Princípios SOLID - Correção Exercícios

Engenharia de Software Tecnologia

PUC-SP

Prof. Carlos Eduardo de Barros Paes
Departamento de Computação
Pontifícia Universidade Católica de São Paulo
E-mail: carlosp@pucsp.br

Exercício 01

- O princípio da responsabilidade única está sendo “quebrado” neste exemplo.
- Este princípio diz que uma classe deve possuir apenas uma responsabilidade, sendo uma classe coesa e simples.
- A classe Conta está com três responsabilidades, fazer a autenticação do usuário, cuidar do saldo e imprimir extrato.
- A primeira vista pode parecer correto, afinal a responsabilidade da classe é a conta de usuário. Mas se por algum motivo você precisar mudar a lógica de autenticação, você pode sem querer afetar a funcionalidade de transferência entre contas por exemplo.

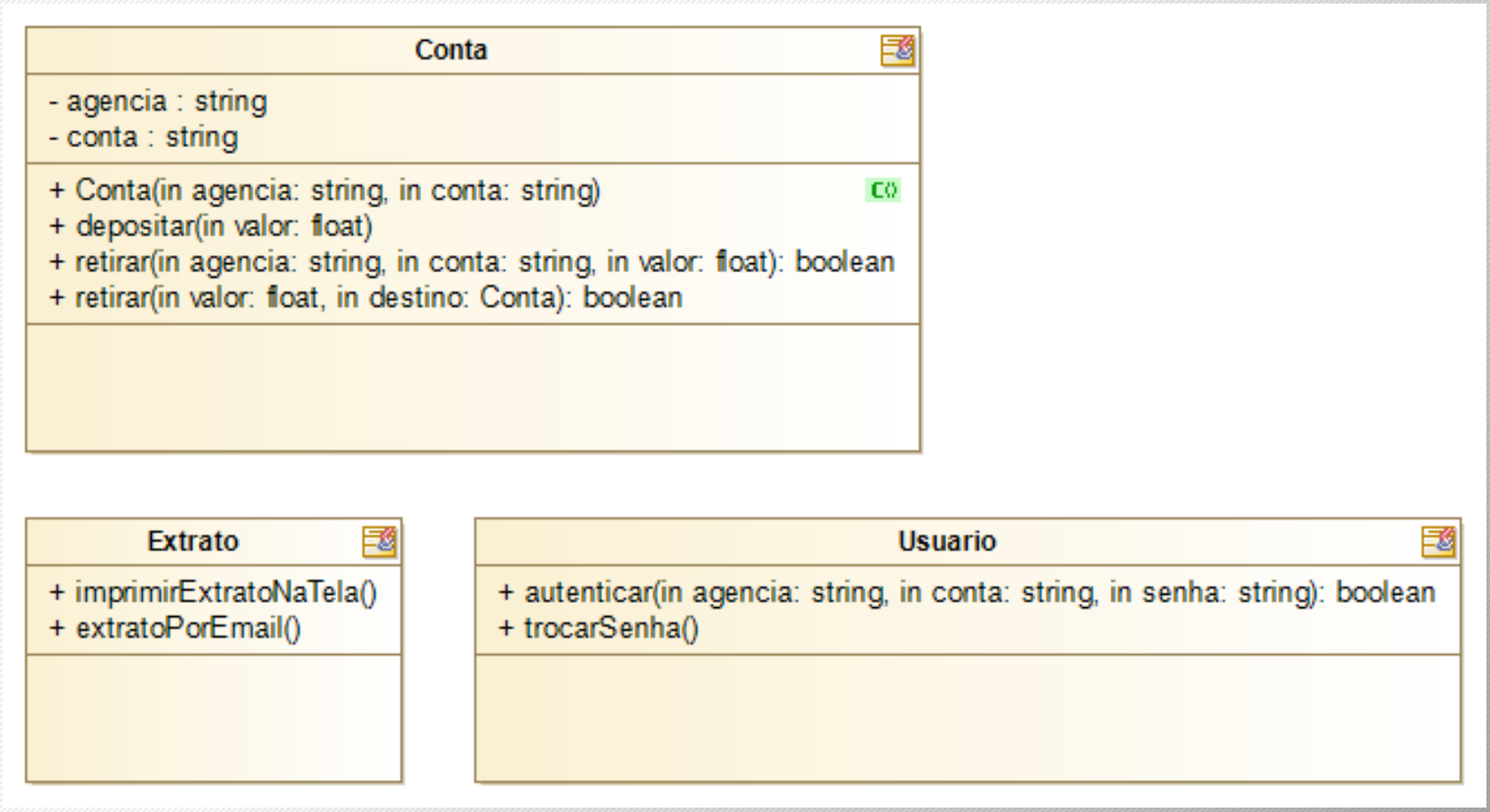
Exercício 01

- A classe apresentada não deve ter a responsabilidade de fazer *login* (autenticação) e imprimir extrato! Não é responsabilidade dela fazer isso.
- A seguinte refatoração de código poderia ser realizar para deixar a classe mais coesa e com responsabilidade única.

Exercício 01

- Código parcial Java Refatorado

Exercício 01 – Diagrama de Classes UML



Exercício 02

- Neste exemplo está sendo “quebrado” o princípio de Open/Close.
- Este princípio estabelece que uma classe deve ser aberta para extensão e fechada para modificação.
- A classe deve ser projetada de modo que não haja necessidade de fazer alterações no código cada vez que um novo comportamento for necessário. Como fazer isso?
  - Separando todo o comportamento mutável da classe e manter nela somente aquilo que nunca mudará.

Exercício 02

- Para cada novo comportamento que adicionamos, a classe fica mais complexa
- Poderíamos criar métodos separados getSalarioVendedor(), getSalarioSupervisor(), getSalarioSecretaria, getSalarioGerente().
- Mais isso só iria tornar a classe mais complexa do ponto de vista de quem a utiliza.
- Então o que faremos agora? Bom, basta separar o que muda do que não muda. O que nunca muda? A forma como calculamos o salário é sempre salário base mais comissão. E o que muda? A forma de calcular a comissão!

Exercício 02

- Vamos então refatorar a classe. Primeiro vamos fixar nossa regra de salários.

```
class Funcionario{

    public float getSalario(float comissao){
        return salarioBase + comissao;
    }
}
```

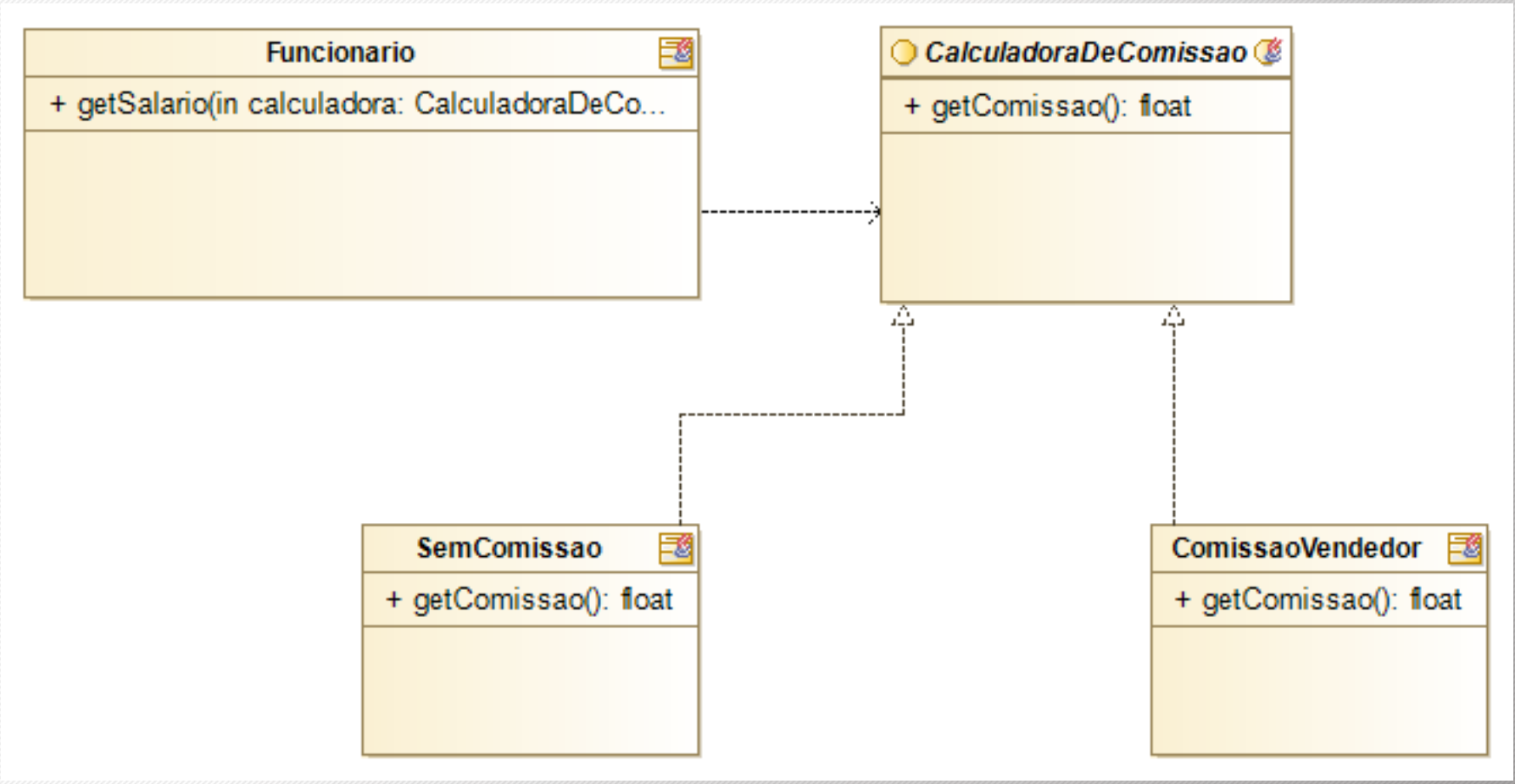
Exercício 02

- Veja como o método ficou extremamente mais simples,
- Precisamos ainda resolver como calcular a comissão.
- Vamos usar o polimorfismo!

Exercício 02

- Código Java Refatorado

Exercício 02 – Diagrama de Classes UML



Exercício 02 – Diagrama de Classes UML

- Pronto! Com isso cada vez que precisamos estender o comportamento da classe, basta criar uma nova classe que contenha esse comportamento e implementar a interface CalculadoraDeComissao.
- De quebra implementamos o padrão de projeto ***Strategy***

Exercício 03

- O Princípio da Substituição de Liskov está sendo violado neste exemplo
- Proposto por Barbara Liskov em 1988
- Princípio: Programe para a interface e não para sua implementação!

Exercício 03

- “Dado um Tipo T, todos os seus subtipos S podem ser usados como seus substitutos sem que haja impactos no sistema.”
- Caso possua uma classe qualquer, pode-se substituir suas chamadas por chamadas de quaisquer classes que herdem dela.
- Mas qual a utilidade disso? Isso evita que haja muita desordem nas relações de herança em todo o sistema.

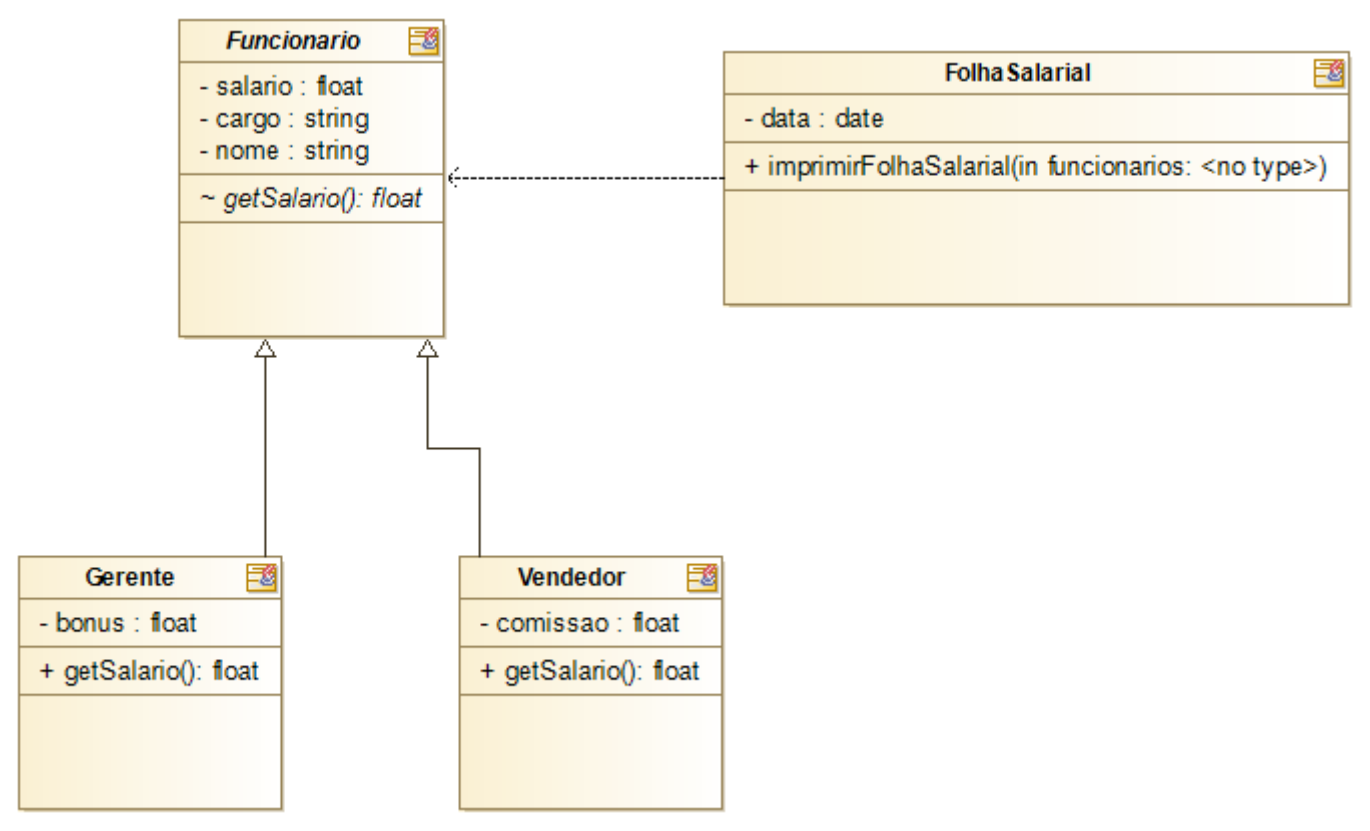
Exercício 03

- Exemplo do exercício 03
- Violação do Princípio de Liskov
  - Não podemos substituir Funcionário por um subtipo Gerente ou Vendedor, pois cada subtipo possui um método diferente para calcular o salário.
- Violação do Princípio do Aberto/Fechado
  - Cada vez que criarmos um novo cargo, precisamos vir no módulo de folha de pagamento e alterá-lo.

Exercício 03

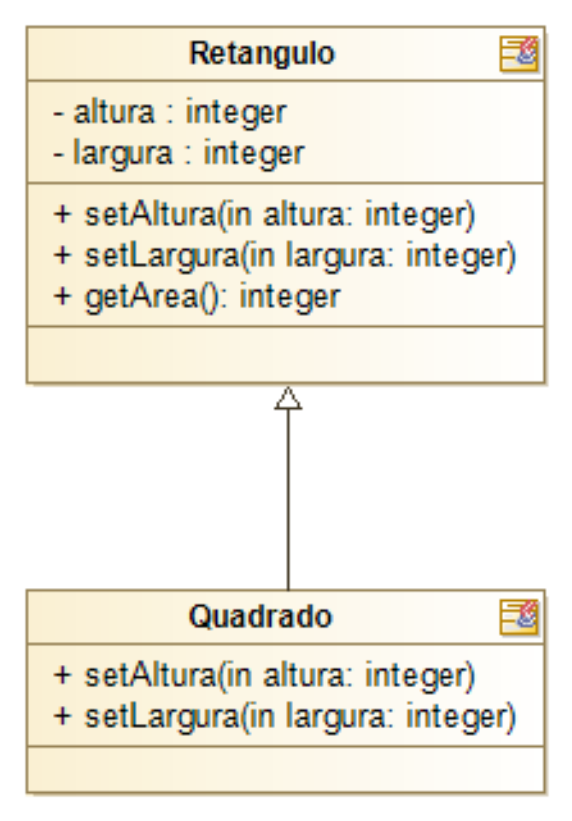
- Para resolver o problema, basta que todos os subtipos tenha o mesmo método para calcular salário. Para isso criamos um método abstrato getSalario(), e todas os subtipos terão que implementá-los
- Código Java Refatorado
- Agora podemos criar quantos subtipos de funcionários desejarmos e nunca precisaremos nos preocupar com o módulo de Folha Salarial, pois ele conhece a abstração de funcionário e sabe que topos os subtipos possuem o método getSalario(), independente de como cada um irá implementar.

Exercício 03 – Diagrama de Classes UML



Exercício 03 – Outro Exemplo

- Considere o exemplo do quadrado: Retângulo, que é o supertipo e o quadrado que é o subtipo
- Diagrama de Classes UML



Exercício 03 – Outro Exemplo

- Código Java
- Como sabemos, o quadrado é um retângulo que possui altura e largura iguais, sendo assim, quando alteramos a altura, precisamos alterar a largura também, e vice-versa.
- Usando as classes:

```
Retangulo q1 = new Quadrado();
q1.setAltura(5);
q1.setLargura(10);

if(q1.getArea() == 50){
    //comportamento esperado
} else {
    //comportamento inesperado
}
```

Exercício 03 – Outro Exemplo

- Qual o resultado?
- Conclusão?

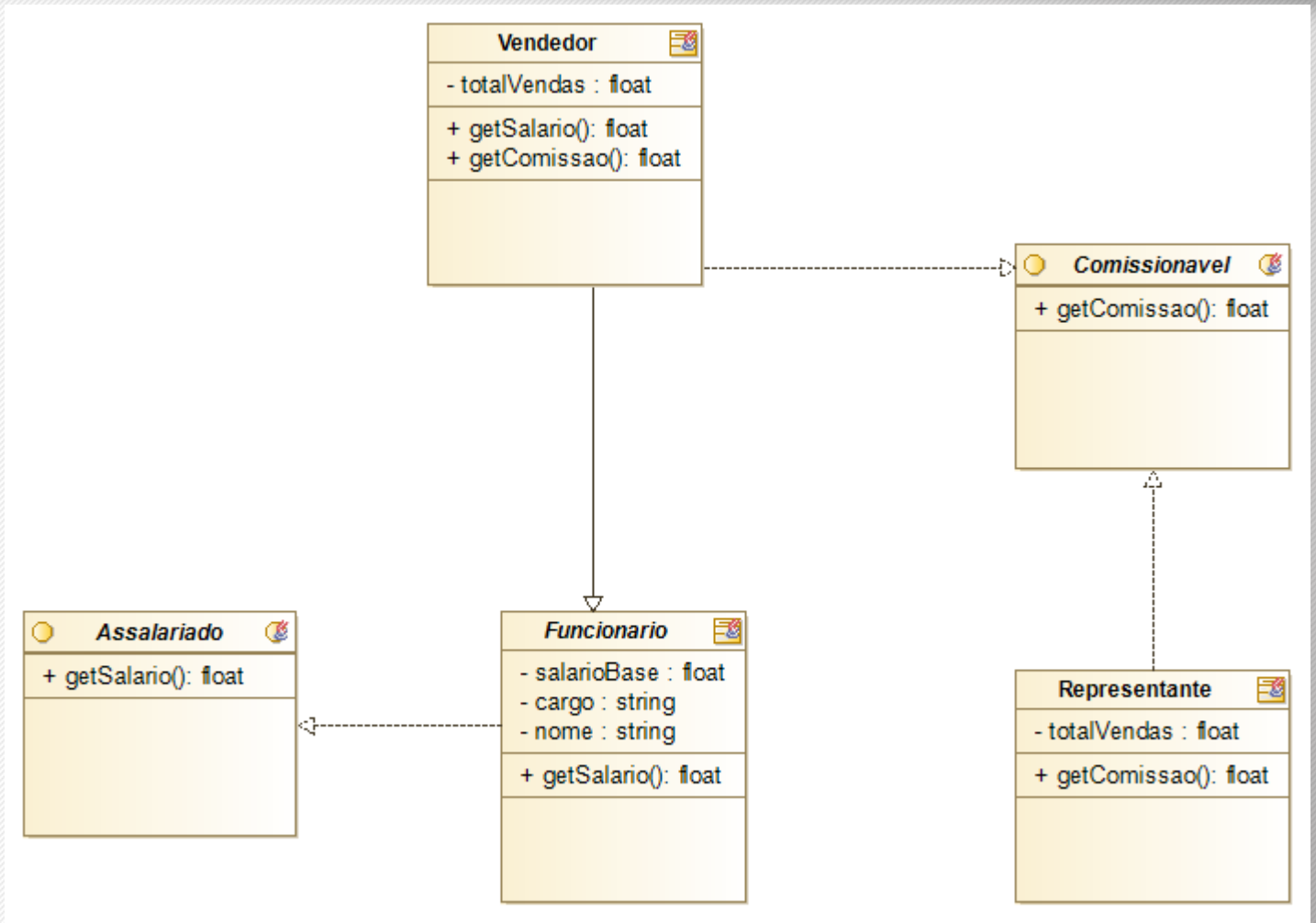
Exercício 04

- Princípio da Segregação de Interfaces é violado
- Este princípio nos diz que uma classe consumidora não deve conhecer (depender) métodos que não necessitam.
- Para ter uma classe coesa e reutilizável, devemos atribuir a ela uma única responsabilidade. Mas as vezes, mesmo essa única responsabilidade pode ser quebrada em responsabilidades menores ainda, tornando sua interface mais amigável

Exercício 04

- Código Java Refatorado
  - Aplicando o método de refatoração **Extrair Interface**, para extrair as interfaces *Assalariado* e *Comissionável*.

Exercício 04 – Diagrama de Classes UML



Exercício 04

- Código Java Refatorado
  - Aplicando o método de refatoração **Extrair Interface**, para extrair as interfaces *Assalariado* e *Comissionável*.
- Extração das interfaces da Classe Funcionário
  - Deixou de ser abstrata e não possui mais dos métodos abstratos getSalario() e getComissao().
  - Mas como funcionário ainda recebe um salário, então a classe Funcionário passou a ter a Interface Assalariado, obrigando-a a implementar o método getSalario(). Assim a classe Atendente de Caixa perdeu o sentido de existir e passou ser representada como um Funcionário.

Exercício 04

- Classe Vendedor (que é um funcionário) estende a classe Funcionário que é um Assalariado.
  - Mas como Vendedor possui sua própria maneira de calcular o salário, então sobrescrevemos o método getSalario(), para que contemple essa sua lógica.
  - Vendedor possui uma comissão e deve implementar a interface Comissionavel, para poder calcular sua comissão.
- Classe Representante (Comercial) que não é mais tratada como um funcionário
  - Deve implementar apenas a interface Comissionavel, assim podemos ter o seu calculo de comissão, não mais precisamos mais ter conhecimento sobre salário.

Exercício 04

- O princípio da Segregação de Interfaces ajuda a aumentar a granularidade dos objetos, aumentando a coesão de suas interfaces e diminuindo drasticamente o acoplamento.
- Consequentemente melhora a manutenção do código, pois interfaces mais simples são mais fácies de serem entendidas e implementadas

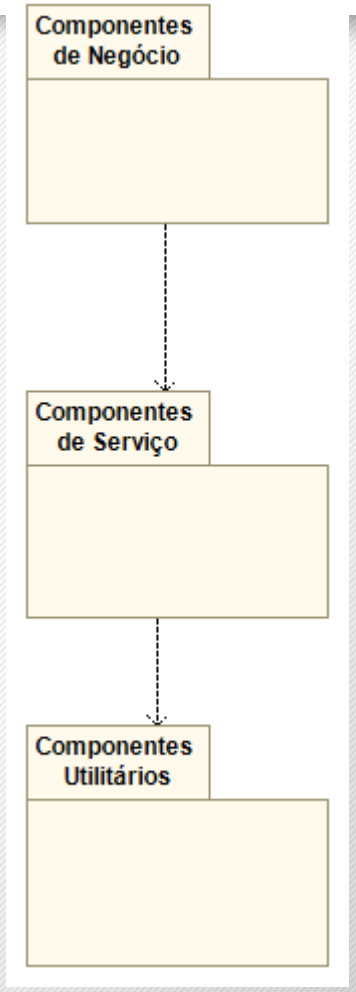
Exercício 05

- Está sendo violado neste exercício o princípio da Inversão de Dependência
- Este princípio trata de uma maneira específica para desacoplar as dependências entre os objetos, modificando a maneira tradicional como estabelecemos as dependências entre nossos objetos
- Padrão apresentado por Martin Fowler
- https://martinfowler.com/articles/injection.html

Exercício 05

- Componentes de mais alto nível não devem depender de componentes de níveis mais baixos, mas ambos devem depender de abstrações.
- Abstrações não devem depender de implementações, mas as implementações devem depender de abstrações.

Exercício 05

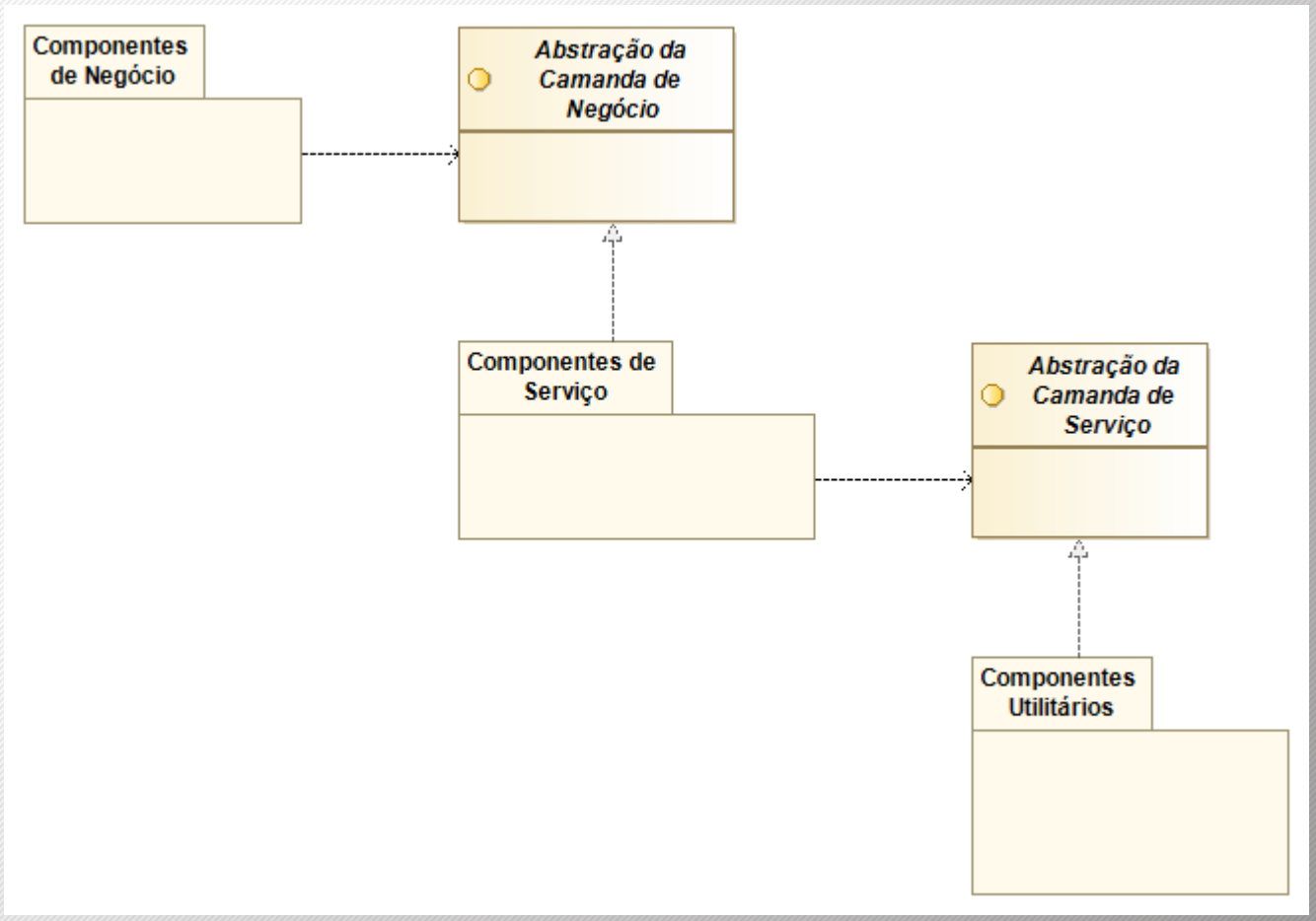


Exercício 05

- Componentes de nível inferior são projetados para serem consumidos por módulos de nível superior, de modo que estes possam ir incrementando de complexidade conforme o sistema vai sendo construído.
- Componentes da camada superior dependem dos componentes mais baixos apenas para que possam realizar alguma tarefa, isto reduz as possibilidades de reuso dos componentes de alto nível, devido ao excessivo acoplamento entre os componentes.
- O objetivo deste princípio é a redução do acoplamento entre os componentes por meio de uma camada de abstração

Exercício 05

- Inversão de Dependência



Exercício 05

- As abstrações pertencem as camadas superiores, sendo assim as camadas inferiores herdam/implementam as interfaces dessas abstrações.
- A inversão de dependências encoraja a reutilização dos componentes de camadas superior, que contêm as maior parte da lógica de verdade. As camadas superiores podem conter diferentes implementações das camadas inferiores, enquanto as camadas inferiores podem ser fechadas ou extensíveis, conforme a sua necessidade.

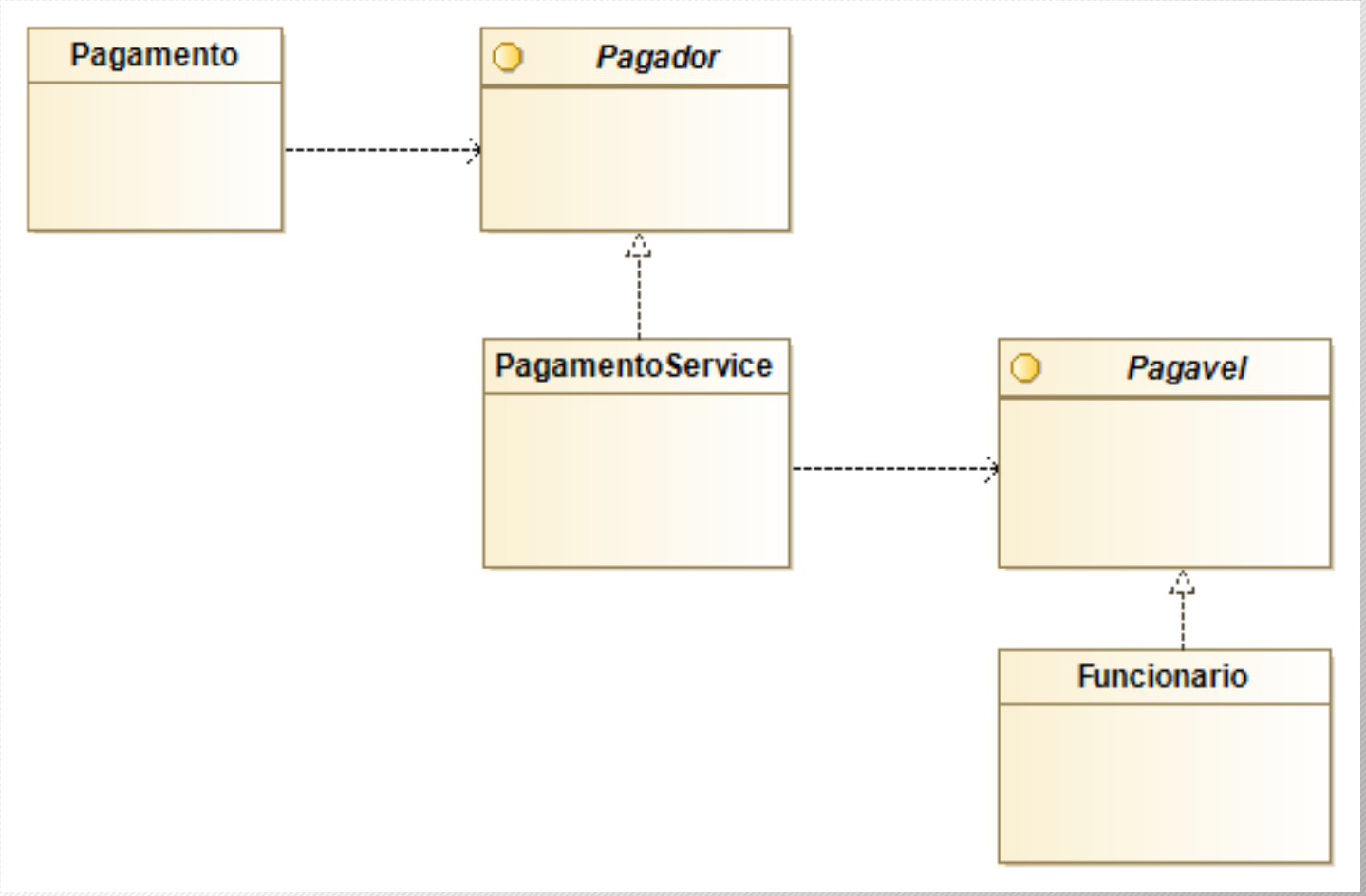
Exercício 05

- Construindo uma camada de abstração
  - todas as variáveis membro da classe devem ser interfaces ou classes abstratas
  - todos os pacotes contendo classes concretas devem se comunicar somente através de interfaces ou classes abstratas
  - Nenhuma classe deve derivar de uma outra classe concreta
  - Nenhum método deve sobrescrever um método já implementado
  - Todas as instâncias de objetos devem ser criadas através de Padrões de Projetos de criação como Factory ou Injeção de Dependências.

Exercício 05

- Exemplo apresentado no exercício
- Classe Pagamento conhece os detalhes da implementação de Funcionário, ou seja, Pagamento depende da implementação de Funcionário e não da sua abstração, o que deixa o código fortemente acoplado.

Exercício 05



Exercício 05

- Código Java Refatorado
- Foram criadas duas interfaces: Pagável que é a abstração para o Funcionário e Pagador que é uma abstração para Pagamento.
- Por fim foi criada uma camada intermediária chamada serviço responsável pelo gerenciamento dos Pagadores e os Pagáveis.

Exercício 05

- Inversão de dependências é um princípio muito utilizado e presente na maioria dos frameworks mais conhecidos.
- EX: Spring, PicoContainer, Castle Windsor, StructureMap, Unity entre outros.