirmando que:

"Integridade conceitual é a consideração mais importante no projeto de sistemas. É melhor um sistema omitir algumas funcionalidades e melhorias anômalas, de forma a oferecer um conjunto coerente de ideias, do que oferecer diversas ideias interessantes, mas independentes e descoordenadas."

.

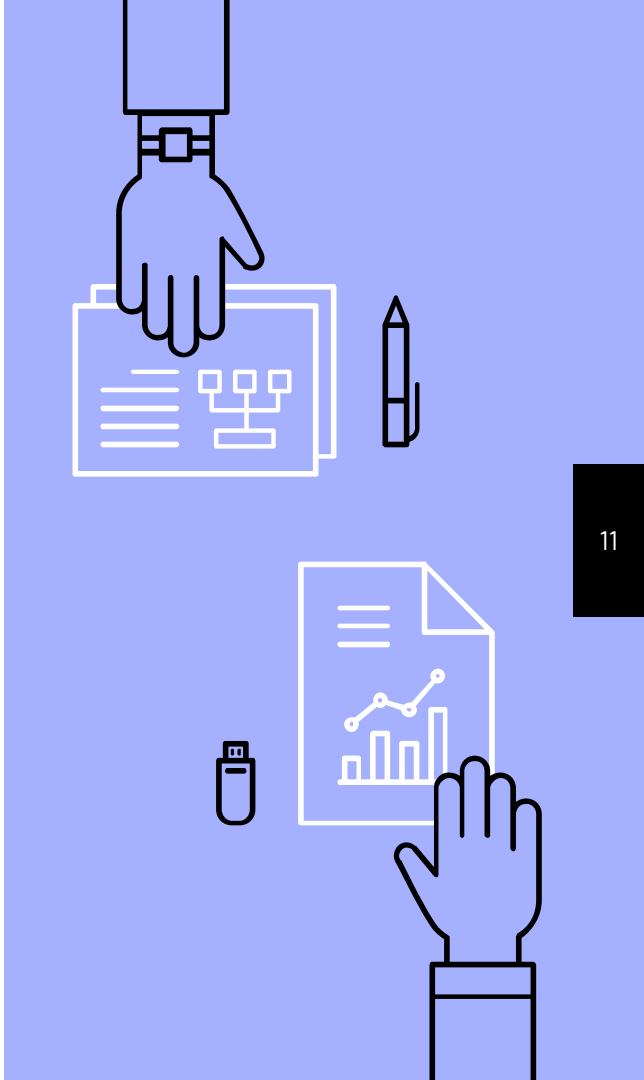


Propriedades de Projetos

Integridade Conceitual

Sempre que falamos de integridade conceitual, surge uma discussão sobre se o princípio requer que uma autoridade central um único arquiteto ou gerente de produto, por exemplo, seja responsável por decidir quais funcionalidades serão incluídas no sistema. Sobre essa questão, temos que ressaltar que essa pré-condição — o projeto ser liderado por uma pessoa apenas, não faz parte da definição de integridade conceitual.

No entanto, existe um certo consenso de que decisões importantes de projeto não devem ficar nas mãos de um grande comitê, no qual cada membro tem direito a um voto. Quando isso ocorre, a tendência é a produção de sistemas com mais funcionalidades do que o necessário, isto é, sistemas sobrecarregados (bloated systems). Por exemplo, um grupo pode defender uma funcionalidade A e outro grupo defender uma funcionalidade B. Talvez, as duas não sejam necessárias; porém, para obter consenso, o comitê acaba decidindo que ambas devem ser implementadas. Assim, os dois grupos vão ficar satisfeitos, embora a integridade conceitual do sistema ficará comprometida.

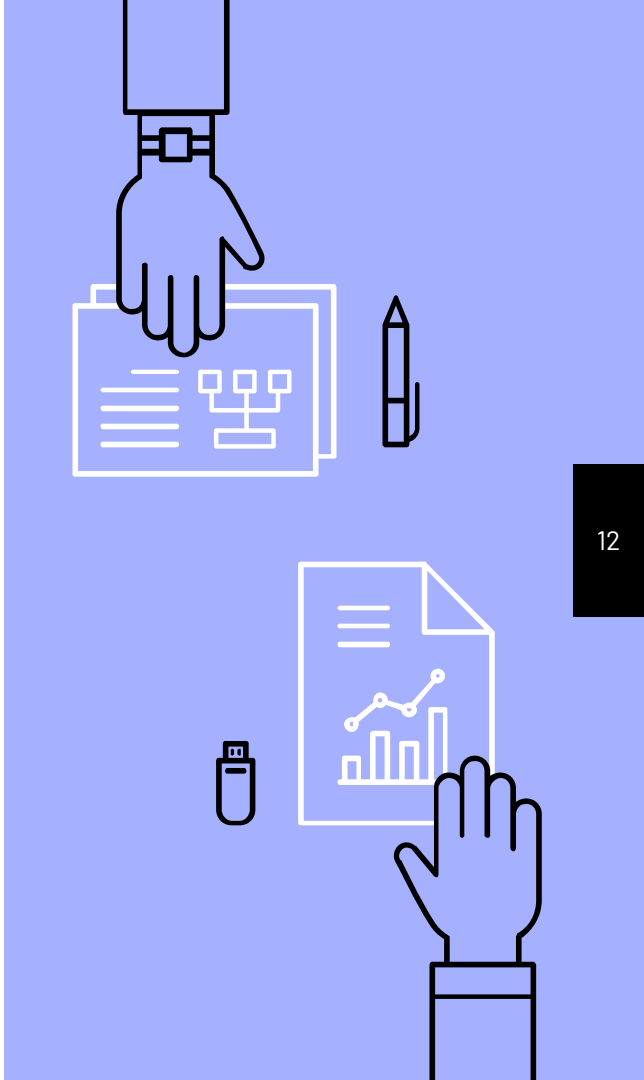


Propriedades de Projetos

Integridade Conceitual

Nos parágrafos anteriores, enfatizamos o impacto da falta de integridade conceitual nos usuários finais de um sistema. No entanto, o princípio se aplica também ao design e código de um sistema. Nesse caso, os afetados são os desenvolvedores, que terão mais dificuldade para entender, manter e evoluir o sistema. A seguir, mencionamos exemplos de falta de integridade conceitual em nível de código:

- Quando uma parte do sistema usa um padrão de nomes para variáveis (por exemplo, camel case, como em notaTotal), enquanto em outra parte usa-se um outro padrão (por exemplo, snake case, como em nota_total).
- Quando uma parte do sistema usa um determinado framework para manipulação de páginas Web, enquanto em outra parte usa-se um segundo framework ou então uma versão diferente do primeiro framework.
- Quando em uma parte do sistema resolve-se um problema usando-se uma estrutura de dados X, enquanto que, em outra parte, um problema parecido é resolvido por meio de uma estrutura Y.

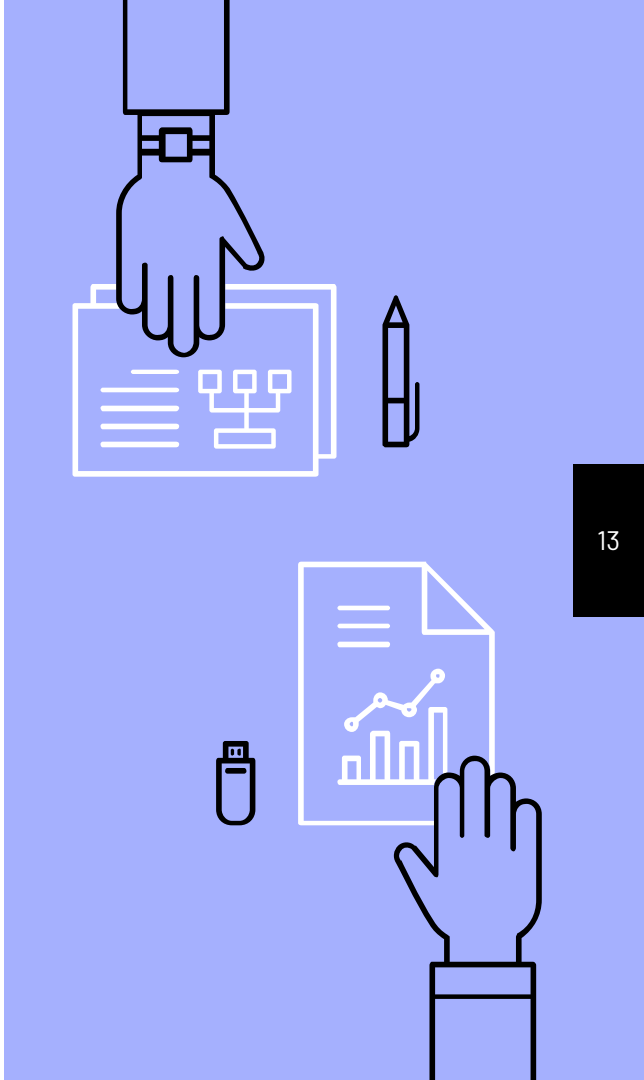


Propriedades de Projetos

Ocultamento de Informação

Essa **propriedade**, uma tradução da expressão **information hiding**, foi discutida pela primeira vez em 1972, por **David Parnas**, em um dos artigos mais importantes e influentes da área de Engenharia de Software, de todos os tempos, cujo título é **On the criteria to be used in decomposing systems into modules**. O resumo do artigo começa da seguinte forma:

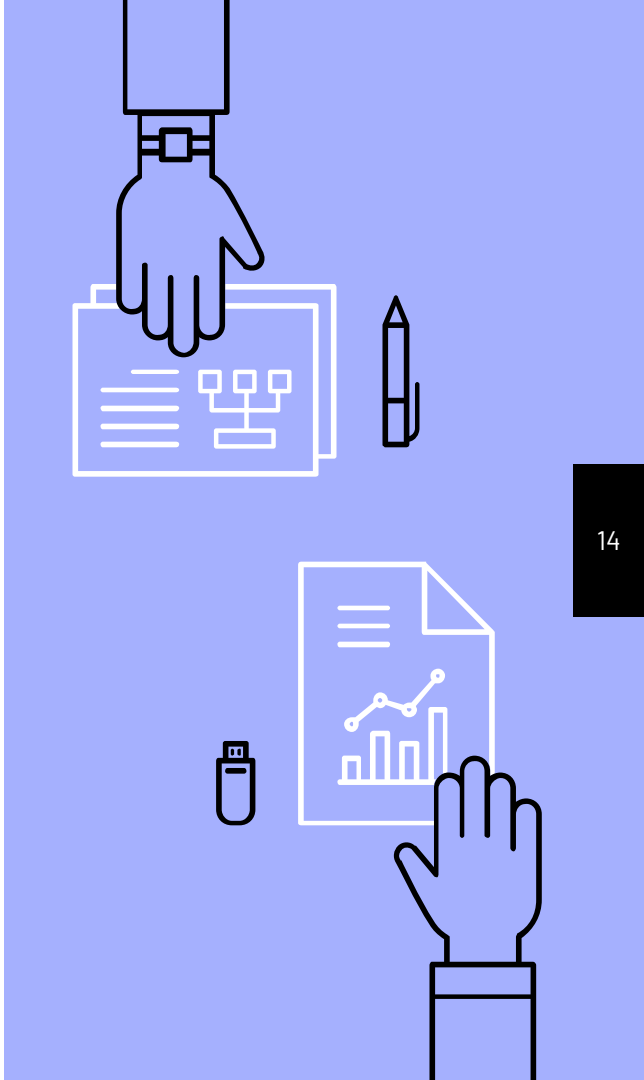
"Este artigo discute modularização como sendo um mecanismo capaz de tornar sistemas de software mais flexíveis e fáceis de entender e, ao mesmo tempo, reduzir o tempo de desenvolvimento deles. A efetividade de uma determinada modularização depende do critério usado para dividir um sistema em módulos."



Propriedades de Projetos

Ocultamento de informação traz as seguintes vantagens para um sistema:

- **Desenvolvimento em paralelo.** Suponha que um sistema X foi implementado por meio de classes C1, C2, ..., Cn. Quando essas classes ocultam suas principais informações, fica mais fácil implementá-las em paralelo, por desenvolvedores diferentes. Consequentemente, teremos uma redução no tempo total de implementação do sistema.
- **Flexibilidade a mudanças.** Por exemplo, suponha que descobrimos que a classe Ci é responsável pelos problemas de desempenho do sistema. Quando detalhes de implementação de Ci são ocultados do resto do sistema, fica mais fácil trocar sua implementação por uma classe Ci', que use estruturas de dados e algoritmos mais eficientes. Essa troca também é mais segura, pois como as classes são independentes, diminui-se o risco de a mudança introduzir bugs em outras classes.
- **Facilidade de entendimento.** Por exemplo, um novo desenvolvedor contratado pela empresa pode ser alocado para trabalhar em algumas classes apenas. Portanto, ele não precisará entender toda a complexidade do sistema, mas apenas a implementação das classes pelas quais ficou responsável.

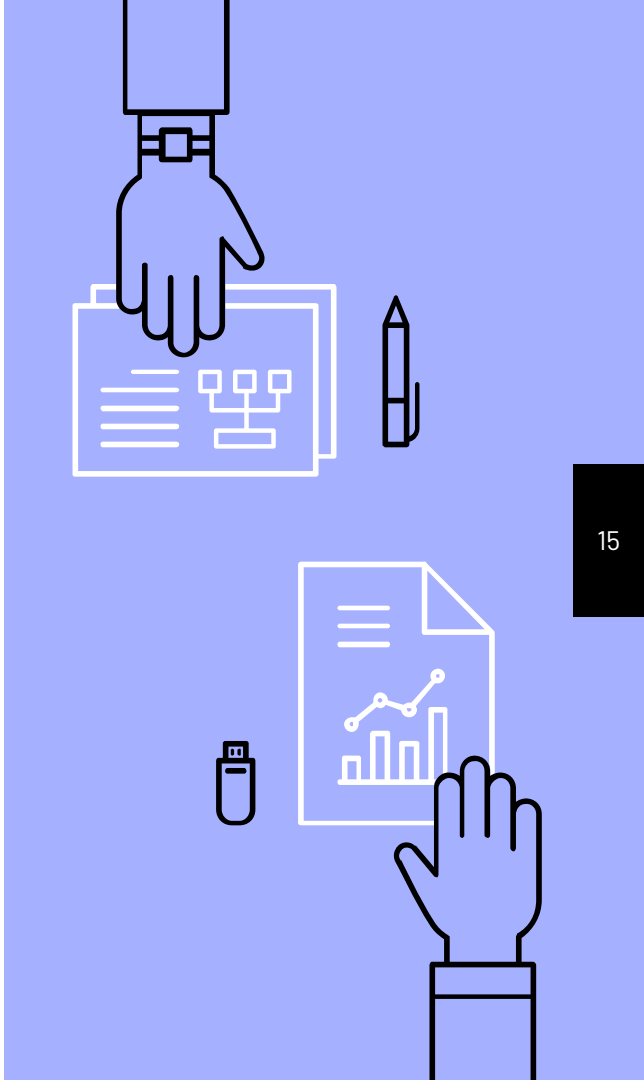


Propriedades de Projetos

Ocultamento de Informação

No entanto, para se atingir os benefícios acima, classes devem satisfazer à seguinte condição (ou critério): elas devem esconder decisões de projeto que são sujeitas a mudanças. Devemos entender decisão de projeto como qualquer aspecto de projeto da classe, como os requisitos que ela implementa ou os algoritmos e estruturas de dados que serão usados no seu código. Portanto, ocultamento de informação recomenda que classes devem esconder detalhes de implementação que estão sujeitos a mudanças. Modernamente, os atributos e métodos que uma classe pretende encapsular são declarados com o modificador de visibilidade **privado**, disponível em linguagens como Java, C++, C# e Ruby.

Porém, se uma classe encapsular toda a sua implementação ela não será útil. Dito de outra forma, uma classe para ser útil deve tornar alguns de seus métodos públicos, isto é, permitir que eles possam ser chamados por código externo. Código externo que chama métodos de uma classe é dito ser **cliente** da classe. Dizemos também que o conjunto de métodos públicos de uma classe define a sua interface. A definição da **interface** de uma classe é muito importante, pois ela constitui a sua parte visível.

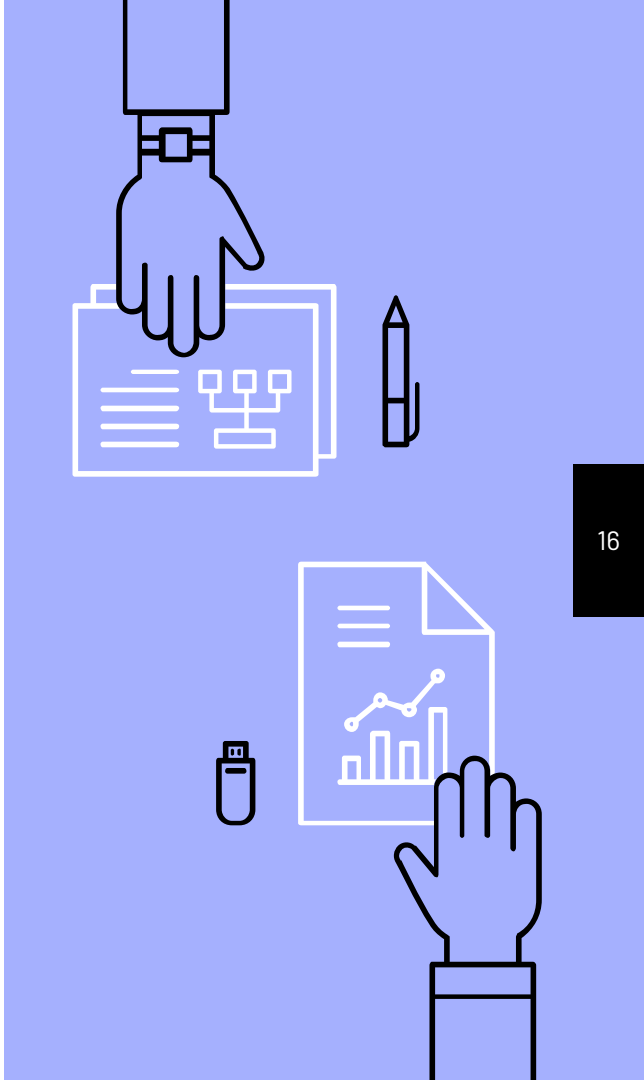


Propriedades de Projetos

Ocultamento de Informação

Interfaces devem ser **estáveis**, pois mudanças na interface de uma classe podem demandar atualizações em seus clientes. Para ser mais claro, suponha uma classe **Math**, com métodos que realizam operações matemáticas.

Suponha um método **sqrt**, que calcula a raiz quadrada de seu parâmetro. Suponha ainda que a assinatura desse método seja alterada para, por exemplo, retornar uma exceção caso o valor do parâmetro seja negativo. Essa alteração terá impacto em todo código cliente do método sqrt, que deverá ser alterado para tratar a nova exceção.



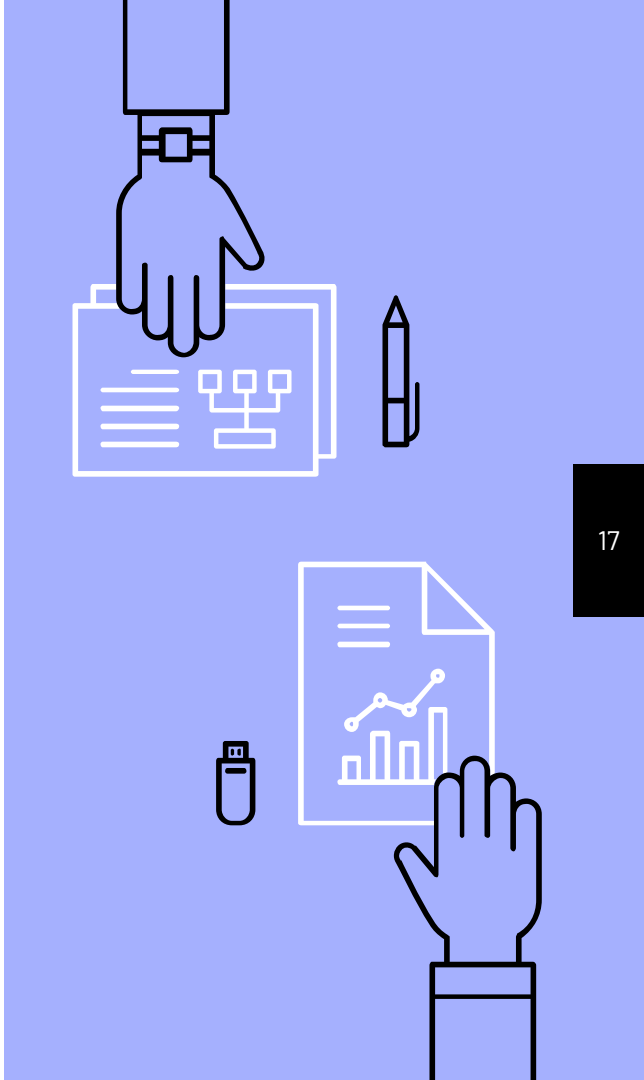
Propriedades de Projetos

Coesão

A implementação de qualquer classe deve ser coesa, isto é, toda classe deve implementar uma única funcionalidade ou serviço. Especificamente, todos os métodos e atributos de uma classe devem estar voltados para a implementação do mesmo serviço. Uma outra forma de explicar coesão é afirmando que toda classe deve ter uma única responsabilidade no sistema. Ou, ainda, afirmando que deve existir um único motivo para modificar uma classe.

Coesão tem as seguintes vantagens:

- Facilita a implementação de uma classe, bem como o seu entendimento e manutenção.
- Facilita a alocação de um único responsável por manter uma classe.
- Facilita o reúso e teste de uma classe, pois é mais simples reusar e testar uma classe coesa do que uma classe com várias responsabilidades.



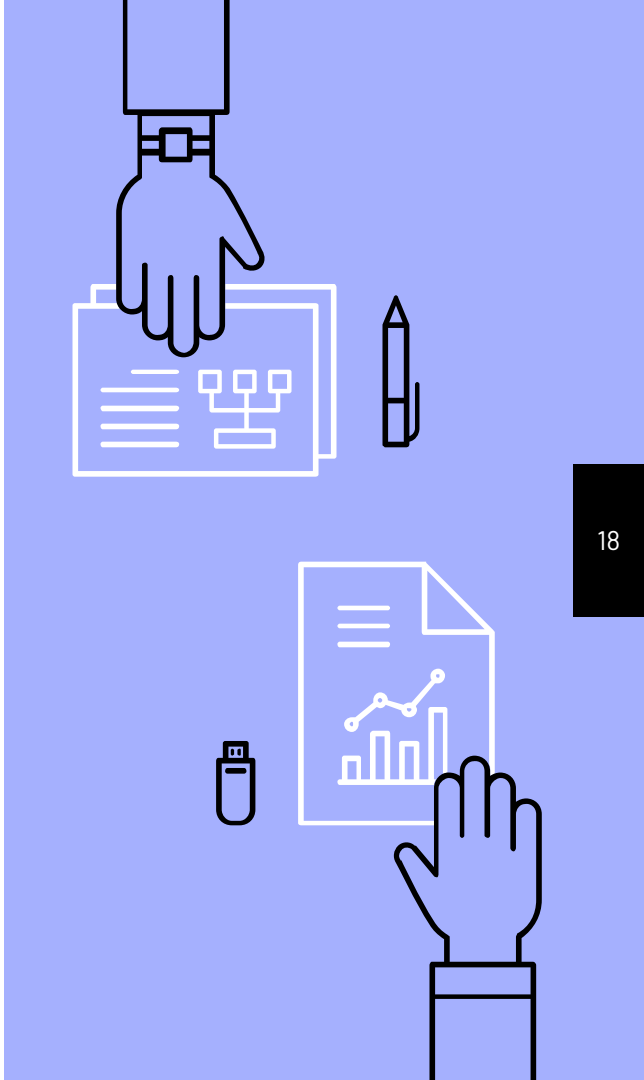
Propriedades de Projetos

Coesão

Separação de interesses (separation of concerns) é uma outra propriedade desejável em projetos de software, a qual é semelhante ao conceito de coesão. Ela defende que uma classe deve implementar apenas um interesse (concern). Nesse contexto, o termo interesse se refere a qualquer funcionalidade, requisito ou responsabilidade da classe.

Portanto, as seguintes recomendações são equivalentes:

- uma classe deve ter uma única responsabilidade;
- uma classe deve implementar um único interesse;
- uma classe deve ser coesa.



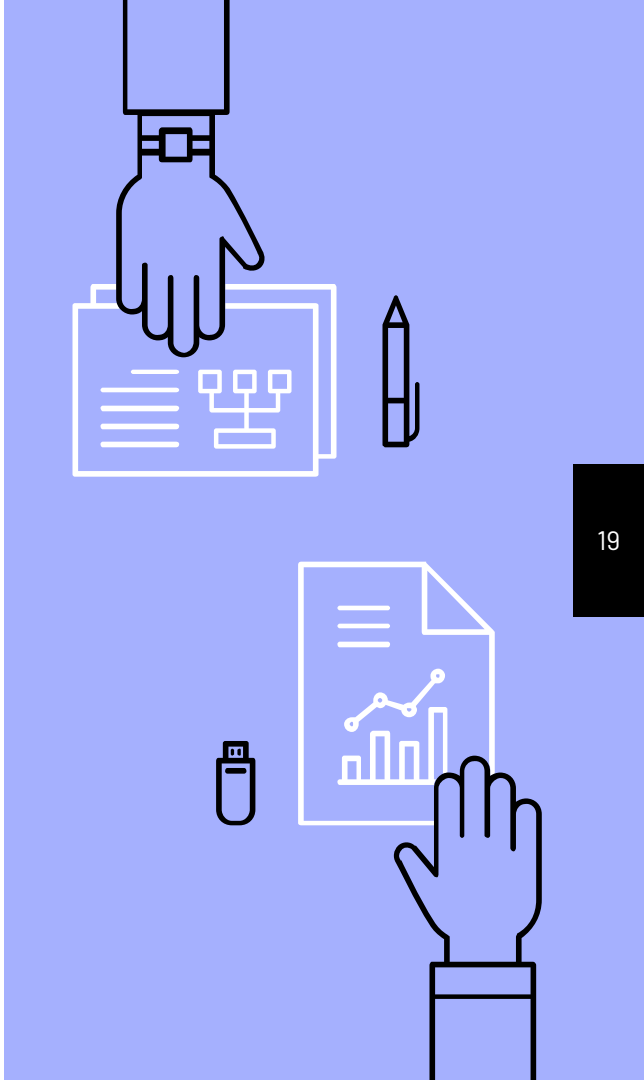
Propriedades de Projetos

Acoplamento

Acoplamento é a **força da conexão entre duas classes**. Apesar de parecer simples, o conceito possui algumas nuances, as quais derivam da existência de dois tipos de acoplamento entre classes: **acoplamento aceitável** e **acoplamento ruim**.

Dizemos que existe um **acoplamento aceitável** de uma classe A para uma classe B quando:

- A classe A usa apenas métodos públicos da classe B.
- A interface provida por B é estável do ponto de vista sintático e semântico. Isto é, as assinaturas dos métodos públicos de B não mudam com frequência; e o mesmo acontece como o comportamento externo de tais métodos. Por isso, são raras as mudanças em B que terão impacto na classe A.

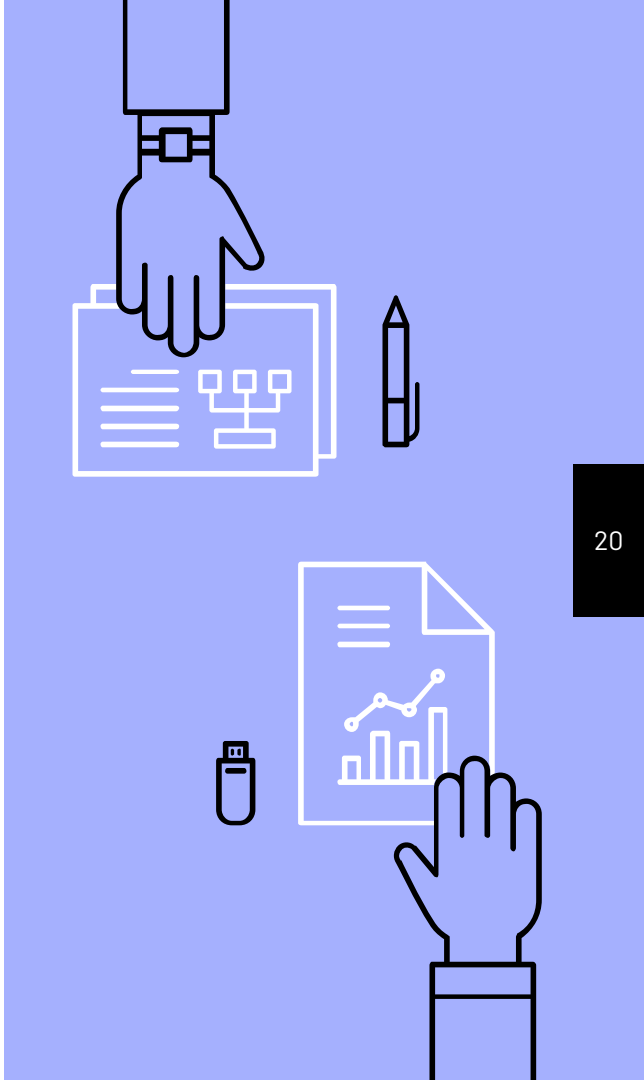


Propriedades de Projeto

Acoplamento

Por outro lado, existe um **acoplamento ruim** de uma classe A para uma classe B quando mudanças em B podem facilmente impactar A. Isso ocorre principalmente nas seguintes situações:

- Quando a classe A realiza um acesso direto a um arquivo ou banco de dados da classe B.
- Quando as classes A e B compartilham uma variável ou estrutura de dados global. Por exemplo, a classe B altera o valor de uma variável global que a classe A usa no seu código.
- Quando a interface da classe B não é estável. Por exemplo, os métodos públicos de B são renomeados com frequência.

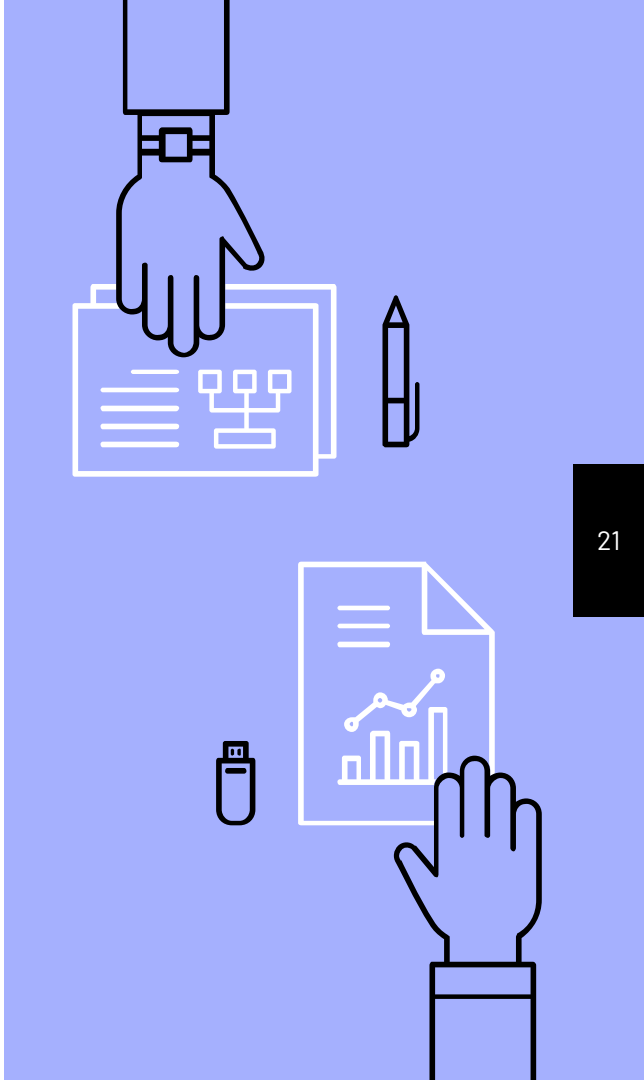


Propriedades de Projetos

Acoplamento

Em essência, o que caracteriza o acoplamento ruim é o fato de que a dependência entre as classes não é mediada por uma interface estável. Por exemplo, quando uma classe altera o valor de uma variável global, ela não tem consciência do impacto dessa mudança em outras partes do sistema. Por outro lado, quando uma classe altera sua interface, ela está ciente de que isso vai ter impacto nos clientes, pois a função de uma interface é exatamente anunciar os serviços que uma classe oferece para o resto do sistema.

Resumindo: acoplamento pode ser de grande utilidade, principalmente quando ocorre com a interface de uma classe estável que presta um serviço relevante para a classe de origem. Já o acoplamento ruim deve ser evitado, pois é um acoplamento não mediado por interfaces. Mudanças na classe de destino do acoplamento podem facilmente se propagar para a classe de origem.



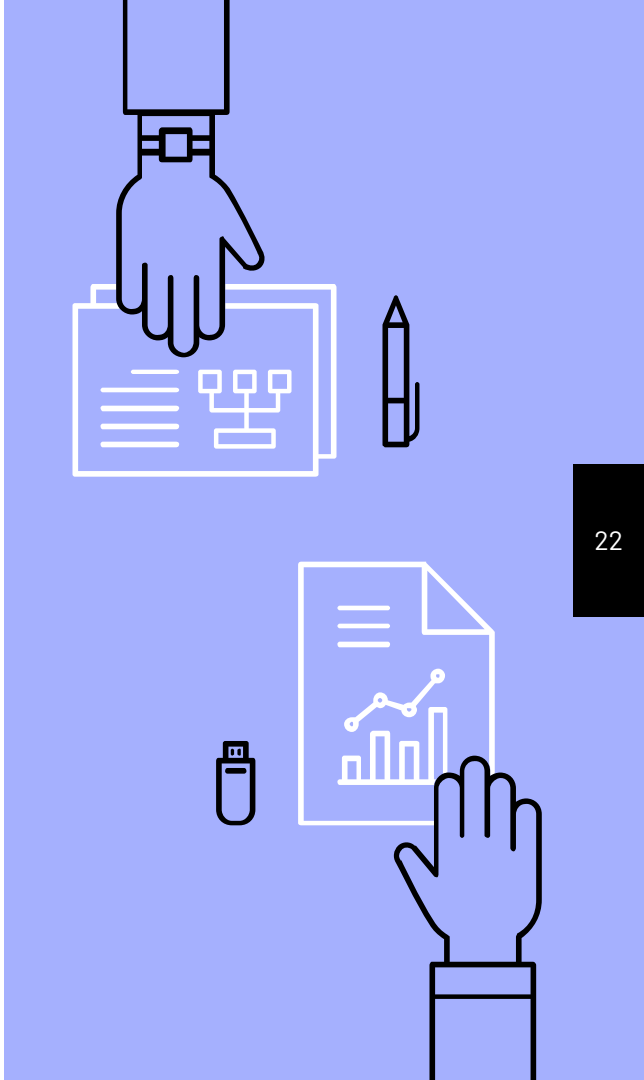
Propriedades de Projetos

Acoplamento

Frequentemente, as recomendações sobre acoplamento e coesão são reunidas em uma única recomendação:

Maximize a coesão das classes e minimize o acoplamento entre elas.

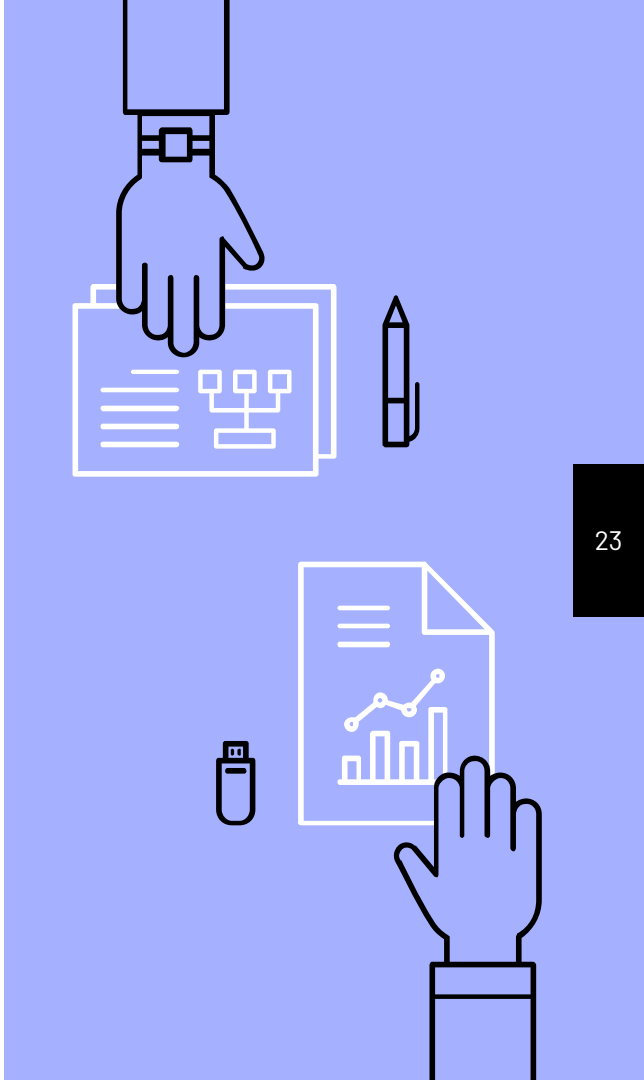
De fato, se uma classe depende de muitas outras classes, por exemplo, de dezenas de classes, ela pode estar assumindo responsabilidades demais, na forma de funcionalidades não coesas. Lembre-se que uma classe deve ter uma única responsabilidade (ou um único motivo para ser modificada). Por outro lado, devemos tomar cuidado com o significado do verbo minimizar. O objetivo não deve ser eliminar completamente o acoplamento de uma classe com outras classes, pois é natural que uma classe precise de outras classes, principalmente daquelas que implementam serviços básicos, como estruturas de dados, entrada/saída, etc.



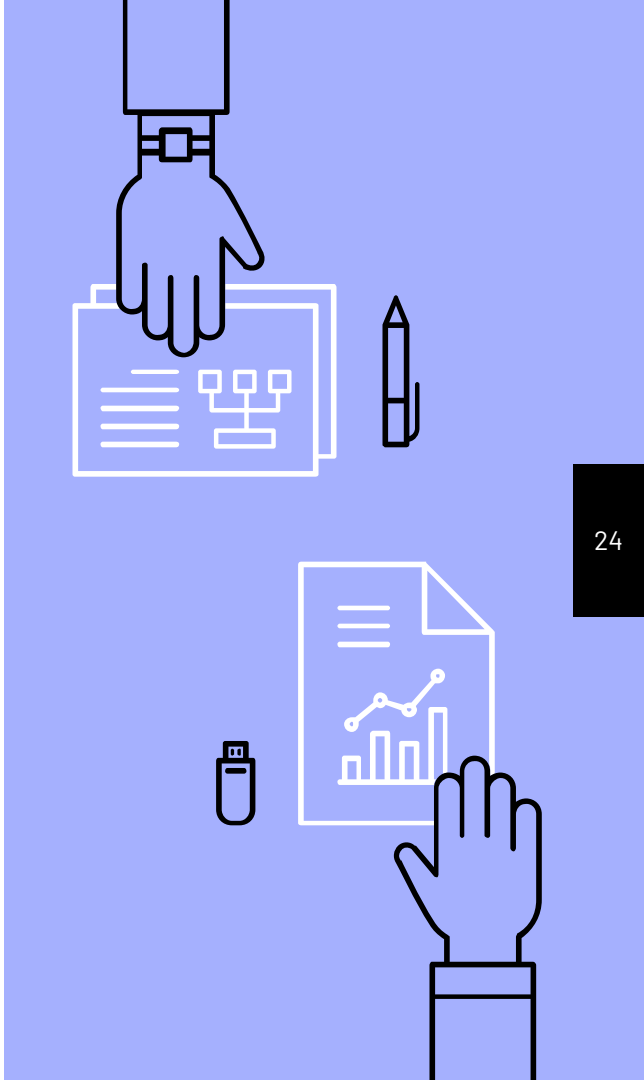
Princípios de Projetos

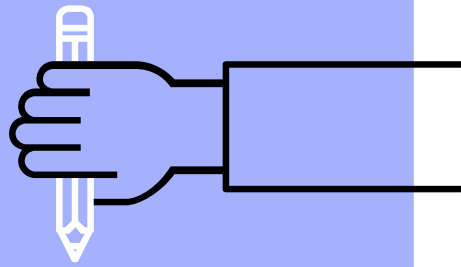
Princípios de projeto são **recomendações** mais concretas que desenvolvedores de software devem seguir para atender às **propriedades de projeto** que estudamos anteriormente. Assim, propriedades de projeto podem ser vistas como recomendações ainda **genéricas (ou táticas)**, enquanto que os princípios que estudaremos agora estão em um **nível operacional**.

Agora, iremos estudar os **nove princípios de projeto** listados na próxima tabela. A tabela mostra ainda as propriedades de projeto que são contempladas ao seguir cada um desses princípios.



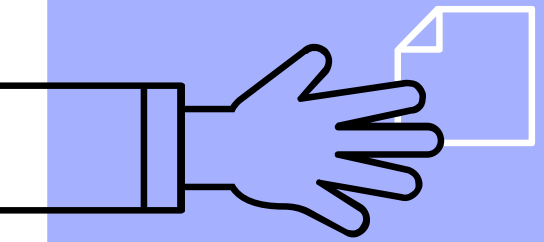
Princípios de Projeto	Propriedade de Projeto
DRY (Não se Repita)	Coesão
YAGNI (Você não vai precisar disto)	Coesão
KISS (Mantenha estupidamente simples)	Coesão
LoD (Lei de Demeter)	Ocultamento de Informação
Tell, Don´t Ask (Diga, não pergunte)	Ocultamento de Informação
Prefira Composição do que Herança	Acoplamento
Programe para Abstrações não para Implementações	Acoplamento
Encapsule o que varia	Coesão
Princípio Hollywood	Acoplamento





1. DRY (Don’t Repeat Yourself)

Não se Repita

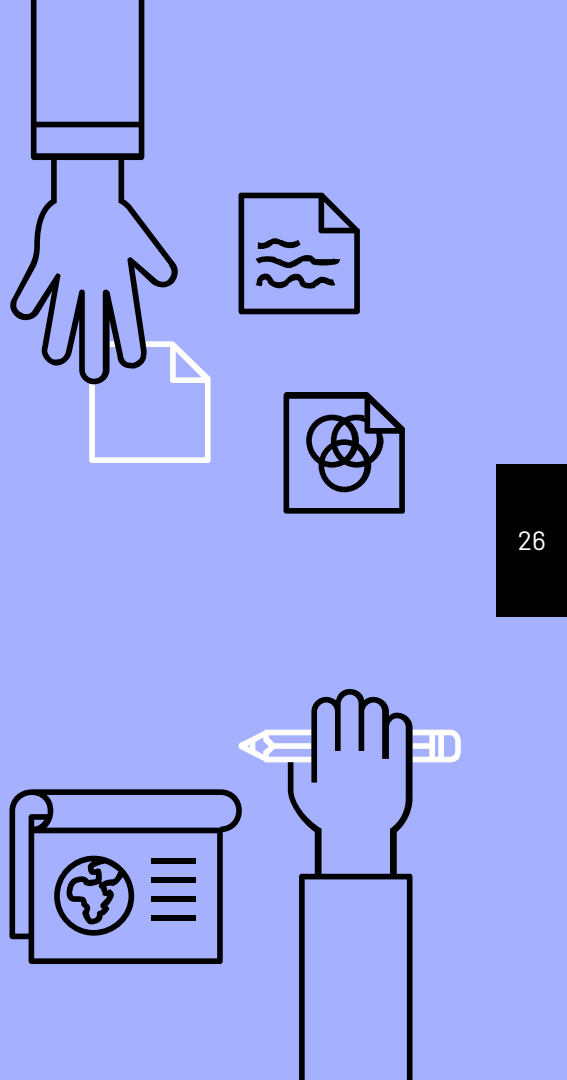


DRY – Não se Repita

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, a fim 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.

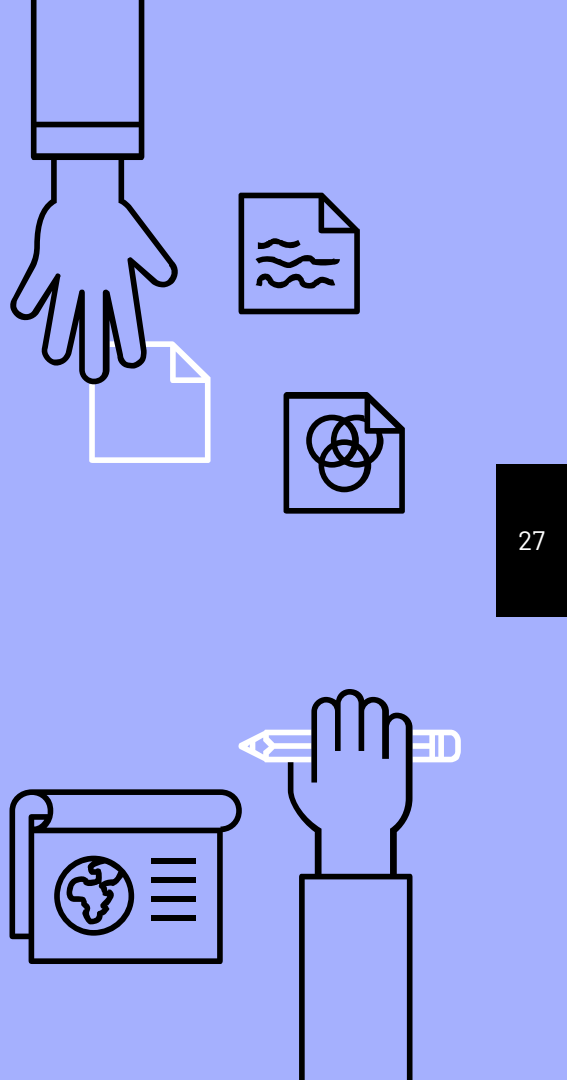


DRY – Não se Repita

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!!!

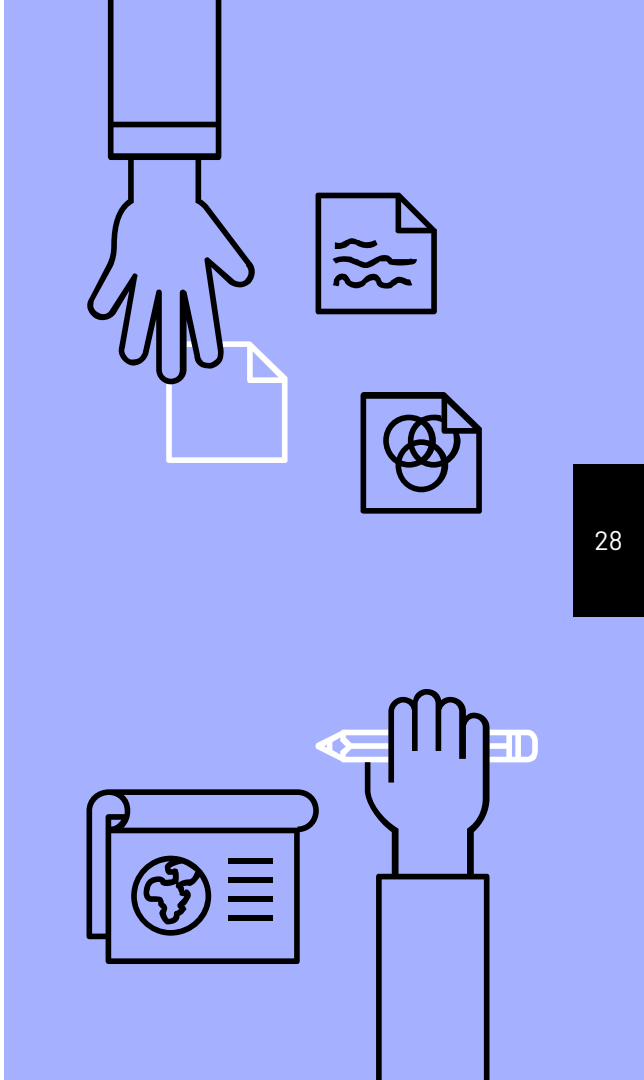


DRY – Não se Repita

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**.

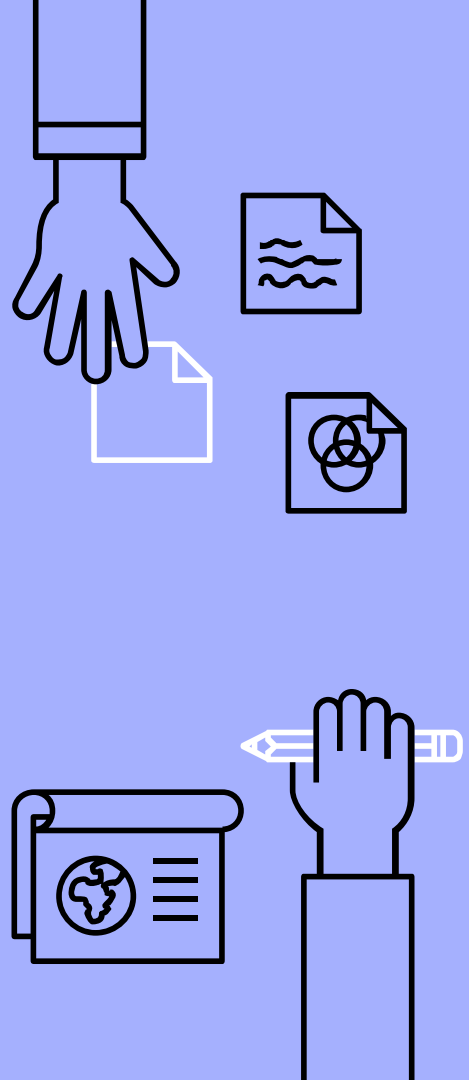


DRY – Não se Repita

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.



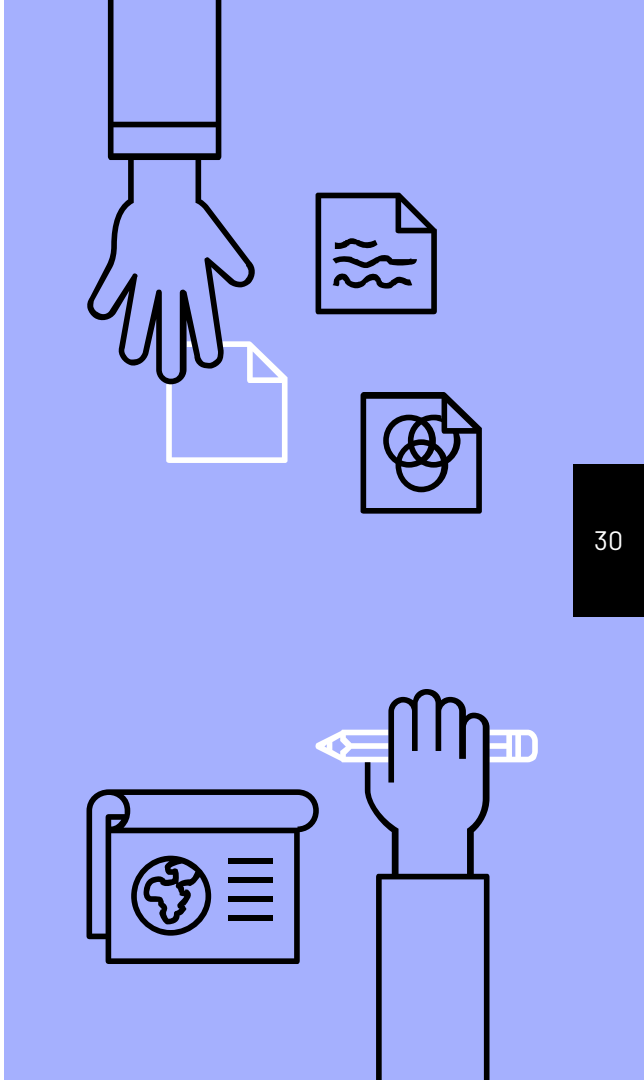
DRY – Não se Repita

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.



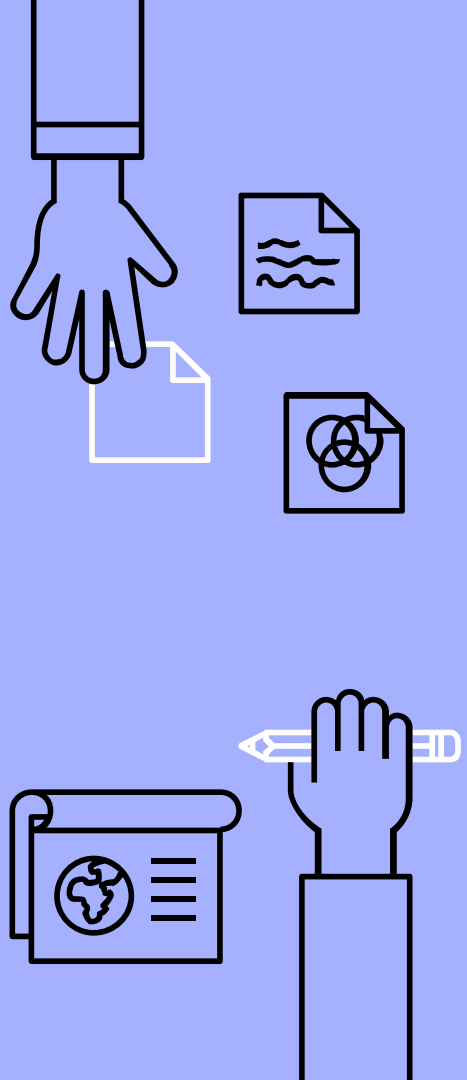
DRY – Não se Repita

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.

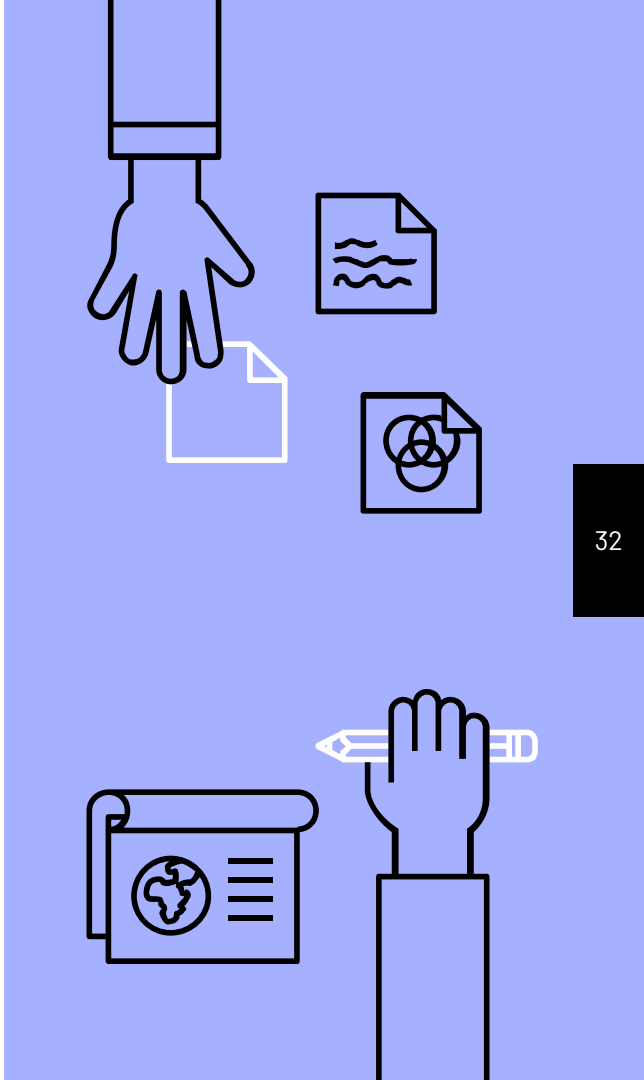


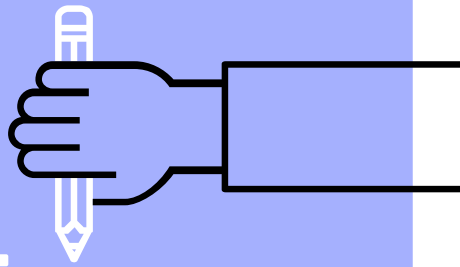
DRY – Não se Repita

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.

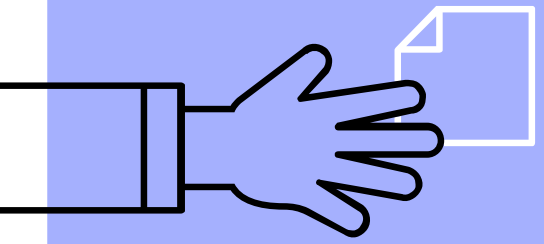
A utilização do DRY evita diversos problemas como manutenção de código desnecessária, aumento de bugs por conta da duplicidade de responsabilidades e até mesmo desempenho da aplicação.





2. YAGNI (You Aren’t Gonna Need It)

Você não vai precisar disto.



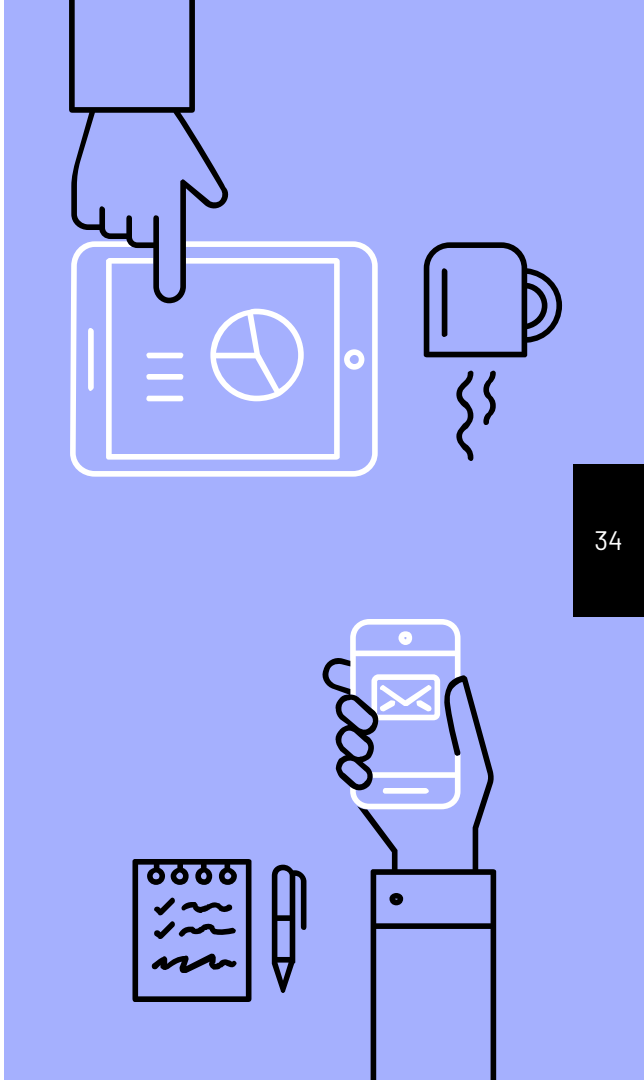
YAGNI - Você não vai precisar disto

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".*

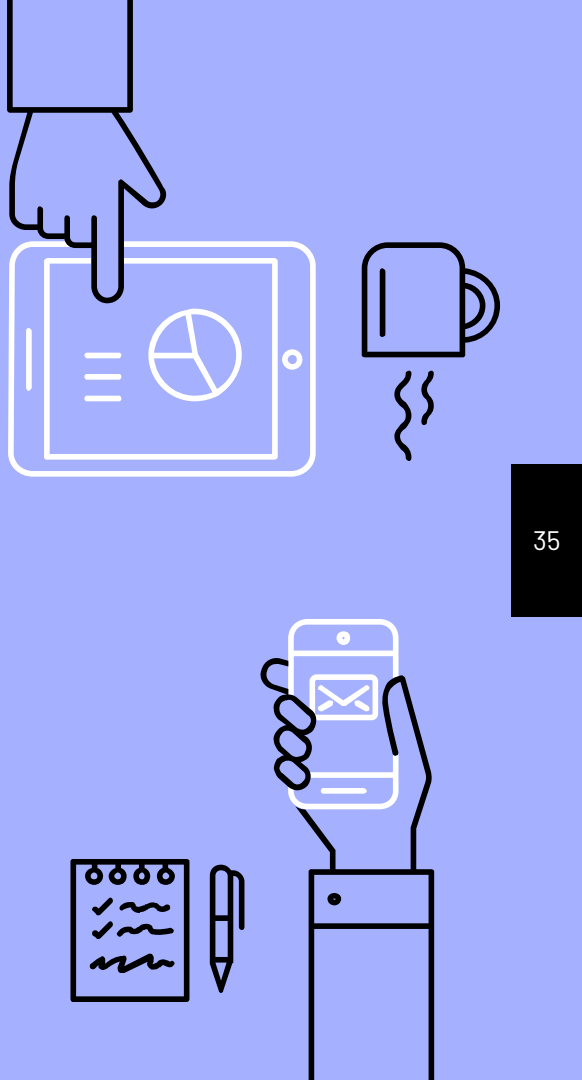


YAGNI - Você não vai precisar disto

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

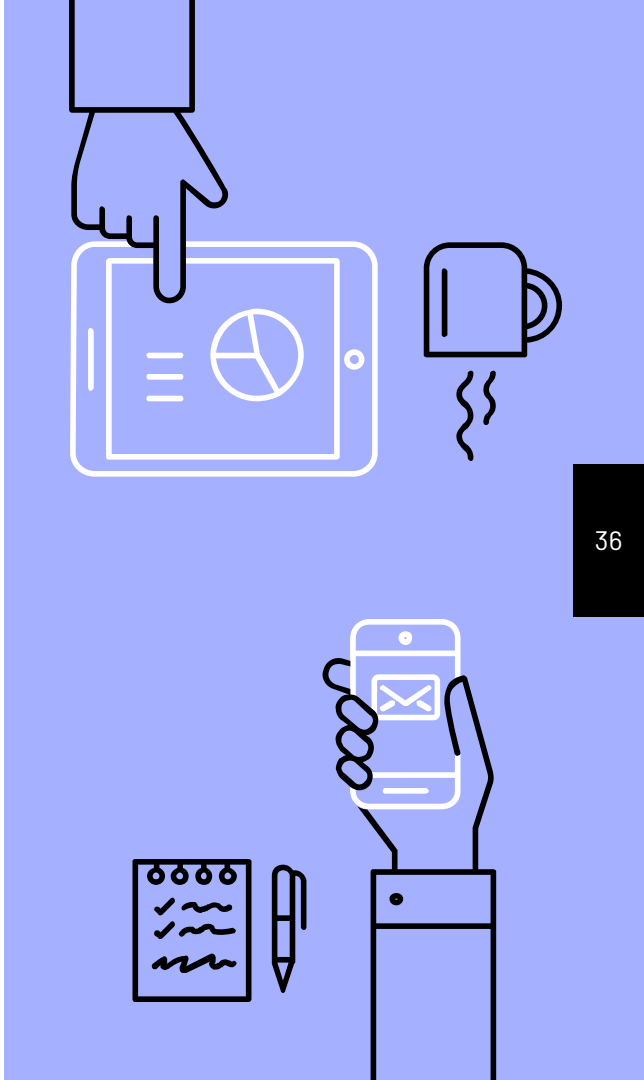
Conclusão

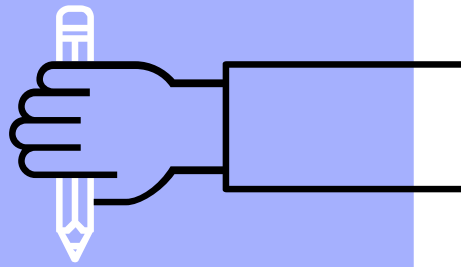
O princípio do desenvolvimento de *software* **YAGNI**, foca nas necessidades reais e já conhecidas. Economizando tempo, esforço e custos para apenas aquilo que de fato será utilizado e irá retornar algum valor para empresa ou para o cliente.

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.

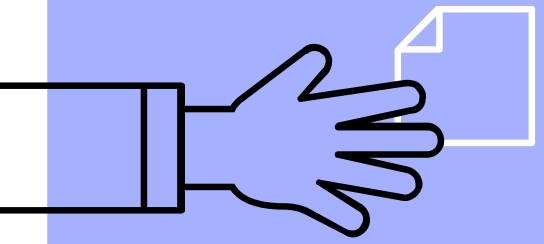
Resista à tentação de codificar agora o que você não precisa, afinal **"You Aren't Gonna Need it"**.





3. KISS (Keep It Simple, Stupid)

Mantenha Estupidamente Simples



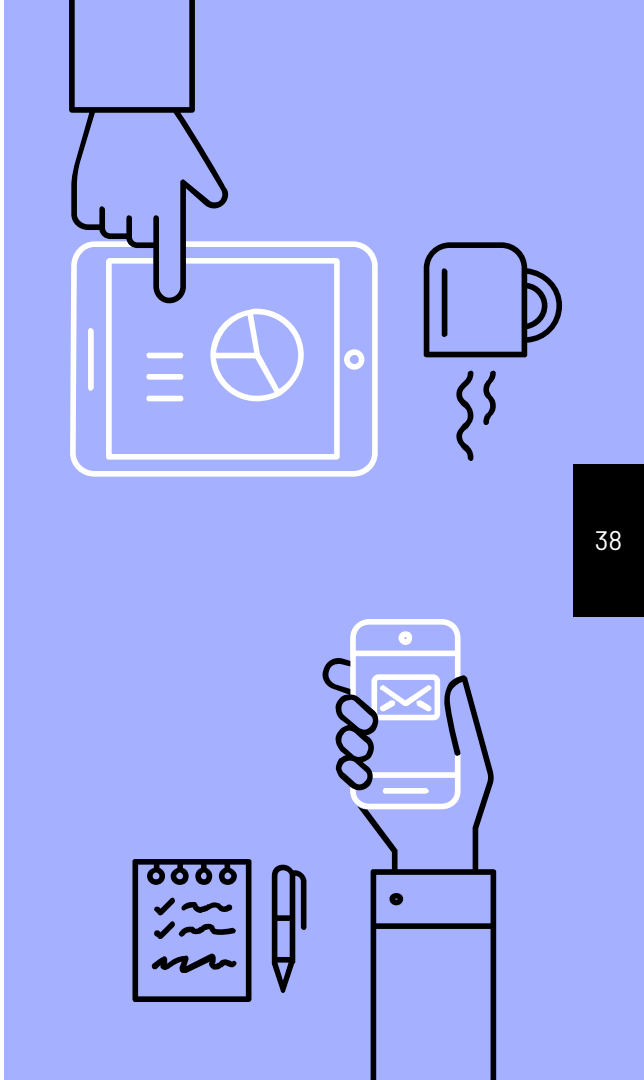
KISS - Mantenha as coisas simples

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!



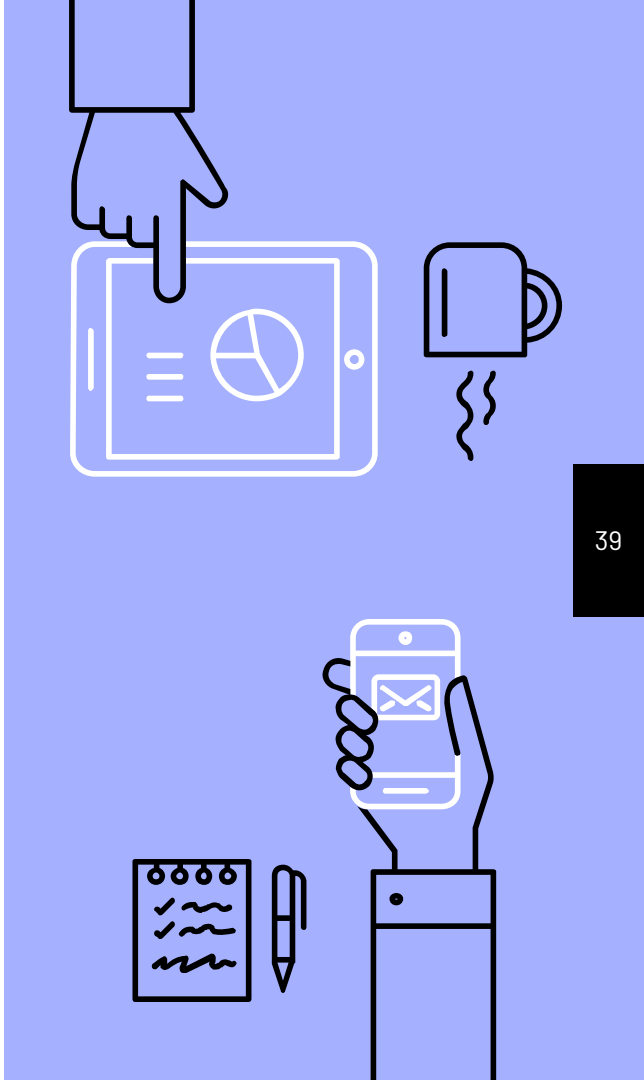
KISS - Mantenha as coisas simples

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".



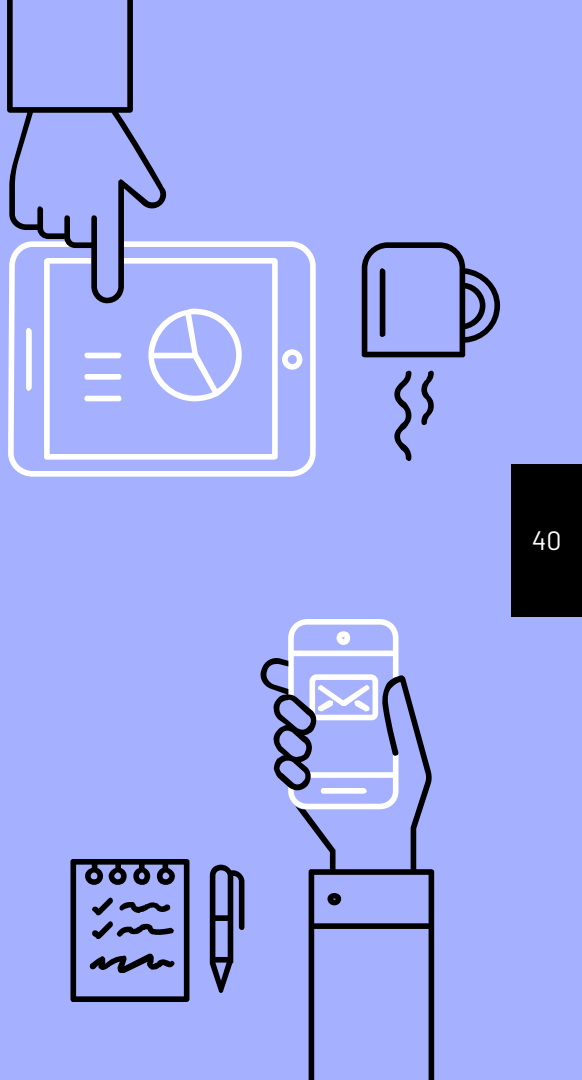
KISS - Mantenha as coisas simples

Como devo aplicar este princípio ?

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 princípio:

- 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.

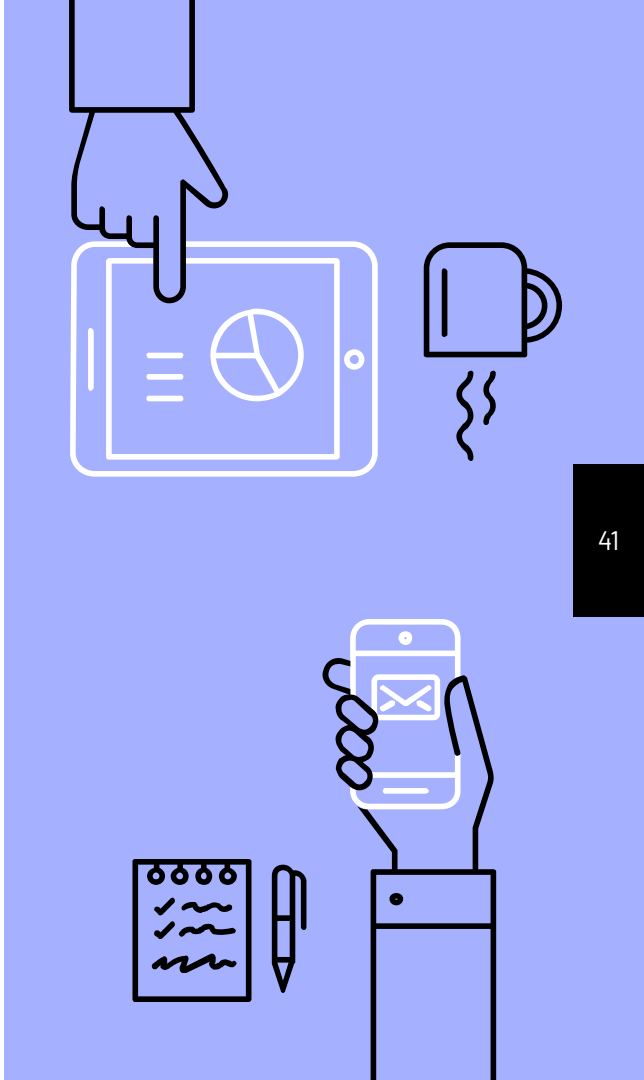


KISS - Mantenha as coisas simples

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

Benefícios ao aplicar esse princípio:

- 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

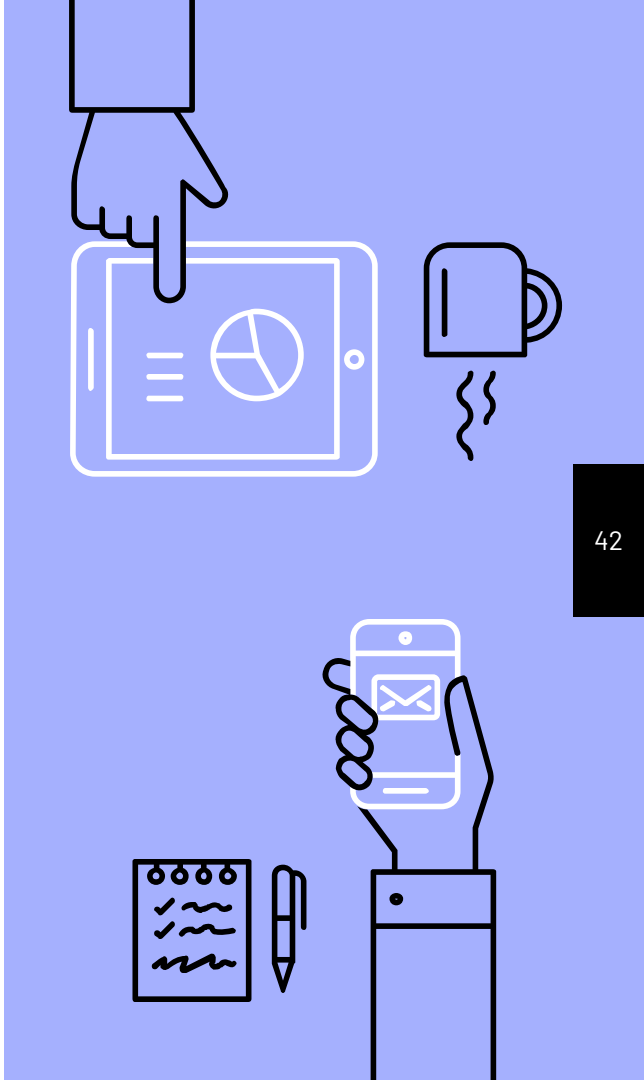
A aplicação deste princípio é um processo que mesmo sendo simples exige paciência. Onde a maior parte dessa paciência será com você mesmo.

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.



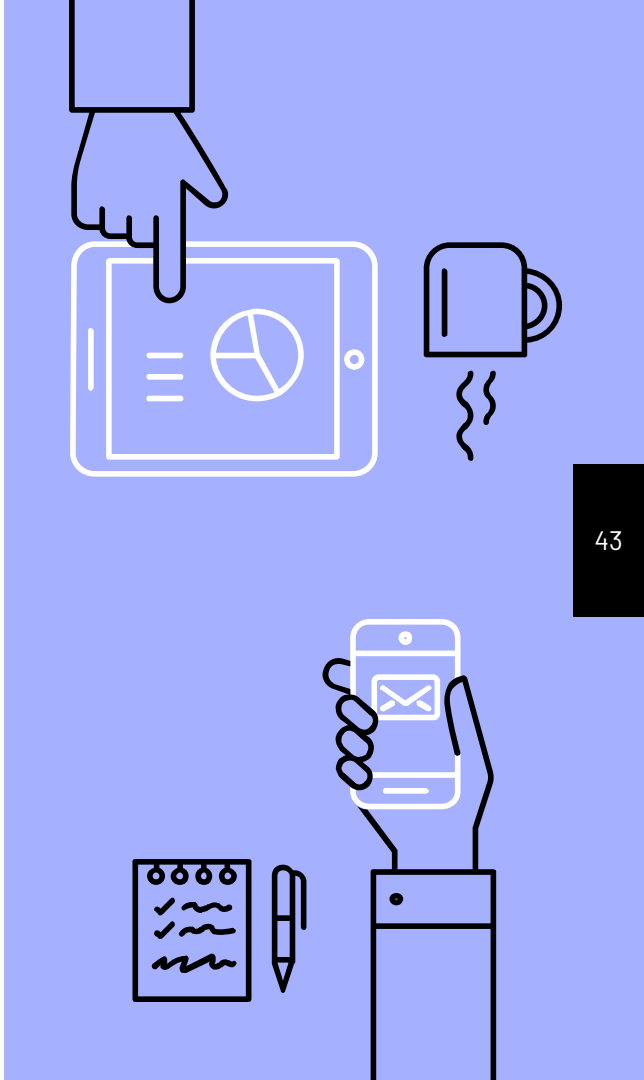
KISS - Mantenha as coisas simples

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.



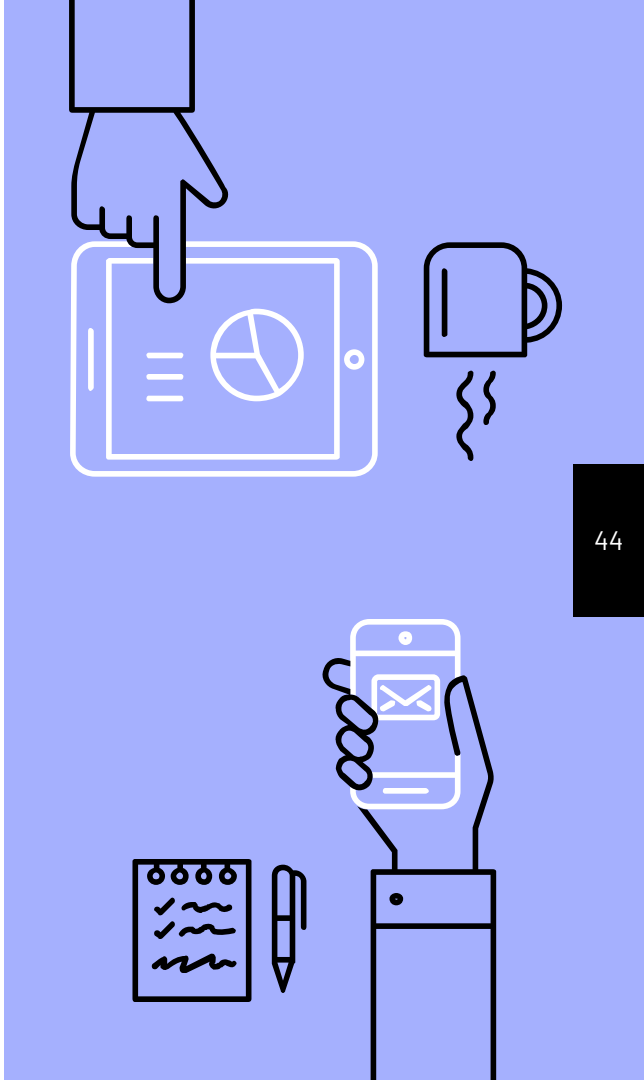
KISS - Mantenha as coisas simples

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.



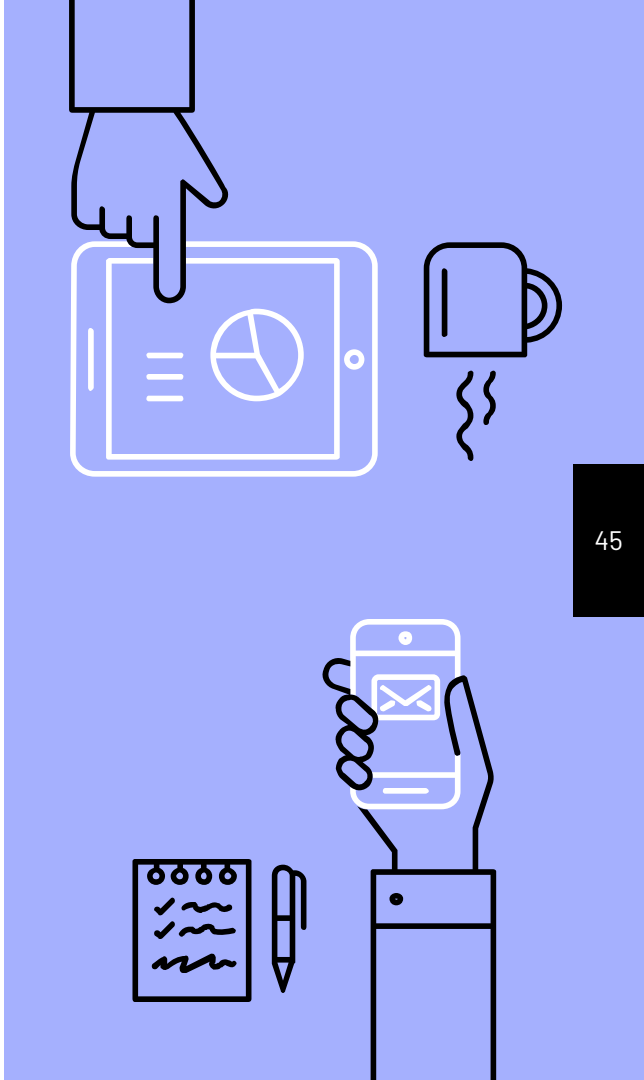
KISS - Mantenha as coisas simples

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

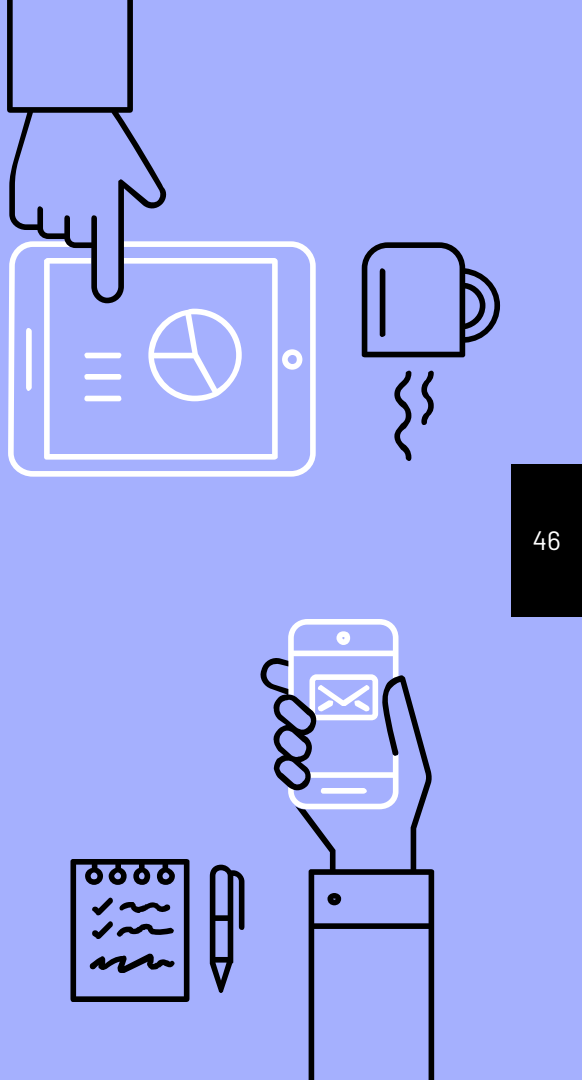
Se você já é veterano de guerra, muitos anos como profissional de desenvolvimento. Então é provável que este conteúdo não tenha lhe falado muito do que você já não sabia.

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;



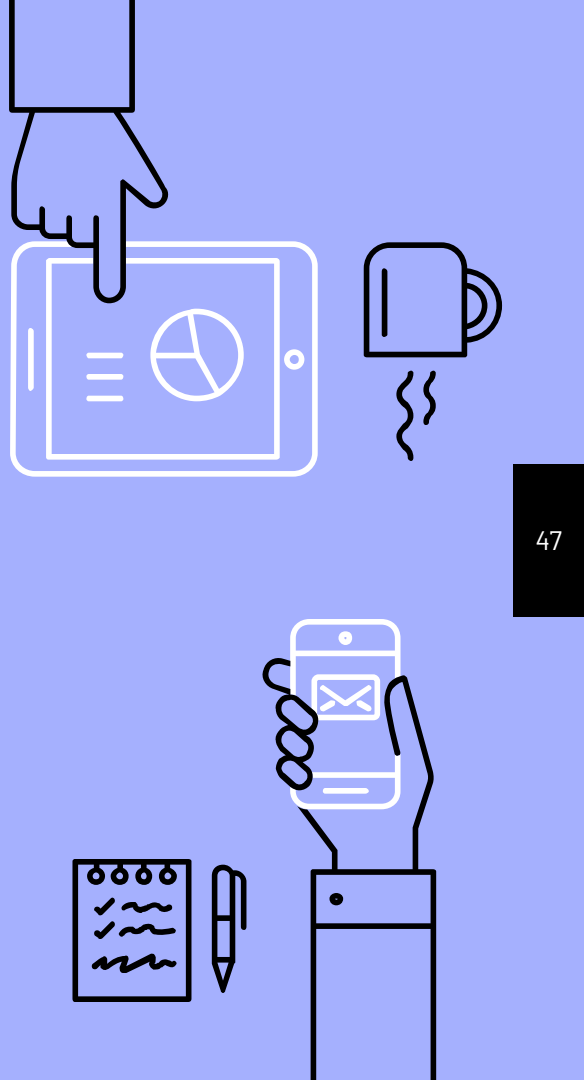
KISS - Mantenha as coisas simples

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:

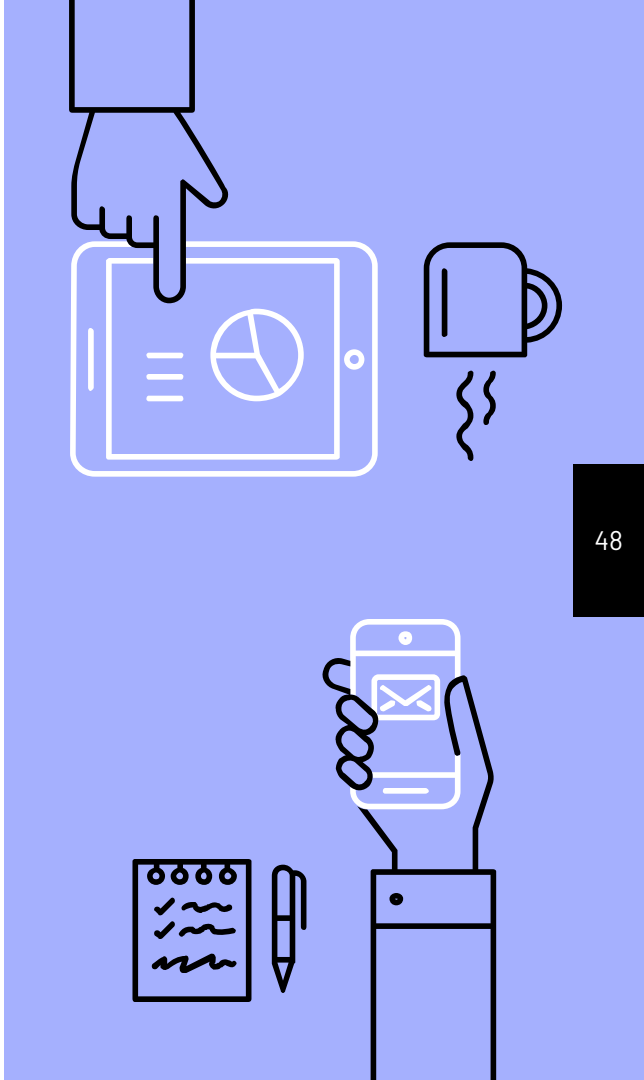
"A proporção de tempo gasto lendo código versus a escrita dele é bem mais do que 10 para 1... portanto tornando-o fácil de ler faz com que seja mais fácil de escrever [refatorar]."

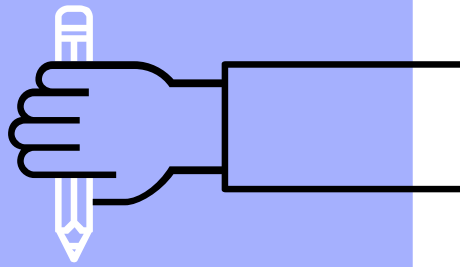


KISS - Mantenha as coisas simples

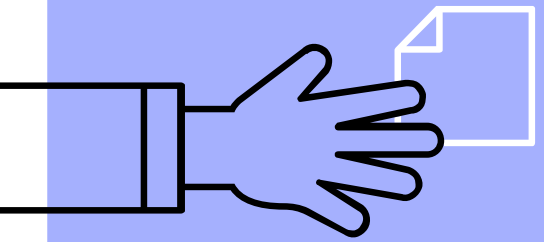
Conclusão

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.





4. LoD (Law of Demeter)



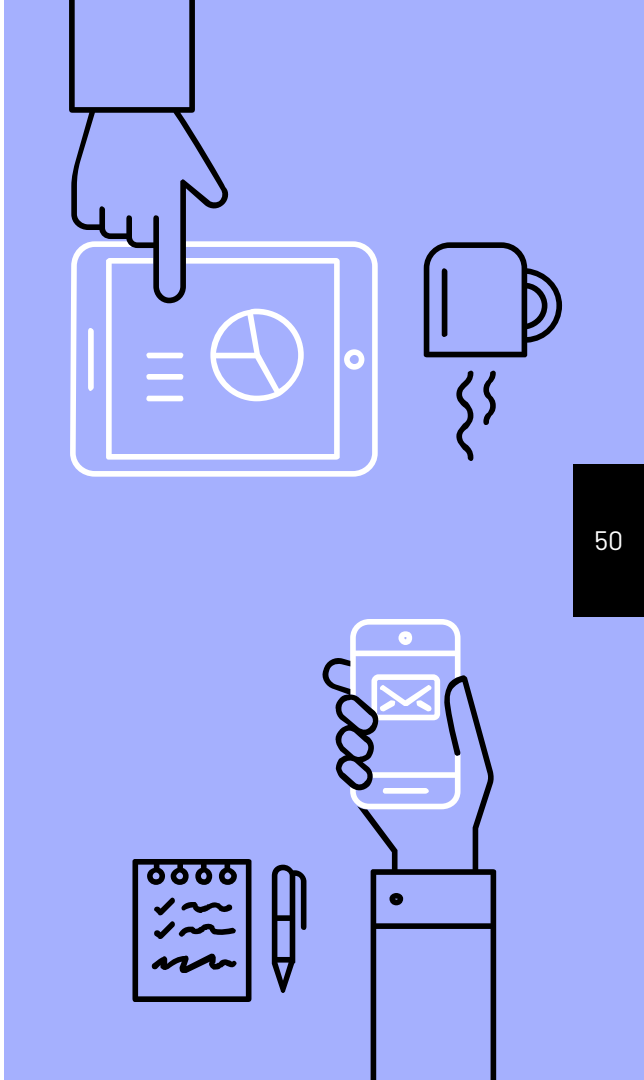
Não Fale com Estranhos

LoD - Não Fale com Estranhos

LoD é um acrônimo que significa **Law of Demeter (Lei de Demeter)**, que é um princípio de programação orientada a objetos que afirma que um objeto deve ter **informações limitadas** sobre outros objetos.

Isso significa que um objeto deve interagir apenas com seus objetos vizinhos imediatos e não com objetos mais distantes. Isso torna o código mais fácil de entender e menos propenso a erros e **reduz o acoplamento entre objetos** em um software.

O nome desse princípio faz referência a um grupo de pesquisa da Northeastern University, em Boston, EUA. Esse grupo, chamado Demeter, desenvolvia pesquisas na área de modularização de software. No final da década de 80, em uma de suas pesquisas, o grupo enunciou um conjunto de regras para **evitar problemas de encapsulamento** em projeto de sistemas orientados a objetos, as quais ficaram conhecidas como Princípio ou Lei de Demeter.

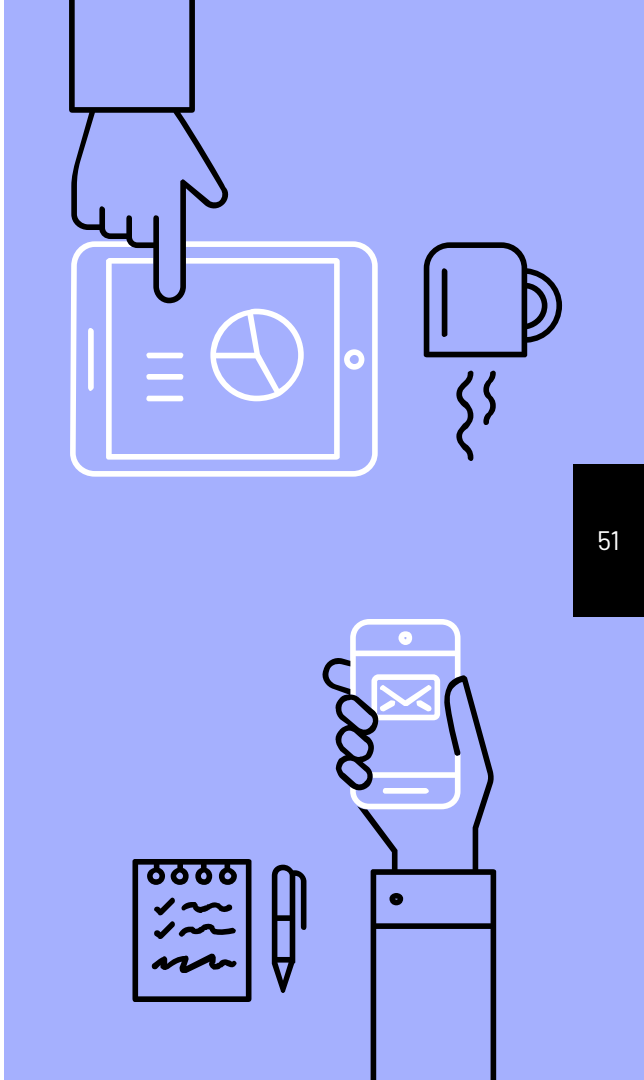


LoD - Não Fale com Estranhos

O Princípio de Demeter — também chamado de **Princípio do Menor Privilégio (Principle of Least Privilege),** defende que a implementação de um método deve invocar apenas os seguintes outros métodos:

- de sua própria classe (caso 1)
- de objetos passados como parâmetros (caso 2)
- de objetos criados pelo próprio método (caso 3)
- de atributos da classe do método (caso 4)

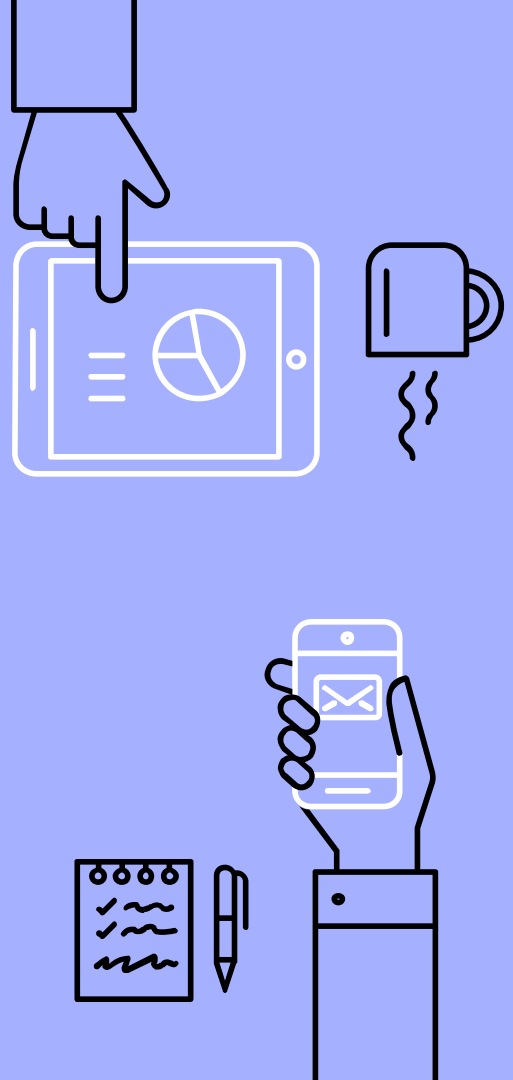
Nesse exemplo, somamos todos os itens de um carrinho de compras, para isso existe a classe **Product** que representa um produto contendo seu nome e valor. Há também a classe Item responsável por compor os itens do carrinho de compras, ela guarda o produto e a quantidade. Por fim, temos a classe **Cart** que representa o carrinho de compras, responsável por somar o valor de todos os itens.



LoD - Não Fale com Estranhos

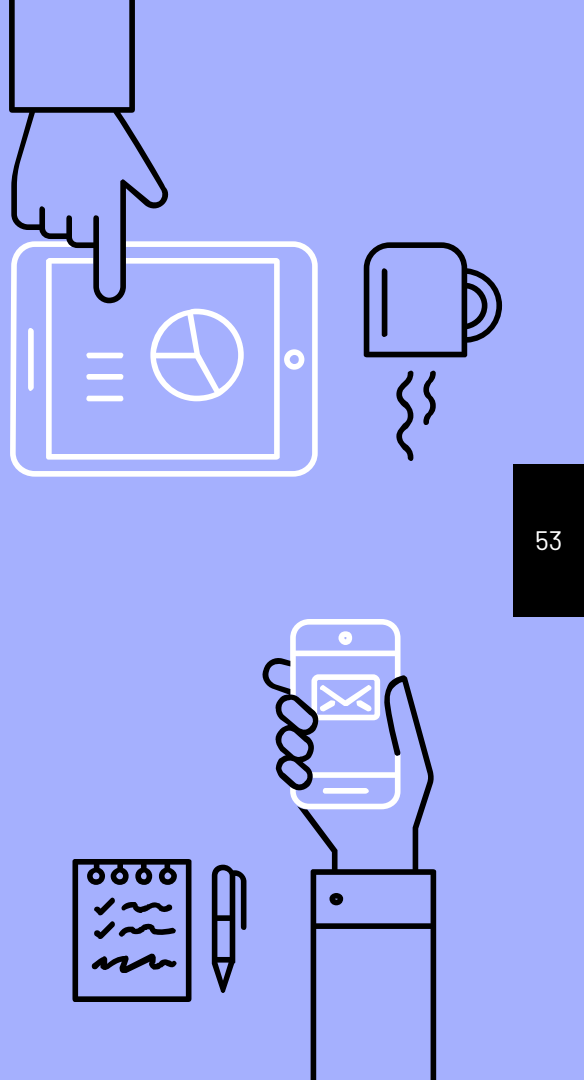
```php
class Product
{
    private $value;
    private $name;

    public function __construct($name, $value) {
      $this->name  = $name;
      $this->value = $value;
    }

    public function getValue() {
      return $this->value;
    }
}
```

```php
class Item
{
    private $product;
    private $quantity;

    public function __construct($product, $quantity) {
      $this->product  = $product;
      $this->quantity = $quantity;
    }

    public function getProduct() {
      return $this->product;
    }
    public function getQuantity() {
      return $this->quantity;
    }
}
```



LoD - Não Fale com Estranhos

```php
class Cart
{
    private $owner;
    private $items;

    public function __construct($owner, $items) {
      $this->owner = $owner;
      $this->items = $items;
    }

    public function totalValueOfItems() {
      $total = 0;
      foreach ($this->items as $item) {
        $total += $item->getProduct()->getValue() * $item->getQuantity();
      }
      return $total;
    }
  }
```



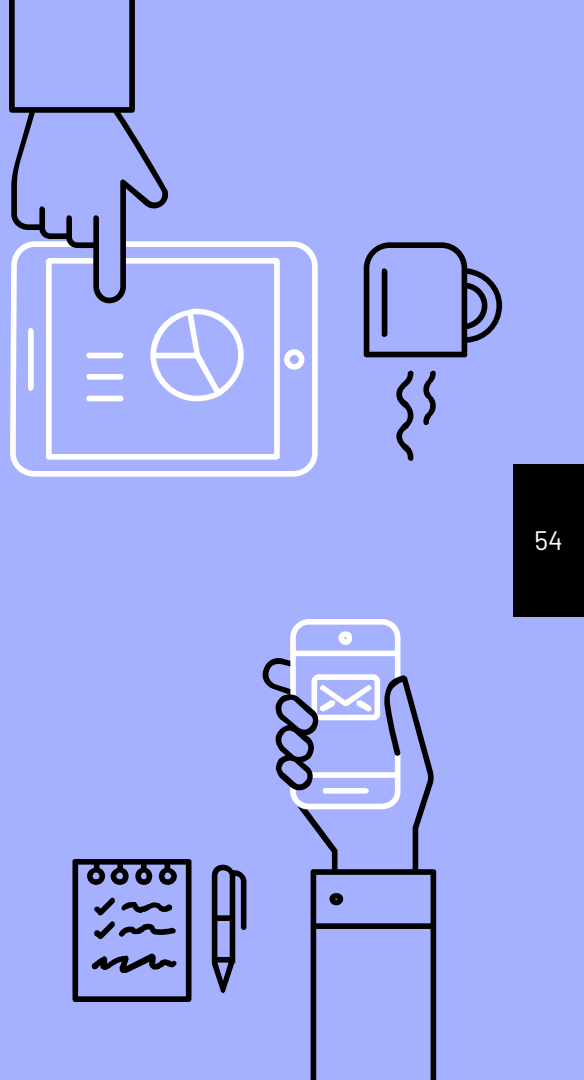
LoD - Não Fale com Estranhos

Embora o código acima parece simples e inofensivo ele apresenta um problema de encapsulamento. Olhando para o método **totalValueOfItems**, podemos notar que a variável total é chamado o produto do item para então chamar seu valor.

```php
public function totalValueOfItems() {
  $total = 0;
  foreach ($this->items as $item) {
    $total += $item->getProduct()->getValue() * $item->getQuantity();
  }
  return $total;
}
```

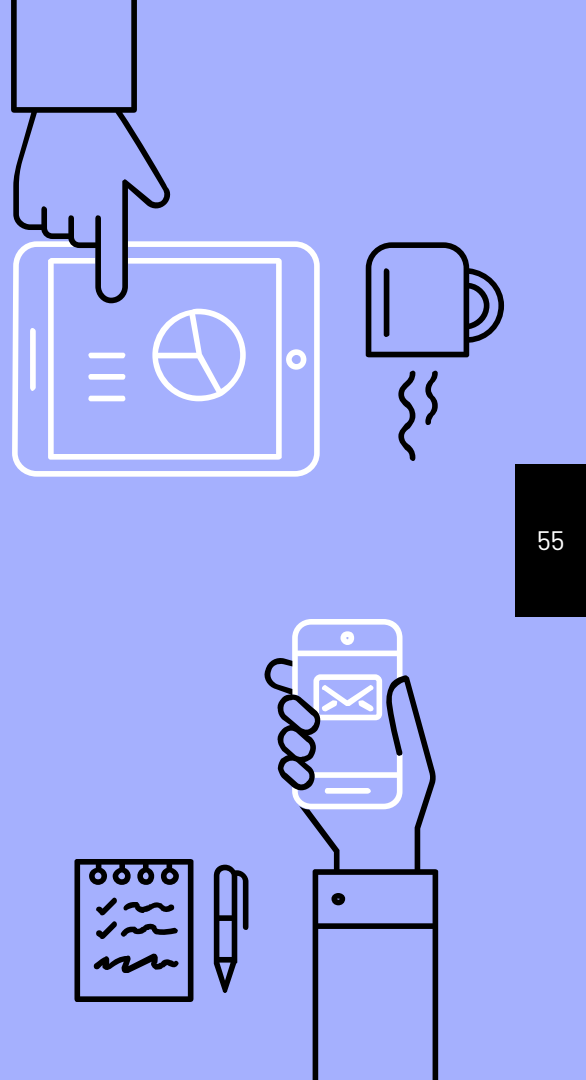
Isso é um problema, porque caso a classe Item mude, o código desse método irá quebrar, e se o projeto não tiver testes para detectar isso? E se códigos assim estiverem espalhados por toda nossa aplicação? É certo o próximo desenvolvedor tem que usar ctrl + F para saber onde deve mudar?

Evitar esses problemas é simples, apenas devemos deixar os donos da informação manipulá-la, para isso vamos refatorar as classes.



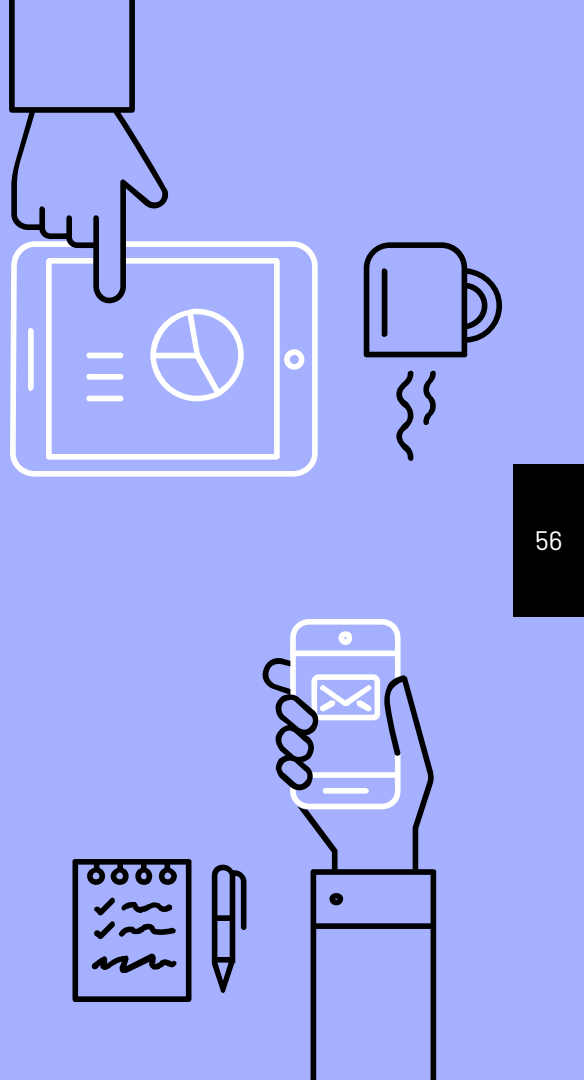
LoD - Não Fale com Estranhos

```php
class Item
{
    private $product;
    private $quantity;

    public function __construct($product, $quantity) {
      $this->product  = $product;
      $this->quantity = $quantity;
    }

    public function getProduct() {
      return $this->product;
    }
    public function getQuantity() {
      return $this->quantity;
    }

    public function totalValue() {
      return $this->product->getValue() * $this->quantity;
    }
}
```



LoD - Não Fale com Estranhos

```php
class Cart
{
    private $owner;
    private $items;

    public function __construct($owner, $items) {
      $this->owner = $owner;
      $this->items = $items;
    }

    public function totalValueOfItems() {
      $total = 0;
      foreach ($this->items as $item) {
        $total += $item->totalValue();
      }
      return $total;
    }
}
```

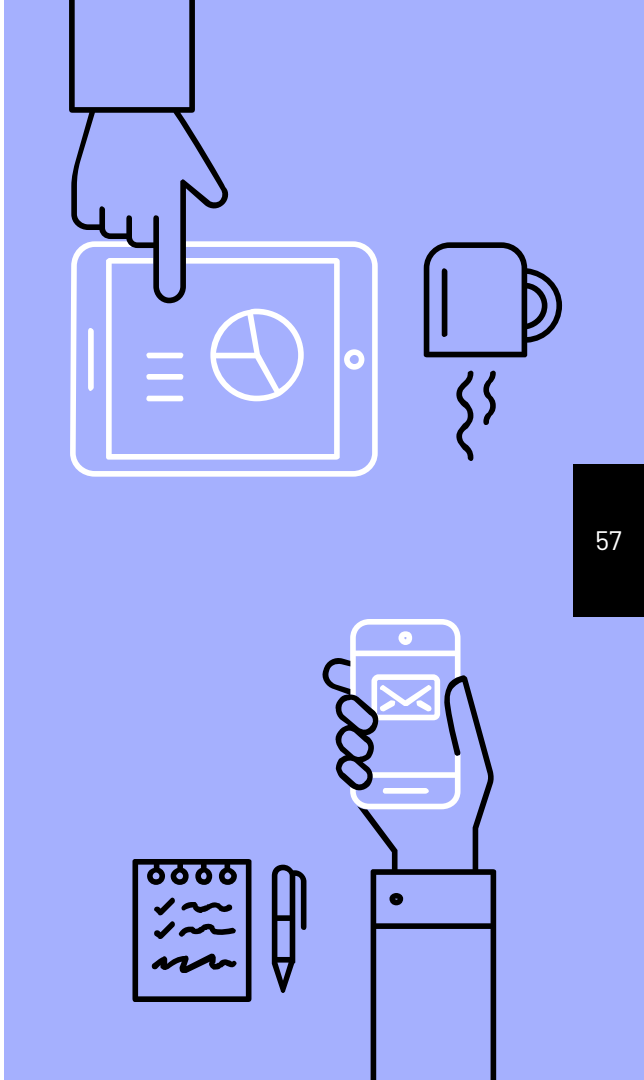


LoD - Não Fale com Estranhos

O que mudou com essa pequena alteração ? Agora, está melhor encapsulado porque a classe que conhece o produto e a quantidade está fazendo essa soma e a classe **Cart** que conhece todos os itens está acumulando o valor para retornar o total no carrinho.

Quando vamos utilizar métodos de outras classes devemos saber sempre o que o método faz e nunca como ele faz. Seguindo isso, evitamos o **code smell** chamado **Feature Envy**.

Feature Envy é um dos elementos que causam esse **"mal cheiro"** e ocorre quando uma classe começa a utilizar, em excesso, as propriedades internas de outra classe. Em outras palavras, a classe não se limita a utilizar as propriedades que possui e acaba por utilizar propriedades externas. Na definição de Uncle Bob, é como se uma classe **"desejasse"** ser outra.



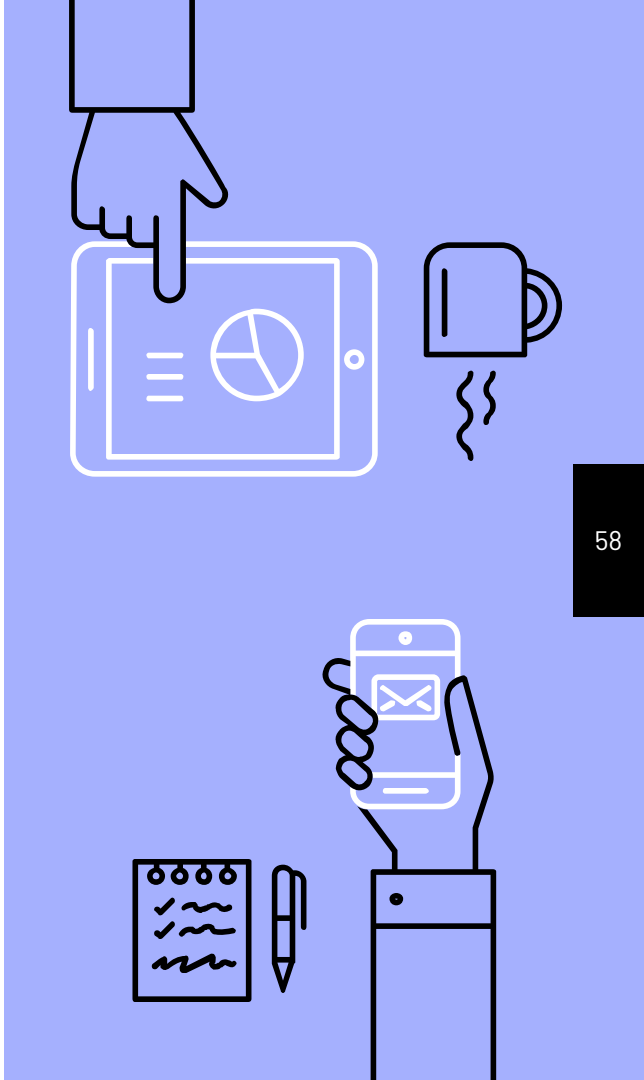
LoD - Não Fale com Estranhos

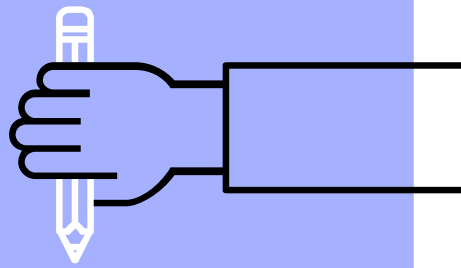
Conclusão

A ideia da Lei de Demeter é **" Não fale com estranhos, converse apenas com seus amigos"**, levando isso em consideração temos na classe Item o método **totalValue** que trabalha com o valor do produto apenas porque ele conhece a classe do produto por isso pode pedir seu valor, com isso mantemos um dos propósitos do **encapsulamento** da Orientação a Objetos. Não conhecemos as propriedades e nem os cálculos que ocorrem dentro do método **totalValue**, mas sabemos que ele irá nos retornar o total do item da venda.

Então se você deparar com código realizando chamadas de métodos encadeadas como: **a.getB().getC().Metodo()**

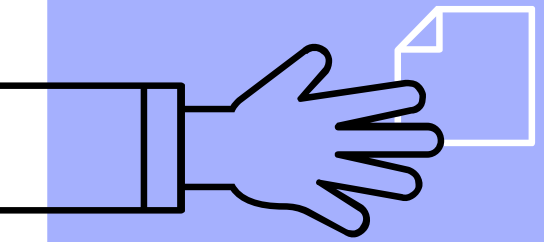
Fique atento e verifique se não seria o caso de aplicar a Lei de Demeter para reduzir o acoplamento e manter o ecapsulamento.





5. Tell, Don’t Ask

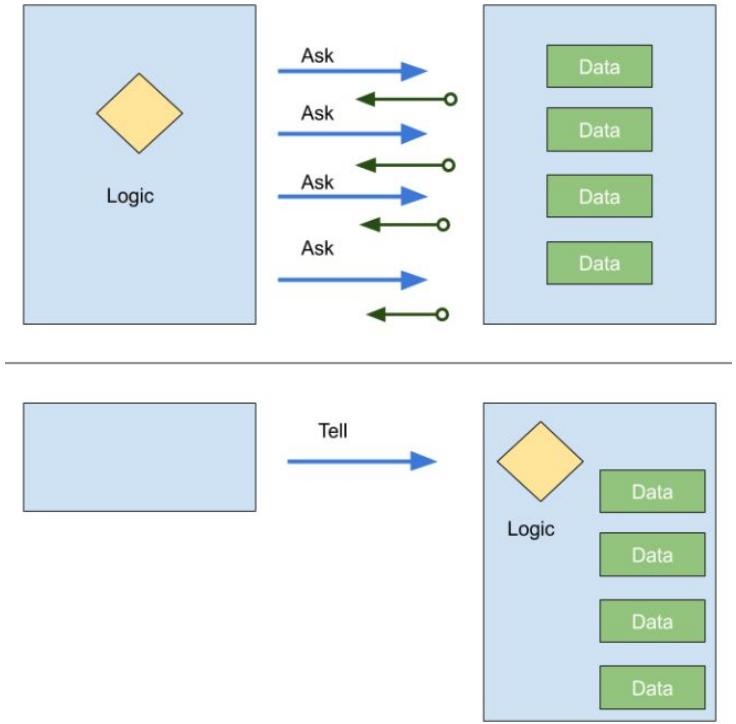
Diga, não pergunte.

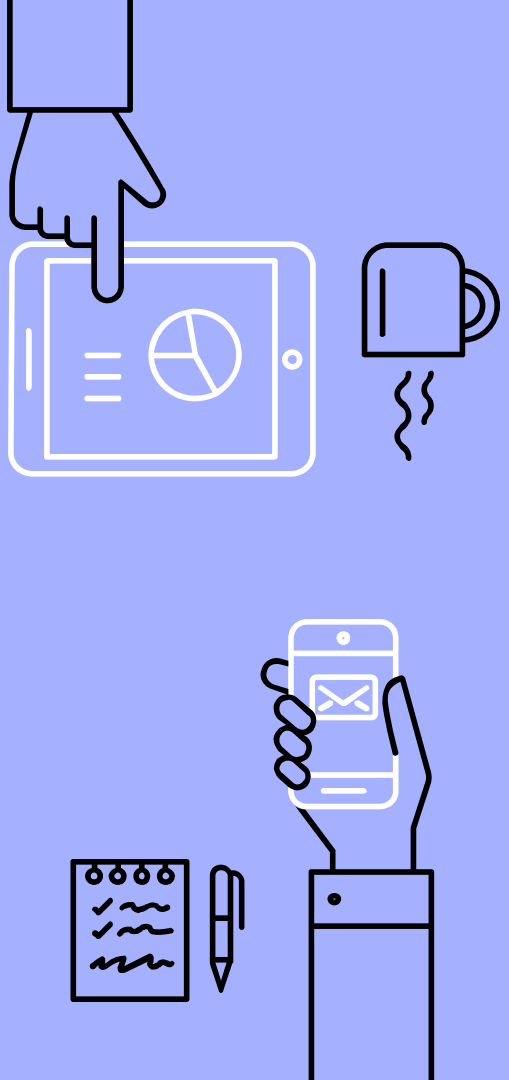


Diga, não pergunte

Um conhecido princípio de Orientação a Objetos é o **Tell, Don't Ask**, ou seja, **"Diga, não pergunte"**.

O princípio nos ajuda a lembrar que orientação a objetos é sobre encapsulamento de dados e sobre operações sobre esses dados. Isso nos lembra que, em vez de solicitar dados a um objeto e agir com base nesses dados, **devemos dizer ao objeto o que fazer**. Isso incentiva a mover o comportamento para um objeto para acompanhar os dados.

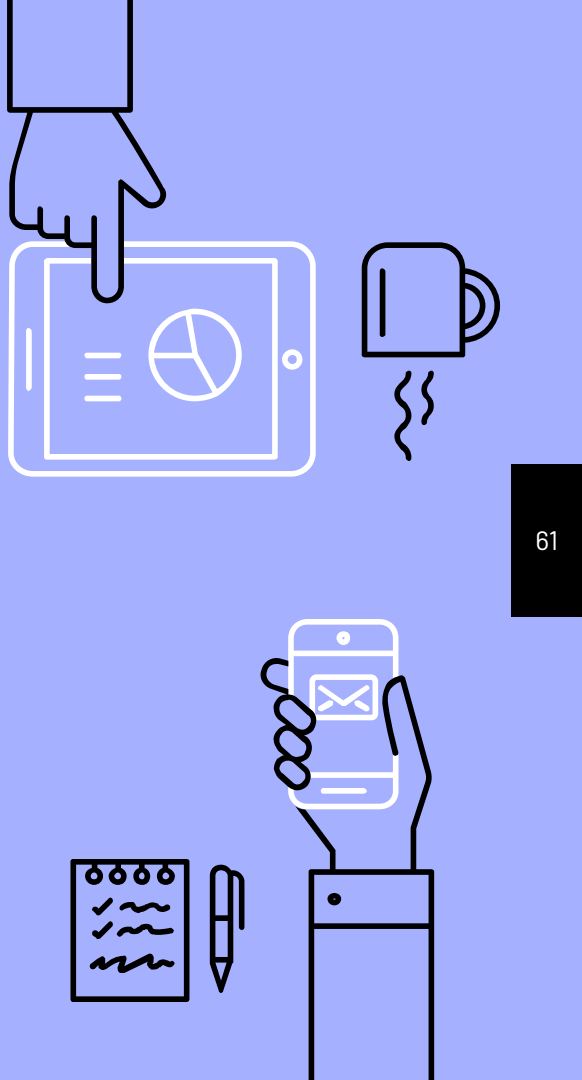




Diga, não pergunte

Mas, como assim, diga e não pergunte? Veja o código abaixo, perceba que a primeira coisa que fazemos para o objeto é uma pergunta (ou seja, um if) e, de acordo com a resposta, damos uma ordem para esse objeto: ou calcula o valor de um jeito ou calcula de outro.

```
NotaFiscal nf = new NotaFiscal();
double valor;

if (nf.getValorSemImposto() > 10000)
{
    valor = 0.06 * nf.getValor();
}
else
{
    valor = 0.12 * nf.getValor();
}
```



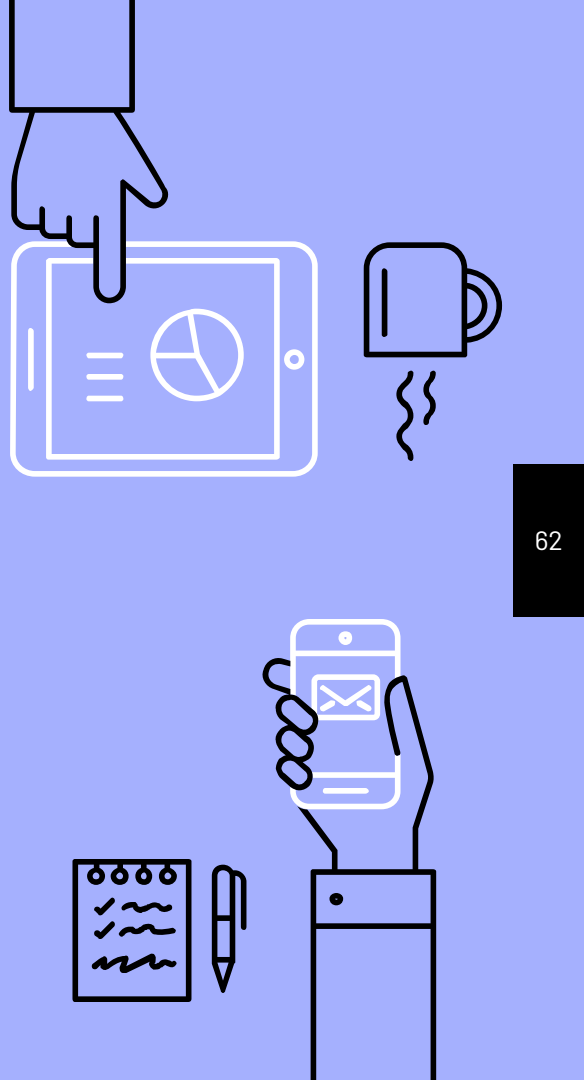
Diga, não pergunte

Quando temos códigos que perguntam uma coisa para um objeto, para então tomar uma decisão, é um código que não está seguindo o Tell, Don't Ask.

A ideia é que devemos sempre dizer ao objeto o que ele tem que fazer, e não primeiro perguntar algo a ele, para depois decidir. O código que refatoramos faz isso direito. Perceba que estamos dando uma ordem ao objeto: calcule o valor do imposto. Lá dentro, obviamente, a implementação será o if anterior, não há como fugir disso. Mas ele está **encapsulado** no objeto:

```
NotaFiscal nf = new NotaFiscal();
double valor = nf.calculaValorImposto();
```

Códigos como esse, que perguntam para depois tomar uma decisão, tendem a ser procedurais. Em linguagens como C, por exemplo, não há muito como fugir disso. Isso é programação procedural. No mundo OO, devemos o tempo todo dar ordens aos objetos. A partir do momento em que perguntamos primeiro para tomar uma decisão, provavelmente estamos furando o encapsulamento.



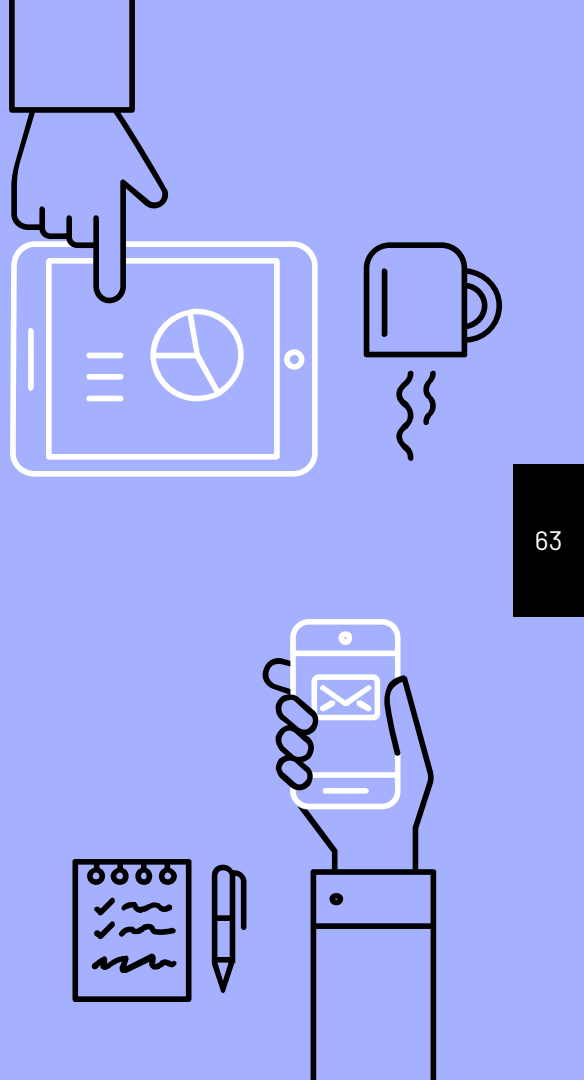
Diga, não pergunte

Procurando por encapsulamentos problemáticos

Perceber se um código está bem encapsulado ou não, não é tão difícil. Olhe, por exemplo, para o código da nota fiscal refatorado. Agora se pergunte: O que esse método faz? Provavelmente sua resposta será: eu sei o que o método faz pelo nome dele, calcula o valor do imposto. É um nome bem semântico, deixa claro que ele faz. Se você conseguiu responder essa pergunta, está no bom caminho.

A próxima pergunta é: Como ele faz isso? Sua resposta provavelmente é: se eu olhar só para esse código, não dá para responder. Não dá para dizer qual é a regra que ele está usando por debaixo dos panos. Não sabemos a implementação do **calculaValorImposto()**. Isso, na verdade, é uma coisa boa. Isso é encapsulamento.

Um exemplo bastante comum de código bem encapsulado e isso as pessoas acertam na maioria das vezes nos códigos OO são os DAOs. O DAO é aquela classe onde escondemos todo o código de acesso aos dados; na maioria das vezes, persistido em um banco de dados. O código a seguir mostra um exemplo de uso de um DAO convencional:

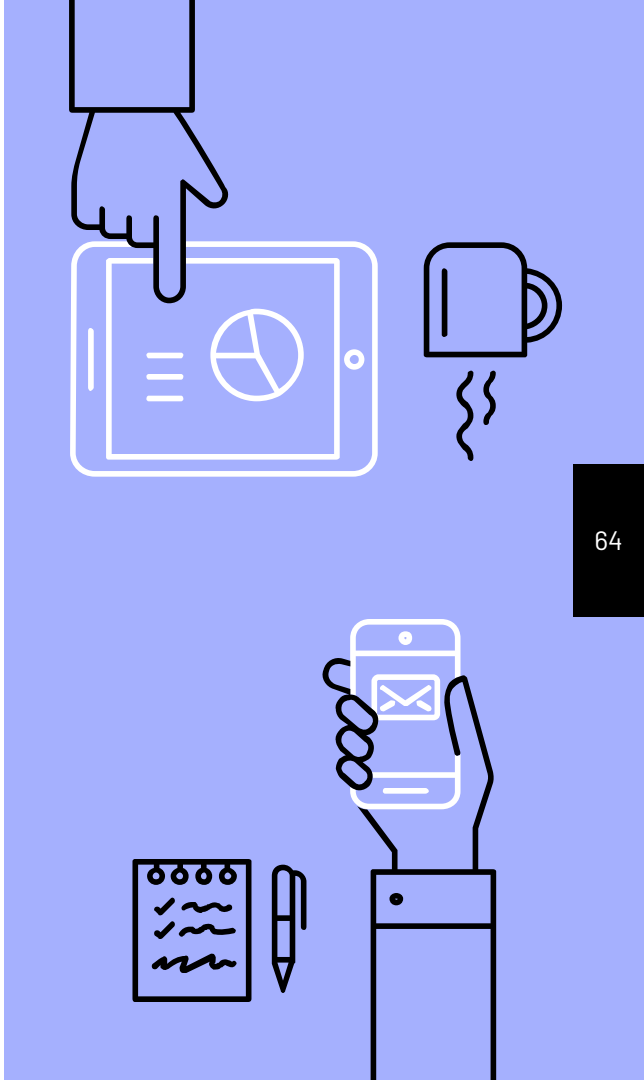


Diga, não pergunte

O que o método **pegaTodos** faz? Pega todas as notas fiscais. Como ele faz? Não sabemos! Não sabemos se vem de um banco de dados, se vem um serviço web, se ele está lendo um arquivo texto. Tudo isso está escondido dentro da implementação do método. Encapsulado.

```
NotaFiscalDao dao = new NotaFiscalDao();
List<NotaFiscal> notasFiscais = dao.pegaTodos();
```

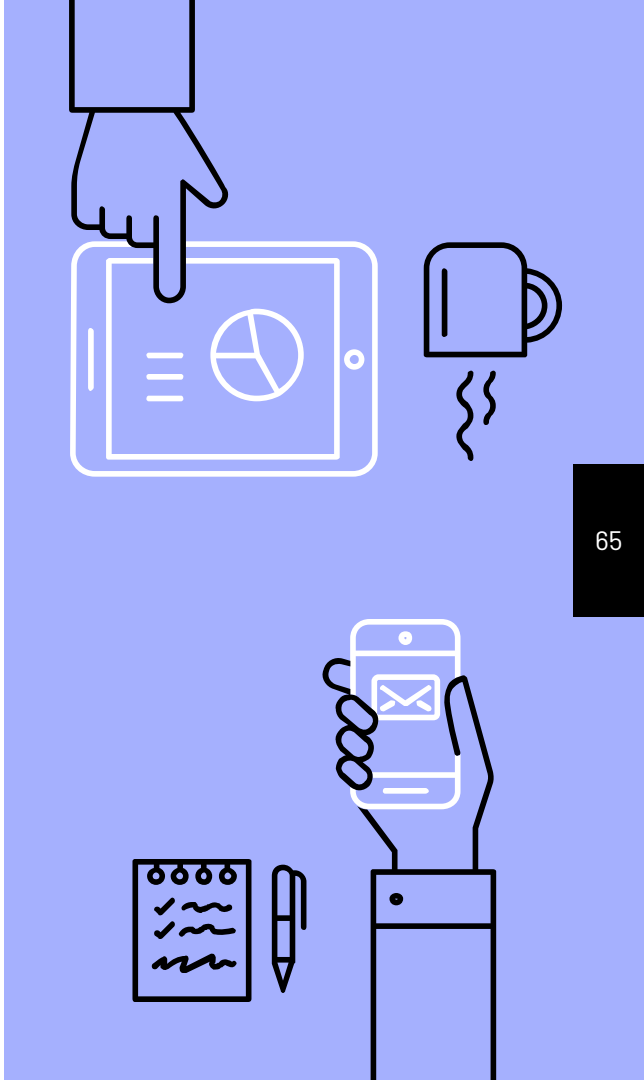
Uma linha de raciocínio bastante interessante na hora de se programar OO é não pensar só na implementação daquela classe, mas também nas classes clientes, que a consumirão. Novamente, é isso que geralmente faz um sistema difícil de se manter. Sistemas difíceis de se manter são aqueles que não pensam na propagação de mudança. É um sistema em que você, para fazer uma mudança, tem que mudar em 10 pontos diferentes. Códigos bem encapsulados geralmente resolvem esse tipo de problema, porque você muda em um lugar e a mudança se propaga.



Diga, não pergunte

Vantagens do Tell Don't Ask

- Favorecer o encapsulamento
- Evitar modelo anêmico
- Evita repetição de código



Diga, não pergunte

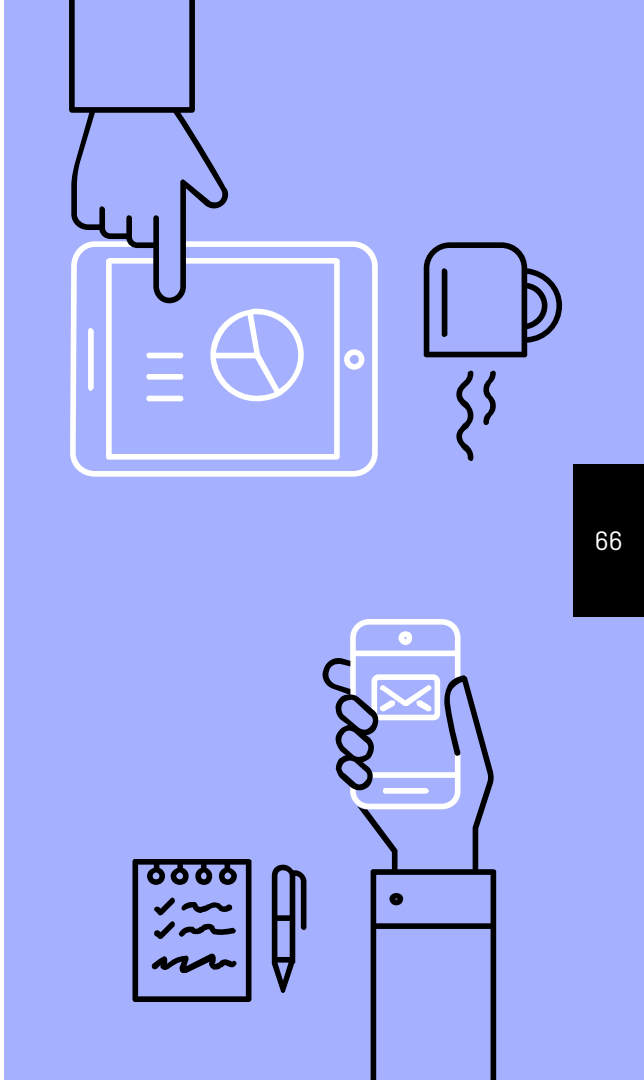
Conclusão

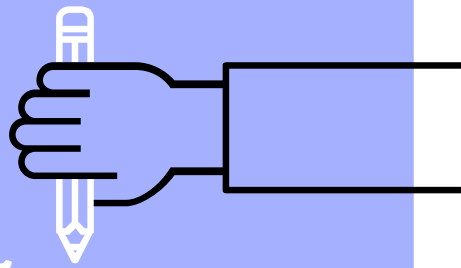
Esse é o conceito **Tell, Don't Ask** onde deixamos toda a responsabilidade para objeto executar a lógicas de negócio e nós apenas solicitar a ação. Códigos que não consideram o "Tell, don't ask" são códigos PROCEDURAIS. Um objeto de verdade possui dados e comportamento para manipular esses dados (princípio básico de orientação a objetos).

Código procedural não expressa claramente que ação está sendo executada. Ao invés de algo objetivo como **"calculaValorImposto"**, podemos ter uma série de condições e atribuições em cima de um ou mais dados do objeto. Além disso, damos margem para que uma mesma lógica se repita em mais de um lugar do software. E sabemos que duplicação de código é do mal!

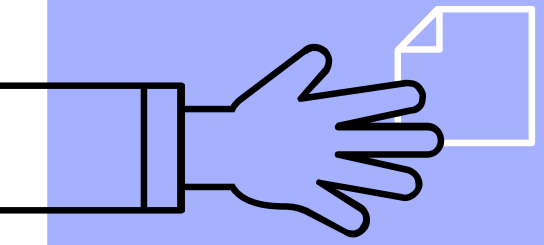
Esses problemas são potencializados a medida que a base de código cresce e se torna mais complexa. Centralizando dados e comportamento em um mesmo lugar, evitamos os problemas acima e facilitamos a manutenção do código.

RESUMINDO TUDO... Diga ao objeto o que você quer que ele faça!





6. Prefira Composição a Herança



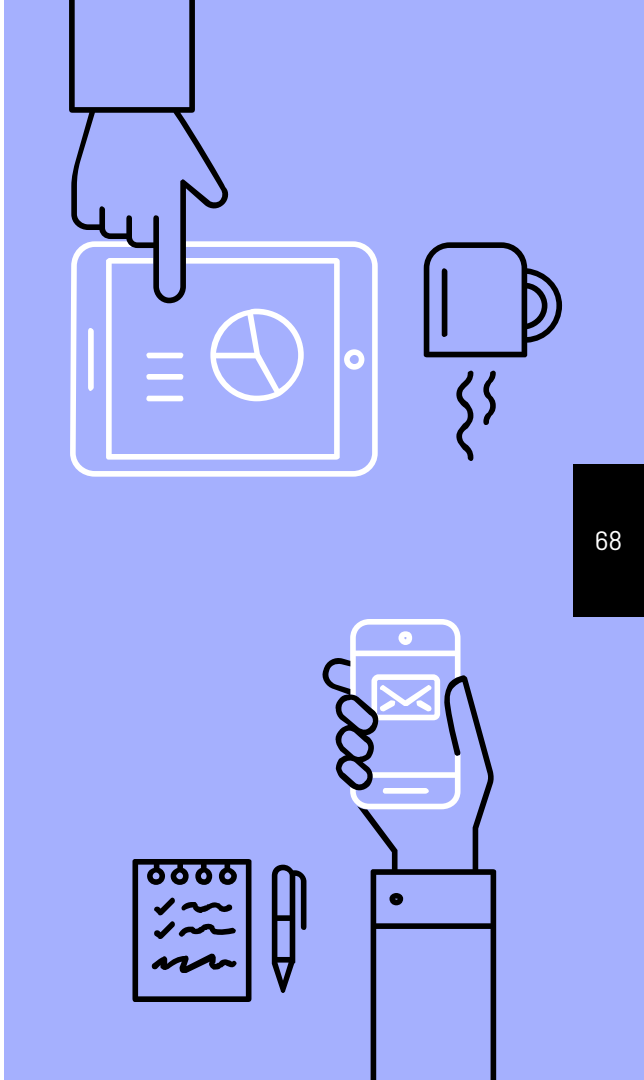
Comportamento em tempo de execução

Prefira Composição a Herança

Antes de explicar o princípio, vamos esclarecer que existem dois tipos de herança:

- **Herança de classes** (exemplo: class A extends B), que é aquela que envolve reúso de código. Não apenas nesta sessão, mas em todo o conteúdo, quando mencionarmos apenas o termo herança estaremos nos referindo a herança de classes.

- **Herança de interfaces** (exemplo: interface I extends J), que não envolve reúso de código. Essa forma de herança é mais simples e não suscita preocupações. Quando precisarmos de nos referir a ela, iremos usar o nome completo: herança de interfaces.

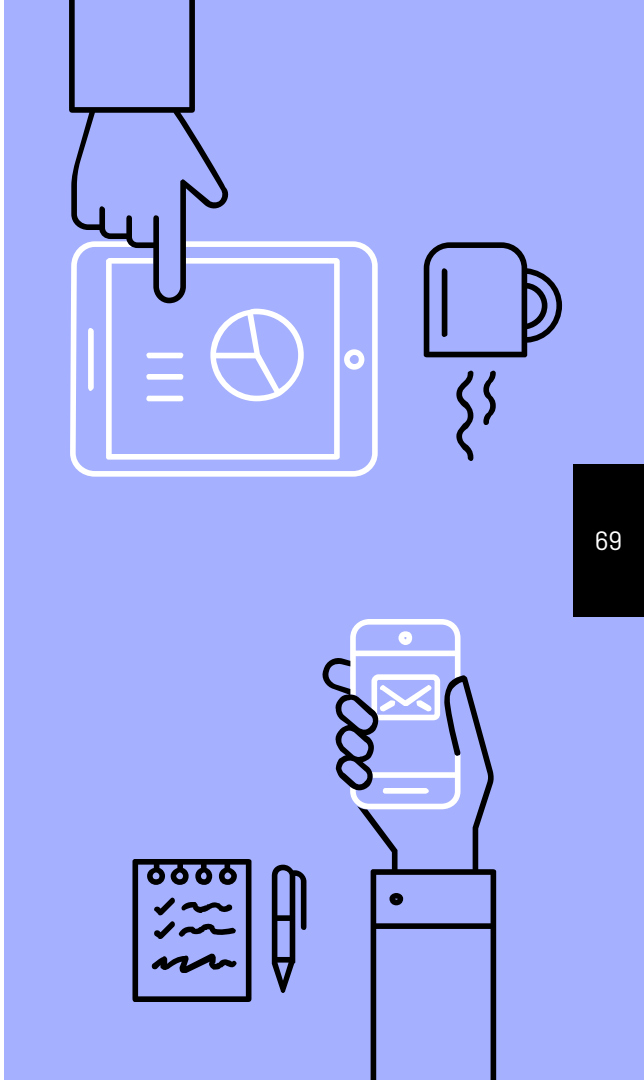


Prefira Composição a Herança

Voltando ao princípio, quando orientação a objetos se tornou comum, na década de 80, houve um incentivo ao uso de herança. Acreditava-se que o conceito seria talvez uma bala de prata capaz de resolver os problemas de reúso de software. Argumentava-se que hierarquias de classes profundas, com vários níveis, seriam um indicativo de um bom projeto, no qual foi possível atingir elevados índices de reúso.

No entanto, com o tempo, percebeu-se que herança não era a tal bala de prata. Pelo contrário, herança tende a introduzir problemas na manutenção e evolução das classes de um sistema. Esses problemas têm sua origem no forte acoplamento que existe entre subclasses e superclasses.

Herança expõe para subclasses detalhes de implementação das classes pai. Logo, frequentemente diz-se que herança **viola o encapsulamento** das classes pai. A implementação das subclasses se torna tão **acoplada** à implementação da classe pai que qualquer mudança nessas últimas pode forçar modificações nas subclasses.



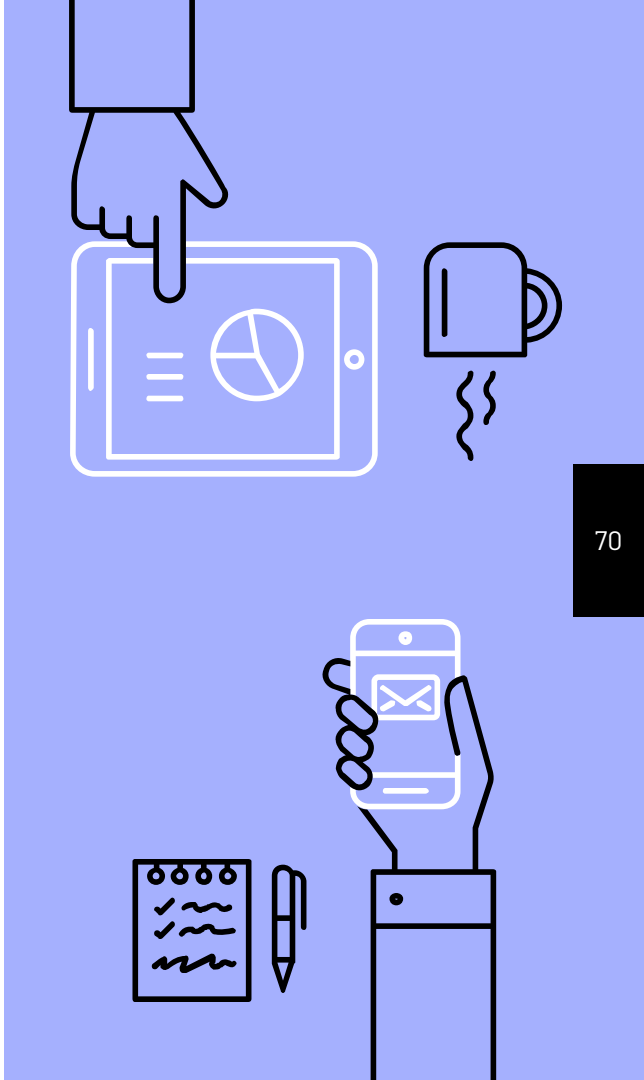
Prefira Composição a Herança

O princípio, porém, **não proíbe o uso de herança**. existem alguns cenários em que devemos considerar o uso de herança:

- Uma classe ser utilizada como **estrutura** para outras **classes** ou **camadas.**
- Se a hierarquia de herança representar um relacionamento **"É UM"** e não relacionamento **"TEM UM".**
- Precisa fazer alterações globais em classes filhas.

Mas ele recomenda: se existirem duas soluções de projeto, uma baseada em herança e outra em composição, a solução por meio de composição, normalmente, é a melhor.Só para deixar claro, existe uma relação de composição entre duas classes A e B quando a classe A possui um atributo do tipo B.

Exemplo: Suponha que temos que implementar uma classe **Stack**. Existem pelo menos duas soluções, por meio de herança ou por meio de composição, conforme mostra o seguinte código:



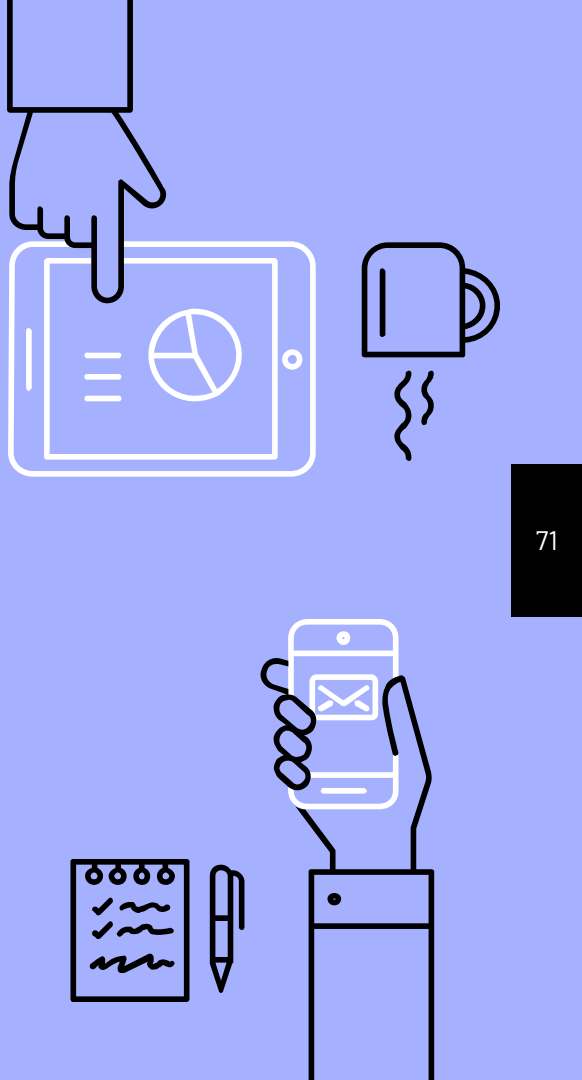
Prefira Composição a Herança

Solução Via Herança

```
class Stack extends ArraList
{
    // Implementação da Classe
}
```

Solução Via Composição

```
class Stack
{
    private ArraList elementos;
    // Implementação da Classe
}
```



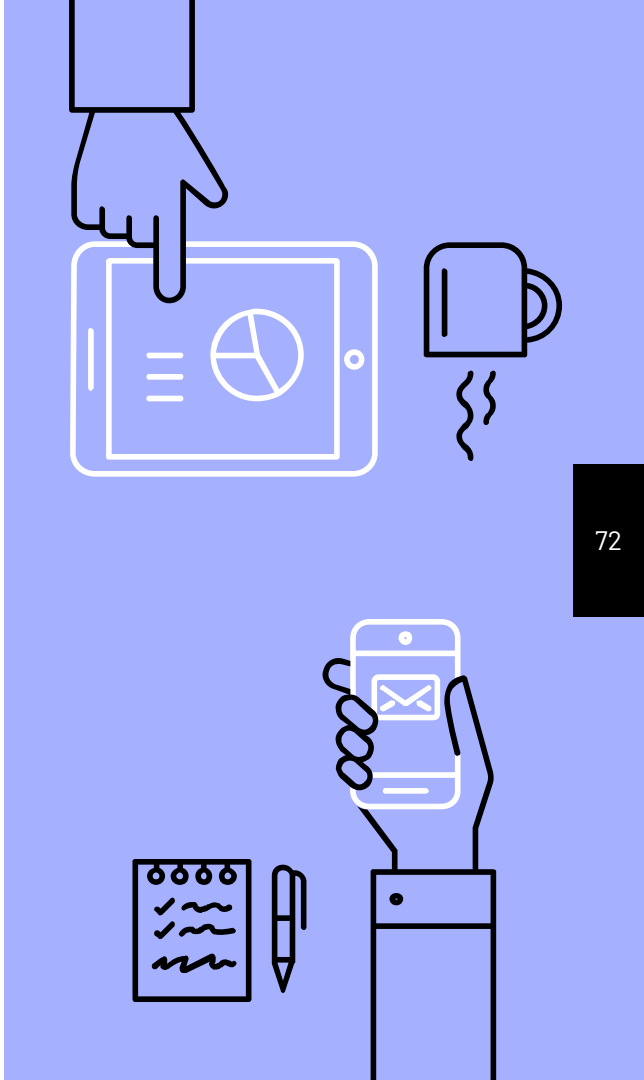
Prefira Composição a Herança

A solução por meio de herança não é recomendada por vários motivos, sendo que os principais são os seguintes:

1. Um Stack, em termos conceituais, não **É UM** ArrayList, mas sim uma estrutura que pode usar um ArrayList na sua implementação interna;

1. Quando se força uma solução via herança, a class Stack irá herdar métodos como get e set, que não fazem parte da especificação de pilhas. Portanto, nesse caso, devemos preferir a solução baseada em composição.

Uma segunda vantagem de composição é que a relação entre as classes não é estática, como no caso de herança. No exemplo, se optássemos por herança, a classe Stack estaria **acoplada estaticamente** a ArrayList, e não seria possível mudar essa decisão em tempo de execução. Por outro lado, quando se adota uma solução baseada em composição, isso fica mais fácil, como mostra o exemplo a seguir:

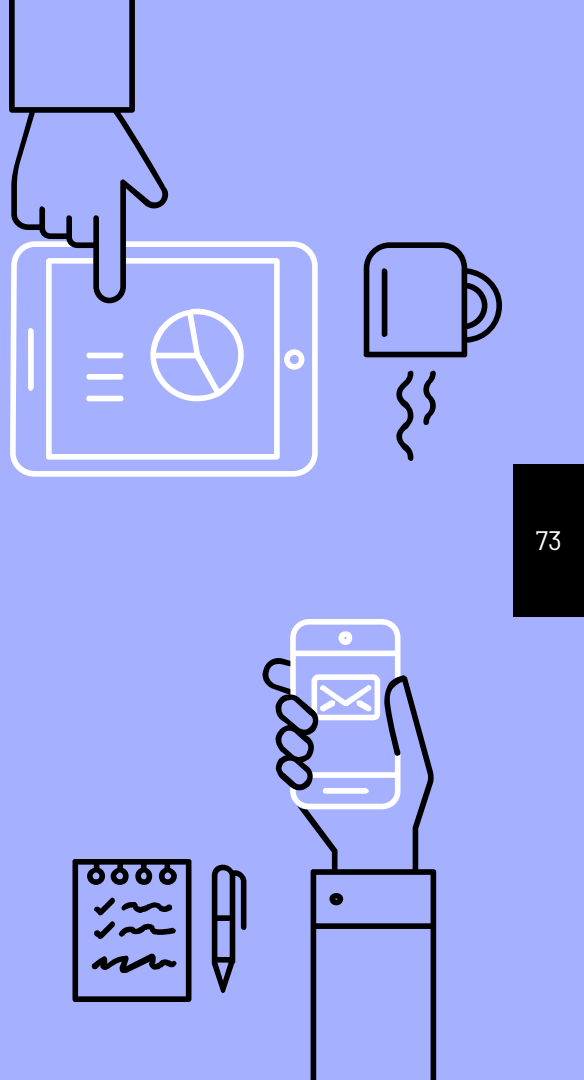


Prefira Composição a Herança

No exemplo, a estrutura de dados que armazena os elementos da pilha passou a ser um parâmetro do construtor da classe Stack. Com isso, torna-se possível instanciar objetos Stack com estruturas de dados distintas. Por exemplo, um objeto no qual os elementos da pilha são armazenados em um **ArrayList** e outro objeto no qual eles são armazenado em um **Vector**.

Como uma observação final, veja que o tipo do atributo elementos de Stack passou a ser um **IList**, ou seja, fizemos uso também do Princípio de Inversão de Dependências (ou Prefira Interfaces a Classes).

```
class Stack
{
    private IList elementos;

    public Stack(IList elementos)
    {
        this.elementos = elementos;
    }
}
```



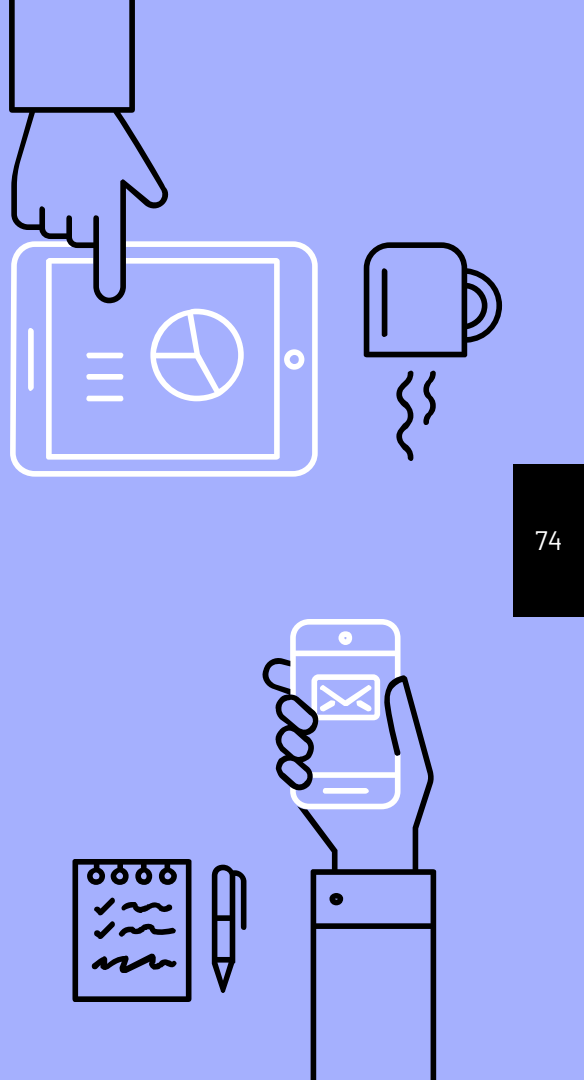
Prefira Composição a Herança

Antes de concluir, gostaríamos de mencionar três pontos suplementares ao que discutimos sobre Prefira Composição a Herança:

- Herança é classificada como um mecanismo de **reuso caixa-branca,** pois as subclasses costumam ter acesso a detalhes de implementação da classe base. Por outro lado, composição é um mecanismo de **reuso caixa-preta**.

- Um padrão de projeto que ajuda a substituir uma solução baseada em herança por uma solução baseada em composição é o Padrão Decorador.

- Por conta dos problemas discutidos nesta seção, linguagens de programação mais recentes — como Go e Rust — não incluem suporte a herança.



Prefira Composição a Herança

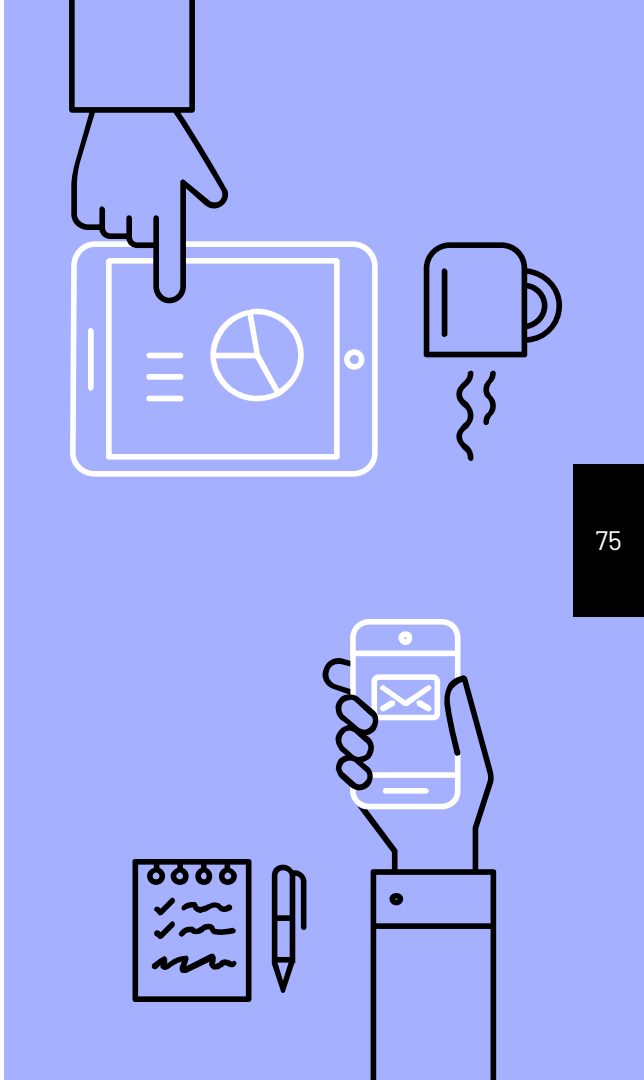
Herança

Vantagem

- Reuso de código

Desvantagem

- Fraco Encapsulamento.
- Forte acoplamento em muitos níveis de herança.
- Comportamento estático (Tempo de Compilação).
- Fazer polimorfismo da maneira correta utilizando **LSP** do **SOLID.**



Prefira Composição a Herança

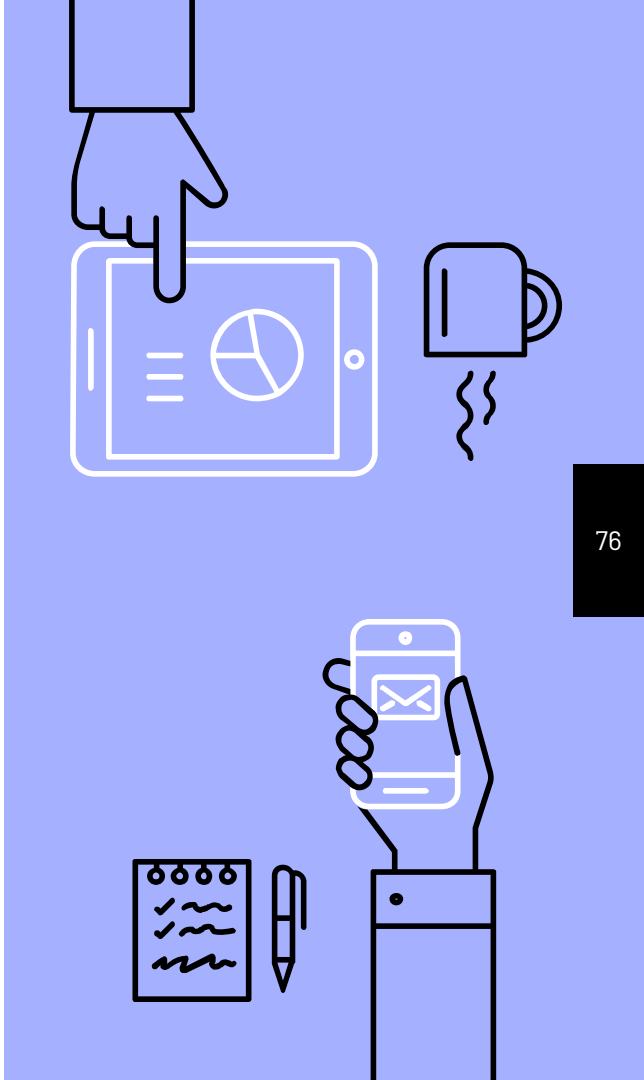
Composição

Vantagem

- Reuso de código
- Comportamento dinâmico (Tempo Execução)
- Forte encapsulamento

Desvantagem

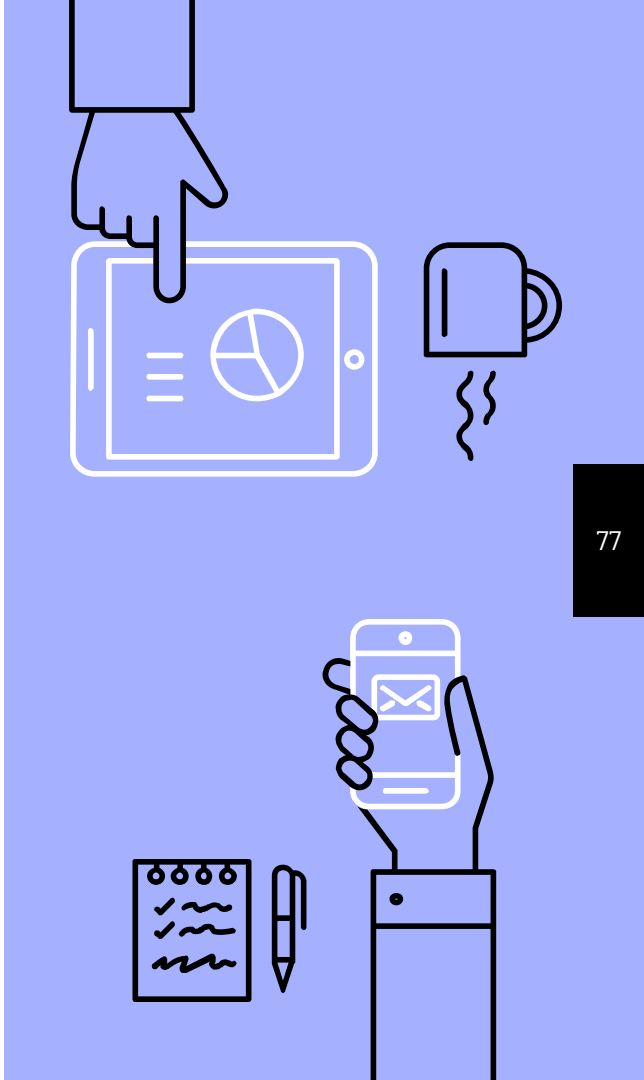
- Complexidade no entendimento, em relação a herança.

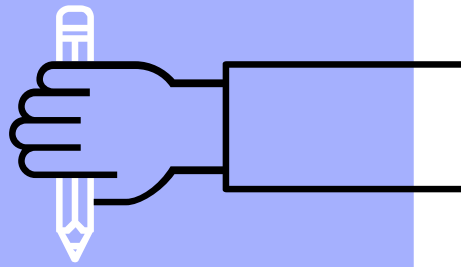


Diga, não pergunte

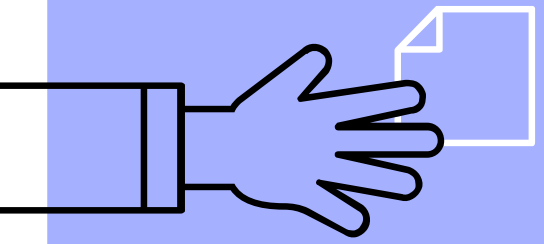
Conclusão

A utilização de composições proporciona interdependência entre os objetos e muito mais flexibilidade, pois permite a alteração do comportamento durante o tempo de execução, desde que o objeto que estivermos compondo, utilize a interface de comportamento correta.





7. Programe para abstrações, não para implementações



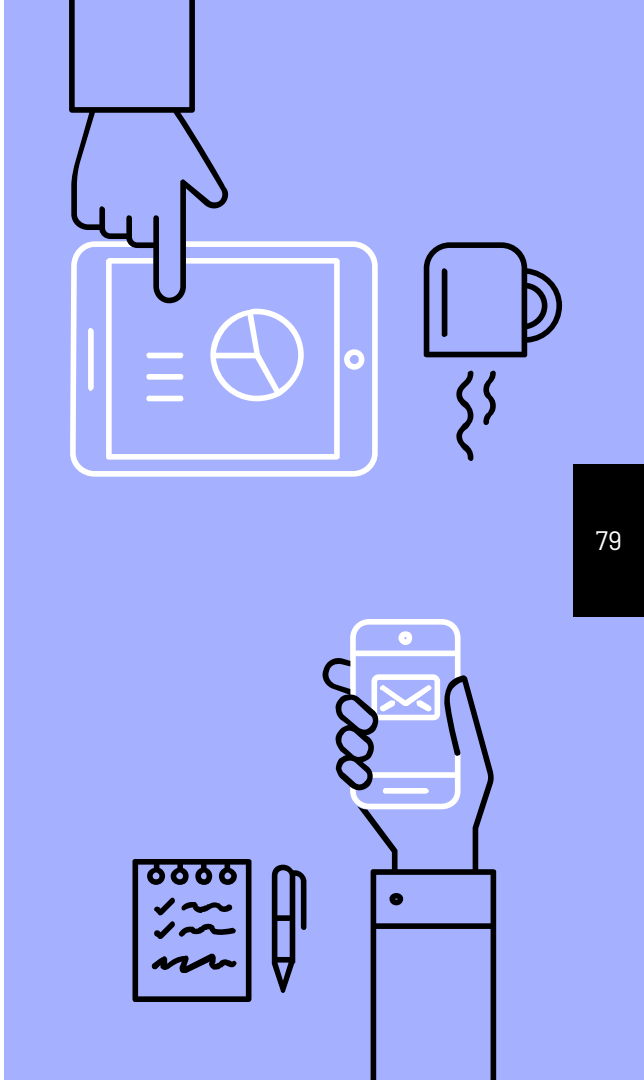
Baixo Acoplamento, Alta Flexibilidade

Programe para Abstrações não para Implementações

Programar orientado a objetos é pensar em **abstrações**.

Um exemplo bem didático que costumo usar é do autor Maurício Aniche do livro **"Orientação a Objetos e SOLID para Ninjas"** ele disse que quando ele está dando aula de Orientação a objetos básica, e o aluno está vendo pela primeira vez todos aqueles conceitos complicados de polimorfismo, herança, encapsulamento etc., uma brincadeira que ele sempre faz com os alunos é: no meio da aula ele fala **"Gato, cachorro e pássaro"**. E espera que os alunos respondam **"animal"** (a abstração mais adequada para o que ele falou). Outro exemplo é quando ele fala **"ISS, PIS, COFINS e outro-imposto-qualquer"** ele espera que as pessoas respondam **"imposto"**.

O intuito dessa brincadeira é fazer com que o aluno pensa o tempo inteiro em abstração. Isso é programar orientado a objetos. É pensar **primeiro na abstração**, e **depois, na implementação**.

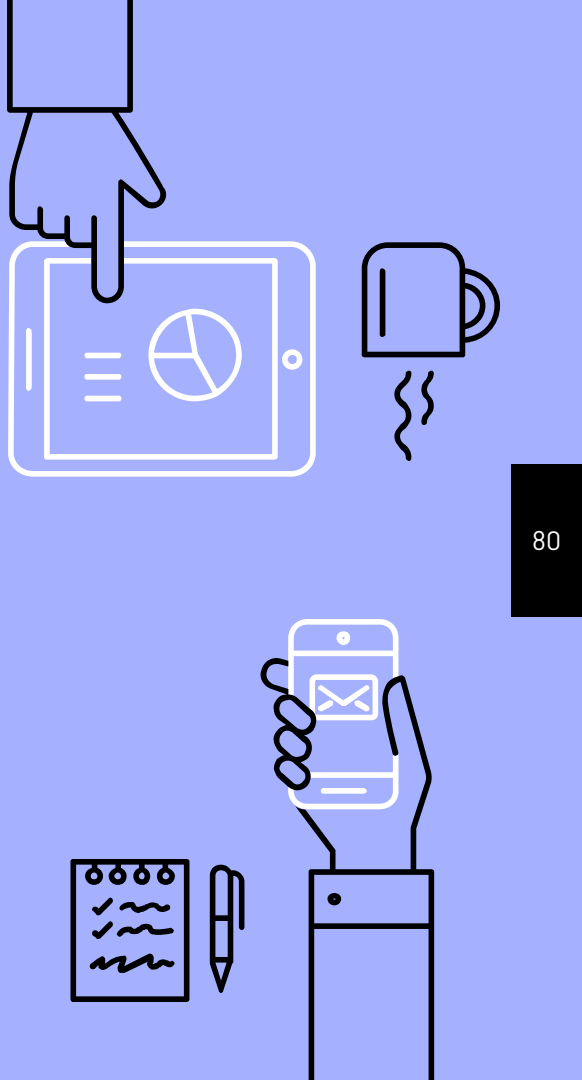


Programe para Abstrações não para Implementações

Essa é uma mudança de pensamento com quem programa procedural. Porque no mundo procedural, você está muito preocupado com a implementação. E é natural. No mundo OO, você tem que inverter: a sua preocupação maior tem que ser com a abstração, com o projeto de classes.

Uma estratégia importante para combater a complexidade de sistemas de software passa pela criação de abstrações. Uma abstração, pelo menos em Computação, é uma representação simplificada de uma entidade. Apesar de simplificada, ela nos permite interagir e tirar proveito da entidade abstraída, sem que tenhamos que dominar todos os detalhes envolvidos na sua implementação.

Em resumo, o primeiro objetivo de projeto de software é decompor um problema em partes menores. Além disso, deve ser possível implementar tais partes de forma independente. Por fim, mas não menos importante, essas partes devem ser abstratas. Em outras palavras, a implementação delas pode ser desafiadora e complexa, mas apenas para os desenvolvedores envolvidos em tal tarefa. Para os demais desenvolvedores, deve ser simples usar a abstração que foi criada.

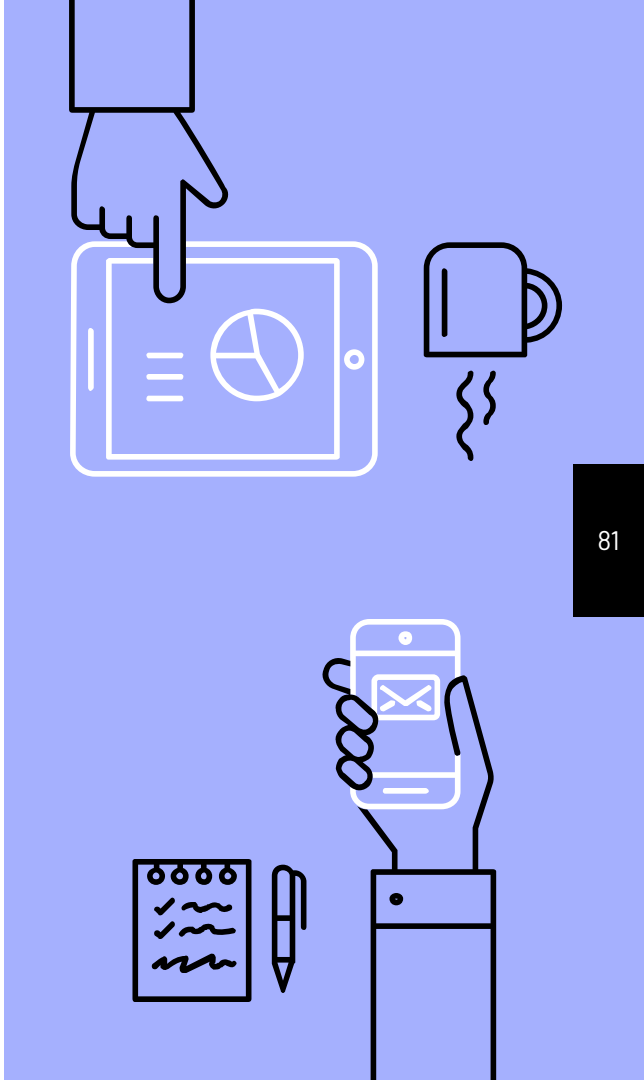


Programe para Abstrações não para Implementações

Classe Abstrata ou Interface?

Classe Abstrata: Quando a abstração for um **conceito** ou **base estrutural** (algo que precisa ser refinado ou especializado).

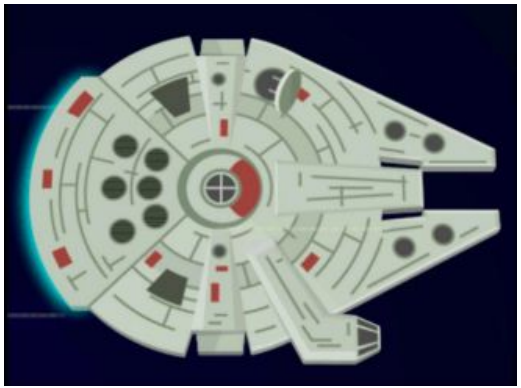
Interface: Quando a abstração for um **comportamento** (algo que uma classe deve saber fazer).

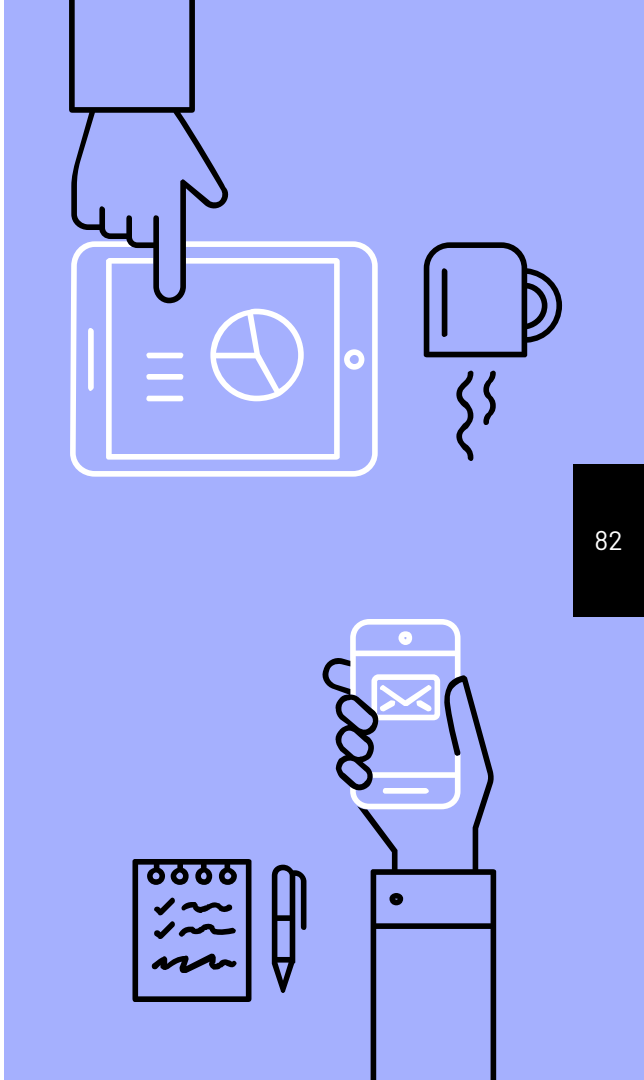


Programe para Abstrações não para Implementações

Classe Abstrata ou Interface?

Imagine um jogo no qual existem naves que se movem. Se sua abstração representa uma nave, então você está representando um **conceito** ou **base estrutural** e deve utilizar uma **classe abstrata**.

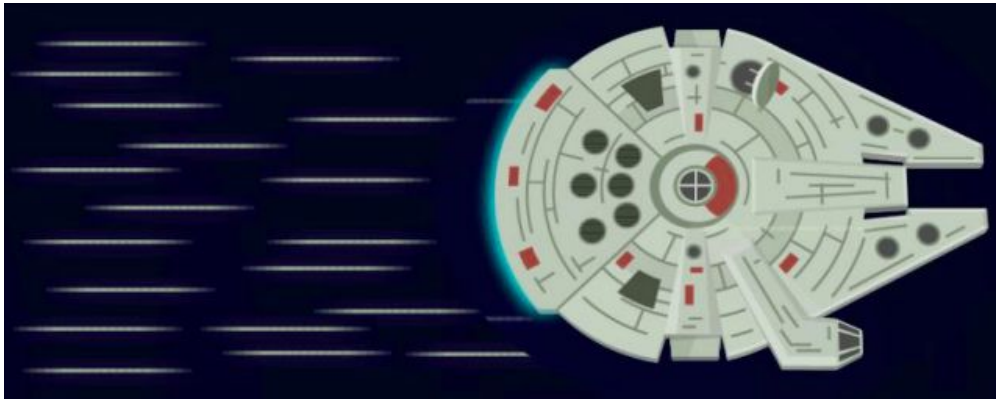


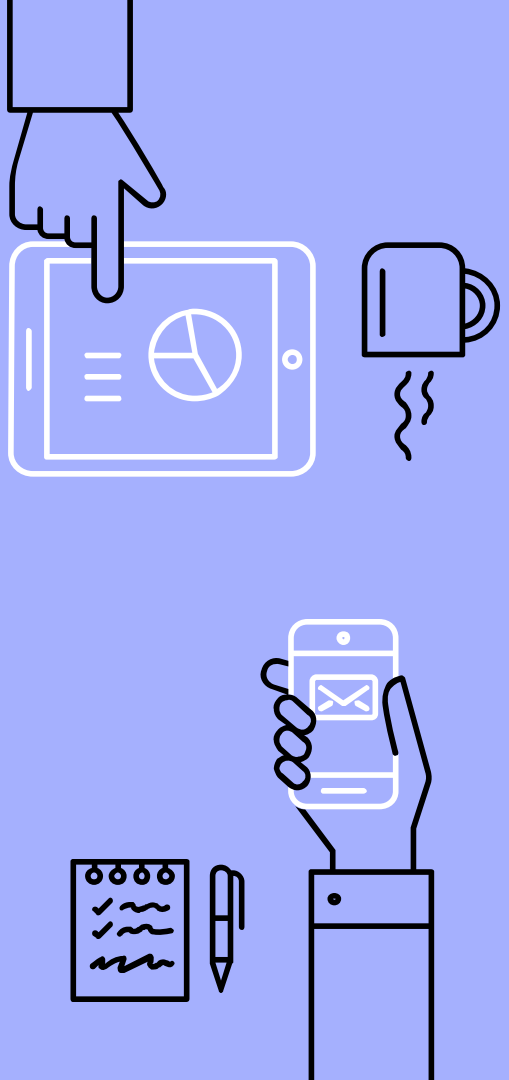


Programe para Abstrações não para Implementações

Classe Abstrata ou Interface?

Imagine um jogo no qual existem naves que se movem. Mas, se sua abstração representa algo que se move, então o que está sendo abstraído é um **comportamento** e a melhor solução é usar uma interface.





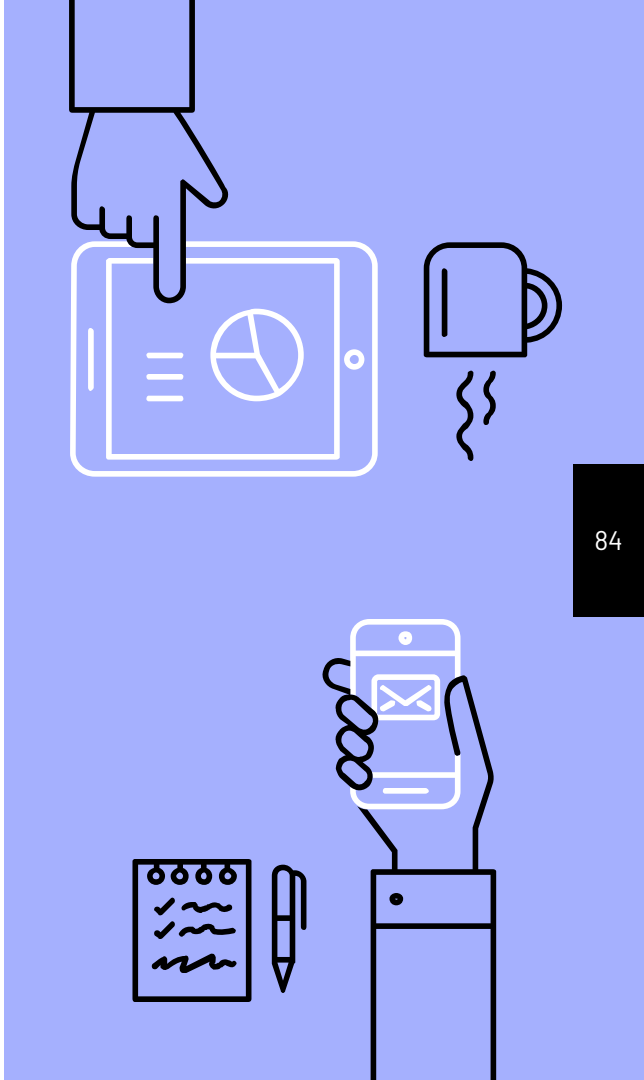
Programe para Abstrações não para Implementações

Exemplo:

Ao trabalhar com coleções, escolher a implementação certa para cada caso é uma tarefa difícil. Cada uma delas, como **ArrayList**, **LinkedList** ou **HashSet**, é melhor para resolver determinadas categorias de problemas. Pode ser muito arriscado escrever todo o código da aplicação dependendo de uma decisão antecipada. Apesar disso, grande parte dos desenvolvedores opta por sempre utilizar ArrayList desde o início, sem critério algum.

Considere um **DAO** de funcionários que pode listar o nome de todos que trabalham em determinado turno, devolvendo um ArrayList com os nomes:

```
public class FuncionarioDao {
    public ArrayList<String> buscaPorTurno(Turno turno) { ... }
}
```



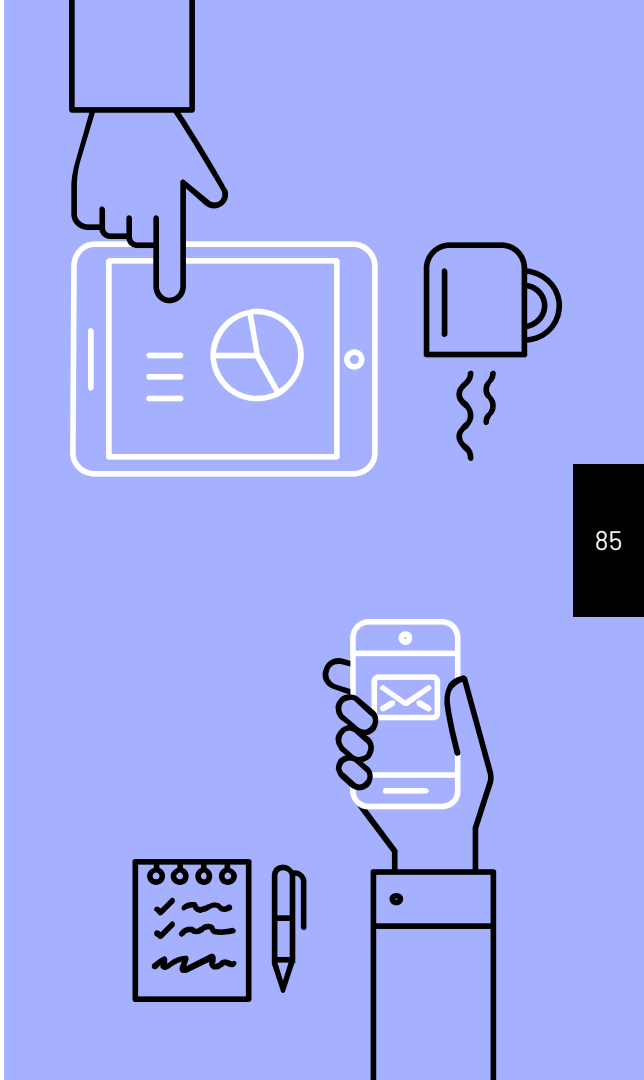
Programe para Abstrações não para Implementações

E um código que precisa saber se um determinado funcionário esteve presente, efetuando uma busca simples na lista devolvida:

Considere um **DAO** de funcionários que pode listar o nome de todos que trabalham em determinado turno, devolvendo um ArrayList com os nomes:

```
FuncionarioDao dao = new FuncionarioDao();
ArrayList<String> nomes = dao.buscaPorTurno(Turno.NOITE);

bool presente = nomes.contains("Anton Tcheckov");
```

Mas a busca com contains em um ArrayList é, em termos computacionais, bastante custosa. Poderíamos utilizar outras alternativas de coleção, trocando o retorno de ArrayList para HashSet, por exemplo, pois sua operação contains é muito mais eficiente computacionalmente, usando internamente uma tabela de espalhamento (hash table).

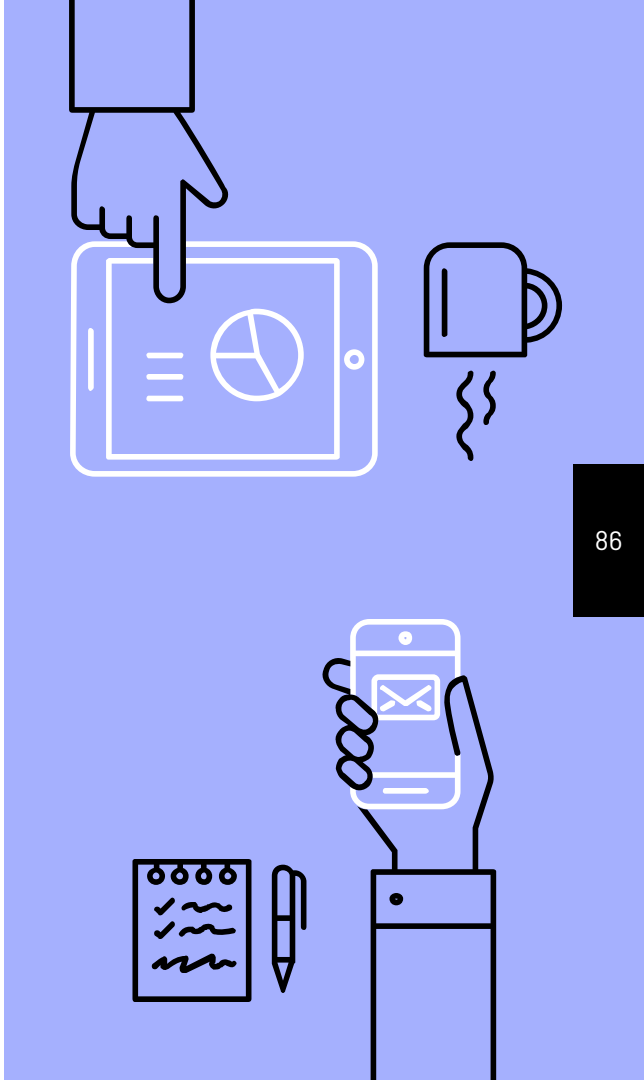


Programe para Abstrações não para Implementações

Para poucos funcionários, a diferença é imperceptível. Entretanto, à medida que a lista aumenta, a diferença de desempenho entre um **ArrayList** e um **HashSet** torna-se mais clara, e até mesmo um gargalo de performance.

O problema em realizar uma mudança de implementação como esta, é que todo código que usava o retorno do método como ArrayList quebra, mesmo que só usássemos métodos que também estão definidos em HashSet. Seria preciso alterar todos os lugares que dependem de alguma forma desse método. Além de trabalhoso, tarefas do tipo search/replace são um forte sinal de código ruim.

Há esse **acoplamento sintático** com a assinatura do método, que conseguimos resolver olhando os erros do compilador. Mas sempre existem também informações semânticas implícitas na utilização desse método, e que não são expostas pela assinatura.

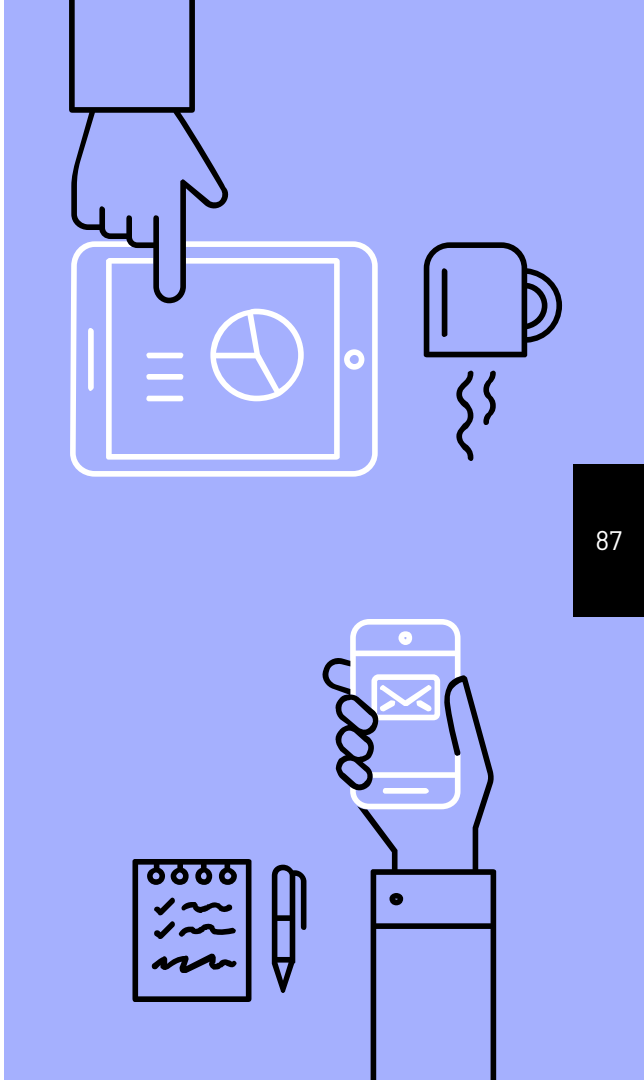


Programe para Abstrações não para Implementações

Um exemplo de acoplamento semântico está em depender da informação de que uma **List** permite dados duplicados, enquanto um **Set** garante unicidade dos elementos. Como problemas no acoplamento sintático são encontrados em tempo de compilação, os semânticos somente são em execução, daí um motivo da importância de testes que garantam o comportamento esperado.

O grande erro do método **buscaPorTurno** da classe **FuncionarioDao** foi atrelar todos os usuários do método a uma implementação específica de Collection. Desta forma, alterar a implementação torna-se sempre muito mais custoso, caracterizando o alto acoplamento que tanto se procura evitar.

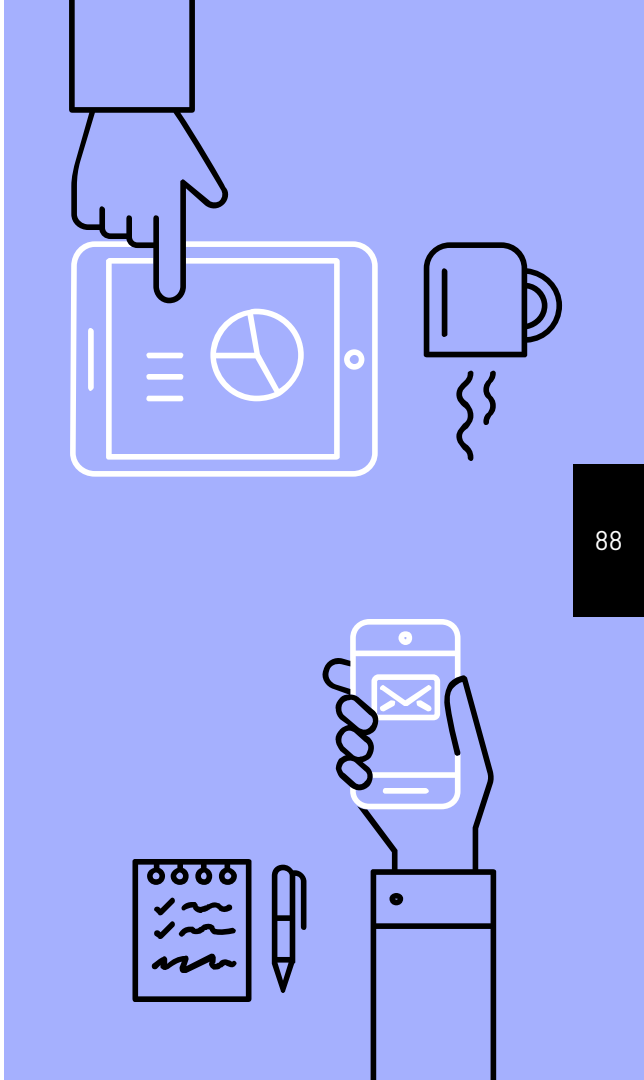
Para minimizar esse problema, é possível usar um tipo de retorno de método mais genérico, que contemple diversas implementações possíveis, fazendo com que os usuários do método não dependam em nada de uma implementação específica. A interface **Collection** é uma boa candidata.



Programe para Abstrações não para Implementações

Com o método desta forma, podemos trocar a implementação retornada sem receio de **quebrar** nenhum código que esteja invocando **buscaPorTurno**, já que ninguém depende de uma implementação específica. Usar **interfaces** é um grande benefício nestes casos, pois ajuda a garantir que nenhum código dependa de uma implementação específica, pois interfaces não carregam nenhum detalhe de implementação.

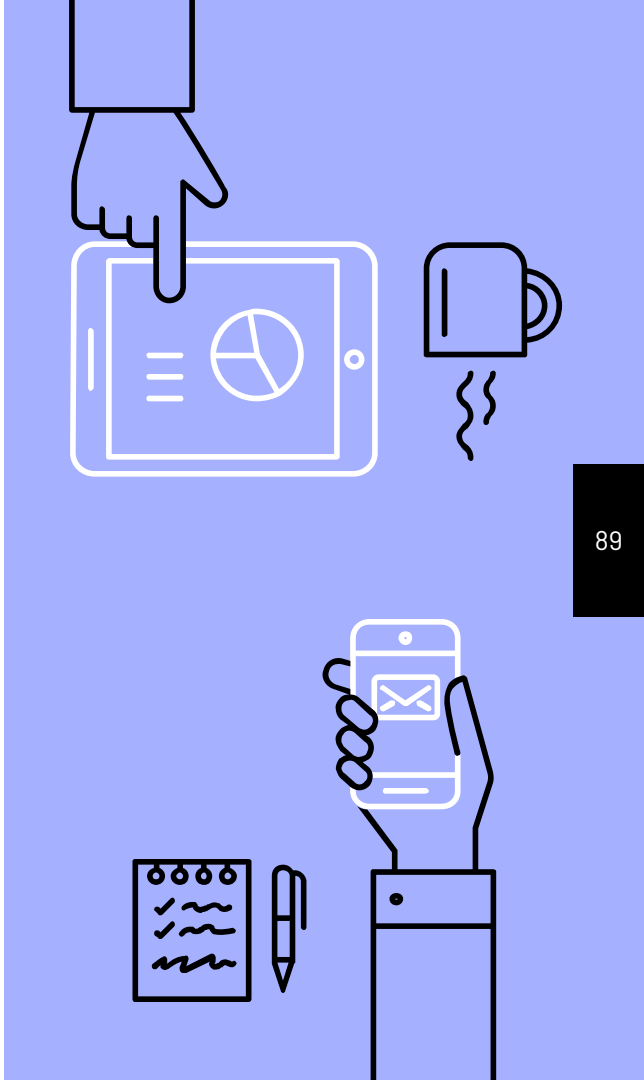
```
public class FuncionarioDao {
    public Collection<String> buscaPorTurno(Turno turno) { ... }
}
```

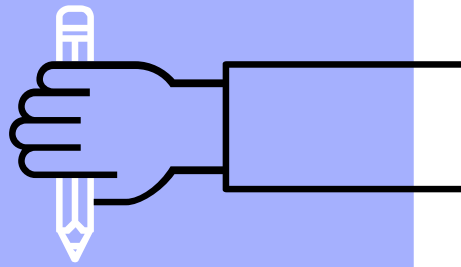


Programe para Abstrações não para Implementações

Conclusão

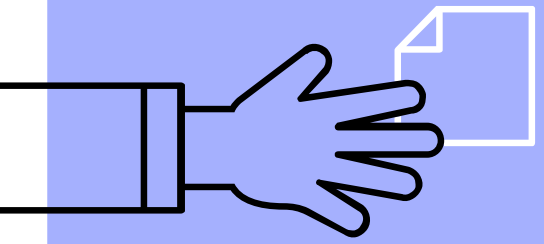
Programar para uma interface significa, na verdade, **programar para um supertipo**, uma classe **abstrata** ou uma **interface** de construção. A vantagem desse princípio é **explorar o polimorfismo** de modo que o objeto fique **vinculado à abstração** e diminua a quantidade de interdependências.





8. Encapsule o que varia

Isolando comportamentos que mudam muito



Encapsule o que varia

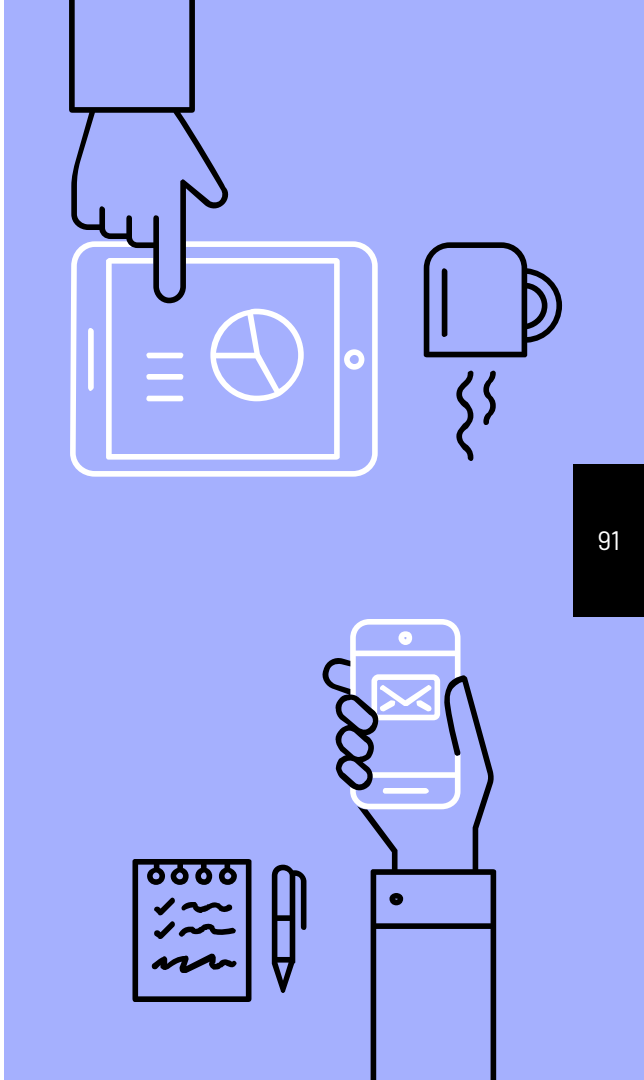
Este Princípio diz para Identificar os aspectos de seu aplicativo que **variam** e **separar do que permanece igual.**

Pegue o que variar e **"encapsule"** para que isso não afete o resto de seu código., com isso garantimos menos consequências indesejadas das alterações de código e mais flexibilidade em seus sistemas!

Em outras palavras, se houver algum aspecto de seu código que está mudando, por exemplo, como cada novo requisito, você sabe que tem um comportamento que precisa ser retirado e separado do que não muda.

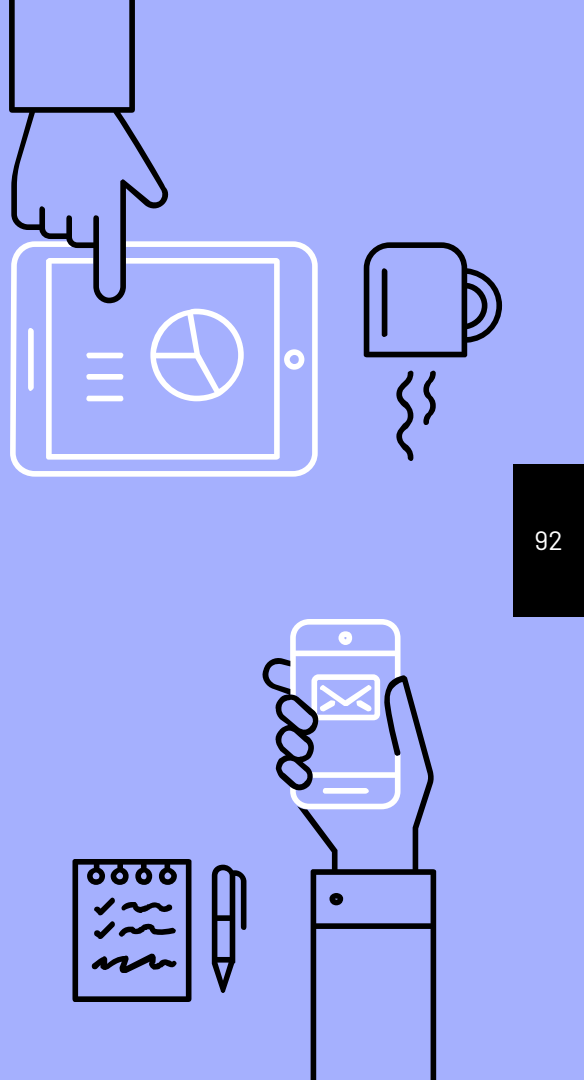
Veja outra maneira de pensar neste princípio: **pegue as partes que variam e encapsule para depois poder alterar ou estender as partes que variam sem afetar as que não variam**.

Esse conceito simples formam a base de todos os padrões de projetos. Todos os padrões fornecem uma maneira de que **alguma parte de um sistema varie independentemente de todas as outras partes**.



Encapsule o que varia

```php
class Pagamento {

  public function calcular(float $valor, string $pagamento) : float {

    if($pagamento === 'Debito'){
      return $valor * 0.10;
    } else if($pagamento === 'Credito'){
      return $valor * 0.05;
    }
  }
}
```

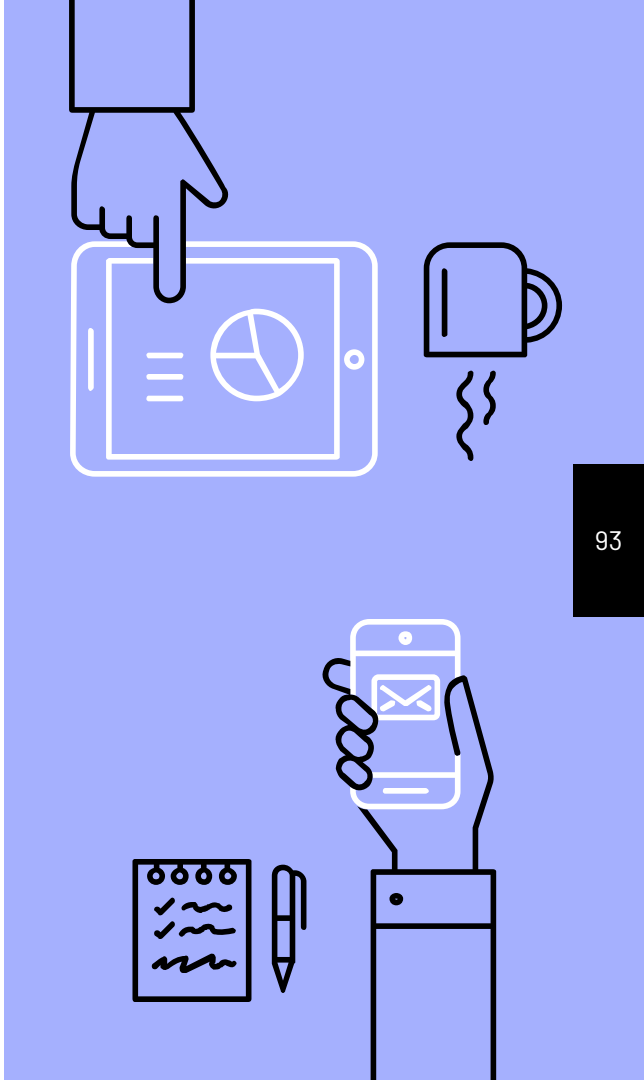


Encapsule o que varia

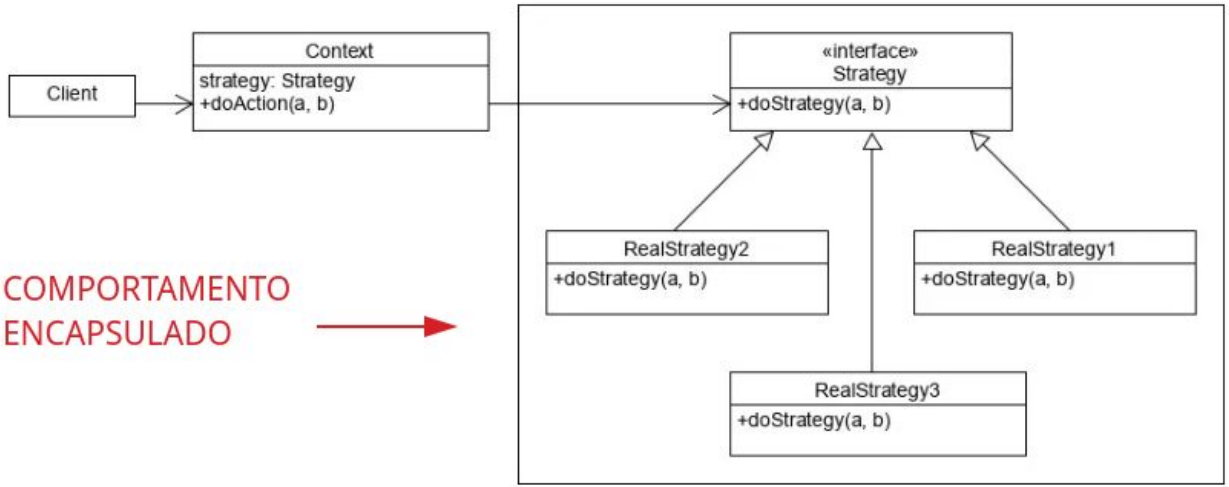
O código é bem simples. Ele basicamente calcula o total venda baseada na forma de pagamento, caso a forma de pagamento seja **"Débito"** acrescenta uma taxa de 10% em cima do valor pago e caso for **"Credito"** tem um acréscimo de 5%.

Agora, Imagine que o sistema é mais complicado que isso. Não existe apenas duas regra de cálculo de taxas, mas várias de acordo com o valor pago e também não existe apenas forma de pagamento cartão, más várias, como dinheiro, cheque e a prazo. Uma maneira (infelizmente) comum de vermos código por aí é resolvendo isso por meio de ifs. Ou seja, o código decide se é a regra A ou B que deve ser executada.

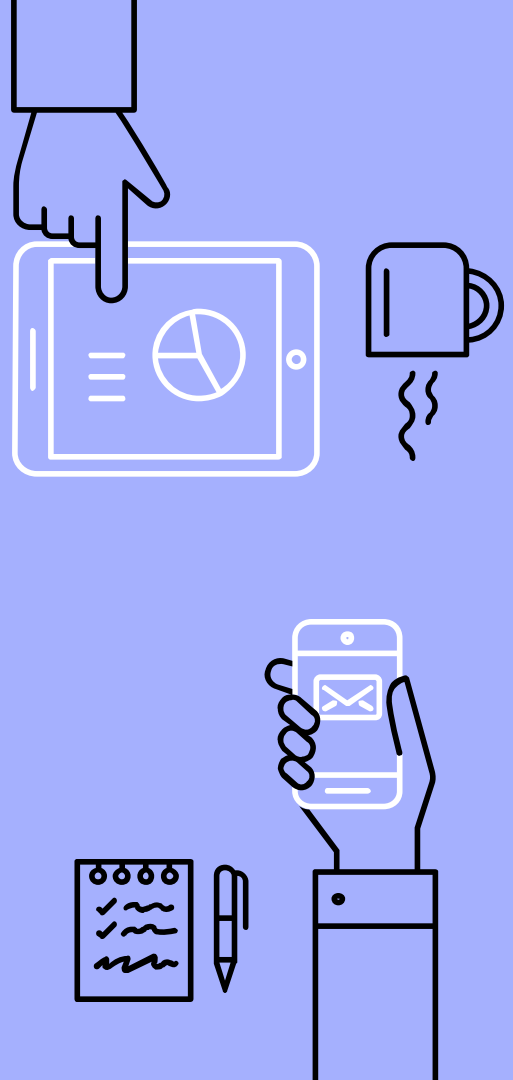
Agora vamos aplicar o princípio **encapsule o que varia**, para ver como nosso código irá ficar:



Encapsule o que varia



COMPORTAMENTO ENCAPSULADO →

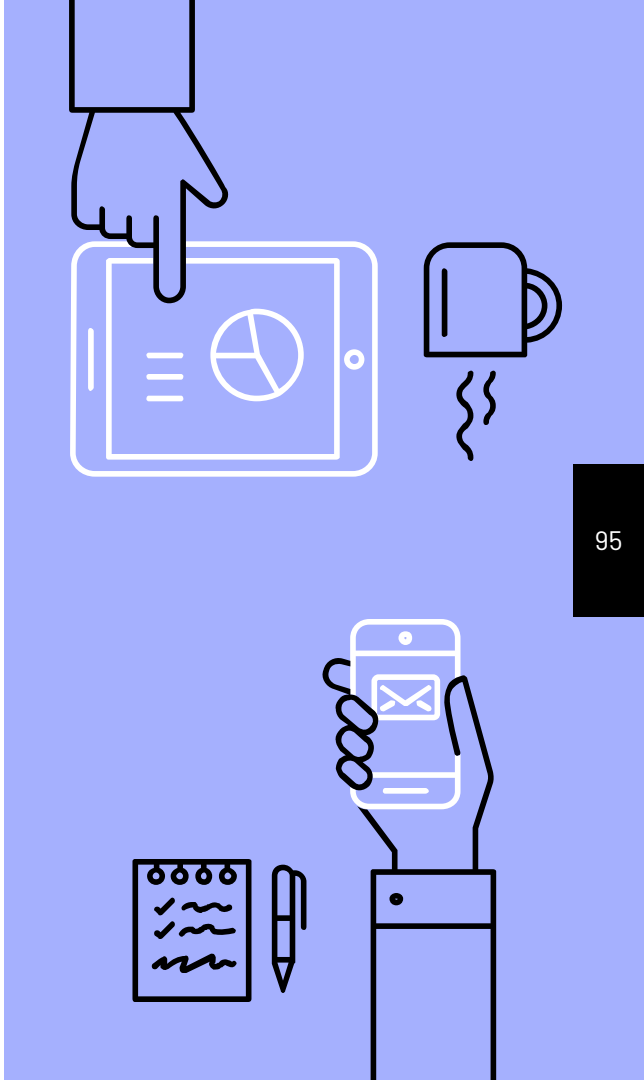


Encapsule o que varia

Aplicando o principio, vamos ter uma **interface IMetodoPagamento** que será utilizada nas classes dos métodos de pagamentos **CartaoDebito** e **CartaoCredito** e uma classe **Pagamento** para interpretar a chamada do index e executar o **cálculo**.

Na classe **IMetodoPagamento**, definimos uma interface na qual os métodos de pagamento (**CartaoDebito** e **CartaoCredito**) que irão herdar e utilizar a mesma estrutura.

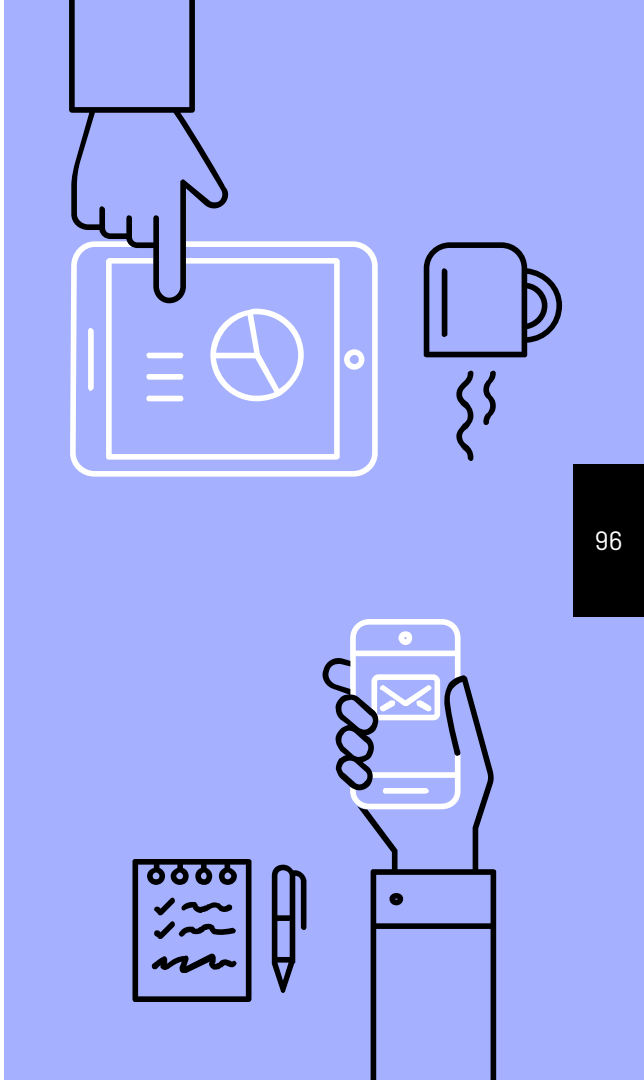
```php
interface IMetodoPagamento {

    public function calcular(float $valor) : float;
}
```



Encapsule o que varia

Agora temos a classe **CartaoDebito** e **CartaoCredito**, onde utilizam a mesma estrutura da interface, porém aplicando o cálculo de acordo com o tipo de pagamento, evitando um monte de if e else if. E também estamos utilizando o **polimorfismo** no método **calcular()**.

```php
class CartaoDebito implements IMetodoPagamento {
  public function calcular(float $valor): float{
    return $valor * 0.10;
  }
}

class CartaoCredito implements IMetodoPagamento {
  public function calcular(float $valor): float{
    return $valor * 0.05;
  }
}
```



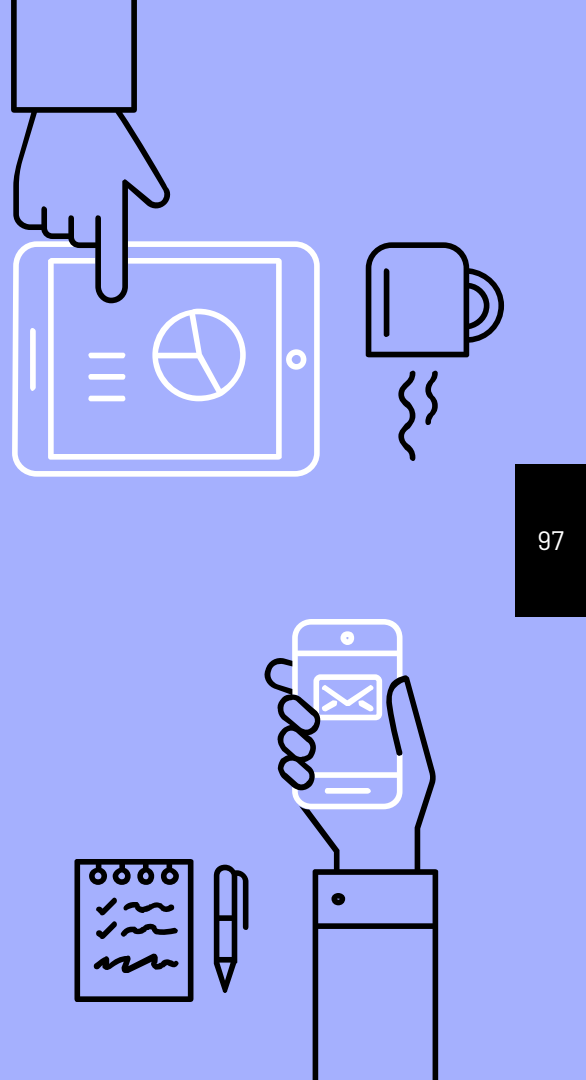
Encapsule o que varia

Agora temos uma classe **Pagamento**, onde vamos realizar a chamada de um **IMetodoPagamento** que será definido na **index**.

```php
class Pagamento {
  public function calcular(float $valor, IMetodoPagamento $pagamento) : float{
    return $pagamento->calcular($valor);
  }
}
```

Na **index**, realizamos a chamada, passando o **IMetodoPagamento** um objeto.

```php
require 'MetodoPagamento.php';
require 'Debito.php';
require 'Credito.php';
require 'Pagamento.php';

$pagamento = new Pagamento();
echo $pagamento->calcular(100, new CartaoDebito());
echo '<br />';
echo $pagamento->calcular(100, new CartaoCredito());
```

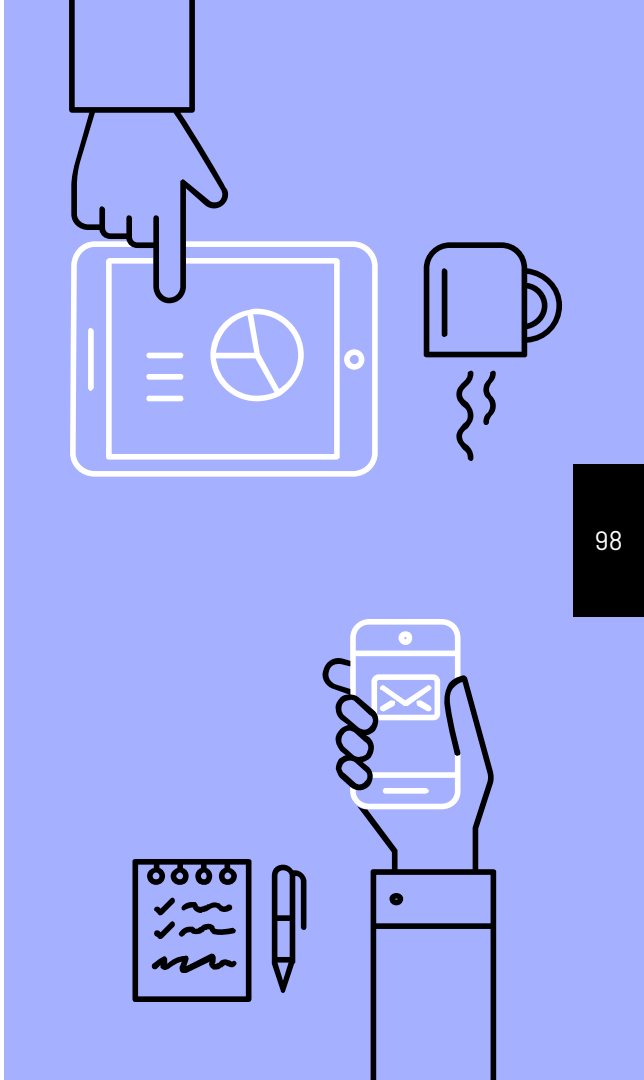


Encapsule o que varia

Como e quando aplicar ?

- Olhar para uma classe, enxergar sua responsabilidade, abstrair suas possíveis alterações e criar uma nova classe.

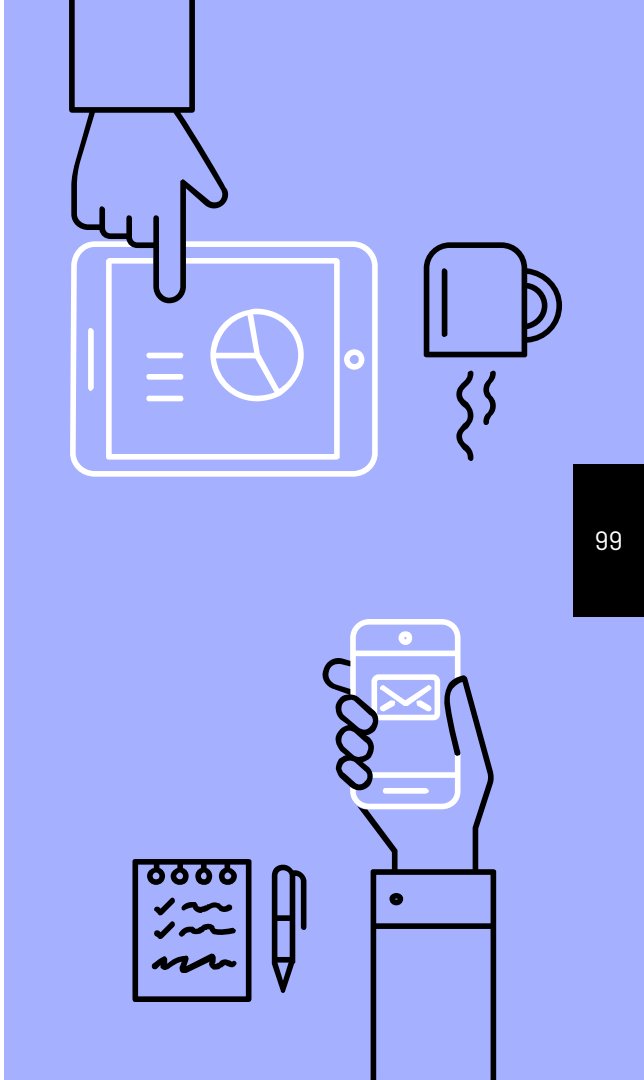
- Se a sua classe precisa ter mais de uma responsabilidade, divida-as e as associe.

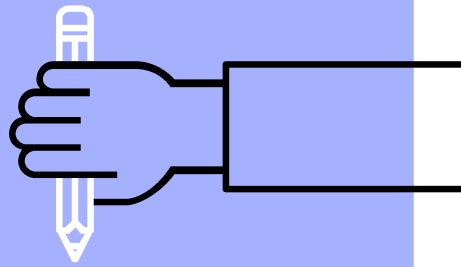


Encapsule o que varia

Conclusão

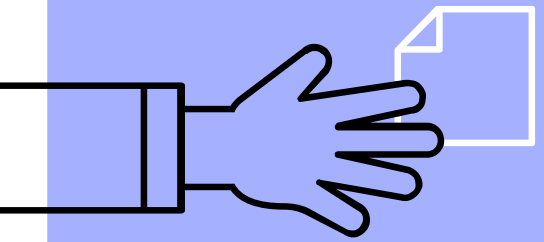
Este princípio possui um conceito simples que forma a base de quase todos os padrões de projeto. Ele defende a idéia de separar as partes que variam e encapsulá-las para que seja possível alterar ou estender essas partes sem afetar as que não variam. Assim é possível que uma parte de um sistema varie independentemente de todas as outras. Isso acarreta um número menor de alterações de código e mais flexibilidade nos sistemas.





9. O Princípio Hollywood

Não nos chame, nós chamaremos você

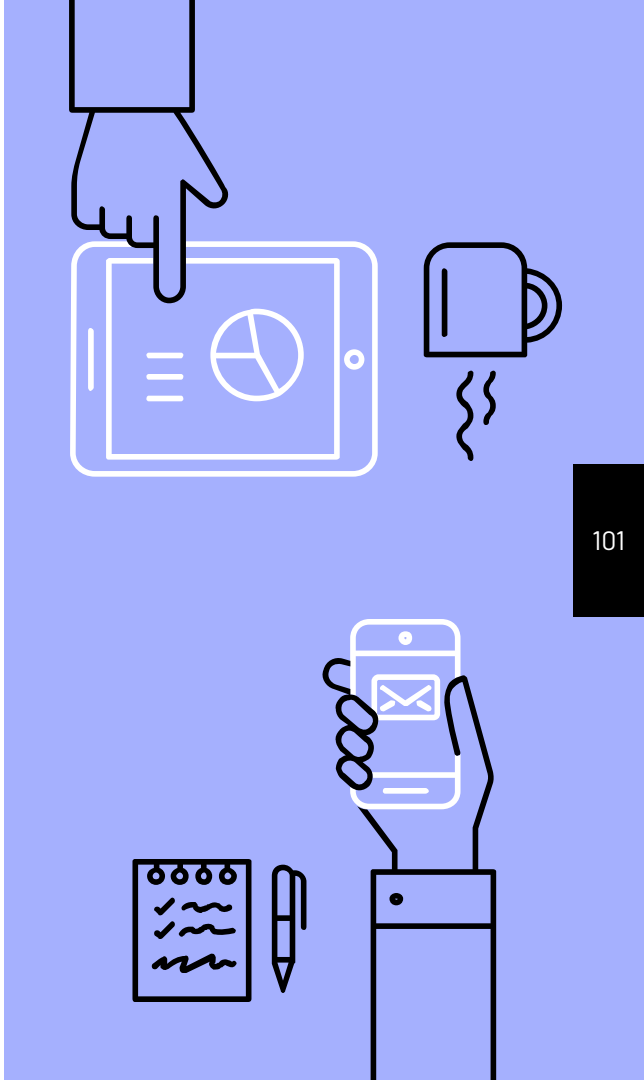


Princípio Hollywood

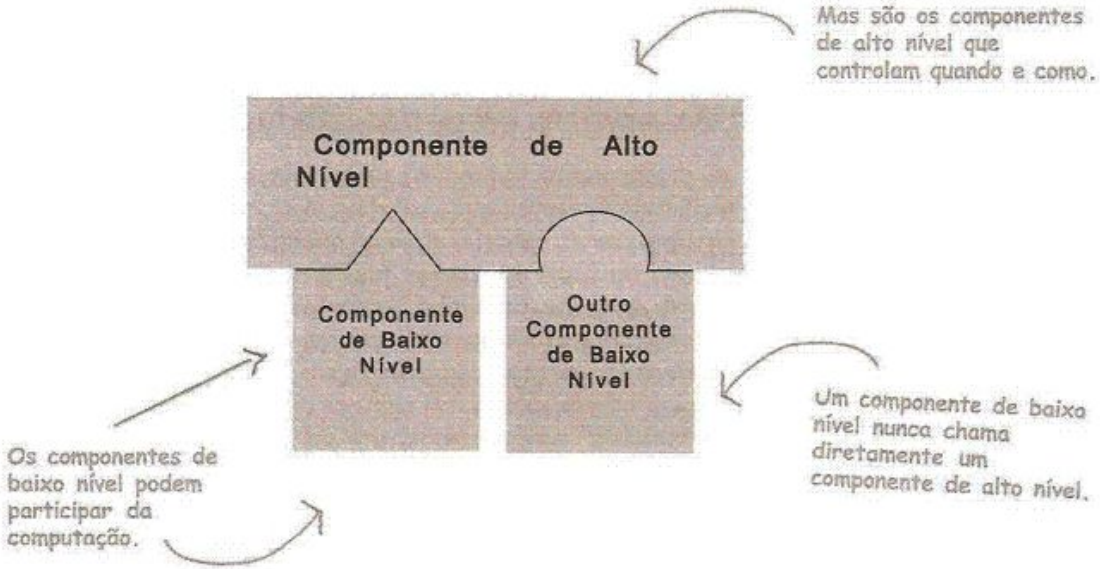
Temos outro princípio de projeto para você. Nós chamamos de Princípio Hollywood. O Princípio Hollywood nos proporciona uma maneira de evitar o **"colapso das dependências"**.

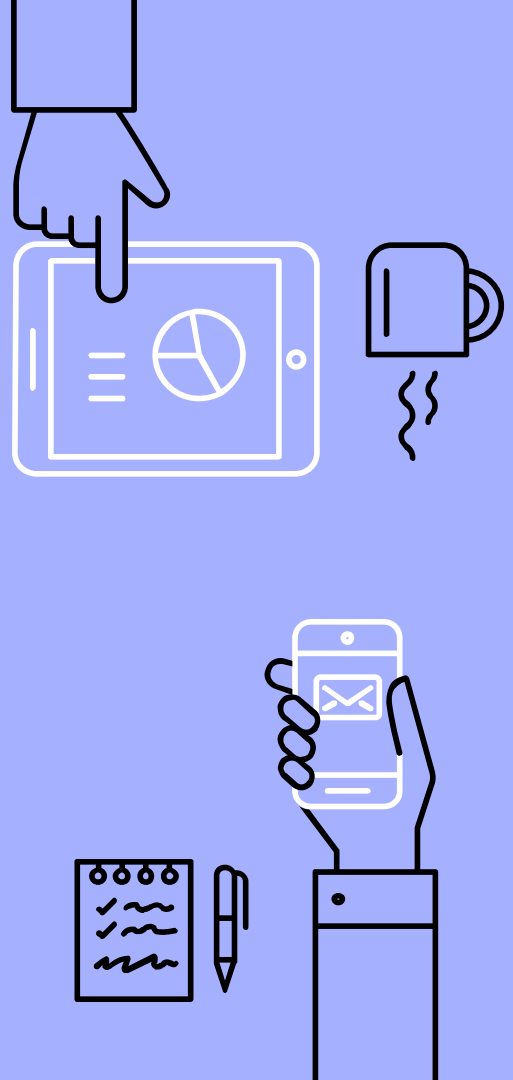
Colapso das dependências é o que acontece quando você tem componentes de alto nível dependendo de componentes de alto nível que dependem de componentes laterais que dependem de componentes de baixo nível, e assim por diante. Quando essa estrutura entra em colapso, ninguém mais consegue entender o sistema foi originalmente projetado.

Com o Princípio Hollywood, nós permitimos que componentes de baixo nível se conectem ao sistema através de ganchos, mas são os componentes de alto nível que determinam quando e como eles serão solicitados. Em outras palavras os componentes de alto nível dizem aos componentes de baixo nível **"Não nos chame, nós chamaremos você"**.



Princípio Hollywood

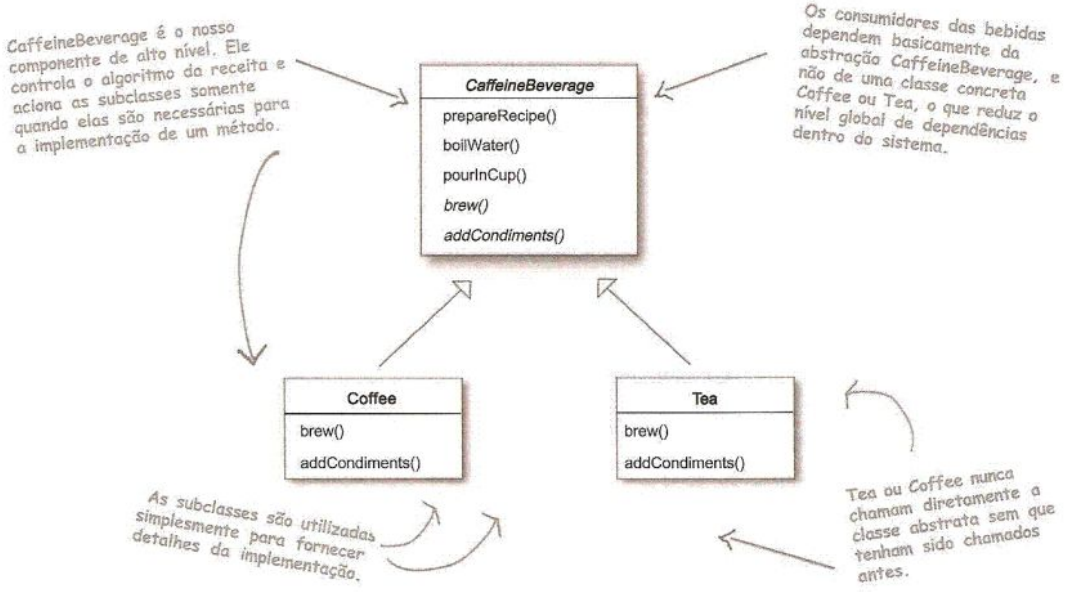




Princípio Hollywood

O Princípio Hollywood e o Template Method

É provável que você já tenha detectado a conexão entre o Princípio Hollywood e o Padrão Template Method, quando usamos um template method, na verdade estamos dizendo às subclasses "Não nos chame, nós chamaremos você".

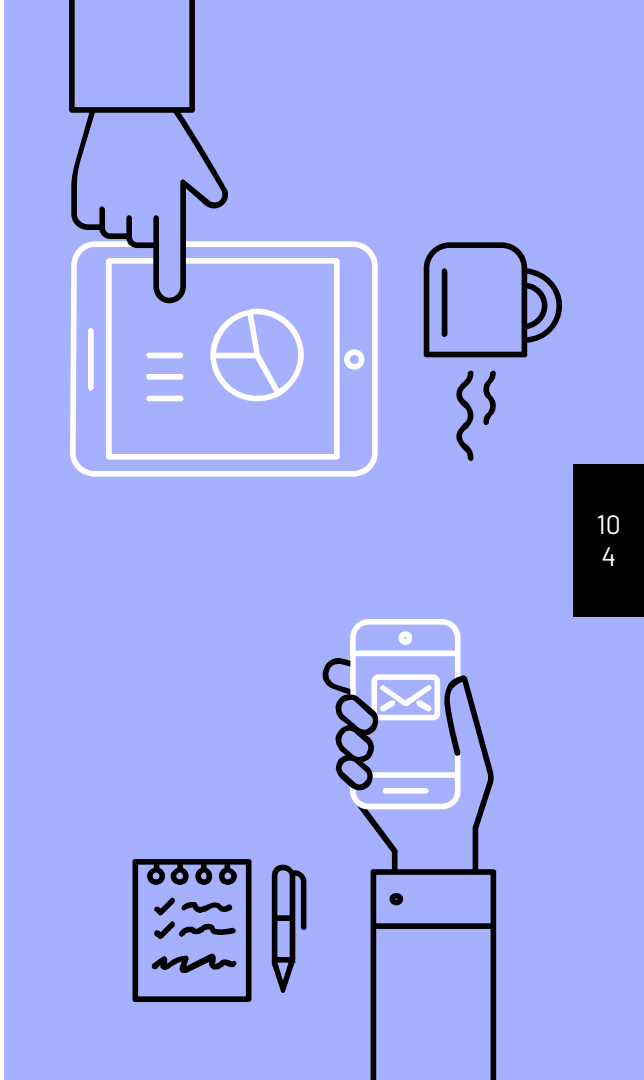


Princípio Hollywood

Os **métodos template** são uma técnica fundamental para a reutilização de código. Eles conduzem a uma estrutura de **inversão de controle** em ou princípio da **inversão de dependência** o **D** do **SOLID**, comumente conhecida como **"o princípio de Hollywood"**.

O **Princípio da Inversão de Dependências** nos ensina que devemos evitar o uso de classes concretas, trabalhando sempre que possível com abstrações. O Princípio Hollywood é uma técnica de para a construção de estruturas ou componentes de forma que os componentes de baixo nível possam conectar à computação sem que sejam criadas dependências entre estes e as camadas de nível mais alto. Portanto, o objetivo básico de ambos é a modularização, mas o Princípio de Inversão de Dependências é uma declaração muito mais genérica e poderosa sobre como evitar dependências em um projeto.

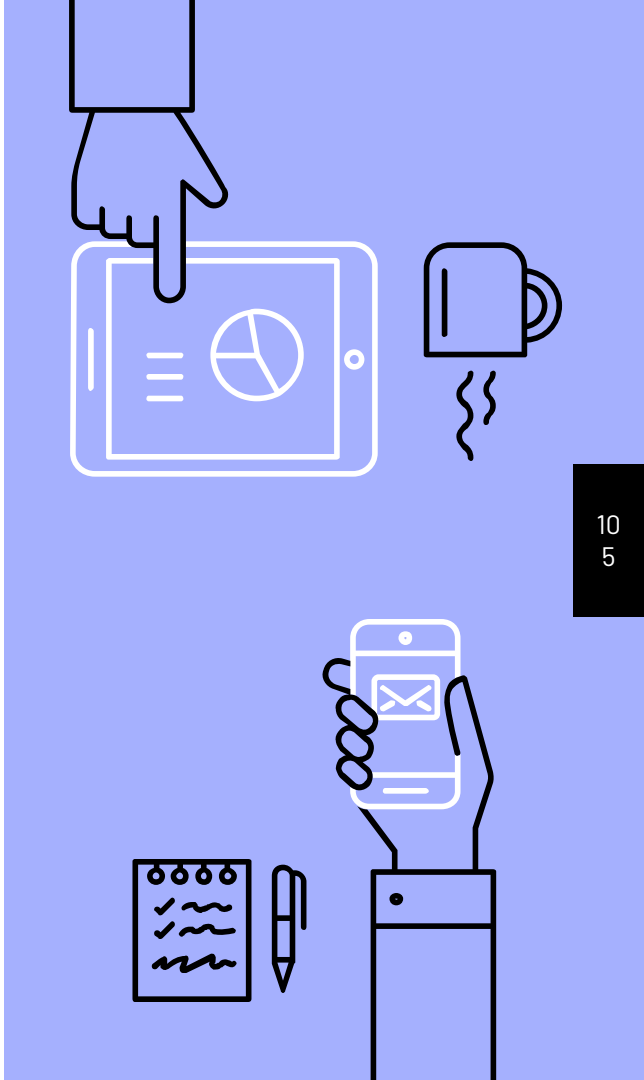
O Princípio Hollywood nos fornece uma técnica para a criação de projetos nos quais as estruturas de baixo nível podem operar entre si sem que outras classes tornem-se excessivamente dependente dela.



Princípio Hollywood

Um componente de baixo nível sempre é impedido de chamar um método de um componente de nível mais alto?

Na verdade, não, Na prática, um componente de baixo nível frequentemente acionará um método definido acima dele na hierarquia de hereditariedade devido às características da própria hereditariedade. O que queremos evitar é a criação de dependências circulares explícitas entre os componentes de baixo e alto níveis.



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 aí.

Obrigado!

Alguma pergunta?

Você pode me encontrar em:

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