Instituto de Computação

Padrões GRASP

Leonardo Gresta Paulino Murta

leomurta@ic.uff.br

Agenda

- Introdução
- Estilo MVC
- Padrões
  - *Expert*
  - *Creator*
  - *Controller*
  - *Low Coupling*
  - *High Cohesion*
  - *Polymorphism*
  - *Pure Fabrication*
  - *Indirection*
  - *Don't Talk to Strangers*

Introdução

- A **qualidade** de um projeto orientado a objetos está fortemente relacionada com a **distribuição de responsabilidades**
- As responsabilidades de um projeto podem ser divididas em "conhecer" e "fazer"
  - As responsabilidades "conhecer" estão relacionadas à **distribuição das características** do sistema entre as classes
  - As responsabilidades "fazer" estão relacionadas com a **distribuição do comportamento** do sistema entre as classes

Introdução

- A **principal característica** da atribuição de responsabilidades está em **não sobrecarregar os objeto** com responsabilidades que poderiam ser **delegadas**
  - O objeto só deve fazer o que está relacionado com a sua abstração. Para isso, delega as demais atribuições para quem está mais apto a fazer
  - Quando o objeto não sabe quem é o mais apto, pergunta para algum outro objeto que saiba

Introdução

- Os **princípios de projeto** fornecem os fundamentos necessários para o entendimento de **o que** é um bom projeto
- Entretanto, não é retratado claramente **como** se pode obter um bom projeto
- Para facilitar o entendimento de **como** fazer um bom projeto, esse conhecimento foi codificado na forma de padrões

Introdução

- **Padrões** descrevem, em um **formato estruturado**, um **problema** e uma **possível solução** para este problema
- Padrões não são criados, são descobertos!
  - A solução descrita foi aplicada com sucesso por especialistas da área inúmeras vezes, podendo ser considerada uma boa solução
- As principais **utilidades de um padrão** estão relacionadas com
  - Formalização e propagação do conhecimento
  - Uniformização do vocabulário

Introdução

- Uma **linguagem de padrões** agrega um conjunto de **padrões relacionados** para um contexto em particular
- No contexto de projeto orientado a objetos foi criada uma linguagem de padrões conhecida como GRASP (Larman, G.; 2007)
  - GRASP – General Responsability Assignment Software Patterns
  - Os padrões GRASP descrevem os princípios fundamentais para a atribuição de responsabilidades em projetos OO

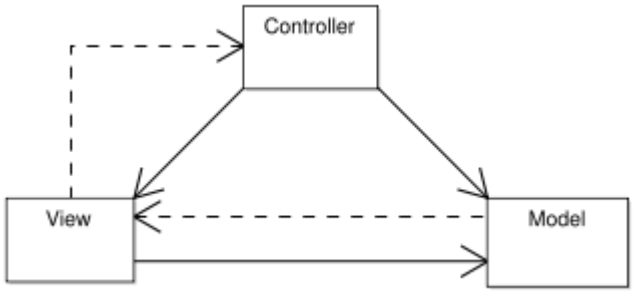
Estilo MVC

- Estilos arquiteturais servem para dar uma diretriz de como a arquitetura do sistema deve ser construída
  - Engenharia de software, diferentemente das demais engenharias, não está sujeita a “leis da natureza”
  - Estilos arquiteturais são “regras artificiais” que devem ser seguidas pelos desenvolvedores
- Um dos principais estilos arquiteturais é o MVC
  - MVC = Model-View-Controller
  - Filosofia básica: as classes do sistema devem ser separadas em três tipos: modelo, visão e controle

Estilo MVC

- Modelo:
  - São as classes derivadas do processo de análise
  - Representam os principais conceitos do domínio
  - São usualmente persistidas em banco de dados
- Visão:
  - São as classes criadas durante o projeto para fazer interface com o usuário
  - Normalmente manipulam classes de modelo
- Controle:
  - São as classes que fazem a orquestração

Estilo MVC



- Regras:
  - Classes de modelo não conhece ninguém (a não ser outras classes de modelo)
  - Classes de visão conhece somente classes de modelo (e outras classes de visão)
  - Classes de controle conhece tanto classes de modelo quanto classes de visão (e outras classes de controle)

Estilo MVC

- Vantagens:
  - Separação de responsabilidades clara
  - Alto grau de substituição da forma de interface com o usuário
  - Alto grau de reutilização das entidades do domínio (classes de modelo)
  - Possibilidade de múltiplas interfaces com o usuário, trabalhando de forma simultânea (modo texto, janelas, web, celular, etc.)

Padrão *Expert*

- Problema:
  - Qual é o princípio mais básico de atribuição de responsabilidades em projeto OO?
  - Em um sistema com centenas de classes, como selecionamos quais responsabilidades devem estar em quais classes?
- Exemplo:
  - Em um sistema de PDV, quem deveria ser o responsável pelo cálculo do total de um pedido?

Padrão *Expert*

- Solução:
  - Atribuir a responsabilidade ao especialista
  - O especialista é a classe que tem a informação necessária para satisfazer a responsabilidade
- Exemplo:
  - Em um sistema de PDV, o responsável pelo cálculo do total de um pedido deveria ser a própria classe Pedido

Padrão *Expert*

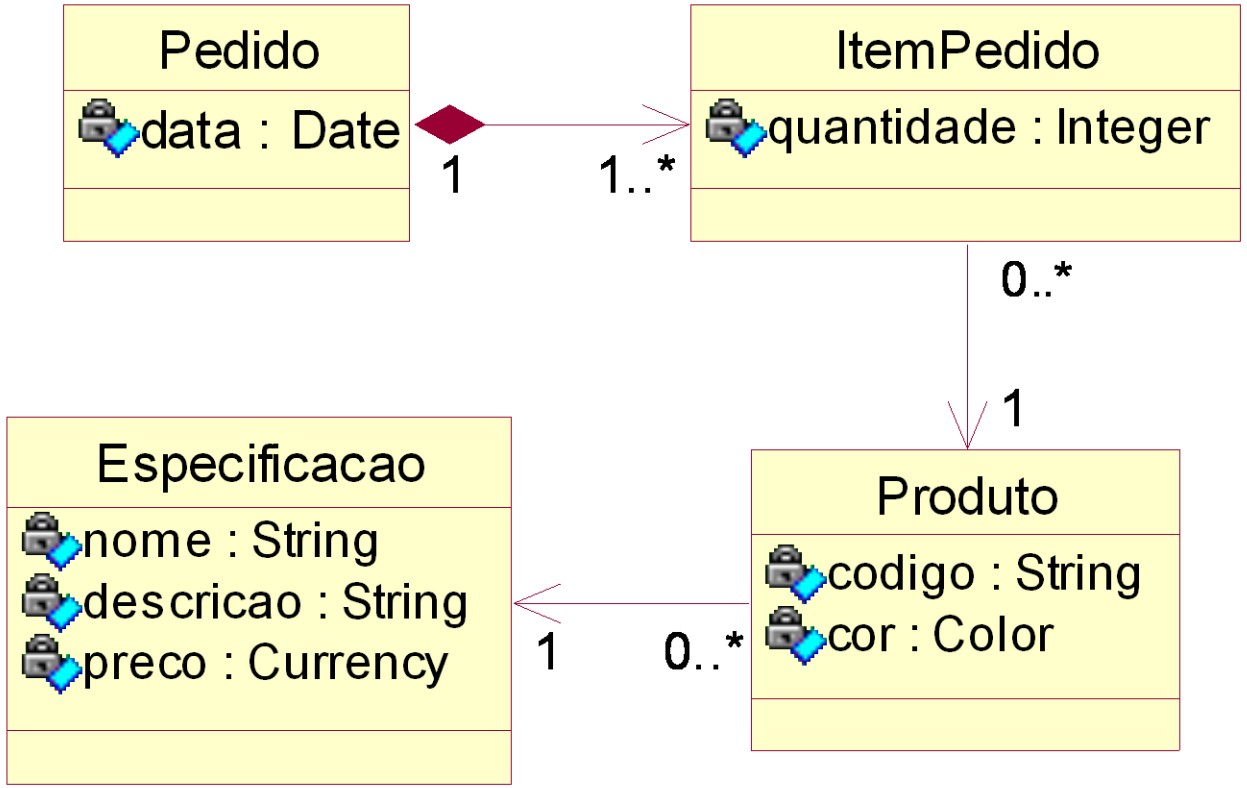
- Benefícios:
  - Leva a projetos onde o objeto de software faz o que o objeto real faria
  - Mantém o encapsulamento e não aumenta o acoplamento, pois utiliza informações próprias
  - Distribui o comportamento uniformemente entre as classes do sistema, aumentando a coesão das mesmas

Padrão *Expert*

- Sinônimos:
  - Colocar responsabilidades com os dados
  - Aquele que conhece faz
  - Quem sabe faz
  - Animação
  - Fazê-lo eu mesmo
  - Colocar os serviços com os atributos que eles utilizam
  - Especialista na informação

Padrão *Expert* *(Exercício)*

Desenvolva o modelo abaixo (através de modelo de seqüência) para inserir a responsabilidade de cálculo do total do pedido segundo o padrão *Expert*:



Padrão *Creator*

- Problema:
  - Quem deveria ser responsável pela criação de uma nova instância de uma classe?
  - A criação de objetos é uma atividade comum em sistemas OO. É necessário um princípio que atribua responsabilidades para essa criação
- Exemplo:
  - Em um sistema de PDV, quem deveria ser responsável por criar um novo item de pedido?

Padrão *Creator*

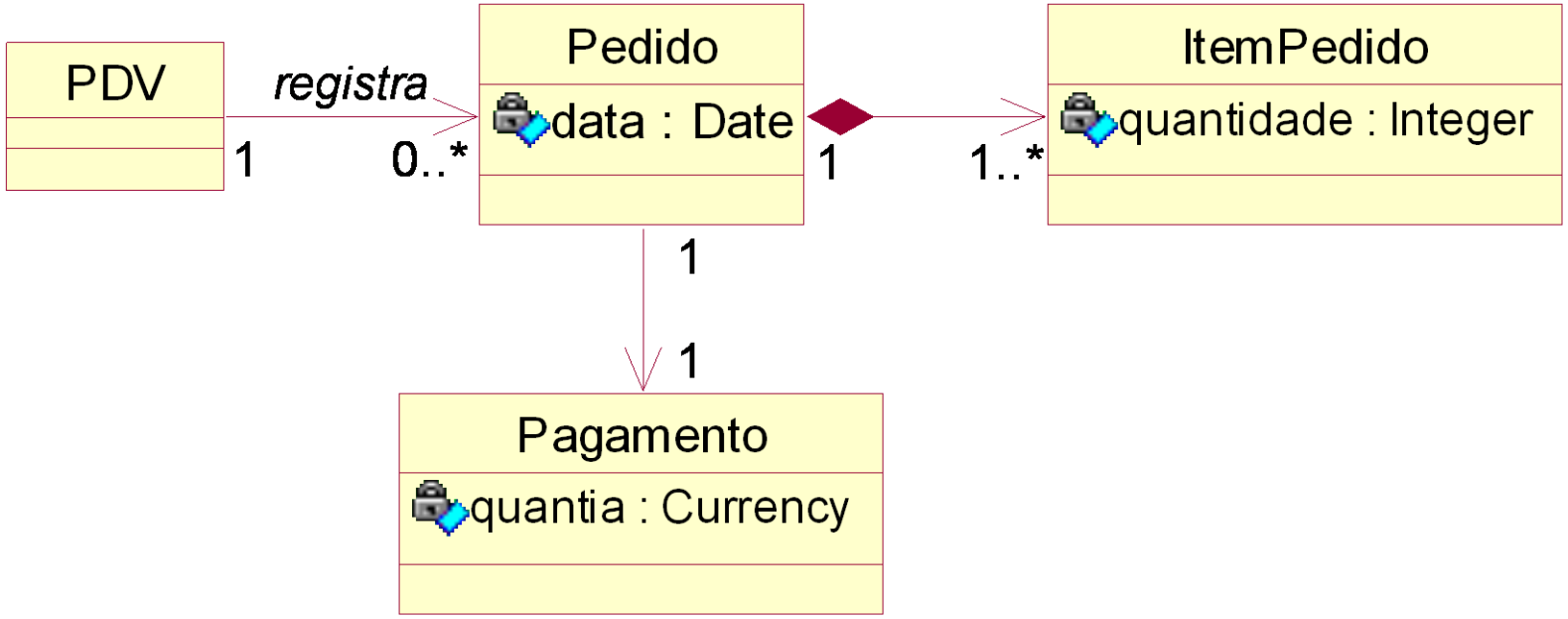
- Solução: Atribua à classe A a responsabilidade de criar instâncias da classe B se ao menos:
  - **A contém objetos de B**
  - **A agrega objetos de B**
  - A registra objetos de B
  - A usa de maneira muito próxima objetos de B
  - A tem os dados necessários para a construção de objetos de B

Padrão *Creator*

- Exemplo:
  - Em um sistema de PDV, a classe Pedido deveria ser responsável por criar instâncias de ItemPedido
- Benefício:
  - Não aumenta o acoplamento, pois a visibilidade entre as classes envolvidas já existia
- Padrão relacionado:
  - Acoplamento fraco

Padrão *Creator* *(Exercício)*

Desenvolva o modelo abaixo (através de modelo de seqüência) para determinar a responsabilidade de criação de pagamento, segundo o padrão *Creator*:



Padrão *Controller*

- Problema:
  - Quem deveria ser responsável por tratar um evento de sistema?
  - Os eventos de sistema estão associados às mensagens de sistema, que são geradas a partir dos passos dos casos de uso
- Exemplo: o caixa registra o código e a quantidade de um item de pedido. Qual classe deve receber e tratar esse evento?

Padrão *Controller*

- Solução: Os eventos de sistemas devem ser tratados por uma das classes abaixo:
  - Representante do sistema como um todo (*Facade)*
  - Representante do negócio ou da organização
  - Representante do caso de uso em questão (tratador artificial)

Padrão *Controller*

- Importante:
  - independente da escolha de solução, ela deve ser mantida para todo o sistema
  - Cada caso de uso deve estar todo em uma mesma classe;
  - Uma classe pode ter mais de um caso de uso
  - Para manter a separação view-controller, é importante que classes de visão (formulários, janelas, applets, etc.) não sejam responsáveis por esse tratamento

Padrão *Controller*

- Exemplo: o evento de registro de um item de pedido, executado pelo método registraItemPedido(codigo, quantidade) pode ser executado por:
  - PDV, representando o sistema como um todo
  - Loja, representando o negócio
  - ControleCompraItens, representando o caso de uso de compra de itens

Padrão *Controller*

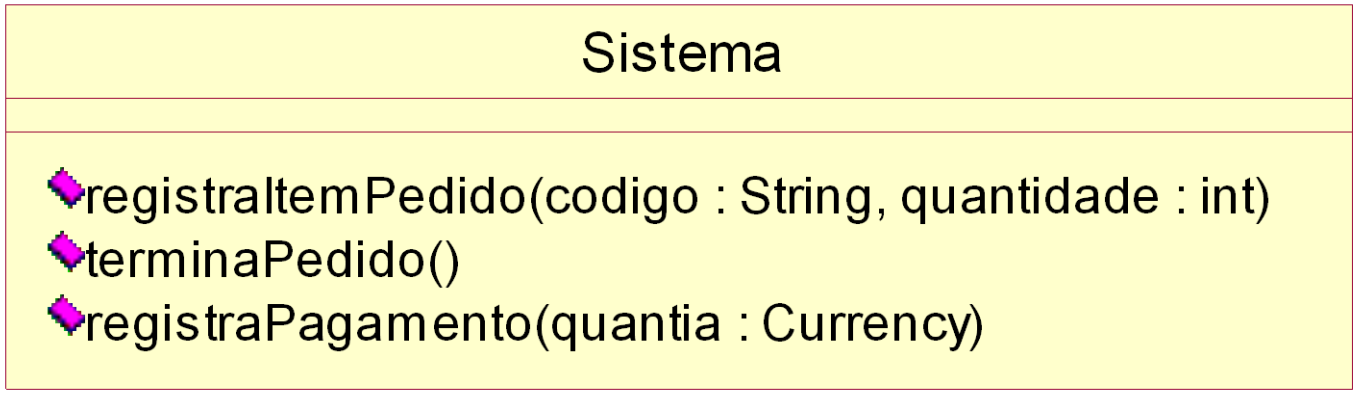
- Os controladores devem somente coordenar a tarefa, delegando a sua execução para os outros objetos do sistema
- O uso de controladores *Facade* (representantes do sistema ou do negócio) é válido somente quando existem poucos eventos de sistema
- Em sistemas com muitos eventos, o uso de tratadores artificiais, um por cada caso de uso, é indicado

Padrão *Controller*

- Benefícios:
  - A separação *view-controller* facilita a reutilização de componentes específicos de negócio e permite o uso dos serviços através de processamento em lote (*batch)*
  - A utilização de uma única classe para todos os eventos de um caso de uso possibilita a manutenção de estado do caso de uso como atributos dessa classe
- O controlador deve ser somente o maestro... não deve tocar nenhum dos instrumentos (mesmo que dê muita vontade)

Padrão *Controller* *(Exercício)*

Para os eventos de sistemas listados abaixo na classe virtual Sistema (vindos do caso de uso Compra de itens), defina uma atribuição de responsabilidades e argumente o motivo da sua escolha:



Padrão *Low Coupling*

- Problema:
  - Como suportar uma dependência baixa e aumentar a reutilização?
  - Modificações em uma classe forçam modificações em outras classes
  - Dificuldade de compreensão de uma classe isoladamente
  - Dificuldade de reutilização por excesso de dependência entre classes (para pegar uma tem que pegar todas)

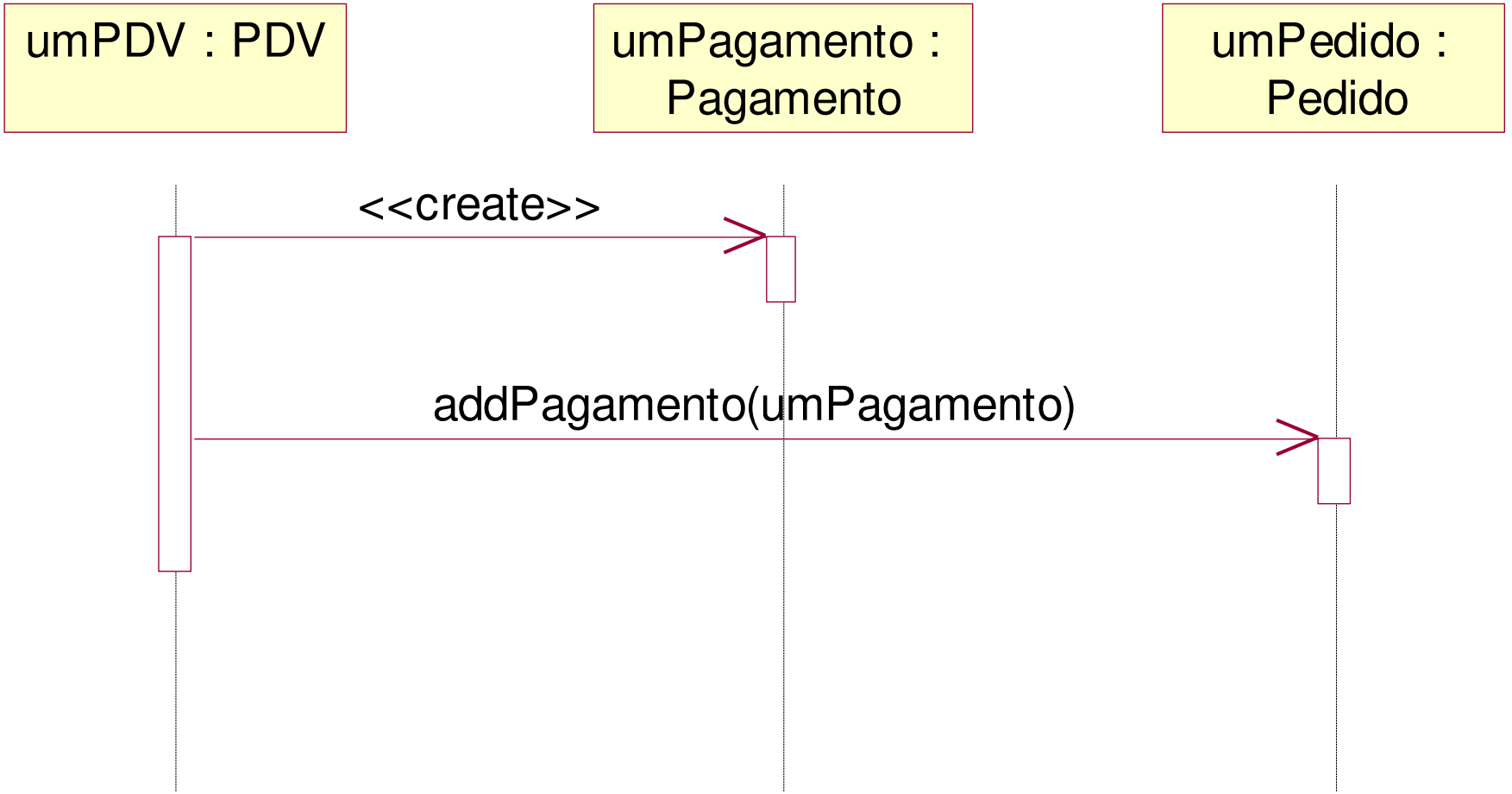
Padrão *Low Coupling*

- Tipos de acoplamento entre as classes A e B:
  - A classe A é subclasse da classe B
  - A classe A tem um atributo do tipo da classe B
  - A classe A tem um método que referencia a classe B através de parâmetro, variável local ou retorno de mensagem
  - A classe A depende de uma interface implementada pela classe B

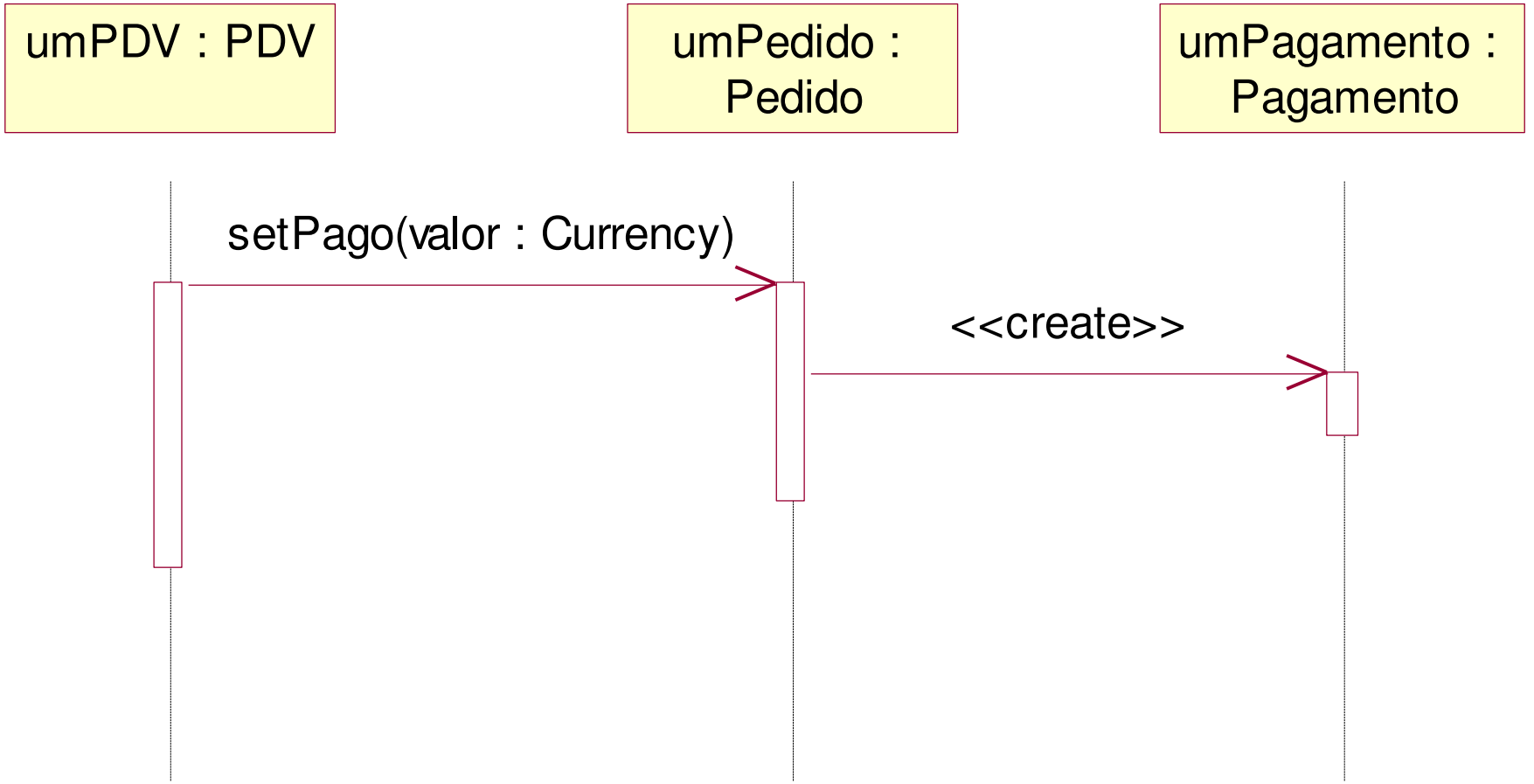
Padrão *Low Coupling*

- Exemplo:
  - Segundo o padrão *Creator*, existem duas soluções para a atribuição da responsabilidade de pagamento:
    - Um PDV é quem registra um pagamento fisicamente, logo a classe PDV é quem deve criar a instância
    - A classe Pedido tem os dados necessários para a construção da instância do pagamento, logo ela é quem deve criar

Padrão *Low Coupling*



Padrão *Low Coupling*



Padrão *Low Coupling*

- Solução:
  - Atribuir as responsabilidades de maneira que o acoplamento permaneça fraco
  - O padrão *Low Coupling* deve ser utilizado como um padrão de avaliação de outros padrões
- Exemplo:
  - Se a classe Pedido criar a instância de Pagamento, a classe PDV não terá seu acoplamento aumentado (PDV não conhece Pagamento)

Padrão *Low Coupling* *(Exercício)*

Dê a sua avaliação sobre o acoplamento nos problemas abaixo e indique se é uma situação desejável:

Cenário A: um sistema onde todos os objetos herdam da classe ObjetoPersistente, para que seja possível implementar um mapeamento objeto-relacional automático

Cenário B: um sistema onde as classes tem somente atributos primitivos e métodos, sem nenhuma associação ou herança

Padrão *High Cohesion*

- Problema:
  - Como manter a complexidade sob controle?
  - As classes são difíceis de compreender
  - As classes são difíceis de reutilizar
  - As classes são difíceis de manter
  - As classes são frágeis, sendo afetadas por praticamente todas as modificações

Padrão *High Cohesion*

- Exemplo:
  - No exemplo da atribuição de responsabilidade de pagamento, qual das duas soluções tem melhor impacto em relação à coesão?
- Solução:
  - Atribuir responsabilidade de forma que as classes não fiquem sobrecarregadas, e que as suas atribuições sejam relacionadas
  - O padrão *High Cohesion* também deve ser visto como um padrão de avaliação de padrões

Padrão *High Cohesion*

- Exemplo:
  - Caso o PDV assuma a responsabilidade de pagamento, provavelmente assumirá todas as demais responsabilidades do sistema
  - Assim, caso o sistema seja grande, a classe PDV ficará extremamente complexa e pouco coesa
  - A solução de delegação para a classe Pedido ajuda a aumentar a coesão geral do sistema

Padrão *High Cohesion*

- Alta coesão na prática:
  - Classes têm usualmente poucos métodos altamente relacionados
  - Tarefas mais complexas são delegadas a objetos associados
- Analogia ao mundo real:
  - Pessoas que não delegam responsabilidades e fazem muitas coisas diferentes ao mesmo tempo tendem a não ser eficientes

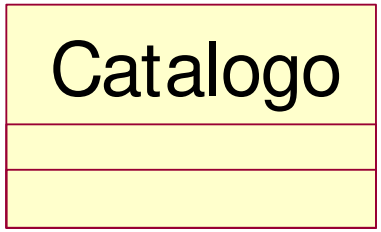
Padrão *High Cohesion*

- Benefícios:
  - Clareza e facilidade de compreensão do projeto
  - Simplificação das atividades de manutenção
  - Favorecimento indireto do baixo acoplamento
  - Facilidade de reutilização, graças à classe ser muito específica
- Alguns casos de coesão moderada trazem benefícios, como, por exemplo, o padrão *Facade*

Padrão *High Cohesion* *(Exercício)*

Como seria possível desenvolver o modelo abaixo para possibilitar o envio do catálogo de produtos através de mala direta, inicialmente via correio, e-mail, celular e fax, colocando em prática o padrão *High Cohesion*?

Obs.: Lembre-se que, no futuro, novas formas de comunicação surgirão.



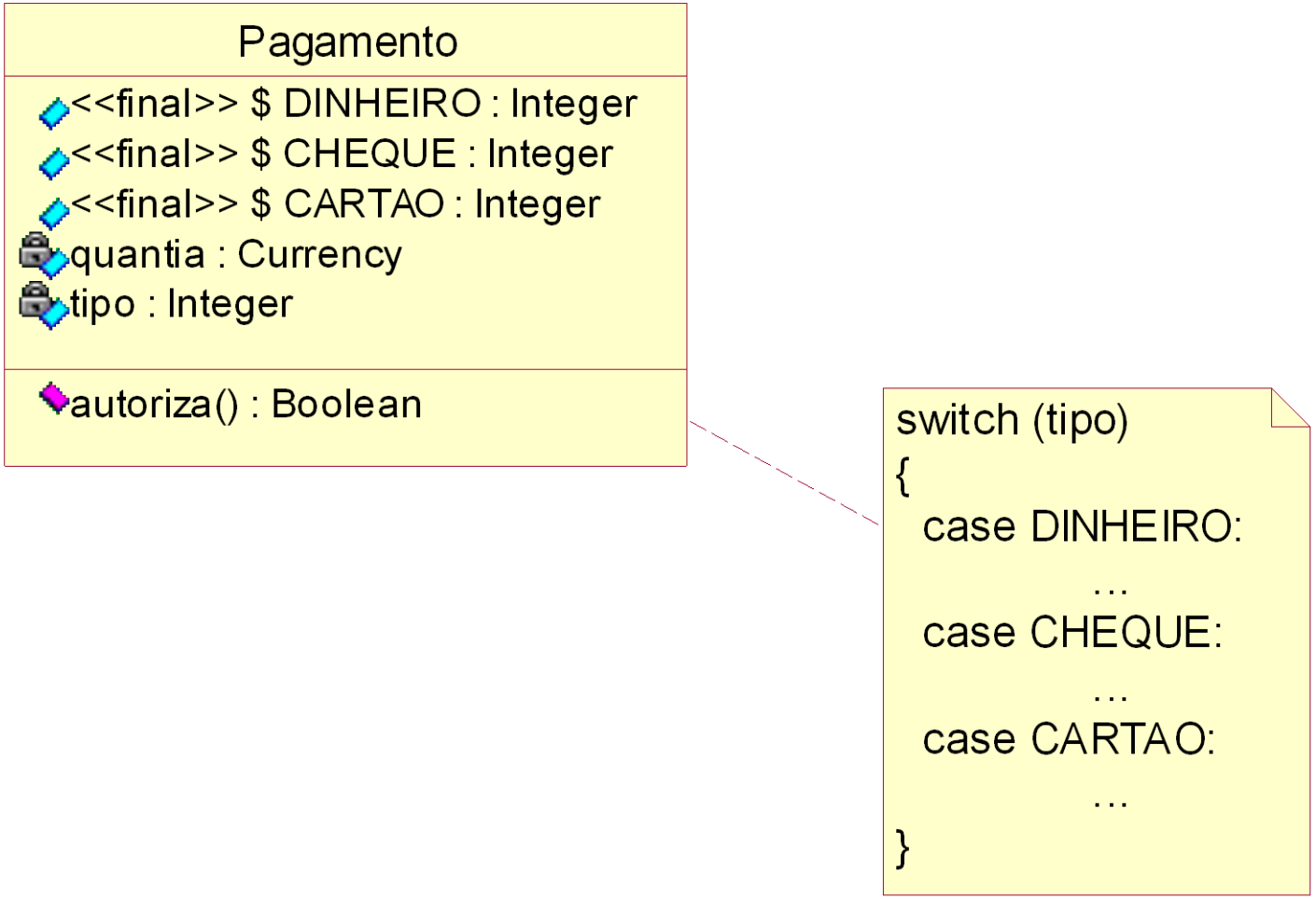
Padrão *Polymorphism*

- Problema:
  - Como tratar alternativas em função do tipo da classe?
    - O uso de ifs aninhados ou switch-case para selecionar comportamento em função do tipo de classe espalha-se por todo o código, dificultando a manutenção
  - Como criar componentes de software substituíveis?
    - A necessidade de substituição de parte de um sistema pode se tornar um problema caso o sistema não tenha sido projetado para isso

Padrão *Polymorphism*

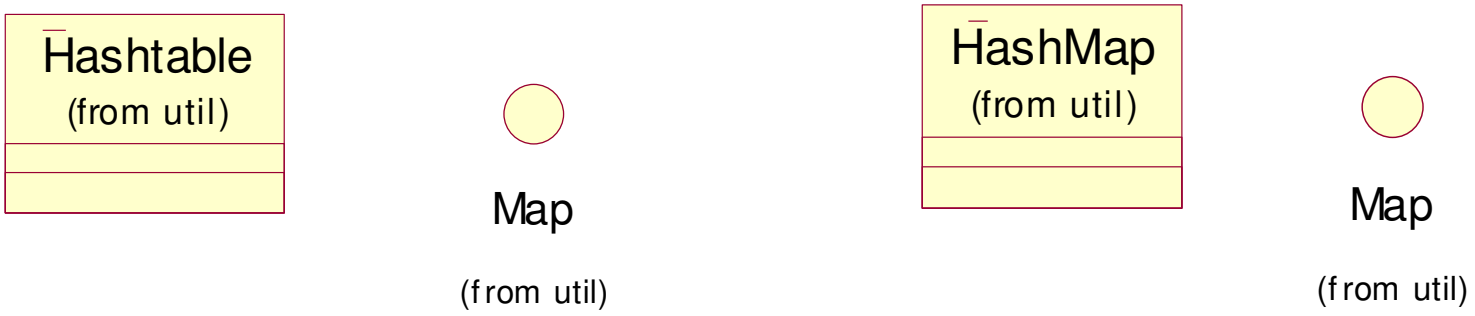
- Exemplo:
  - No sistema de PDV, como seria possível a seleção do tipo de autorização de pagamento?
  - Segundo o padrão *Expert*, o próprio pagamento deveria saber se autorizar

Padrão *Polymorphism*



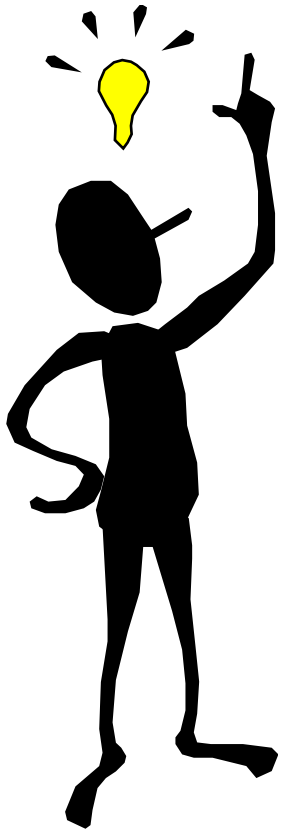
Padrão *Polymorphism*

- Solução:
  - Seleção do comportamento desejado através do mecanismo de polimorfismo
  - Utilização de polimorfismo aplicado ao conceito de interfaces para permitir a substituição de componentes

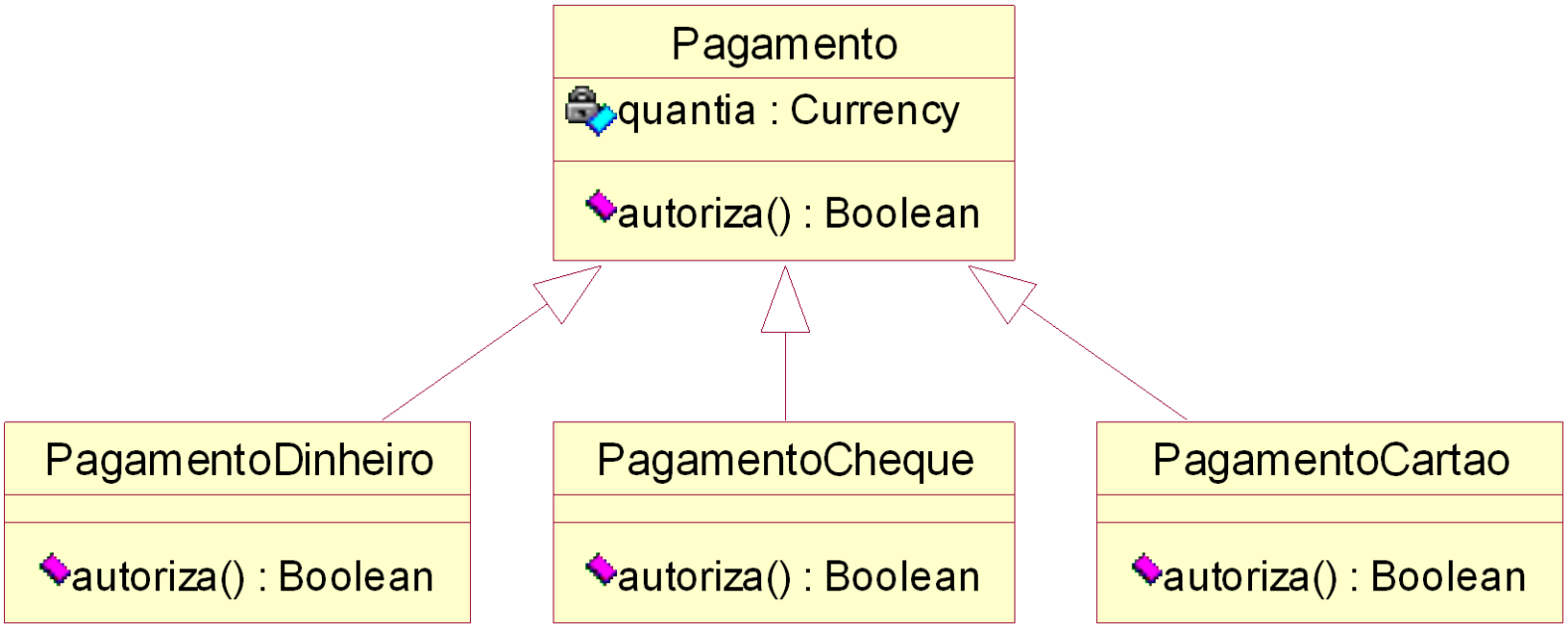


Padrão *Polymorphism*

- Exemplo:
  - Para evitar a necessidade de seleção de comportamento, a classe Pagamento deve definir o método *autoriza()*
  - As subclasses de Pagamento devem aplicar polimorfismo sobre o método *autoriza()*



Padrão *Polymorphism*

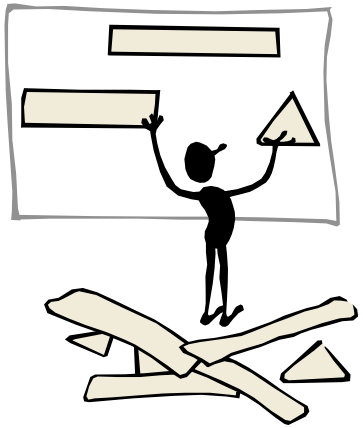


- Benefício:
  - Facilidade de manutenção
  - Facilidade de inserção de um novo tipo de autorização

Padrão *Polymorphism* *(Exercício)*

A utilização do conceito de interfaces ajuda na substituição de componentes. Entretanto, como poderia ser feito *refactoring* de um sistema que precisa ter partes substituídas mas que não foi projetado para isso (não usa componentes)?

Tente estruturar uma sistematização simplificada para essa questão



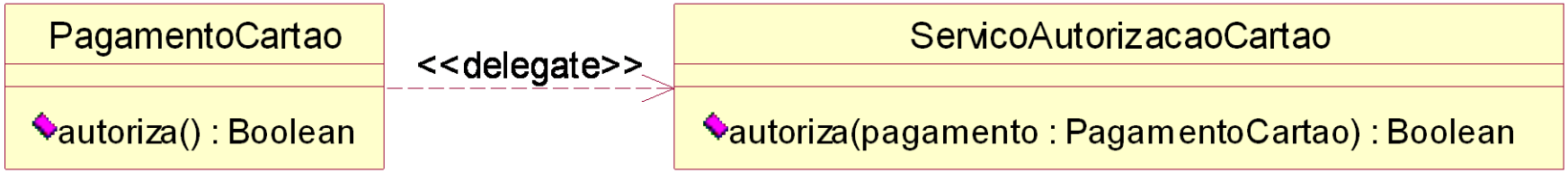
Padrão *Pure Fabrication*

- Problema:
  - A quem atribuir uma responsabilidade quando todas as opções ferem os princípios de acoplamento baixo e coesão alta?
- Exemplo:
  - Segundo o padrão *Expert*, a autorização de pagamento deve ficar na classe Pagamento
  - Entretanto, essa abordagem poderá implicar em baixa coesão e alto acoplamento

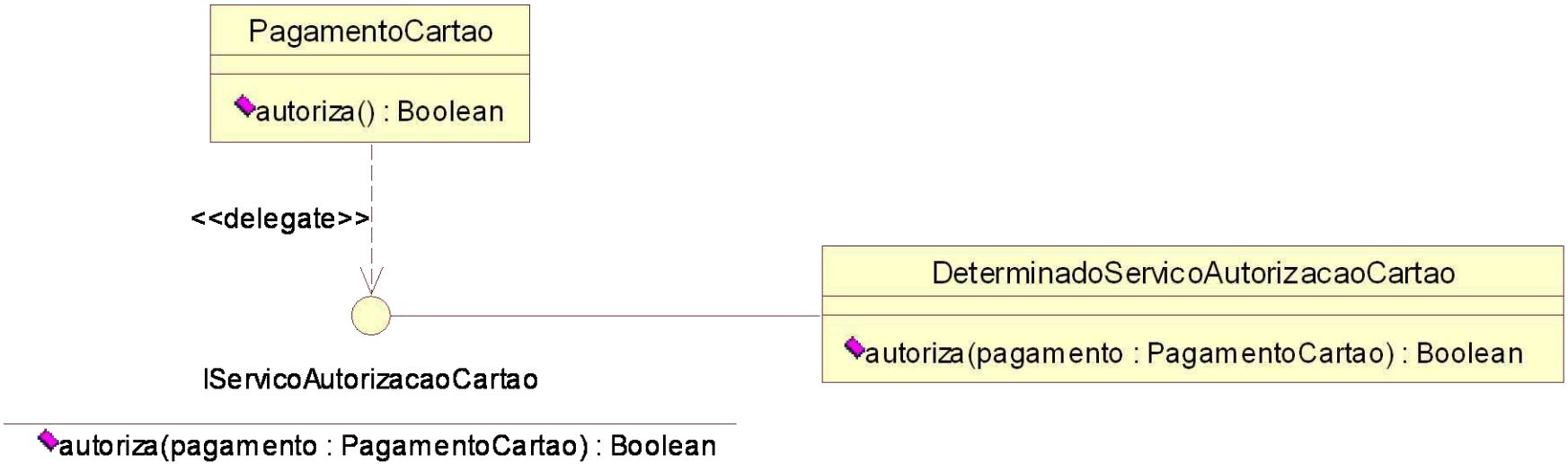
Padrão *Pure Fabrication*

- Solução:
  - Criar classes artificiais (invenções) que tenham alta coesão e baixo acoplamento (classes puras)
- Exemplo:
  - As classes artificiais ficariam responsáveis por todo o conhecimento sobre a comunicação
  - As classes de pagamento continuariam aplicando o padrão *Polymorphism*, aliado a um mecanismo de delegação às classes artificiais

Padrão *Pure Fabrication*



OU

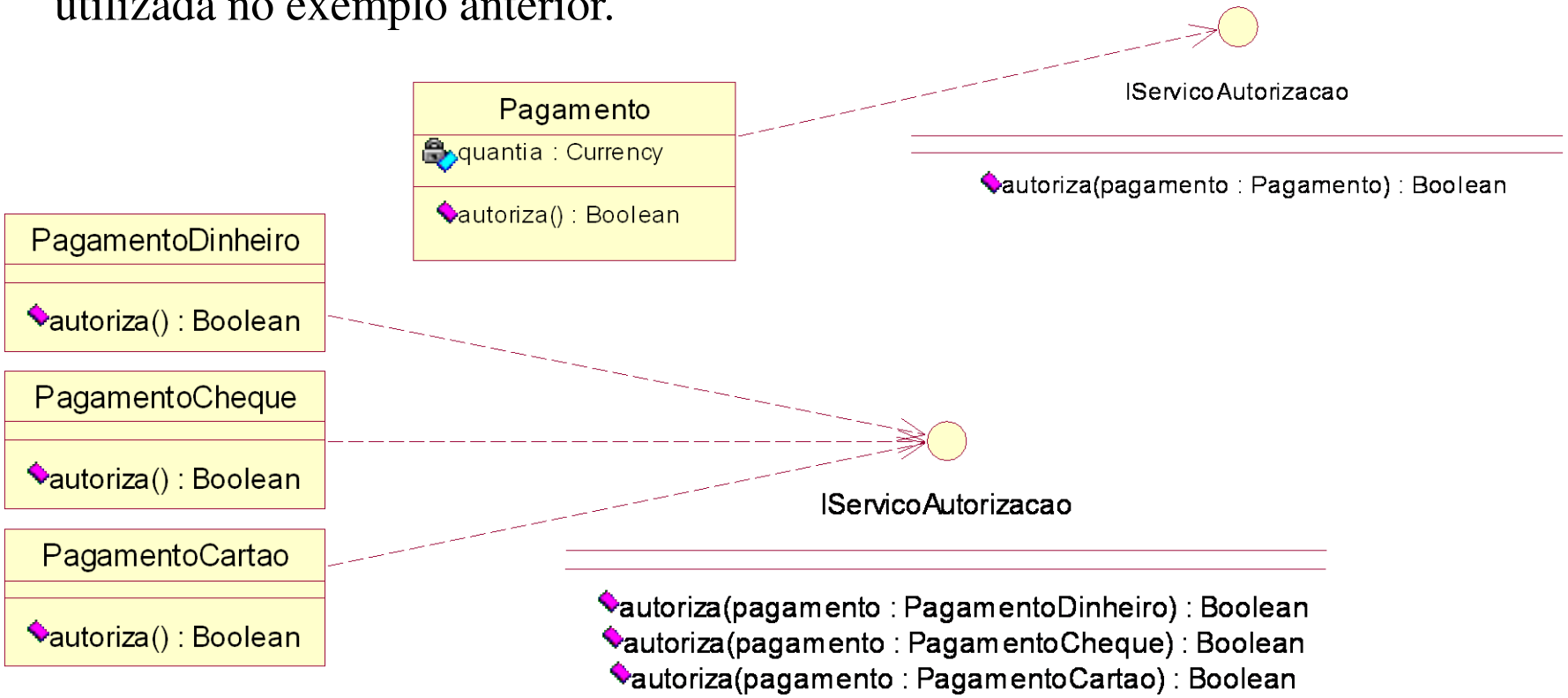


Padrão *Pure Fabrication*

- Benefícios:
  - Remove as características não coesas das classes do domínio de negócio
  - Cria classes muito coesas com essas características
- Problemas:
  - Cria classes altamente funcionais, que não fazem parte da realidade
  - Se utilizado em excesso, poderá transformar um sistema OO em um sistema orientado a eventos

Padrão *Pure Fabrication* *(Exercício)*

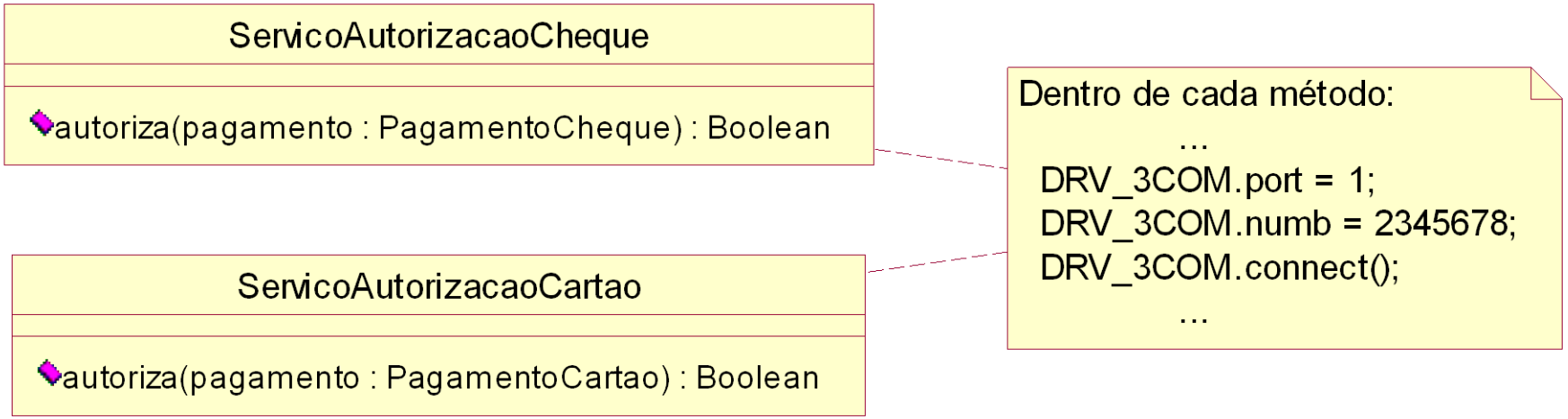
Qual das duas abordagens abaixo você considera melhor? Justifique citando os prós e contras de cada abordagem e compare com a abordagem utilizada no exemplo anterior.



Padrão *Indirection*

- Problema:
  - Como posso evitar o acoplamento direto?
  - Caso uma classe seja acoplada a serviços, será impossível reutilizar esses serviços
- Exemplo:
  - Para que o pagamento via cartão ou cheque funcione, devem existir chamadas ao driver de modem
  - No caso do pagamento via dinheiro, devem existir chamadas ao leitor de moeda (contra falsificação)

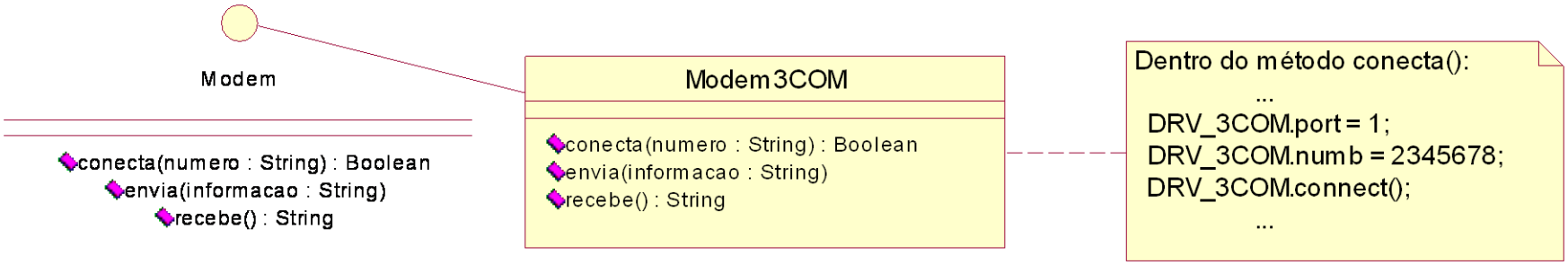
Padrão *Indirection*



- Solução:
  - Criar um objeto intermediário, fazendo indireção para o serviço
  - São criados um ou mais níveis de indireção, para possibilitar a reutilização e substituição de código

Padrão *Indirection*

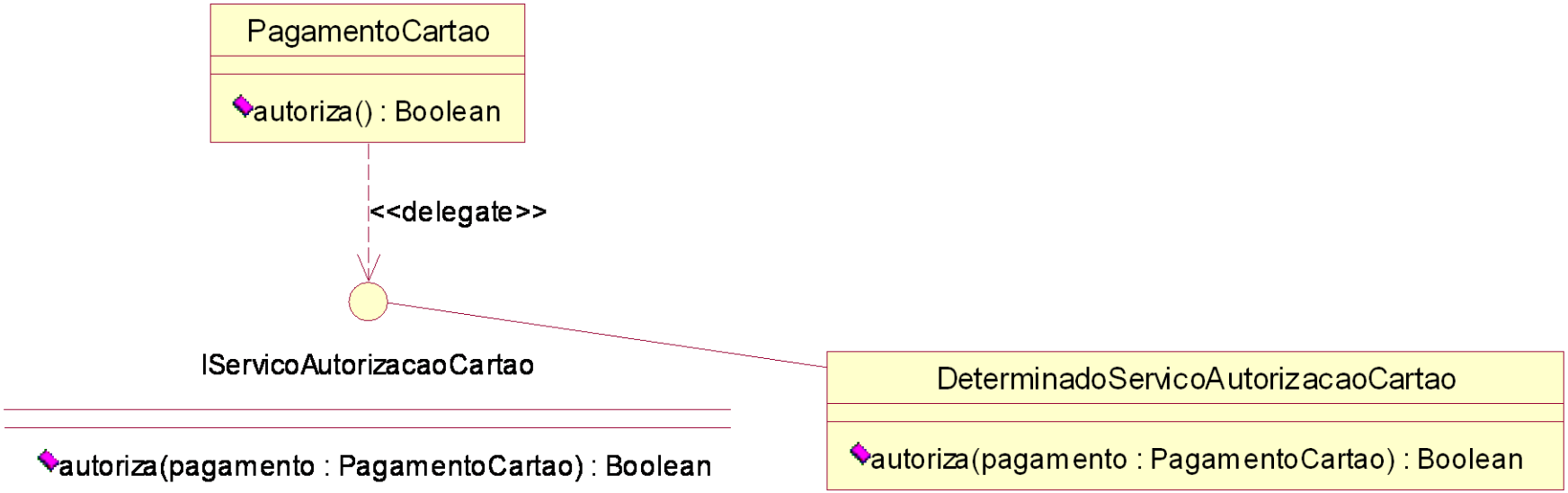
- Exemplo:
  - Uma classe deve ser criada para representar (fazer indireção para) o modem
  - Uma classe deve ser criada para representar (fazer indireção para) o leitor de moeda



Padrão *Indirection* *(Exercício)*

Evolua o modelo abaixo para contemplar o padrão *Indirection* para as três formas de pagamento.

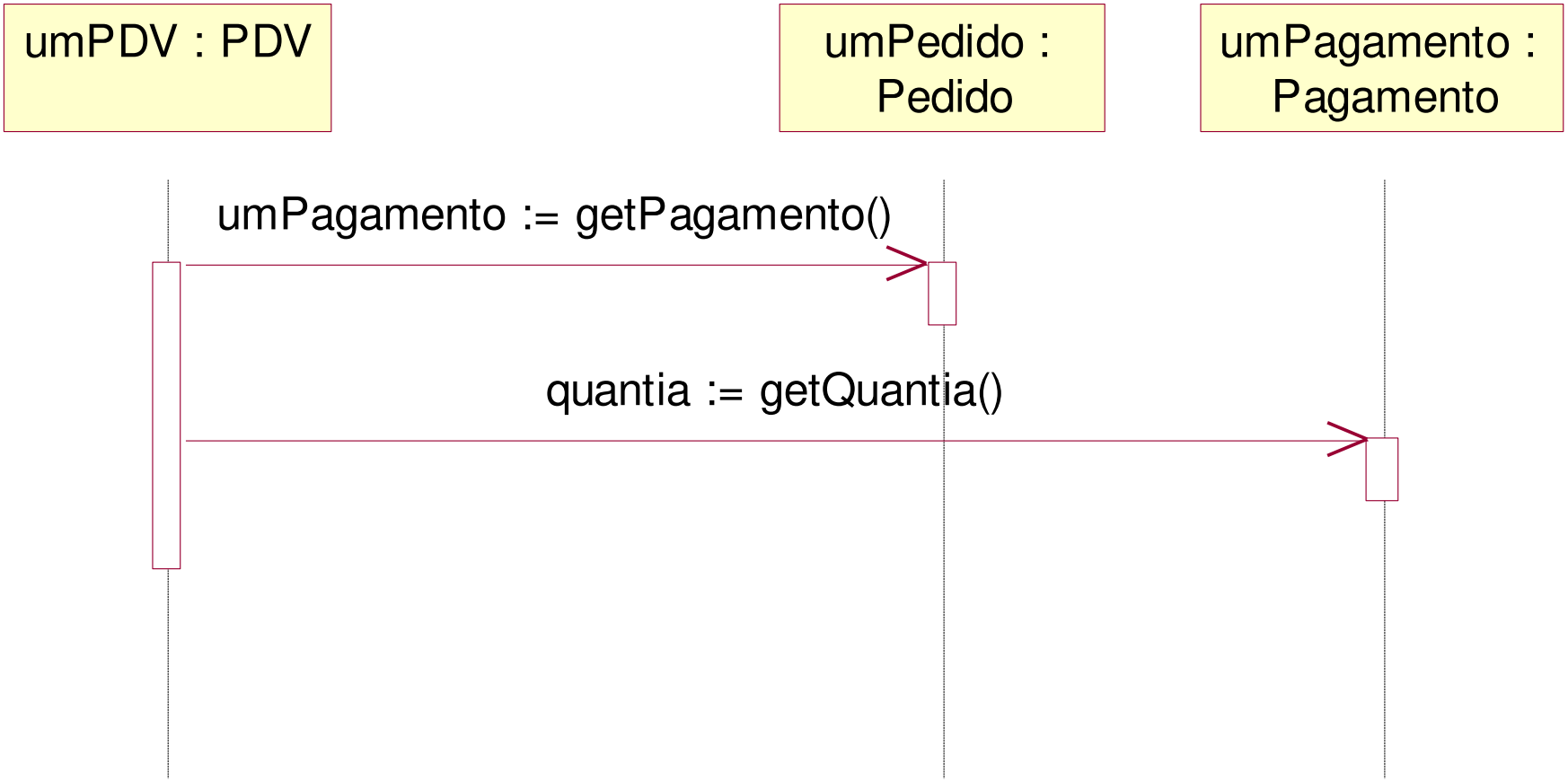
Sobre o modelo gerado, perceba que um determinado serviço de autorização pode ser visto como um componente (interfaces providas e requeridas).



Padrão *Don't Talk to Strangers*

- Problema:
  - Como seria possível fortalecer o encapsulamento?
  - O conhecimento da estrutura interna de relacionamentos de uma classe pode dificultar a manutenção
- Exemplo:
  - Para a classe PDV saber qual é a quantia de um pagamento, ela irá consultar a classe Pedido, que fornecerá o Pagamento que contém a informação

Padrão *Don't Talk to Strangers*



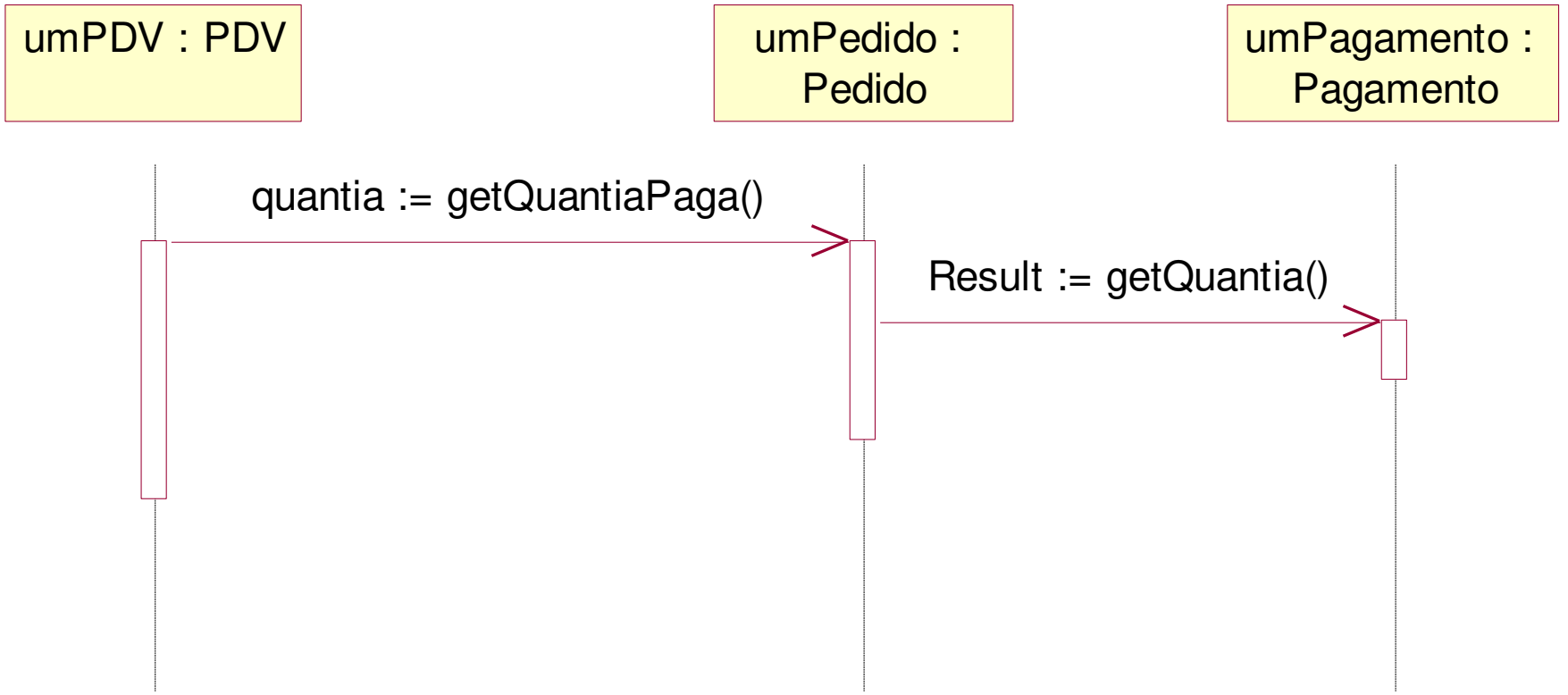
Padrão *Don't Talk to Strangers*

- Solução: Segundo a Lei de Demeter, um método "A" deve enviar mensagens somente para:
  - O próprio objeto (this)
  - Os próprios atributos
  - Os argumentos do próprio método "A"
  - Um objeto criado no método "A"
  - Um elemento de alguma coleção que seja atributo, argumento ou criada no método "A"

Padrão *Don't Talk to Strangers*

- A solução proposta é conhecida como **promoção de interface**, pois promove algumas operações para a interface mais externa
- Exemplo:
  - A classe Pedido deve adicionar à sua interface um método chamado *getQuantiaPaga()*, que forneceria a quantia da classe Pagamento
  - Assim, a classe PDV perde sua dependência em relação a classe Pagamento, o que diminui o seu acoplamento

Padrão *Don't Talk to Strangers*

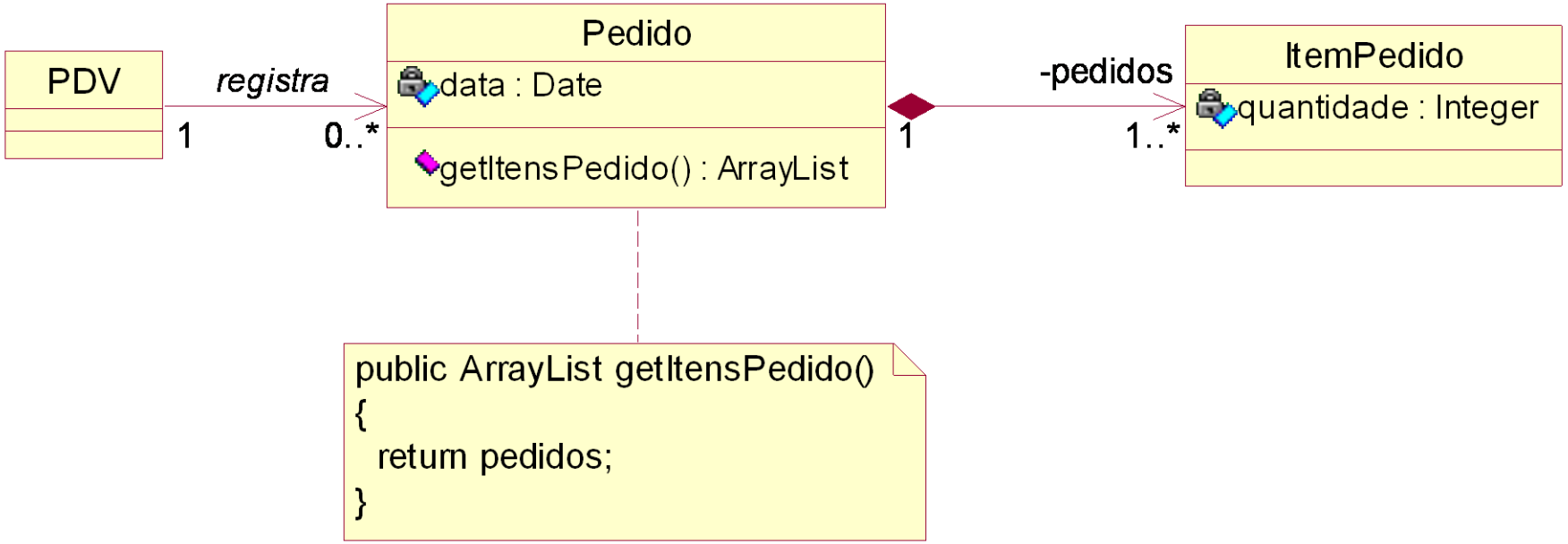


Padrão *Don't Talk to Strangers*

- A Lei de Demeter deve ser rompida em algumas situações:
  - Uso de classes intermediárias (broker)
  - Uso de classes representante (proxy)
  - Sempre que seja utilizado algum tipo de classe que tem o objetivo de fornecer objetos de outras classes (ex.: Singleton, Factory, etc.)

Padrão *Don't Talk to Strangers (Exercício)*

Identifique falhas no modelo abaixo, referentes ao projeto do método *getItensPedido()* e argumente possíveis soluções:



Bibliografia

- Craig Larman, 1999, “Utilizando UML e Padrões”, 1ª ed., Prentice-Hall
- Craig Larman, 2007, “Utilizando UML e Padrões”, 3ª ed., Bookman.

Instituto de Computação

Padrões GRASP

Leonardo Gresta Paulino Murta

leomurta@ic.uff.br