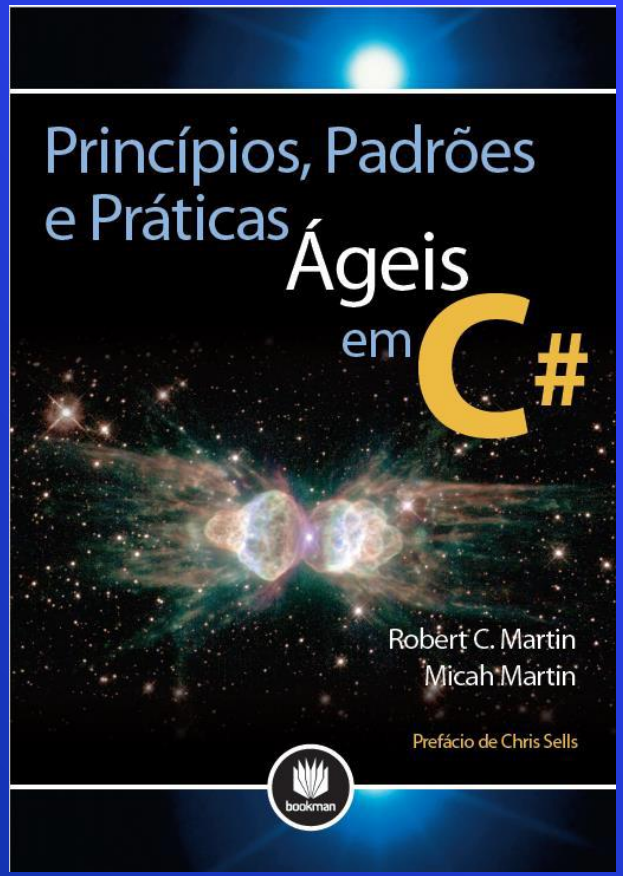
Princípios de Projeto de Componentes

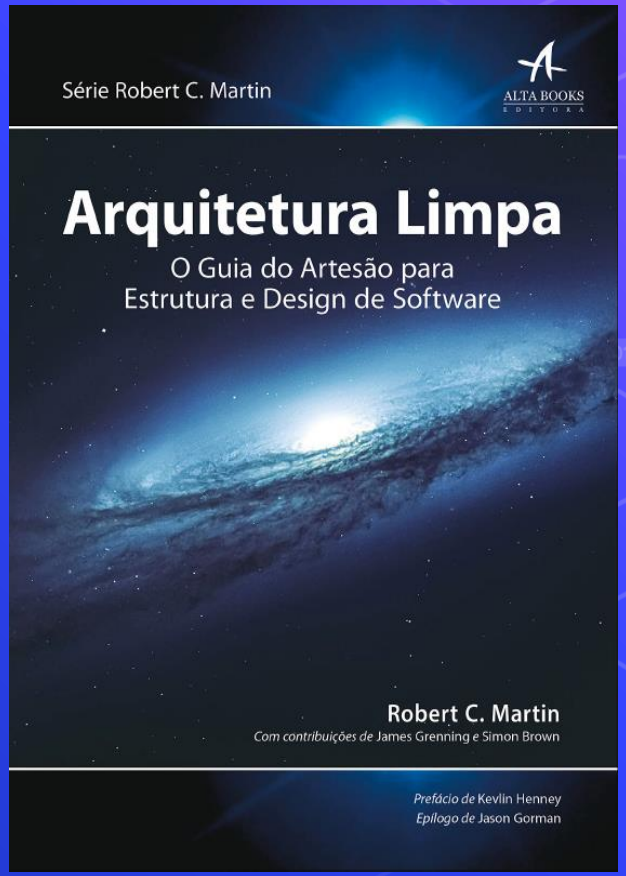
Olá!

Eu sou Willian Brito

- Desenvolvedor Full Stack 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:





Resumo dos Princípios de Componentes

Sigla	Nome	Definição
REP	Princípio da Equivalência entre o Reuso e a Distribuição	A granularidade de reuso não deve ser menor que a granularidade da entrega.
CCP	O Princípio do Fechamento Comum	Classes que mudam juntas, permanecem juntas.
CRP	O Princípio Comum de Reutilização	As classes que não são reutilizadas não devem ser agrupadas.
ADP	O Princípio das Dependências Acíclicas	As dependências entre componentes não devem formar ciclos.
SDP	O Princípio das Dependências Estáveis	Dependa na direção da estabilidade.
SAP	O Princípio das Abstrações Estáveis	Componentes estáveis devem ser componentes abstratos.

Módulos, componentes ou pacotes ?

Em seus textos mais antigos, Uncle Bob usava o termo **packages**, ou pacotes, para se referir ao que chamamos de módulos. Porém, pacote é um termo com um significado bem definido na plataforma Java, que define tanto a estrutura de diretórios como o nome completo de uma classe. Em outras tecnologias OO, como C++ e C#, esse tipo de agrupamento de classes é conhecido como **namespace**.

No post The Principles of OOD (MARTIN, 2005), ele esclarece a confusão:

[No contexto dos princípios de pacotes] um pacote é um entregável binário como um arquivo .jar ou uma dll, em oposição a um package Java ou a um namespace C++.

Em livros mais recentes, Uncle Bob usa o termo **binary components** ou simplesmente **components** ou, em português, **componentes.** Este será o termo que iremos utilizar neste conteúdo.

O que são Componentes ?

Componentes são unidades de implantação. Eles são as menores entidades que podem ser implantadas como parte de um sistema. Em Java, são arquivos jar. Em Ruby, arquivos gem. Em .Net, DLLs. À medida que aplicações crescem em tamanho e complexidade, é necessária alguma maneira de organizar o código para além de classes.

Características Componentes ?

Algumas características dos componentes são:

- **Implantáveis:** são entregáveis que podem ser executados em runtime.
- **Reusáveis:** são nativamente reusáveis por diferentes aplicações, sem a necessidade de comunicação pela rede.
- **Testáveis:** podem ser testados independentemente, com testes de unidade.
- **Gerenciáveis:** em um sistema de módulos, podem ser instalados, reinstalados e desinstalados.
- **Sem estado:** classes podem ter estado, módulos não.
- **Unidades de Composição:** podem se unir a outros módulos para compor uma aplicação.

Porque componentizar ?

- Capacidade de trocar um componente por outro, com uma implementação diferente, desde que a interface pública seja mantida.
- Facilidade de compreensão de cada componente individualmente.
- Possibilidade de desenvolvimento em paralelo, permitindo que tarefas sejam divididas entre diferentes times.
- Testabilidade melhorada, permitindo um outro nível de testes, que trata um componente como uma unidade.
- Flexibilidade, permitindo o reuso de componentes em outras aplicações.

“

Os 6 princípios do desenvolvimento de Componentes são divididos em granularidade (coesão) e estabilidade (acoplamento).

Granularidade: Os Princípios de Coesão de Componentes

- **Princípio da Equivalência entre Reúso e a Distribuição** - *The Reuse Release Equivalence Principle* (REP): a granularidade do reuso é a granularidade do lançamento. Ou todas as classes em um componente são reutilizáveis ou nenhuma delas são.
- **Princípio do Reuso Comum** - *The Commom Reuse Principle* (CRP): as classes em um componente são todas reutilizadas juntas. Se você reusa uma classe no pacote, reusará todas elas.
- **Princípio do Fechamento Comum** - *The Commom Closure Principle* (CCP): as classes em um componente devem ser fechadas em conjunto contra os mesmos tipos de alterações. Uma alteração que afeta um componente afeta todas as classes nesse componente e nenhuma em outro componente.

Os Princípios da Coesão de Componentes

- Ao pensarmos em componentes, uma grande questão é: como "fatiar" o código para definir o que deve ficar em cada componente?
- Ou melhor: quais classes devem ficar em quais módulos? Quais as **responsabilidades** de um componente?
- Coesão, algo importante no nível de classes, também é importante no nível de componentes?
- Por isso, Uncle Bob, define alguns princípios relativos à **coesão** dos componente.

1.
The Release Reuse Equivalency Principle

“A granularidade de reuso não deve ser menor que a granularidade da entrega.”

“

O REP é um princípio de nível de componente. Reutilizar refere-se a um grupo de classes ou componentes reutilizáveis. Entrega se refere a publicá-lo com um número de versão. Este princípio diz que tudo o que você liberar deve ser reutilizável como uma unidade coesa. Não deve ser uma coleção aleatória de classes não relacionadas.

O que é Granularidade ?

Granularidade é a extensão à qual um sistema é dividido em partes pequenas. Ela é a “extensão até a qual uma entidade grande é subdividida. Por exemplo, um quintal dividido em centímetros possui granularidade mais baixa que um quintal dividido em metros.”

- Corresponde ao nível de detalhamento
- Quanto mais granular um sistema, mais partes.

O que é Reuso ?

“Reuso é a capacidade de reaproveitar código ou funcionalidade no desenvolvimento de um software”

Uncle Bob diz em seu artigo Granularity (MARTIN, 1996e) que **copiar e colar código dos outros não é reuso** porque você se torna dono do código que você copia:

- se algo tiver que ser adaptado, você deve alterar o código
- se você achar bugs, você deve corrigi-los
- se o autor original encontrar bugs, você que deverá ficar sabendo e descobrir o que deve ser alterado na sua cópia

Copiar e colar pode ser mais fácil em um primeiro momento mas é muito caro no longo prazo, em que manutenções são frequentes. No mesmo artigo, Uncle Bob diz:

- *Eu reuso código se, e somente se, eu nunca preciso olhar para o código fonte. Ou seja, eu espero que o código que eu estou reusando seja tratado como um produto. Não é mantido por mim. Não é distribuído por mim. Eu sou o consumidor. O autor, ou outra entidade, é responsável por mantê-lo.*

Mas como um componente pode ser entregue para reuso?

O reuso de um componente pode ser feito por terceiros ou por outros times da mesma empresa.
A entrega pode ser através de:

- um site próprio
- pelo GitHub, GitLab ou equivalente
- por uma ferramenta como Team Foundation Server da Microsoft

Uma vez fechado um grupo de alterações no código, acontece o processo de compilação, teste, documentação e entrega dos componentes, publicando uma nova versão.

Quem usa esse componente, deve ser avisado sobre a nova versão. Só os componentes que são versionados, entregues e disponibilizados por quem os mantém podem ser considerados reusáveis. Além disso, não disponibilizamos classes diretamente. O que usamos são os componentes, que são agrupamentos de classes. E usamos apenas a totalidade de um componente. Não é possível usar apenas parte do que está contido em um componente.

Conclusão do REP

Do ponto de vista do design e da arquitetura de software, esse princípio significa que as classes e módulos formados em um componente devem pertencer a um grupo coeso, ou seja “fazer sentido”. O componente não pode simplesmente consistir de uma mistura aleatória de classes e módulos, mas deve haver algum contexto ou propósito que todos esses módulos compartilhem.

As classes e módulos agrupados em um componente devem ser *passíveis de release* em conjunto. O fato de compartilharem o mesmo número de versão e o mesmo rastreamento de release e de estarem incluídos na mesma documentação de release deve fazer sentido tanto para o autor quanto para os usuários.

2.
The Common Closure Principle

"Classes que mudam juntas, permanecem juntas."

“

Esse princípio diz que os componentes devem ser uma coleção de classes que mudam pelo mesmo motivo ao mesmo tempo. Se houver motivos diferentes para mudar ou se as classes mudarem em taxas diferentes, o componente deve ser dividido.

O Princípio do Fechamento Comum

Mudanças em uma aplicação são necessárias. Se o mundo muda, a aplicação muda.

- Para isolar o impacto dessas mudanças no nível de classes, temos o SRP do SOLID. O SRP nos diz que devemos agrupar, em uma classe, apenas código que tem o mesmo motivo para ser modificado.

Mas e no nível de módulos?

- Quando há necessidade de uma modificação na aplicação, o ideal é que uma alteração seja isolada no código de apenas um componente. Dessa forma, é minimizado o trabalho de publicar uma nova versão: ao invés de entregar vários componentes, será entregue apenas um.

Uncle Bob definiu um princípio que é o equivalente ao SRP para módulos:

- *“Agrupe em módulos as classes que são modificadas pelos mesmos motivos e ao mesmo tempo. Separe em módulos diferentes as classes que são modificadas em momentos e por motivos diferentes.”*

Conclusão do CCP

CCP agrupa classes para minimizar impactos de mudanças.

- Agrupar classes que achamos que mudarão juntas
- Exigem testes de validação em conjunto quando modificadas
- Devem estar no mesmo componente para alta coesão e baixo acoplamento

3.
The Common Reuse Principle

"As classes que não são reutilizadas não devem ser agrupadas."

> O CRP afirma que você não deve depender de um componente que possui classes que você não precisa. Esses componentes devem ser divididos para que os usuários não precisem depender de classes que não usam. Isso é basicamente a mesma coisa que o *Princípio de Segregação de Interface* do SOLID.

O Princípio do Reuso Comum

- O ISP define que classes não devem depender de métodos que não usam. Isso implica em interfaces coesas, que definem contratos menores e, também, em usar interfaces para expor o mínimo possível de uma classe.

Qual o equivalente no nível de componentes?

- Não deveríamos colocar, em um mesmo componente, classes que seriam utilizadas apenas por parte dos clientes desse componente.
- Uma mudança em uma classe forçaria a publicação de uma nova versão do componente já que não é possível entregar apenas parte de um componente.
- Uma nova versão iria requerer recompilação, reteste e reimplantação de todos que usam o componente. Mesmo daqueles que não usam a classe modificada.

O equivalente ao ISP para módulos foi definido por Uncle Bob no seguinte princípio:

- *“As classes de um* componente *são reusadas em grupo. Quem reusa uma dessas classes, reusa todas.”* ou *“Não force quem usa um* componente *a depender de coisas de que eles não precisam.”*

Conclusão do CRP

CRP divide componentes em menores, pensando em evitar dependências e entregas desnecessárias.

- É o contrário do CCP as classes que não são reusadas juntas não deveriam estar agrupadas.
- Dificuldade de reusar um componente sem carregar classes e componentes que não será utilizado (Parecido com o principio ISP do SOLID).
- A modificação de um componente só devia afetar as classes que o usa.

A tensão entre os princípios de coesão de componentes

Uncle Bob diz, no livro Clean Architecture (MARTIN, 2017), que há uma tensão entre o REP, o CCP e o CRP:

- REP agrupa classes para facilitar o reuso por outros projetos
- CCP agrupa classes para minimizar impactos de mudanças
- CRP divide componentes em menores, pensando em evitar dependências e entregas desnecessárias

A tensão entre os princípios de coesão de componentes

- O REP e CCP levariam à inclusão de mais classes, tendendo a criar componentes maiores, com granularidade mais alta. Porém, os motivos de agrupamento são diferentes: o REP por reuso, o CCP por mudanças.
- Já o CRP levaria à exclusão de classes, tendendo a criar componentes menores, com granularidade mais baixa.
- Se a granularidade for muito alta, com muitas classes em poucos componentes, uma mudança levará a novas versões sem muitas novidades para boa parte de quem usa esses módulos.
- Se, por outro lado, a granularidade dos componentes for muito baixa, com poucas classes em muitos componentes, o impacto de uma mudança levará a alterações em diversos componentes diferentes.
- Saber qual princípio deve ser favorecido é uma decisão que deve levar em conta o contexto atual do projeto. À medida que o projeto avança, o contexto muda e as decisões devem ser reavaliadas.

Estabilidade: Os Princípios do Acoplamento de Componentes

- **Princípio das Dependências Acíclicas** - *The Acyclic-Dependencies Principle* (ADP): não permitir ciclos no grafo pacote-dependência.

- **Princípio das Dependências Estáveis** - *The Stable-Dependencies Principle* (SDP): dependa na direção da estabilidade.

- **Princípio das Abstrações Estáveis** - *The Stable-Abstractions Principle* (SAP): um pacote deve ser tão abstrato quanto estável.

Os Princípios de Acoplamento de Componentes

- Dificilmente, uma aplicação será composta por apenas um componente. Mesmo se o código da aplicação em si estiver modularizado, usaremos componentes (os JARs) de bibliotecas e frameworks. E esses componentes, por sua vez, comumente tem outros componentes como **dependência**.

- Podemos dizer, portanto, que (quase) todo componente tem **acoplamento** com outros componentes.

- É importante minimizar o acoplamento e ter o controle dessas dependências.

- Por isso, Uncle Bob, define alguns princípios relativos ao acoplamento de componentes.

4.
The Acyclic Dependencies Principle

"As dependências entre componentes não devem formar ciclos."

“

O princípio ADP significa que você não deve ter nenhum ciclo de dependência em seu projeto. Por exemplo, se o componente A depende do componente B e o componente B depende do componente C e o componente C depende do componente A, então você tem um ciclo de dependência.

O Princípio das Dependências Acíclicas

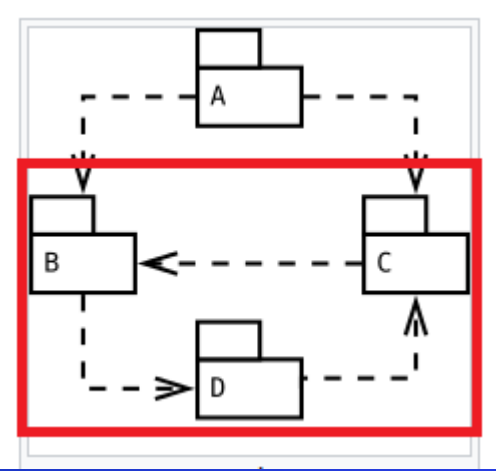
Alguma vez você já trabalhou o dia todo, fez com que alguma coisa funcionasse e então foi para casa, só pra chegar na manhã seguinte e descobrir que as suas coisas não funcionam mais? Por que não funcionam? Porque alguém ficou até mais tarde e mudou algo do qual você dependia! isso é chamado de "a síndrome da manhã seguinte".

A "síndrome da manhã seguinte" ocorre em ambientes de desenvolvimento onde muitos desenvolvedores modificam os mesmos arquivos-fonte. Em projetos relativamente pequenos, com apenas alguns desenvolvedores, isso não é um grande problema. Mas à medida que o tamanho do projeto e da equipe de desenvolvimento aumentam, as manhãs seguintes podem ficar bem parecidas com um pesadelo. Não é incomum passarem-se semanas sem que a equipe consiga criar uma versão estável do projeto. Em vez disso, todo mundo continua modificando o seu código, tentando fazê-lo funcionar com as últimas mudanças feitas por outra pessoa.

Problemas Dependências Acíclicas

Veja abaixo um exemplo de dependência acíclica:

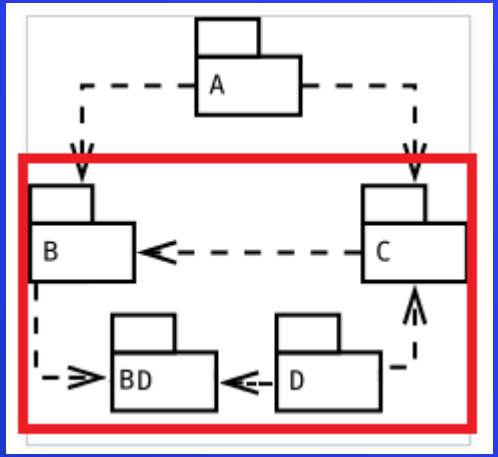
O componente A depende de componentes B e C. O Componente B por sua vez, depende do componente D, que depende do componente C, que por sua vez depende do componente B. As últimas três dependências criam um ciclo, que deve ser quebrado para aderir ao princípio das dependências acíclicas.



Solução para Dependências Acíclicas

Veja abaixo um exemplo de solução para dependência acíclica:

Os ciclos podem ser quebrado de vária maneiras uma delas é criando um novo componente e levando as dependências comum para o mesmo. As classes que o D precisava são extraídas da B e colocadas em um novo componente denominado BD. Tanto a B quanto o D dependem do novo componente quebrando o ciclo acíclico.

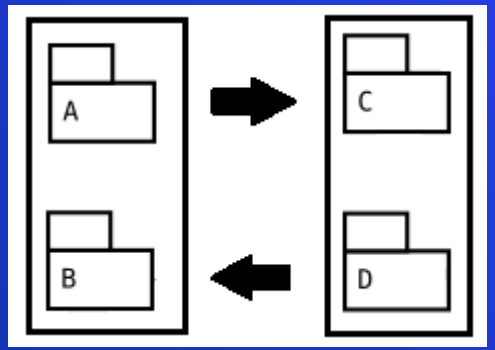


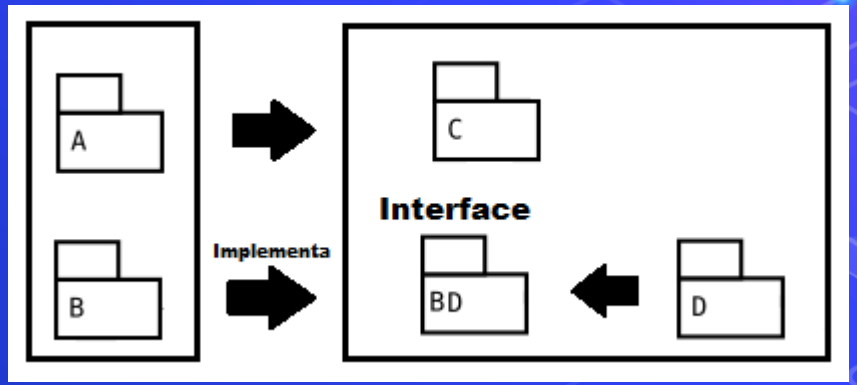
O Princípio das Dependências Acíclicas

Veja abaixo um exemplo de dependência acíclica:

Aqui temos dois componentes que estão ligados por um ciclo. A classe A depende da classe C, e a classe D depende da classe B. Nós quebramos o ciclo aplicando o principio DIP do SOLID, invertendo a dependência entre D e B. Isso é feito adicionando uma nova interface, BD, a B. Essa interface tem todos os métodos que D necessita. D usa essa interface e B a implementa.

Observe o posicionamento de BD. É colocado no pacote com a classe que o utiliza. Este é um padrão que você verá repetido ao longo dos estudos de caso que tratam das idades dos componentes. As interfaces são frequentemente incluídas no componente que as utiliza, e não no componente que as implementa.





Conclusão do ADP

Em geral, sempre é possível quebrar uma cadeia de dependência cíclica. As duas estratégias mais comuns são:

- Princípio de inversão de dependência
- Crie um novo componente e mova as dependências comuns para lá.

5.
The Stable Dependencies Principle

"Dependa na direção da estabilidade."

“

Este princípio diz que as dependências devem ser na direção da estabilidade. Ou seja, componentes menos estáveis devem depender de componentes mais estáveis. Isso minimiza o efeito da mudança. Alguns componentes devem ser voláteis. Tudo bem, mas você não deve fazer com que os componentes estáveis dependam deles.

O que é Estabilidade ?

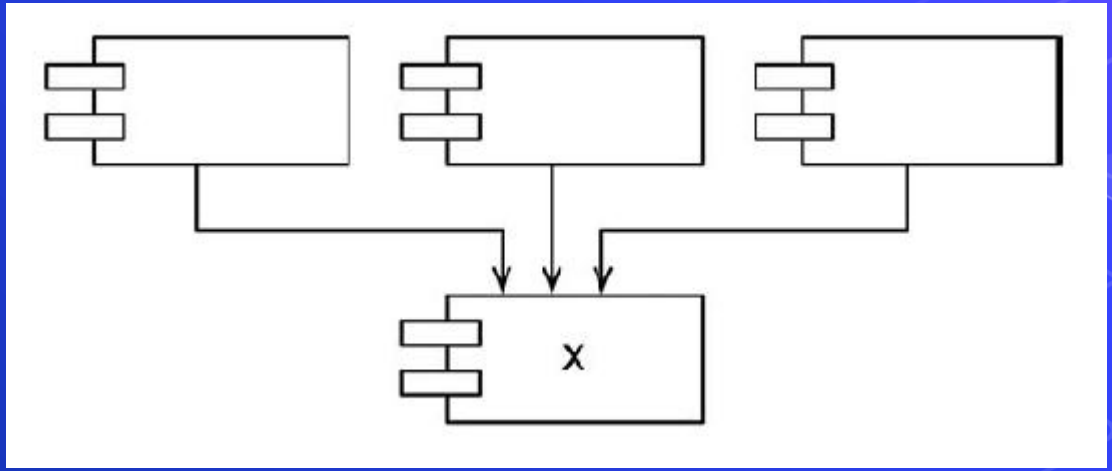
- A **estabilidade** está relacionada com a quantidade de esforço necessário para fazer uma alteração, ou seja módulos estáveis mudam pouco e módulos instáveis mudam frequentemente.

O Princípio das Dependências Estáveis

- Os projetos não podem ser completamente estáticos. Alguma volatilidade é necessária para que o design seja mantido. Ao aplicarmos o Princípio do Fechamento Comum (CCP), criamos componentes sensíveis a determinados tipos de mudanças, mas imunes a outros. Alguns desses componentes são *projetados* para serem voláteis. Portanto, *esperamos* que eles mudem.
- Um componente que seja difícil de mudar não deve ser dependente de qualquer componente que esperamos que seja volátil. Caso contrário, o componente volátil também será difícil de mudar.
- É uma crueldade do software que um módulo que você projetou para ser fácil de mudar seja transformado em algo difícil de mudar devido à simples inclusão de uma dependência por outra pessoa. Nenhuma linha do código-fonte do seu módulo precisa mudar, mas, ainda assim, as mudanças no seu módulo de repente se tornam mais complexas. Ao aplicarmos o Princípio das Dependências Estáveis (SDP), garantimos que os módulos difíceis de mudar não sejam dependentes de módulos projetados para serem fáceis de mudar.

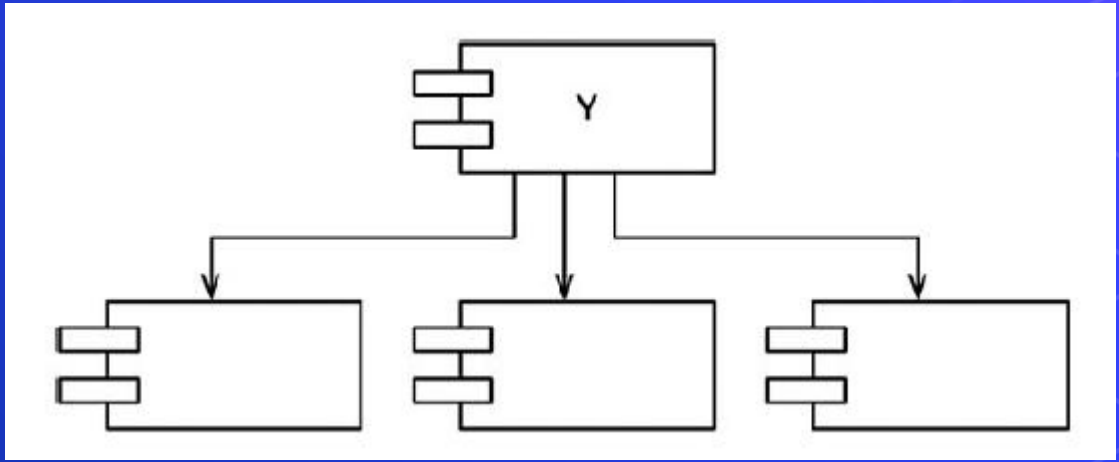
Componente Estável

Este componente tem três componentes dependendo dele e, portanto, tem três boas razões para não mudar. Dizemos que é responsável por esses três componentes. Por outro lado, [X] não depende de nada, então não tem influência externa para fazê-lo mudar. Nós dizemos que é **independente**.



Componente Instável

Por outro lado, mostra um componente muito instável. [Y] não tem outros componentes dependendo dele. Dizemos que não tem responsabilidades. [Y] também tem três componentes dos quais depende, portanto, as alterações podem vir de três fontes externas. Nós dizemos que [Y] é **dependente**.



Métricas de Estabilidade

Podemos calcular a estabilidade de um componente usando um trio de métricas simples.

Supondo que você seja a classe calculada:

- **Ce:** Classes que você depende
- **Ca:** Classes que depende de você

- **Ca:** Acoplamento Aferente, ou seja, o número de classes fora do componente que dependem de classes dentro do componente . (isto é, dependências de entrada);
- **Ce:** Acoplamento Eferente, ou seja, o número de classes fora do componente que as classes dentro do componente dependem (isto é, dependências de saída);
- **I:** Instabilidade, uma métrica que tem o intervalo de “zero” a “um”: [0,1].

Fórmula: I = Ce / Ca + Ce

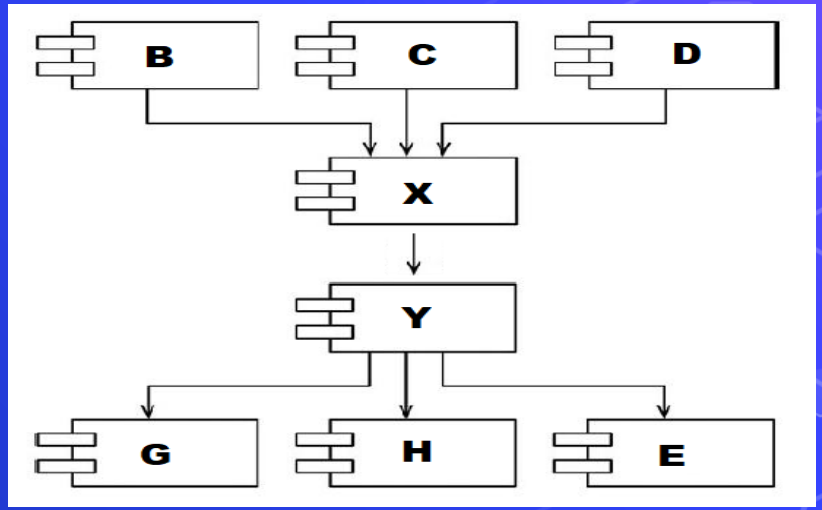
- I = 0 indica um componente estável ao máximo.
- I = 1 indica um componente instável ao máximo.

Violando o SDP

Este exemplo mostra como o SDP pode ser violado. [Y] é um componente que pretendemos tornar fácil de alterar. Queremos que o [Y] seja instável. No entanto, alguns desenvolvedores, trabalhando no pacote chamado [X], colocaram uma dependência no [Y]. Isso viola o SDP, pois a métrica "I" para [X] é muito menor do que a métrica "I" para [Y]. Como resultado, o [Y] não será mais fácil de alterar. Uma mudança para o [Y] nos forçará a lidar com o [X] e todos os seus dependentes.

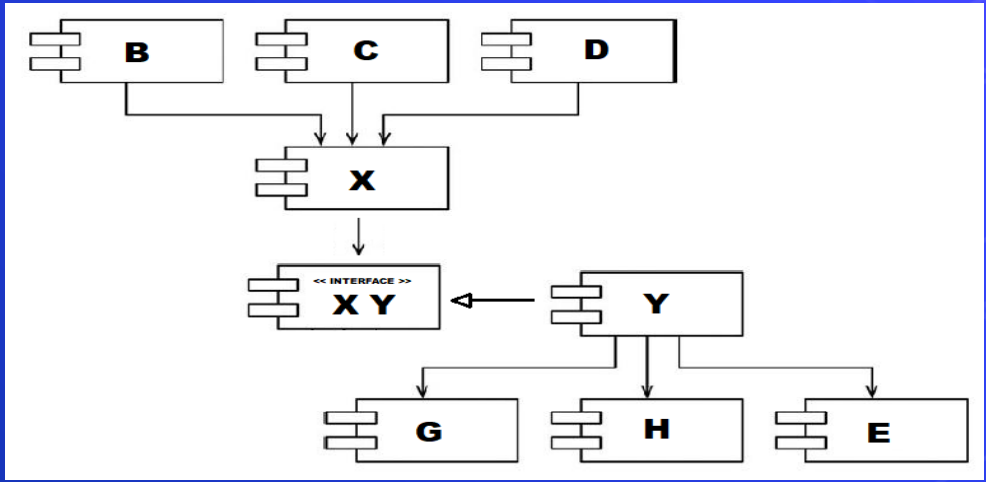
- **X:** I = 1 / (3 + 1)
- **Y:** I = 3 / (1 + 3)

- [X] : I = 0.25
- [Y] : I = 0.75



Aplicando o SDP

Podemos corrigir isso aplicando o DIP do SOLID. Criamos uma interface e a colocamos em um componente chamado [XY]. Garantimos que essa interface declare todos os métodos que [X] precisa usar. Então, fazemos [Y] implementar essa interface como indicado no exemplo abaixo. Isso quebra a dependência entre [X] e [Y] e força ambos os componentes a dependerem de [XY]. O [XY] é muito estável (*I* = 0) e o [Y] retém a sua instabilidade necessária (*I* = 0.75). Todas as dependências agora fluem na direção de um *I decrescente*.



Componentes Abstratos

Você pode achar estranho criar um componente nesse exemplo, [XY] que não contém nada além de uma interface. Esse componente não contém código executável. No entanto, essa é uma tática muito comum e necessária quando usamos linguagens estaticamente tipadas como Java e C#. Esses componentes abstratos são muito estáveis e, portanto, alvos ideais para dependências de componentes menos estáveis.

Conclusão do SDP

Dependências devem acontecer do componente mais instável para o mais estável. No exemplo de violação do SDP o "I" de [Y] é maior que o "I" de [X] que é um relacionamento ruim. Uma boa maneira de aplicar o **Princípio das Dependências Estáveis** é utilizar o **Princípio da Inversão da Dependência** do **SOLID,** com essa técnica podemos projetar softwares mais flexíveis**.**

6.
The Stable Abstractions Principle

“Componentes estáveis devem ser componentes abstratos."

> O princípio SAP declara que quanto mais estável um componente é, mais abstrato ele deve ser, ou seja, mais classes abstratas ele deve conter. As classes abstratas são mais fáceis de estender, portanto, isso evita que os componentes estáveis se tornem muito rígidos.

Onde devemos colocar as Políticas de Alto Nível ?

- Algum software no sistema não deve mudar com muita frequência. Esse software representa a arquitetura de alto nível e as decisões sobre políticas. Não queremos que essas decisões arquiteturais e de negócios sejam voláteis. Assim, o software que engloba as políticas de alto nível do sistema deve ser colocado em componentes estáveis (*I* = 0). Os componentes instáveis (*I* = 1) devem conter apenas software volátil software que podemos mudar com rapidez e facilidade.

- Contudo, se as políticas de alto nível forem colocadas em componentes estáveis, o código-fonte que representa essas políticas será difícil de mudar. Isso pode tornar inflexível a arquitetura geral. Como um componente com estabilidade máxima (*I* = 0) pode ser flexível o bastante para resistir à mudança? A resposta está no OCP. Esse princípio nos diz que é possível e desejável criar classes flexíveis o bastante para serem estendidas sem que haja modificações. Que tipo de classe está de acordo com esse princípio? As **classes *abstratas***.

O Princípio de Abstrações Estáveis

- O Princípio de Abstrações Estáveis (SAP — Stable Abstractions Principles) estabelece uma relação entre estabilidade e abstração. Por um lado, ele diz que um componente estável deve também ser abstrato para que essa estabilidade não impeça a sua extensão. Por outro lado, ele afirma que um componente instável deve ser concreto, já que a sua instabilidade permite que o código concreto dentro dele seja facilmente modificado.
- Assim, para que um componente seja estável, ele deve consistir de interfaces e classes abstratas de modo que possa ser estendido. Os componentes estáveis extensíveis são flexíveis e não restringem demais a arquitetura.
- Os princípios SAP e SDP, quando combinados, formam o DIP aplicável aos componentes. Isso porque o SDP indica as dependências que devem apontar na direção da estabilidade e o SAP determina que a estabilidade implica em abstração. Logo, ***as dependências devem apontar na direção da abstração***.
- O DIP, no entanto, é um princípio que lida com classes de definição precisa. Uma classe é abstrata ou não. A combinação entre o SDP e o SAP se aplica aos componentes e permite que um componente seja parcialmente abstrato e parcialmente estável.

Medindo Abstrações

A métrica A é uma medida do nível de abstração de um componente. Seu valor corresponde à razão entre as interfaces e classes abstratas de um componente e o número total de classes desse mesmo componente.

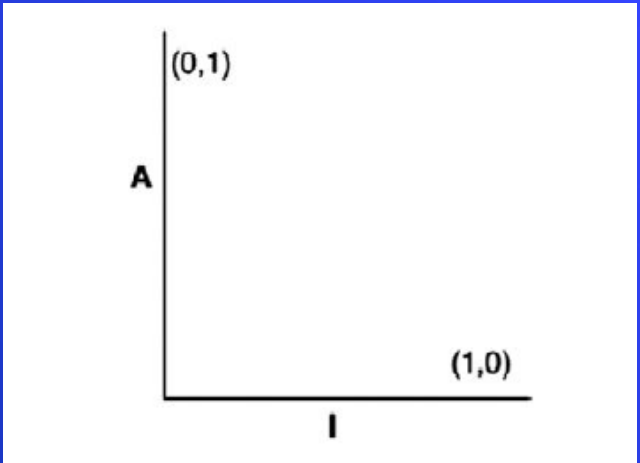
- *Nc:* número de classes de um componente.
- *Na:* número de classes abstratas e interfaces do componente.
- *A: nível de* abstração. *A = Na , Nc.*

A métrica *A* varia de 0 a 1. O valor 0 indica que o componente não tem nenhuma classe abstrata. Já o valor 1 implica que o componente só contém classes abstratas.

Fórmula: A = Na / Nc

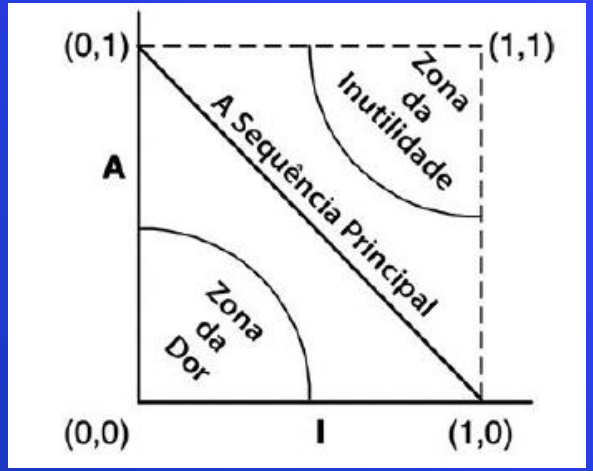
A Sequência Principal

- Agora, podemos definir a relação entre estabilidade (*I*) e abstração (*A*). Para isso, criamos um gráfico com *A* no eixo vertical e *I* no eixo horizontal. Ao incluirmos os dois tipos "bons" de componentes nesse gráfico, observamos os componentes abstratos e com estabilidade máxima no canto esquerdo superior, em (0, 1). Os componentes concretos e com instabilidade máxima estão no canto inferior direito, em (1, 0).



A Sequência Principal

- Como não podemos impor que todos os componentes fiquem em (0,1) ou em (1,0), devemos supor que um lugar geométrico de pontos no gráfico *A* / *I* define posições razoáveis para os componentes. Podemos deduzir qual é esse lugar geométrico encontrando as áreas onde os componentes ***não* devem estar** isto é, as **zonas de *exclusão*.**



A Zona da Dor

- Considere um componente na área de (0, 0). Esse componente é altamente estável, concreto e, por ser rígido, indesejável. Não pode ser estendido porque não é abstrato e é muito difícil de mudar por causa da sua estabilidade. Assim, não esperamos normalmente observar componentes bem projetados próximos de (0, 0). A área em torno de (0, 0) é uma zona de exclusão chamada ***Zona da Dor***.

- Um exemplo de software próximo da área de (0, 0) é uma biblioteca de utilitários concretos. Embora essa biblioteca tenha uma métrica *I* de 1, ela pode não ser realmente volátil. Considere o componente **String**, por exemplo. Embora todas as classes dentro dele sejam concretas, ele é tão comumente usado que mudá-lo poderia provocar um caos. Portanto, o String não é volátil.

- Os componentes não voláteis são inofensivos na zona (0, 0) já que não têm probabilidade de serem mudados. Por isso, apenas os componentes de software voláteis são problemáticos na Zona da Dor. Na Zona da Dor, quanto mais volátil, mais "doloroso" é um componente. De fato, podemos considerar a volatilidade como o terceiro eixo do gráfico. Com base nessa compreensão, o exemplo anterior indica apenas o plano mais doloroso, onde a volatilidade = 1.

A Zona da Inutilidade

- Considere um componente próximo de (1, 1). Essa posição é indesejável porque indica abstração máxima sem dependentes. Componentes como esses são inúteis. Logo, essa área é chamada de ***Zona da Inutilidade***.

- As entidades de software que habitam essa região são um tipo de resíduos. São normalmente restos de classes abstratas que ninguém nunca implementou. Encontramos esses componentes na base de códigos de vez em quando, sem uso.

- Um componente enraizado no interior da Zona da Inutilidade deve conter uma fração significante dessas entidades. Claramente, a presença dessas entidades inúteis é indesejável.

Evitando as Zonas de Exclusão

- Parece claro que os nossos componentes mais voláteis precisam manter a maior distância possível de ambas as zonas. A posição dos pontos mais distante de cada zona é a linha que conecta (1, 0) e (0, 1), que é conhecido como linha de ***Sequência Principal***.

- Na Sequência Principal, um componente não é nem "abstrato demais“ para a sua estabilidade nem "instável demais" para a sua abstração. Não é inútil nem particularmente doloroso. É alvo de dependências proporcionalmente à sua abstração e depende de outros na medida em que é concreto.

- A posição mais recomendável para um componente é em uma das duas extremidades da Sequência Principal. Os bons arquitetos lutam para colocar a maioria dos componentes nessas extremidades. Contudo, uma pequena fração dos componentes de um sistema grande não é perfeitamente abstrata nem perfeitamente estável. Esses componentes apresentarão as melhores características se estiverem na Sequência Principal *ou próximos* dela.

Distância da Sequência Principal

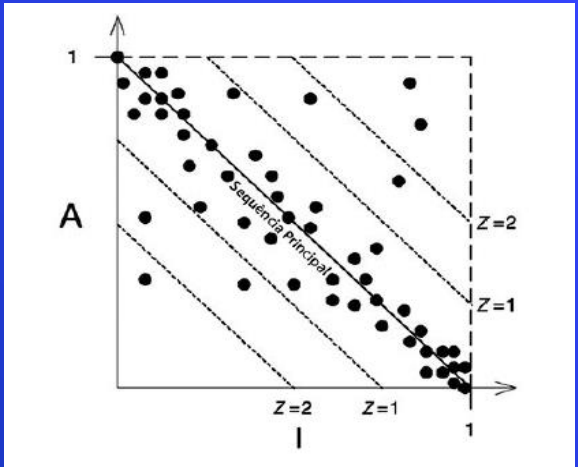
Se é desejável que os componentes estejam na Sequência Principal ou próximos dela, podemos então criar uma métrica que determine a distância entre a posição atual do componente e o seu local ideal.

- *D*: distância. $D = |A+I-1|$. A variação dessa métrica é [0, 1]. O valor 0 indica que o componente está diretamente na Sequência Principal. O valor 1 indica que o componente está o mais distante possível da Sequência Principal.

Com essa métrica, um design pode ser analisado com base na sua conformidade geral à Sequência Principal. Podemos calcular a métrica *D* de cada componente. Qualquer componente com um valor *D* que não esteja próximo de zero pode ser reexaminado e reestruturado.

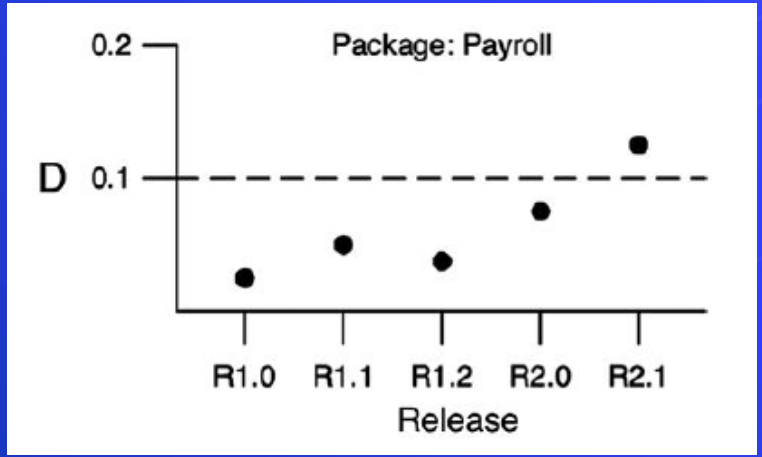
Distância da Sequência Principal

- No gráfico de dispersão do exemplo abaixo, vemos que a maioria dos componentes está situada ao longo da Sequência Principal, embora alguns deles estejam a uma distância superior a um desvio padrão (Z = 1) em relação à média. Vale a pena examinar esses componentes anômalos mais de perto. Por alguma razão, eles ou são muito abstratos e têm poucos dependentes ou são muito concretos e têm muitos dependentes.



Distância da Sequência Principal

- Outra maneira de usar as métricas é calcular a métrica *D* de cada componente em função do tempo. O gráfico do exemplo abaixo é um modelo desse gráfico. Observe que algumas dependências estranhas têm se entranhado no componente Payroll ao longo dos últimos (releases R). O gráfico indica um limite de controle em D = 0.1. Como o ponto R2.1 excedeu esse limite de controle, vale a pena descobrir por que esse componente está tão longe da sequência principal.



Conclusão do SAP

As *métricas de gestão de dependências* determinam a conformidade de um projeto em relação a um padrão de dependência e abstração que é considerado "bom". Existem dependências boas e ruins. Contudo, uma métrica não é um uma verdade absoluta, ela é apenas uma forma de medir a qualidade do software.

É possível que as métricas do principio SAP seja adequado para certos aplicativos, mas não para outros. Talvez métricas bem melhores sejam utilizadas para avaliar a qualidade de um projeto.

Obrigado!

Alguma pergunta?

Você pode me encontrar em:

- willian_brito00@hotmail.com
- www.linkedin.com/in/willian-ferreira-brito
- github.com/willian-brito