Princípios do SOLID

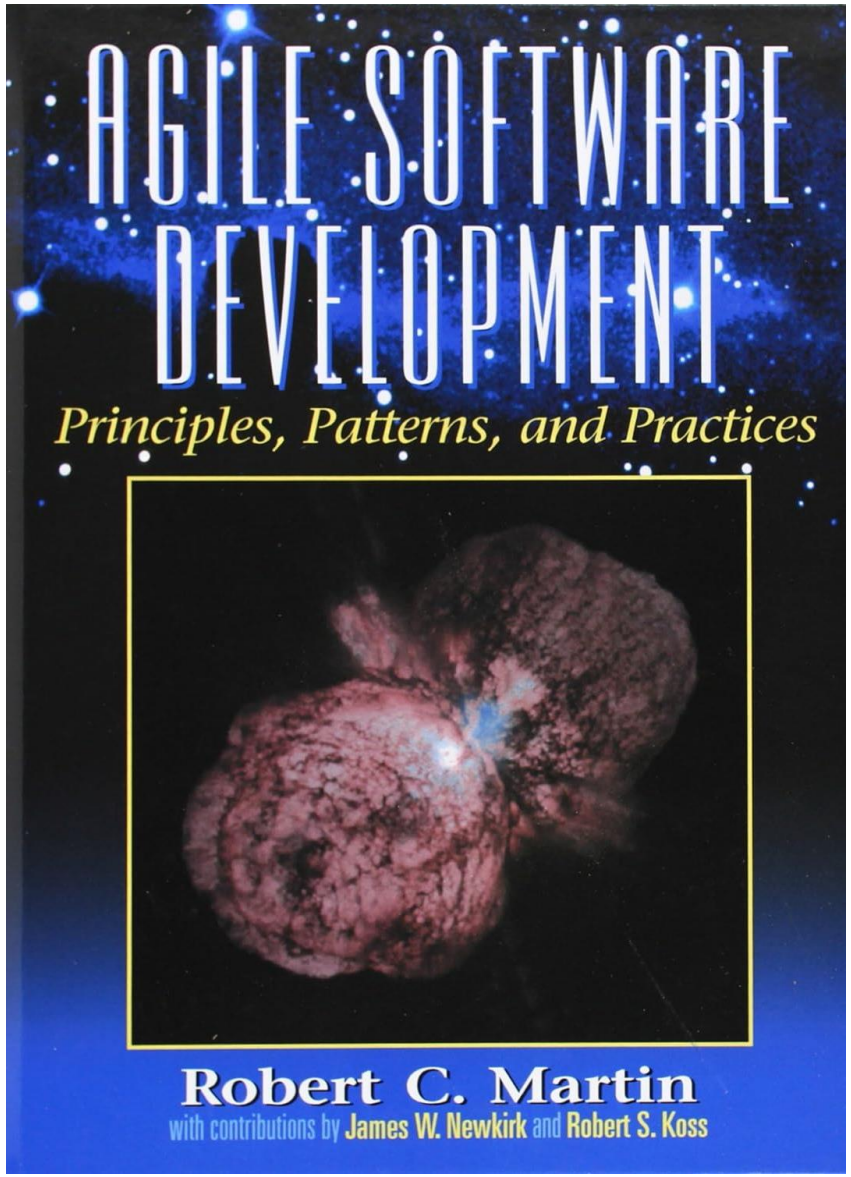
Design Patterns

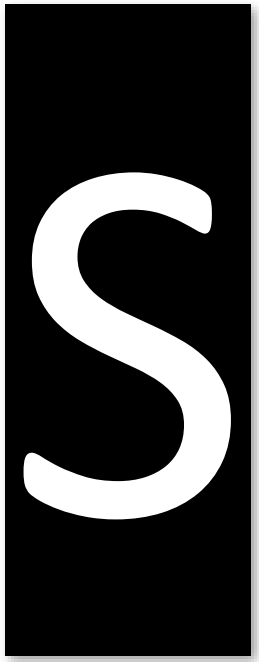
Prof. Me. Jefferson Passerini

Princípios de SOLID

- **Características de um bom projeto de software**

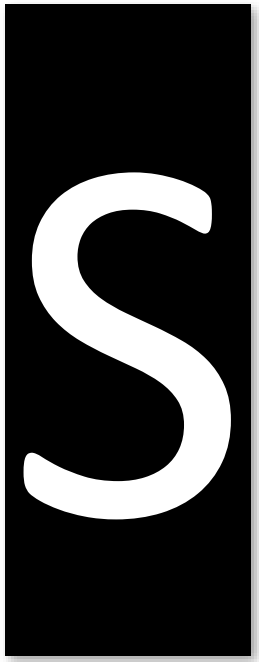
- Agora que conhecemos os princípio básicos de projeto, vamos dar uma olhada em cinco deles que são comumente conhecidos com os ***princípios SOLID***.
- Robert Martin os introduziu no livro ***Agile Software Development, Principles, Patterns, and Pratices*.**
- **O SOLID é uma sigla mnemônica em inglês para cinco princípios de projeto destinados a fazer dos projetos de software algo mais compreensivo, flexível, e sustentável.**
- Como tudo, usar esses princípios sem cuidado pode causar mais males que bem.
- O custo de aplicar estes princípios na arquitetura de um programa pode torná-lo mais complicado do que deveria ser. Tente ser pragmático e não tome tudo o que é apresentado aqui como um dogma.





- ***Single Responsability Principle***
- **Princípio de responsabilidade única**
- ***Uma classe deve ter apenas uma razão para mudar.***

- Uma classe deve ser responsável por uma parte única da funcionalidade fornecida pelo software;
- Faça aquela funcionalidade ser encapsulada pela classe.
- O objetivo principal deste princípio é reduzir a complexidade:
  - Não escreva 200 linhas de código para resolver o problema, quebre-o em vários métodos menores e mais simples.

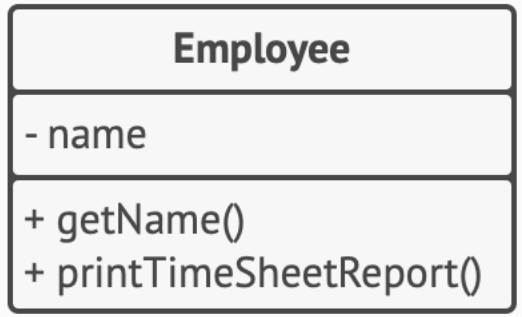


- ***Single Responsability Principle***
- Problemas surgem conforme o programa cresce e se modifica constantemente:
  - A classe fica tão grande e complexa que nem se lembra mais de qual era sua funcionalidade principal;
  - Se a sua classe faz muitas coisas, o número de modificações é maior pois tem que se mexer no código a cada evento em qualquer uma das “coisas” que ela faz.

- Se houver a percepção que está ficando difícil focar em aspectos específicos do programa, tem que analisar se não é o momento de dividir em mais classes.

S

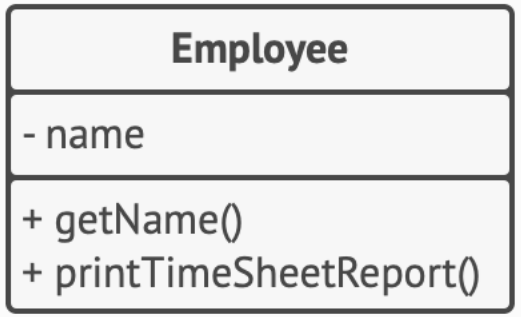
- ***Single Responsability Principle***
- A classe Empregado tem vários motivos para mudar;
- O 1º motivo é relacionado a sua função principal: “gerenciar os dados dos empregados”;
- O método que gera dados empregados no tempo pode ser modificado constantemente, e não se refere aos dados de empregado e sim com formato de um relatório.



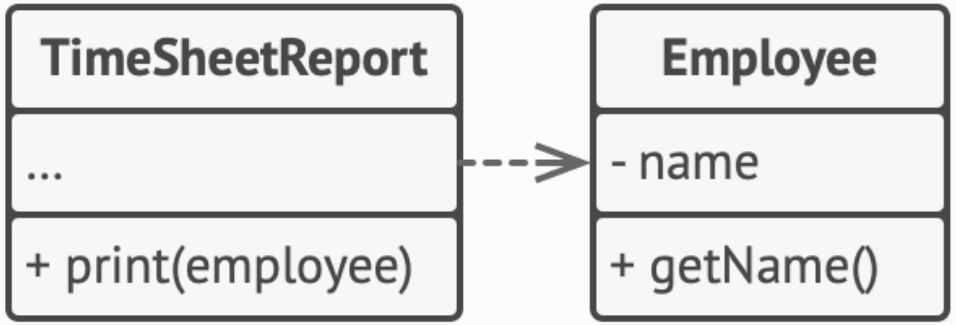
ANTES: a classe contém vários comportamentos diferentes.

S

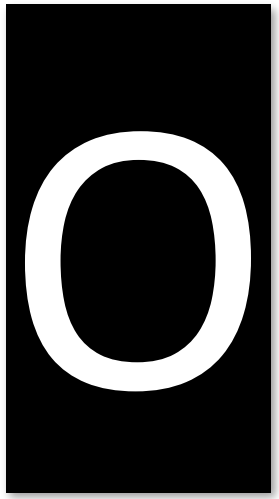
- ***Single Responsability Principle***
- Mova esse comportamento para uma classe separada;
- Essa mudança permite que você também mova outros comportamentos para essa classe especifica.



ANTES: a classe contém vários comportamentos diferentes.



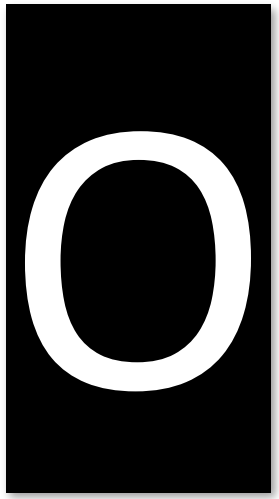
DEPOIS: o comportamento adicional está em sua própria classe.



- ***Open/Closed Principle***
- ***Princípio do Aberto/Fechado***

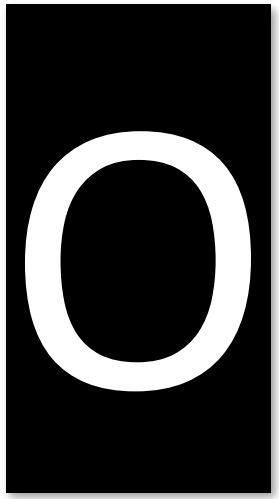
- **As classes devem ser abertas para a extensão, mas fechadas para a modificação.**

- A ideia principal é evitar que o código existente quebre quando você implementar novas funcionalidades.



- ***Open/Closed Principle***

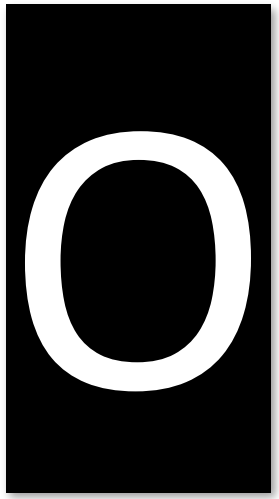
- **Uma classe é aberta quando você pode estendê-la, produzir subclasses, adicionar novos métodos, sobrescrever o comportamento base.**
- Algumas linguagens de programação permitem que se restrinja a futura extensão de uma classe com a palavra-chave como “final”, após isso a classe não é mais aberta.
- A classe é fechada (você também pode chamá-la de completa) se ela estiver 100% pronta para ser usada por outras classes—sua interface está claramente definida e não será mudada no futuro.



- ***Open/Closed Principle***

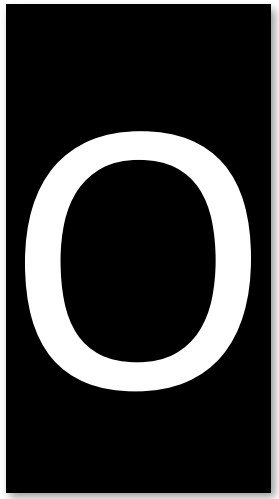
- Este conceito é um tanto confuso pois as palavras aberta e fechada parecem mutualmente exclusivas

- Mas em se tratando deste princípio, uma classe pode ser tanto aberta (para extensão) e fechada (para modificação) ao mesmo tempo.



- ***Open/Closed Principle***

- Se a sua classe já foi desenvolvida, testada e revisada, e incluída em algum *framework* ou é de alguma forma utilizada em sua aplicação, tentar mexer com seu código é arriscado.
- Ao invés de mudar o código da classe diretamente, você pode criar subclasses e sobrescrever partes da classe original que você quer que se comporte de forma diferente.
- Você vai cumprir seu objetivo mas também não quebrará os clientes existentes da classe original.



- ***Open/Closed Principle***

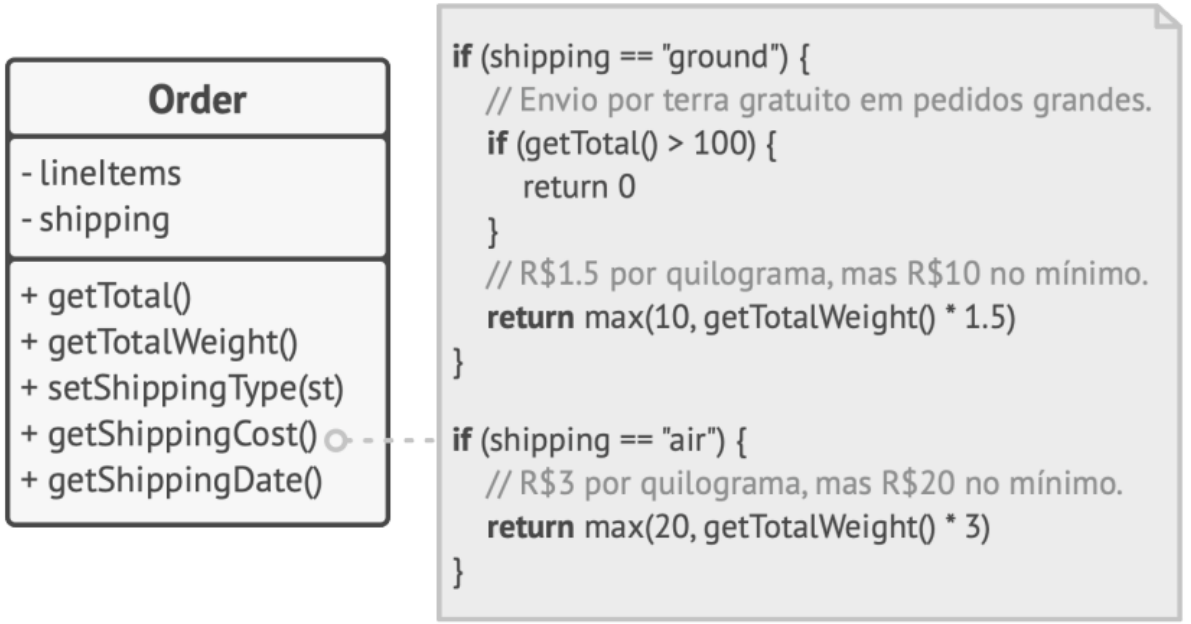
- Atenção!
- Este princípio não foi feito para ser aplicado para todas as mudanças de uma classe.
- Se você sabe que há um bug na classe, apenas vá e corrija-o; não crie uma subclasse para ele. Uma classe filha não deveria ser responsável pelos problemas da classe mãe.

O

- ***Open/Closed Principle***

- Você tem uma aplicação e-commerce com uma classe **Pedido** que calcula os custos de envio e todos os métodos de envio estão codificados dentro da classe.

- Se você precisa adicionar um novo método de envio, você terá que mudar o código da classe Pedido , arriscando quebrá-la.



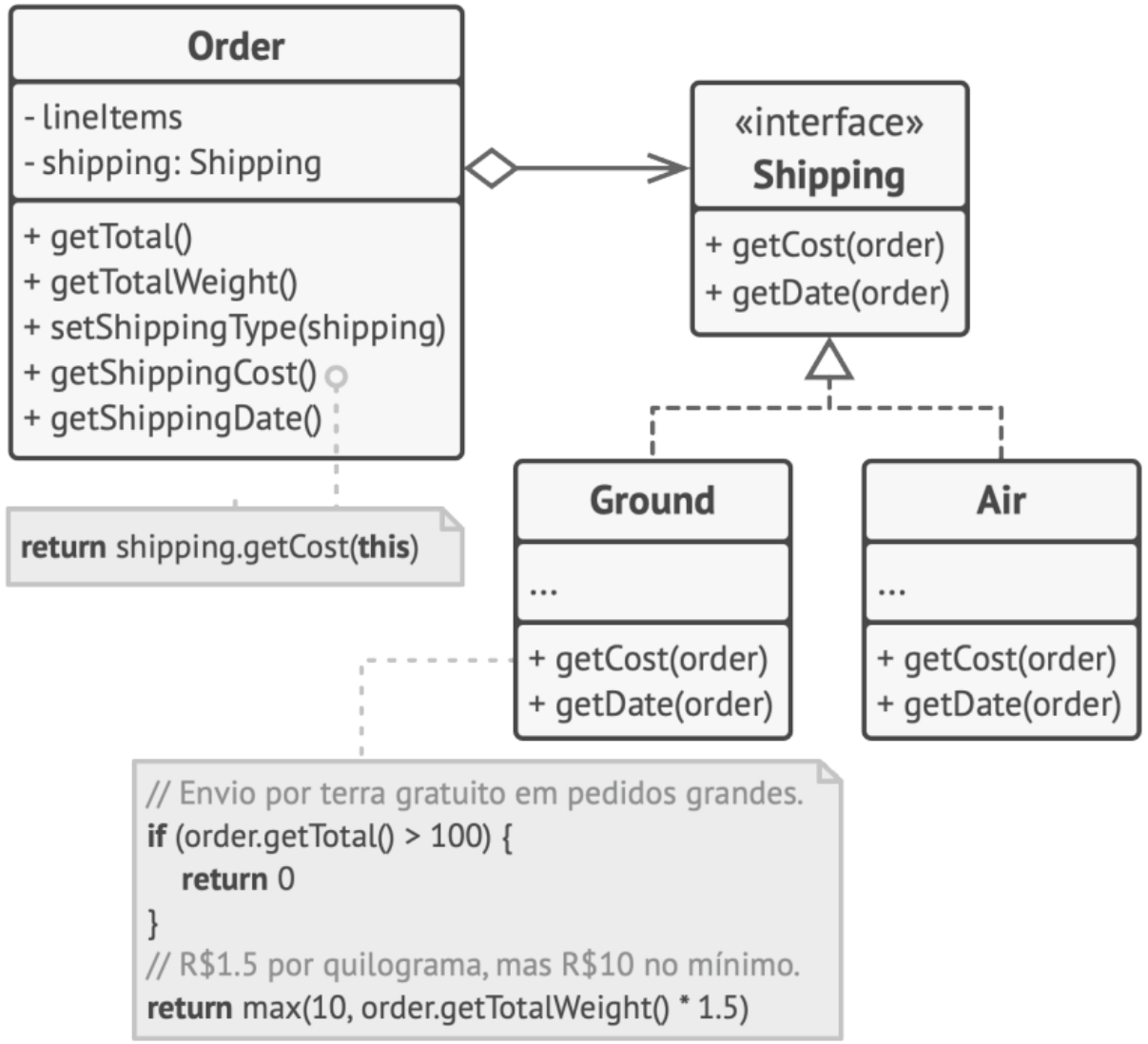
ANTES: você tem que mudar a classe `Pedido` sempre que adicionar um novo método de envio na aplicação.

O

- ***Open/Closed Principle***

- Aplicando o padrão ***Strategy***, comece extraindo os métodos de envio para classes separadas com uma interface em comum.

- Agora quando você precisa implementar um novo método de envio, você pode derivar uma nova classe da interface Envio sem tocar em qualquer código da classe Pedido.



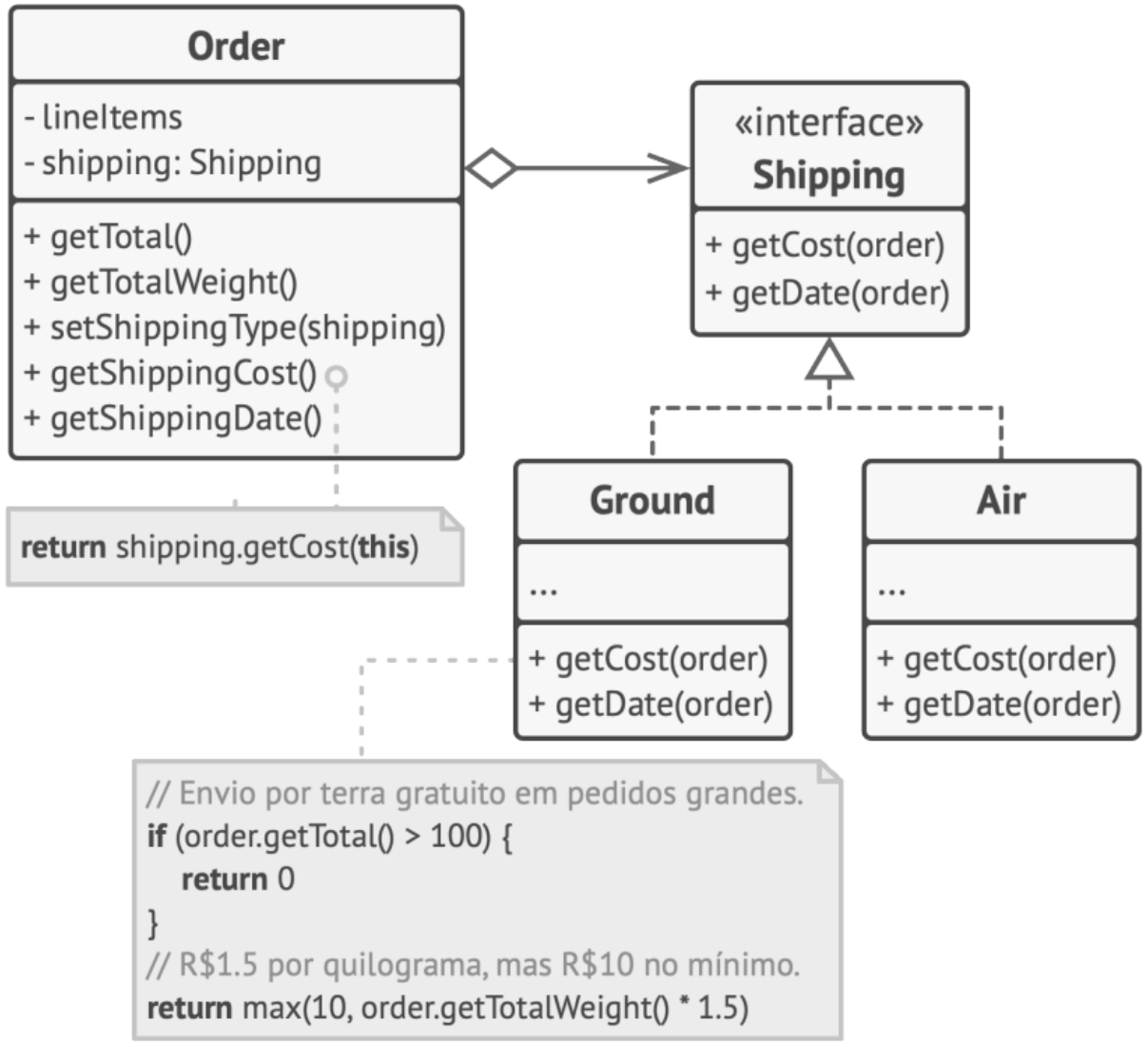
DEPOIS: adicionando um novo método de envio não necessita mudanças em classes existentes.

O

- ***Open/Closed Principle***

- O código cliente da classe Pedido irá ligar os pedidos com o objeto do envio com a nova classe sempre que o usuário selecionar estes métodos de envio na interface gráfica.

- Essa solução permite que você mova o cálculo de tempo de envio para classes mais relevantes, de acordo com o ***princípio de responsabilidade única.***



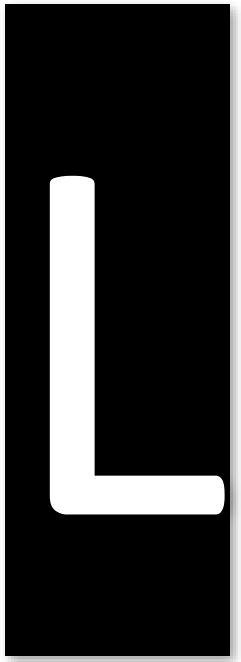
DEPOIS: adicionando um novo método de envio não necessita mudanças em classes existentes.

L

- ***Liskov Substitution Principle***
- ***Princípio de Substituição de Liskov***

- **Quando estendendo uma classe, lembre-se que você deve ser capaz de passar objetos da subclasse em lugar de objetos da classe mãe sem quebrar o código cliente.**

- Este princípio foi nomeado por **Barbara Liskov,** que o definiu em 1987em seu trabalho ***Data Abstraction and Hierarchy :***
  - https://www.semanticscholar.org/paper/Data-Abstraction-and-Hierarchy-Liskov/36bebabeb72287ad9490e1ebab84e7225ad6a9e5?p2df



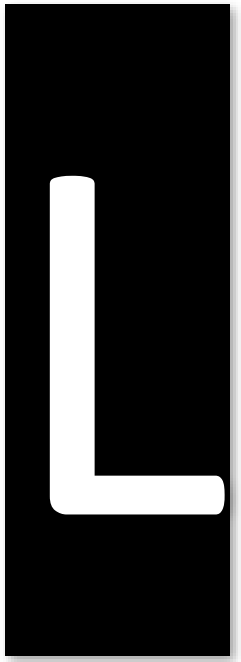
- ***Liskov Substitution Principle***

- Isso significa que a subclasse deve permanecer compatível com o comportamento da superclasse. ***Quando sobrescrevendo um método, estenda o comportamento base ao invés de substituí-lo com algo completamente diferente.***
- O princípio de substituição é um ***conjunto de checagens que ajudam a prever se um subclasse permanece compatível*** com o código que foi capaz de trabalhar com objetos da superclasse.
- Este conceito é crítico quando desenvolvendo bibliotecas e *frameworks*, porque suas classes serão usadas por outras pessoas cujo código você não pode diretamente acessar ou mudar.

L

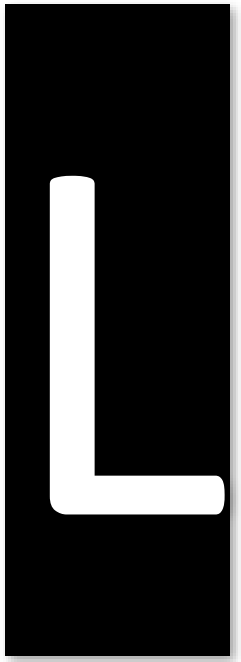
- ***Liskov Substitution Principle***

- Ao contrário de outros princípios de projeto que são amplamente abertos à interpretação, o princípio de substituição tem um conjunto de requerimentos formais para subclasses, e especificamente para seus métodos.
- Vamos ver esta lista em detalhes:



- ***Liskov Substitution Principle***

- **Os tipos de parâmetros em um método de uma subclasse devem coincidir ou serem mais abstratos que os tipos de parâmetros nos métodos da superclasse.**
- **Exemplo:**
- Digamos que temos uma classe com um método que deve alimentar gatos: **alimentar(Gato g)**. O código cliente sempre passa objetos gato nesse método.



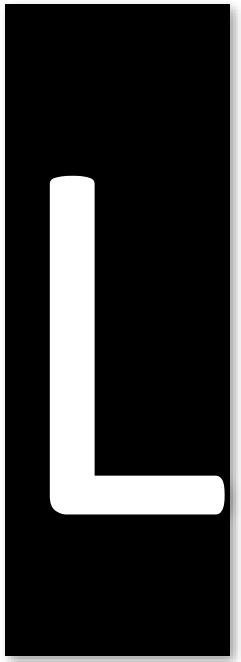
- ***Liskov Substitution Principle***

- **Bom**
- Digamos que você criou uma subclasse que sobrescreveu o método para que ele possa alimentar qualquer animal (uma superclasse dos gatos): **alimentar(Animal g).**
- Agora se você passar um objeto desta subclasse ao invés de um objeto da superclasse para o código cliente, tudo ainda vai funcionar bem. O método pode alimentar quaisquer animais, então ele ainda pode alimentar qualquer gato passado pelo cliente.

L

- ***Liskov Substitution Principle***

- **Ruim**
- Você criou outra subclasse e restringiu o método alimentar para aceitar apenas gatos de Bengala (uma subclasse dos gatos): **alimentar(GatoBengala c)**.
- O que vai acontecer com o código cliente se você ligá-lo com um objeto como este ao invés da classe original?
- Já que o método só pode alimentar uma raça de gatos, ele não vai servir para gatos genéricos mandados pelo cliente, quebrando toda a funcionalidade relacionada.



- ***Liskov Substitution Principle***

- **O tipo de retorno de um método de uma subclasse deve coincidir ou ser um subtipo do tipo de retorno no método da superclasse.**

- **Exemplo:**
- Digamos que você tem uma classe com um método:
- **comprarGato(): Gato**
- O código cliente espera receber qualquer gato como resultado da execução desse método.

L

- ***Liskov Substitution Principle***

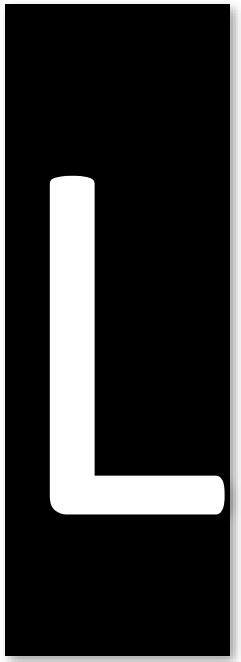
- **Bom**
- Uma subclasse sobrescreve o método dessa forma **comprarGato(): GatoBengala**. O cliente recebe um gato de Bengala, que ainda é um gato, então está tudo bem.

- **Ruim**
- Uma subclasse sobrescreve o método dessa forma: **comprarGato(): Animal**. Agora o código cliente quebra já que ele recebe um tipo desconhecido de animal genérico (um jacaré? um urso?) que não se adequa à estrutura desenhada para um gato.

L

- ***Liskov Substitution Principle***

- **Um método em uma subclasse não deve lançar tipos de exceções que não são esperados que o método base lançaria.**
- Em outras palavras, tipos de exceções devem coincidir ou serem subtipos daquelas que o método base já é capaz de lançar.

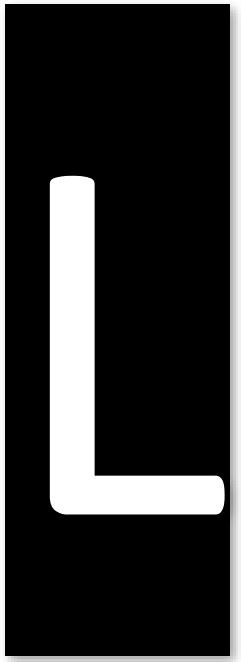


- ***Liskov Substitution Principle***

- Esta regra vem do fato que os blocos **try-catch** no código cliente têm como alvo tipos de exceções específicas que o método base provavelmente lançará.

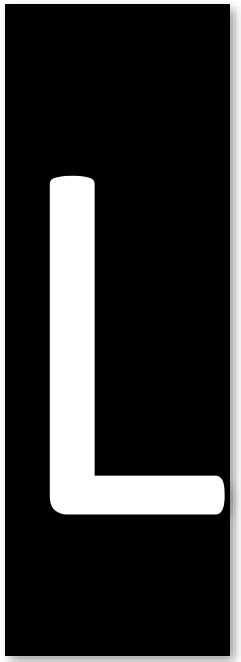
- Portanto, uma exceção inesperada pode escapar atravessar as linhas de defesa do código cliente e quebrar toda a aplicação.

- Na maioria das linguagens de programação modernas, especialmente as de tipo estático (Java, C#, e outras), estas regras eram construídas dentro da linguagem. Você não será capaz de compilar um programa que viola estas regras.



- ***Liskov Substitution Principle***

- **Uma subclasse não deve fortalecer pré-condições.**
- Por exemplo, **o método base tem um parâmetro com tipo int.** Se uma **subclasse sobrescreve este método** e precisa que **o valor** de um argumento passado para o método **deve ser positivo** (lançando uma exceção se o valor é negativo), isso torna as pré-condições mais fortes.
- O código cliente, que funcionava bem até agora quando números negativos eram passados no método, agora quebra se ele começa a trabalhar com um objeto desta subclasse.



- ***Liskov Substitution Principle***

- **Uma subclasse não deveria enfraquecer pós-condições.**

- Digamos que você tem uma classe com um método que trabalha com uma base de dados. Um método da classe deve sempre fechar todas as conexões abertas com a base de dados quando elas retornarem um valor.

- Você criou uma subclasse e mudou ela para que as conexões de base de dados continuem abertas para que você possa reutilizá-las.

- Mas o cliente pode não saber nada sobre suas intenções. Como ele espera que os métodos fechem todas as conexões, ele pode simplesmente terminar o programa logo após chamar o método, poluindo o sistema com conexões fantasmas com a base de dados.

L

- ***Liskov Substitution Principle***

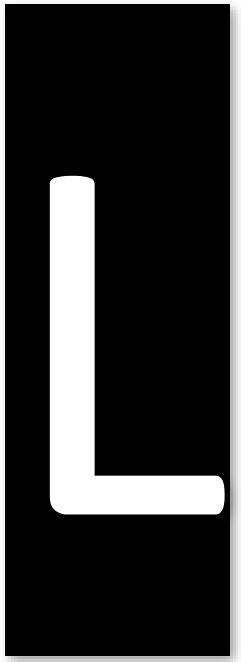
- **Invariantes de uma superclasse devem ser preservadas.**
- Esta é provavelmente a regra menos formal de todas.
- As invariantes são condições nas quais um objeto faz sentido.
- Por exemplo, invariantes de um gato são ter quatro patas, uma cauda, a habilidade de miar, etc.
- A parte confusa das invariantes é que, enquanto elas podem ser definidas explicitamente na forma de contratos de interface ou um conjunto de asserções com métodos, elas podem também serem implícitas por certos testes de unidade e expectativas do código cliente.

L

- ***Liskov Substitution Principle***

- A regra nas invariantes é a mais fácil de violar porque você pode não entender ou não perceber todas as invariantes de uma classe complexa.

- Portanto, a modo mais seguro de estender uma classe é introduzir novos campos e métodos, e não mexer com qualquer membro já existente da superclasse. É claro, isso nem sempre é viável no mundo real.

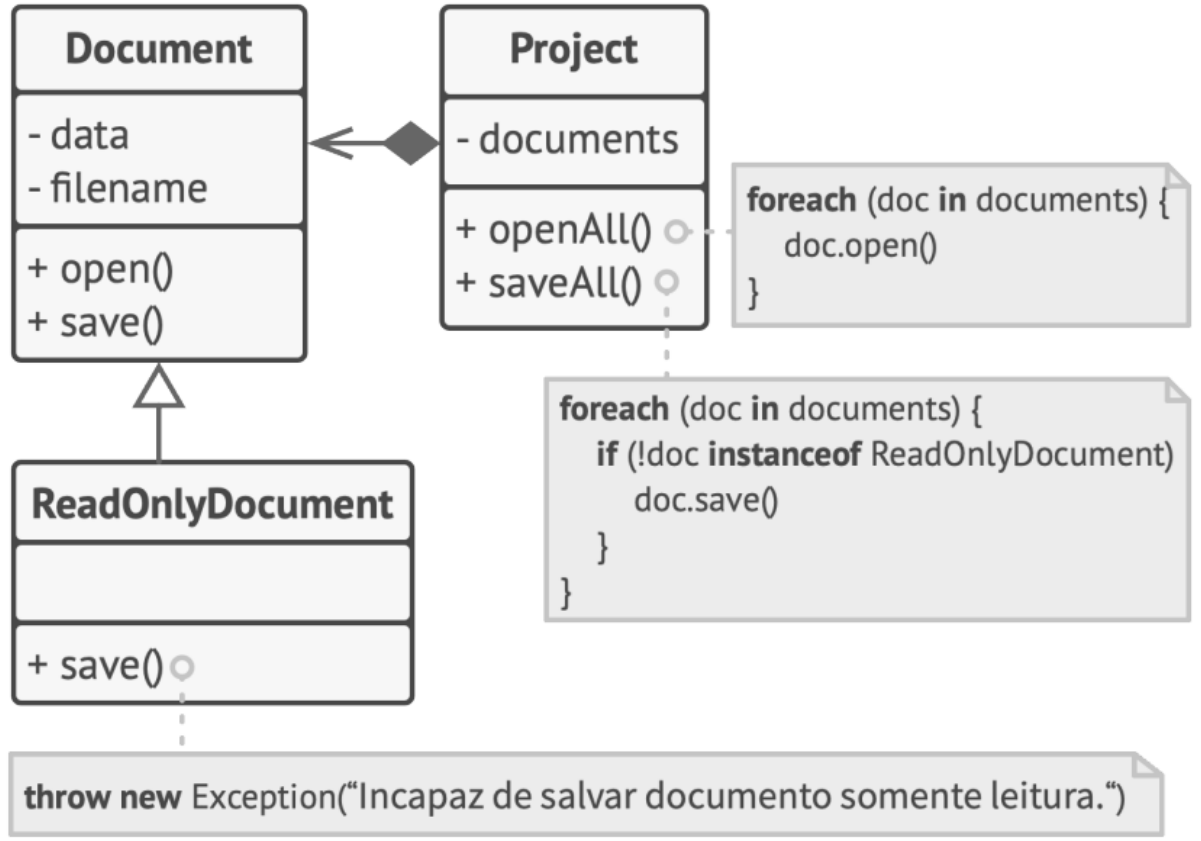


- ***Liskov Substitution Principle***

- **Uma subclasse não deve mudar valores de campos privados da superclasse.**
- *O que? E como isso é possível?*
- Acontece que algumas linguagens de programação permitem que você acesse membros privados de uma classe através de mecanismos reflexivos.
- Outras linguagens (Python, JavaScript) não têm quaisquer proteção para seus membros privados de forma alguma.

L

- ***Liskov Substitution Principle***
- O método salvar na subclasse DocSomenteLeitura lança uma exceção se alguém tenta chamá-lo.
- O método base não tem essa restrição.
- Isso significa que o código cliente pode quebrar se nós não checarmos o tipo de documento antes de salvá-lo.
- O código resultante também viola o princípio aberto/fechado, já que o código cliente se torna dependente das classes concretas de documentos.



ANTES: salvar não faz sentido em um documento somente leitura, então a subclasse tenta resolver isso ao resetar o comportamento base no método sobrescrito.

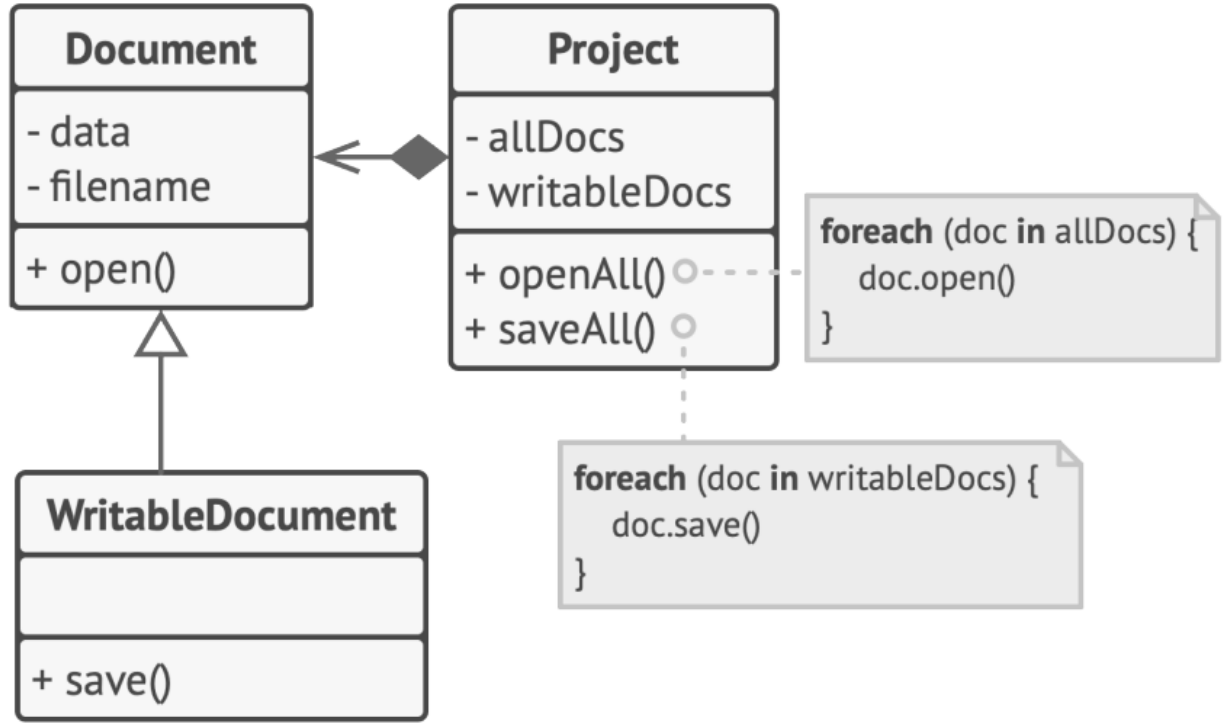
L

- ***Liskov Substitution Principle***

- Pode-se resolver o problema redesenhando a hierarquia de classe:

- Uma subclasse deve estender o comportamento de uma superclasse, portanto, o documento somente leitura se torna a classe base da hierarquia.

- O documento gravável é agora uma subclasse que estende a classe base e adiciona o comportamento de salvar.



DEPOIS: o problema foi resolvido após tornar a classe documento somente leitura como a classe base da hierarquia.

I

- ***Interface Segregation Principle***
- ***Princípio de segregação de interface***

- **Clientes não devem ser forçados a depender de métodos que não usam.**

- Tente fazer com que suas interfaces sejam reduzidas o suficiente para que as classes cliente não tenham que implementar comportamentos que não precisam.
- De acordo com o princípio de segregação de interface, você deve quebrar interfaces “gordas” em tipos mais granulares e específicas.
- Os clientes devem implementar somente aqueles métodos que realmente precisam. Do contrário, uma mudança em uma interface “gorda” irá quebrar clientes que nem sequer usam os métodos modificados.

- ***Interface Segregation Principle***

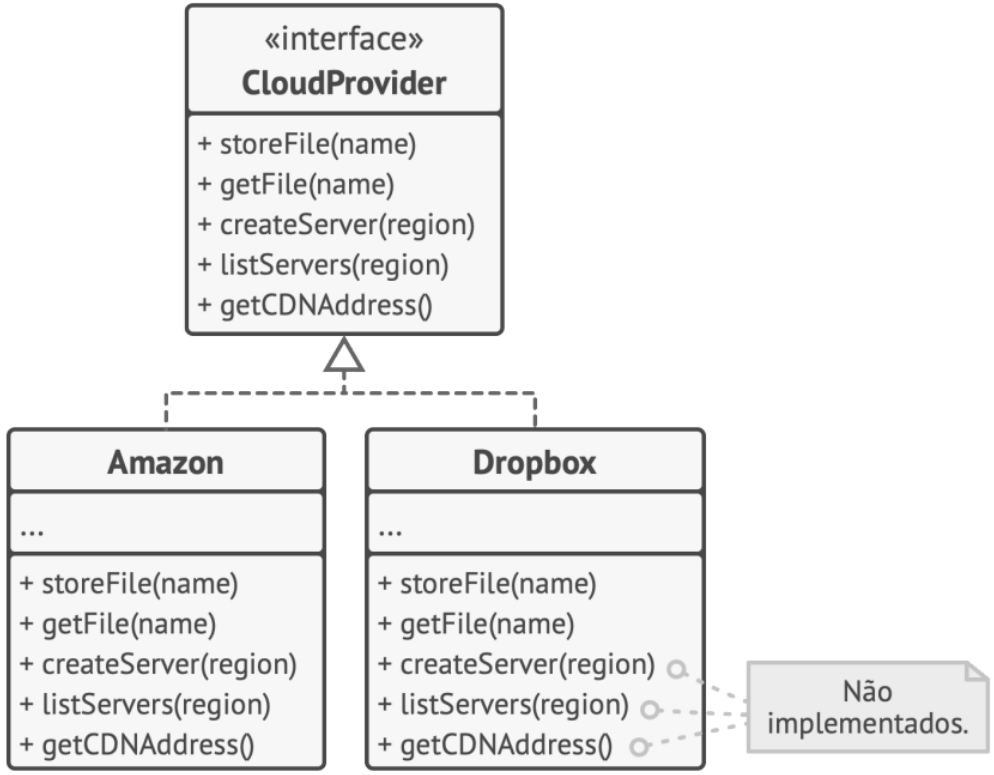
- A herança de classe permite que uma classe tenha apenas uma superclasse, mas ela não limita o número de interfaces que aquela classe pode implementar ao mesmo tempo.

- Portanto não há necessidade de empilhar toneladas de métodos sem relação nenhuma em uma única interface.

- Quebre-as em algumas interfaces refinadas: pode-se implementar todas em uma única classe se necessário. Contudo, algumas classes podem ficar bem implementando apenas uma delas.

- ***Interface Segregation Principle***

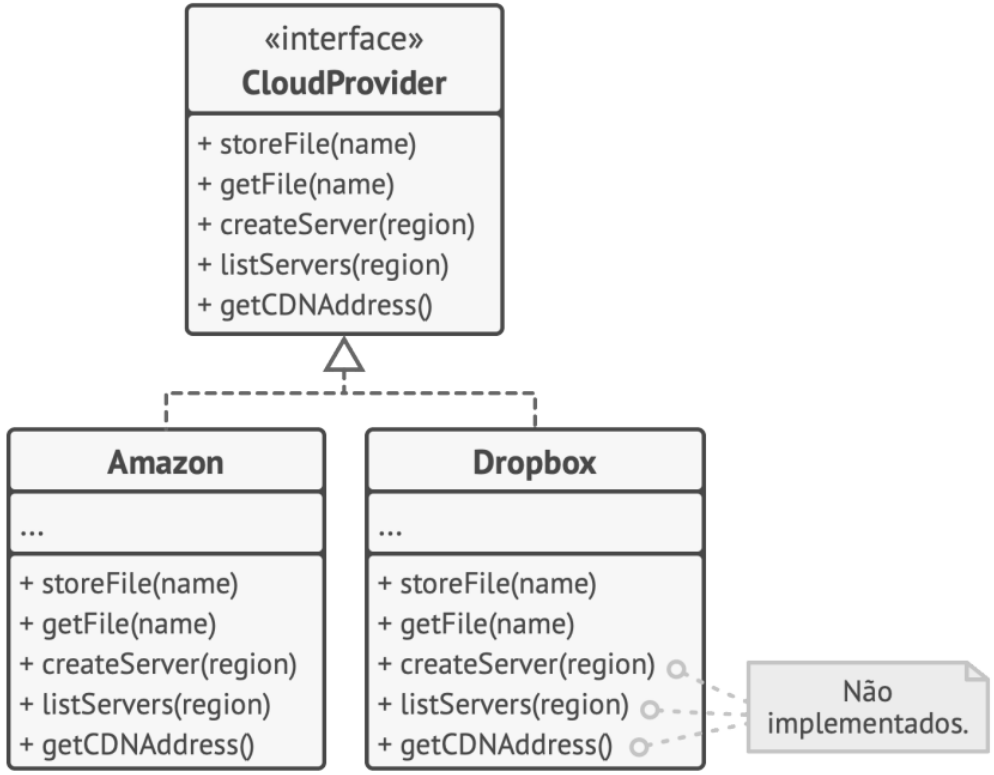
- Exemplo:
- Imagine que você criou uma biblioteca que torna mais fácil integrar aplicações com vários fornecedores de computação da nuvem.
- Embora a versão inicial suportava apenas a Amazon Cloud, ela continha o conjunto completo de serviços e funcionalidades de nuvem.



ANTES: nem todos os clientes podem satisfazer os requerimentos de uma interface inchada.

- ***Interface Segregation Principle***

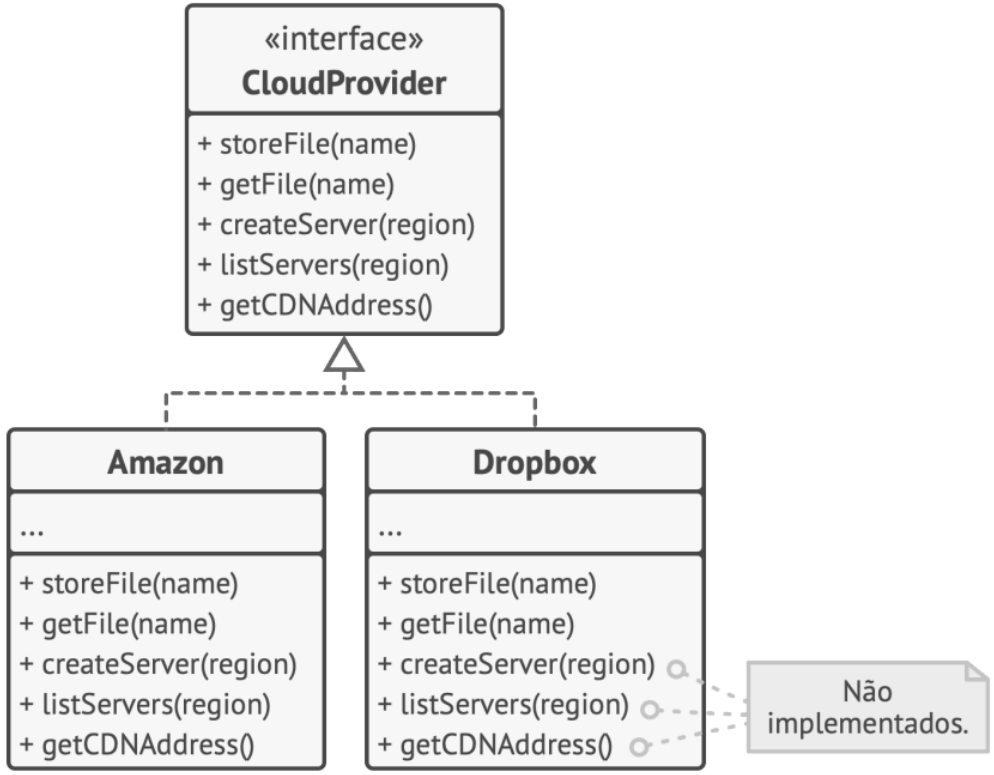
- Naquele momento você assumiu que todos os fornecedores de nuvem teriam a mesma gama de funcionalidade que a Amazon.
- Mas quando chegou a hora de implementar o suporte a outro fornecedor, aconteceu que a maioria das interfaces da biblioteca ficaram muito largas.
- Alguns métodos descreviam funcionalidades que outros servidores de nuvens não tinham.



ANTES: nem todos os clientes podem satisfazer os requerimentos de uma interface inchada.

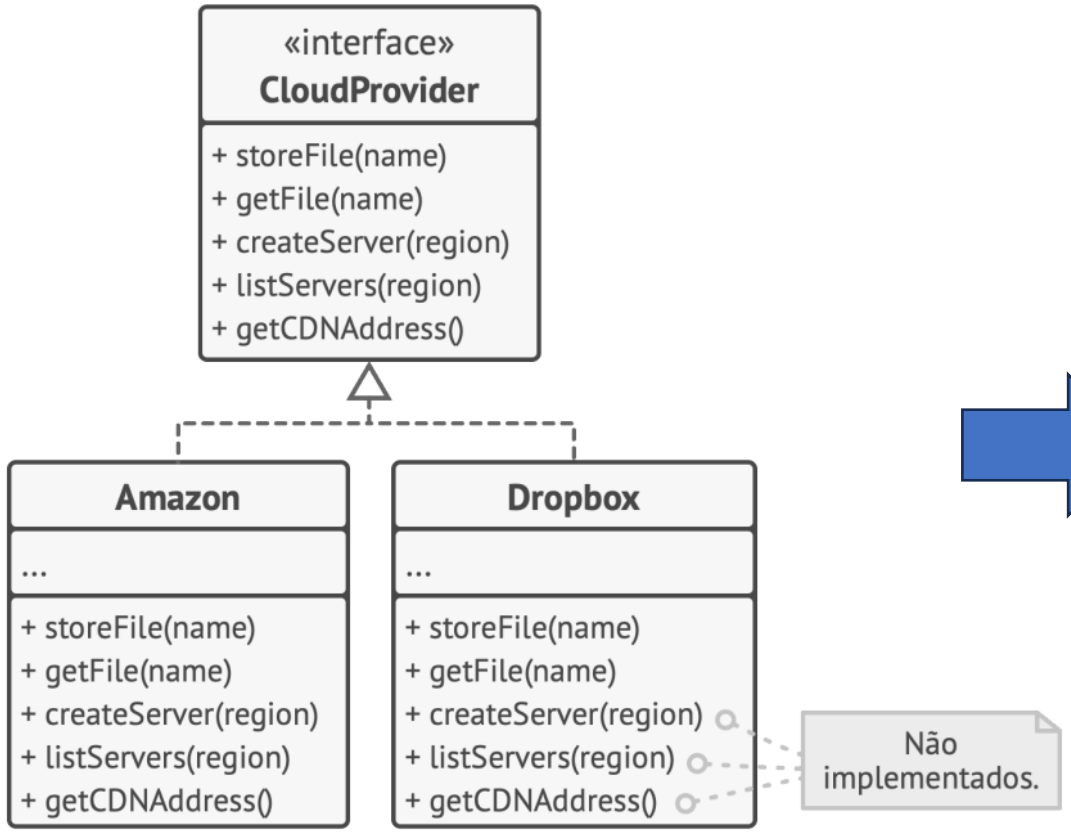
- ***Interface Segregation Principle***

- Embora você ainda possa implementar esses métodos e colocar alguns tocos ali, não seria uma solução bonita.
- A melhor abordagem é quebrar a interface em partes.
- Classes que são capazes de implementar o que a interface original podem agora implementar apenas diversas interfaces refinadas.
- Outras classes podem implementar somente àquelas interfaces as quais têm métodos que façam sentidos para elas.

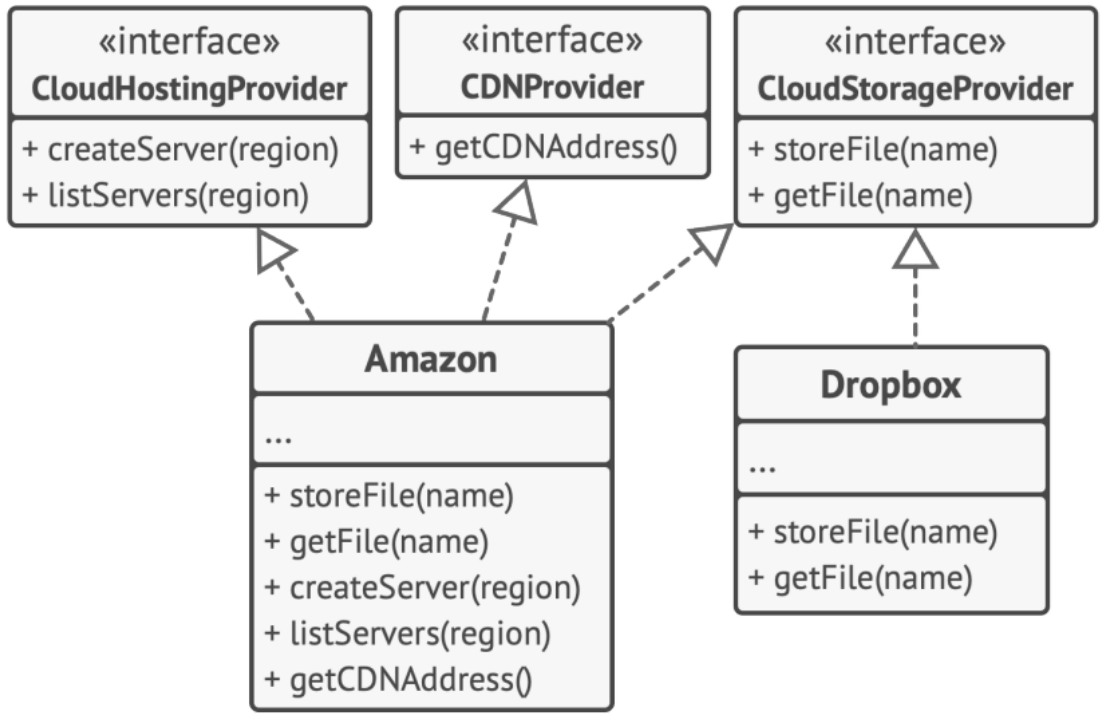


ANTES: nem todos os clientes podem satisfazer os requerimentos de uma interface inchada.

- ***Interface Segregation Principle***



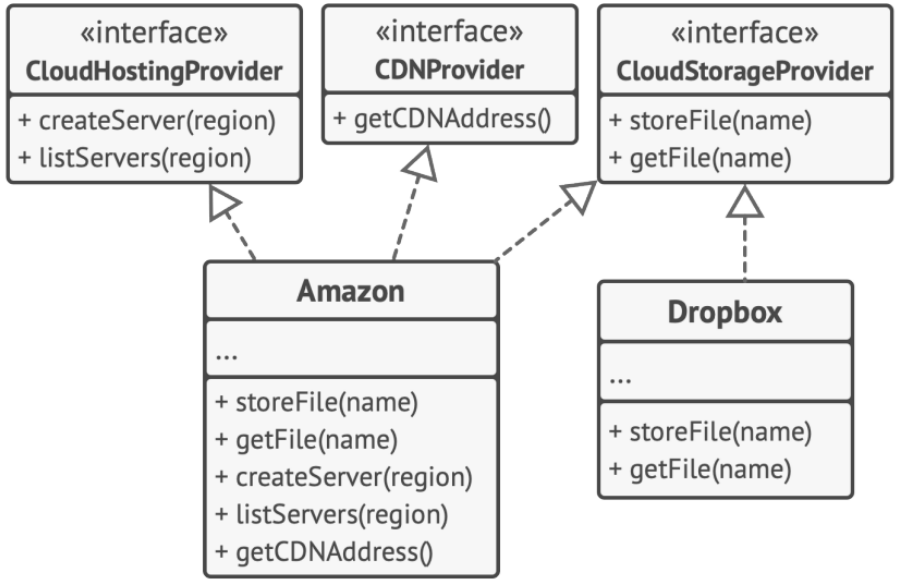
ANTES: nem todos os clientes podem satisfazer os requerimentos de uma interface inchada.



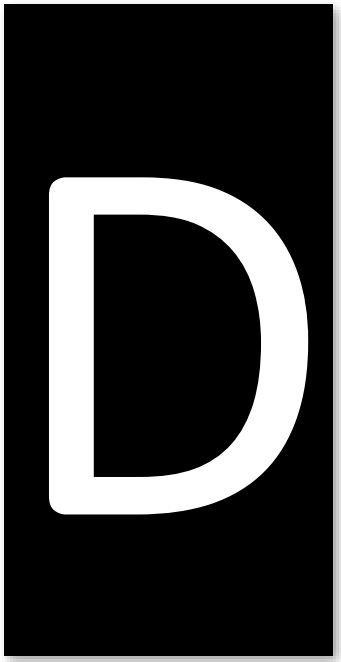
DEPOIS: uma interface inchada foi quebrada em um conjunto de interfaces mais granulares.

- ***Interface Segregation Principle***

- Como com os outros princípios, evite exagerar com este aqui.
- Não divida mais uma interface que já está bastante específica.
- Lembre-se que, quanto mais interfaces você cria, mais complexo seu código se torna. Mantenha o equilíbrio.

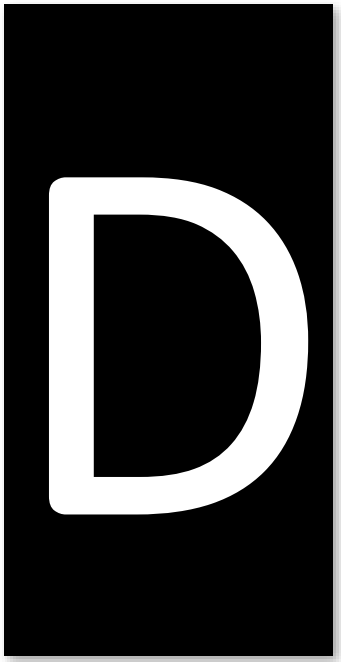


DEPOIS: uma interface inchada foi quebrada em um conjunto de interfaces mais granulares.



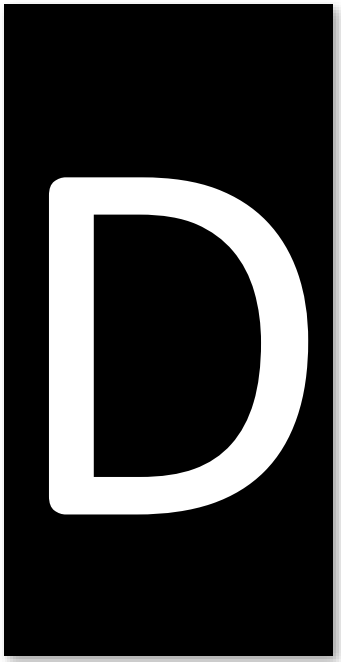
- ***Dependence Inversion Principle***
- ***Princípio da Inversão de Dependência***

- **Classes de alto nível não deveriam depender de classes de baixo nível. Ambas devem depender de abstrações. As abstrações não devem depender de detalhes. Detalhes devem depender de abstrações.**
- **Classes de baixo nível** implementam operações básicas tais como trabalhar com um disco, transferindo dados pela rede, conectar-se a uma base de dados, etc.
- **Classes de alto nível** contém lógica de negócio complexa que direcionam classes de baixo nível para fazerem algo.



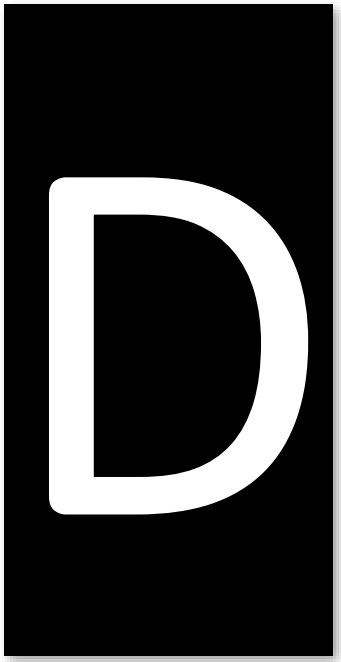
- ***Dependence Inversion Principle***

- Algumas pessoas fazem o projeto de classes de baixo nível primeiro e só depois trabalham nas de alto nível.
- Isso é muito comum quando você começa a desenvolver um protótipo de um novo sistema, e você não tem certeza do que será possível em alto nível porque as coisas de baixo nível não foram implementadas ainda ou não são claras.
- Com tal abordagem, classes da lógica de negócio tendem a ficar dependentes de classes primitivas de baixo nível.



- ***Dependence Inversion Principle***

- O princípio de inversão de dependência sugere trocar a direção desta dependência.
- Para começar, você precisa descrever as interfaces para as operações de baixo nível que as classes de alto nível dependem, preferivelmente em termos de negócio.
- Por exemplo, a lógica do negócio deve chamar um método abrirRelatório(arquivo) ao invés de uma série de métodos abrirArquivo(x), lerBytes(n), fecharArquivo(x).
- Estas interfaces contam como de alto nível.



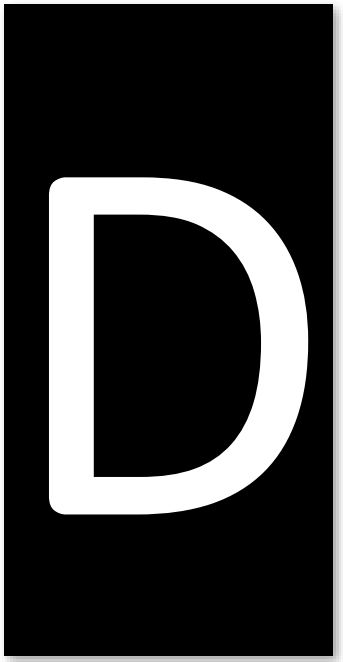
- ***Dependence Inversion Principle***

- O princípio de inversão de dependência sugere trocar a direção desta dependência.

1)Para começar, você precisa descrever as interfaces para as operações de baixo nível que as classes de alto nível dependem, preferivelmente em termos de negócio.

    - Por exemplo, a lógica do negócio deve chamar um método abrirRelatório(arquivo) ao invés de uma série de métodos abrirArquivo(x), lerBytes(n), fecharArquivo(x).

    - Estas interfaces contam como de alto nível.



- ***Dependence Inversion Principle***

- 2) Agora você pode fazer classes de alto nível dependentes daquelas interfaces, ao invés de classes concretas de baixo nível. Essa dependência será muito mais suave que a original.

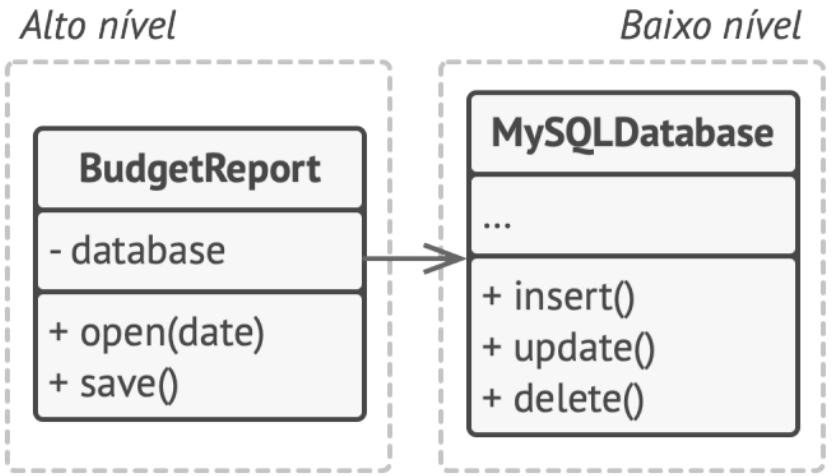
- 3) Uma vez que as classes de baixo nível implementam essas interfaces, elas se tornam dependentes do nível da lógica do negócio, revertendo a direção da dependência original.

- O princípio de inversão de dependência quase sempre anda junto com o princípio aberto/fechado: você pode estender classes de baixo nível para usar diferentes classes de lógica do negócio sem quebrar classes existentes.

D

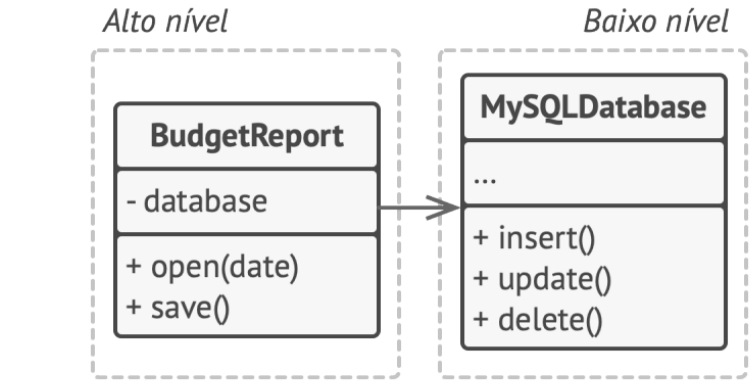
- ***Dependence Inversion Principle***

- Exemplo:
- a classe de alto nível de relatório de orçamento usa uma classe de baixo nível de base de dados para ler e manter seus dados.
- Isso significa que quaisquer mudanças na classe de baixo nível, tais como quando uma nova versão da base de dados for lançada, podem afetar a classe de alto nível, que não deveria se importar com os detalhes do armazenamento de dados.

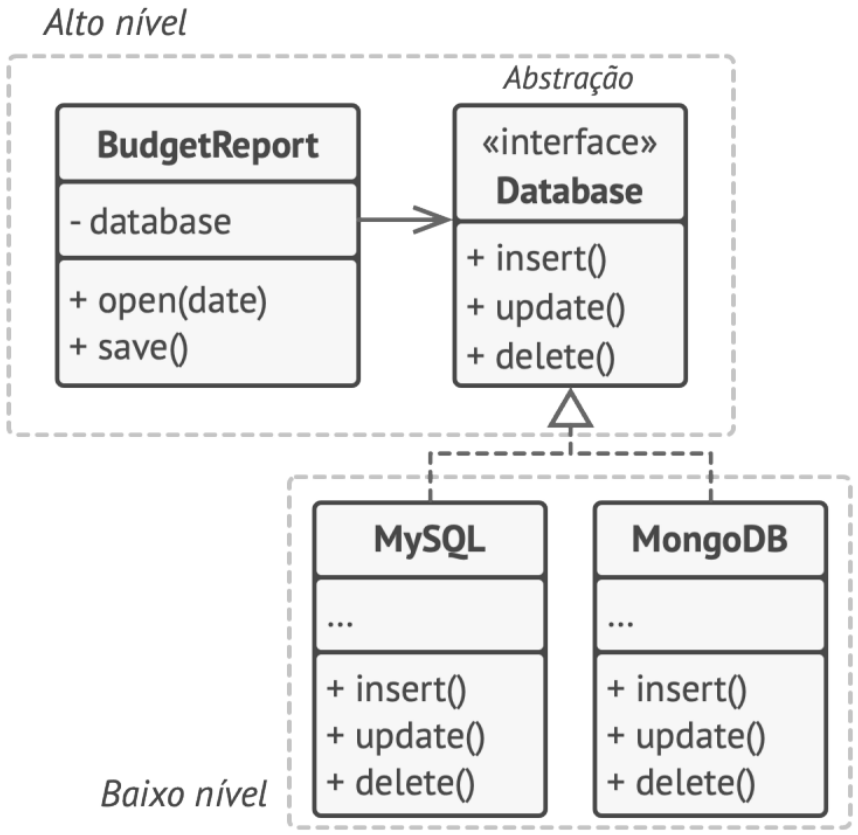


ANTES: uma classe de alto nível depende de uma classe de baixo nível.

D



ANTES: uma classe de alto nível depende de uma classe de baixo nível.

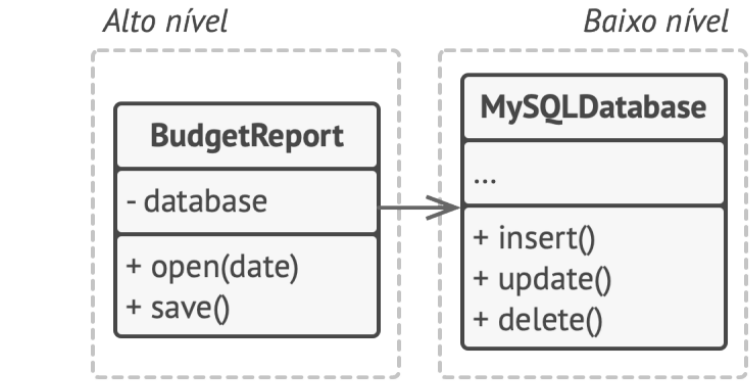


DEPOIS: classes de baixo nível dependem de uma abstração de alto nível.

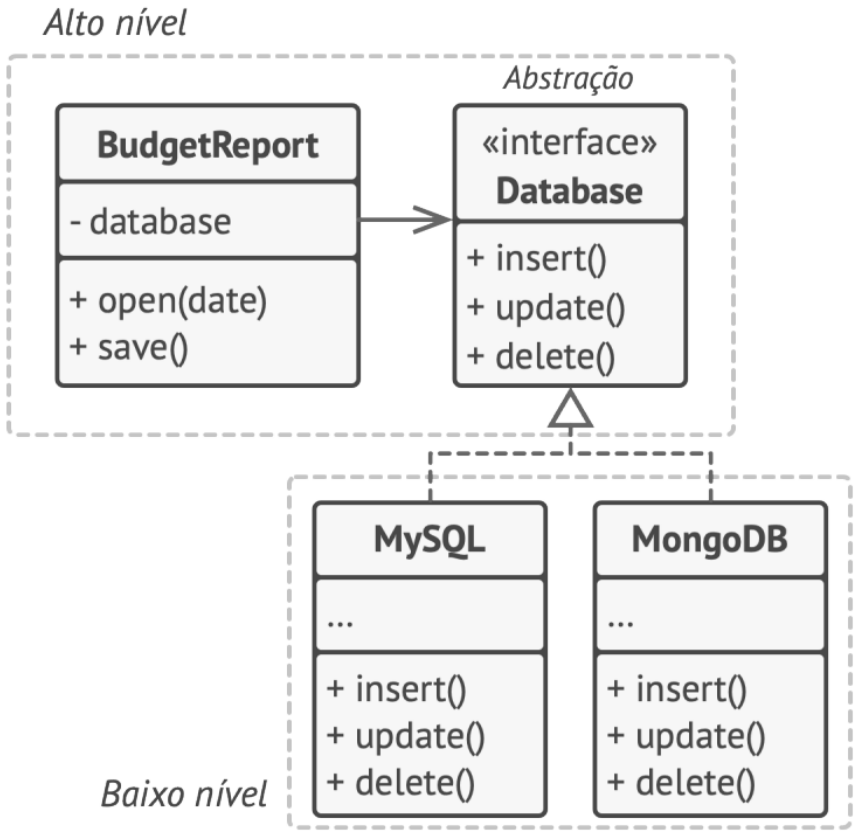
- ***Dependence Inversion Principle***

- Você pode corrigir esse problema **criando uma interface de alto nível** que descreve as operações de ler/escrever;
- e **fazer a classe de relatório usar aquela interface** ao invés da classe de baixo nível;
- então você pode mudar ou estender a classe de baixo nível original para implementar a nova interface de ler/escrever declarada pela lógica do negócio.

D



ANTES: uma classe de alto nível depende de uma classe de baixo nível.



DEPOIS: classes de baixo nível dependem de uma abstração de alto nível.

- ***Dependence Inversion Principle***

- Você pode corrigir esse problema **criando uma interface de alto nível** que descreve as operações de ler/escrever;
- e **fazer a classe de relatório usar aquela interface** ao invés da classe de baixo nível;
- então você pode mudar ou estender a classe de baixo nível original para implementar a nova interface de ler/escrever declarada pela lógica do negócio.
- Como resultado, a direção da dependência original foi invertida: classes de baixo nível agora dependem das abstrações de alto nível.