Projeto de objetos com responsabilidade

Prof. Wladmir Cardoso Brandão

PUC Minas

Bibliografia

- LARMAN, Graig. Utilizando UML e Padrões:Uma introdução a análise e ao projeto orientados a objetos. Porto Alegre: Bookman,
  - 2a Edição, 2004. capítulos 16 e 22
  - 3a Edição, 2007. capítulos 17 e 25

Responsabilidade

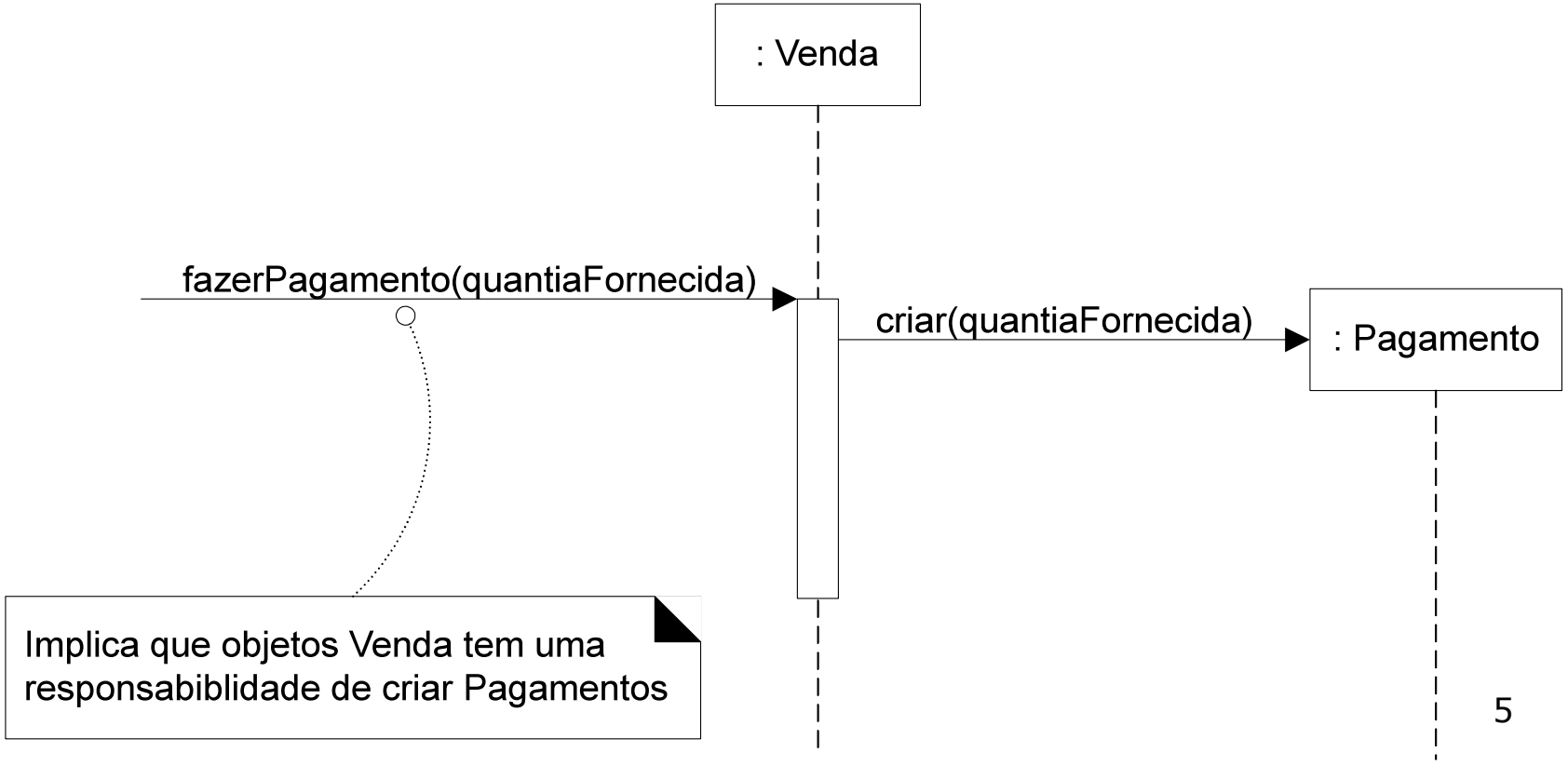
- um contrato ou obrigação de uma classe
  - em termos de comportamento dos seus objetos
- tipos de responsabilidades
  - **conhecer**
    - dados privados encapsulados
    - objetos relacionados
    - dados que podem ser derivados ou calculados
      - *venda* é responsável por conhecer seu total
  - **fazer**
    - criar um objeto, executar um cálculo
    - iniciar uma ação em outros objetos
    - controlar e coordenar atividades em outros objetos
      - *venda* é responsável por criar *LinhasDeItemDeVenda*

Responsabilidades e métodos

- Responsabilidades de conhecer são, em geral, dedutíveis do modelo de domínio
- Uma responsabilidade não é a mesma coisa que um método
  - pode envolver um ou vários métodos de várias classes
- Métodos são implementados para satisfazer às responsabilidades

Responsabilidades e interações

- Diagramas de interação mostram escolhas de atribuição de responsabilidades a objetos
  - as mensagens refletem as responsabilidades atribuídas



Padrões de desenvolvimento

- expressam uma solução para um determinado problema em um determinado contexto incluindo:
  - nome (facilita abstração e a comunicação)
  - conselhos sobre sua aplicação em novas situações
  - discussão sobre as consequências de seu uso
- criados por desenvolvedores experientes
  - sugerem coisas que se repetem, princípios existentes
- baseados na mesma idéia dos padrões arquiteturais de Christopher Alexander

Padrões GRASP

- GRASP = General Responsability Assignment Software Patterns
- O que os padrões GRASP fazem?
  - Os padrões GRASP descrevem princípios fundamentais de projeto baseado em objetos e atribuição de responsabilidades aos mesmos.
- Por que os padrões GRASP são importantes?
  - Um desenvolvedor novato na tecnologia de objetos necessita dominar os princípios básicos rapidamente
  - padrões GRASP são a base de um projeto de sistema

Padrões GRASP fundamentais

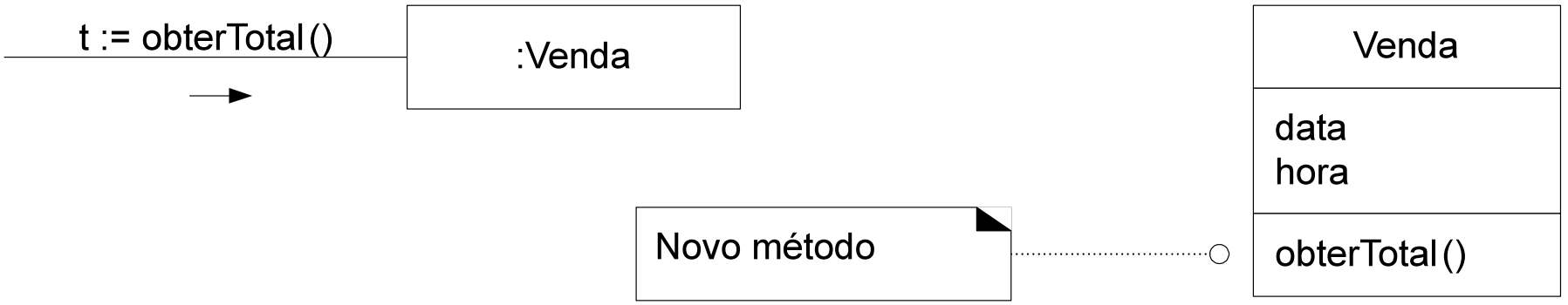
- Especialista na Informação
  - Information Expert
- Criador
  - Creator
- Coesão Alta
  - High Cohesion
- Acoplamento Fraco
  - Low Coupling
- Controlador
  - Controller

Especialista na Informação

- Problema
  - Qual é o princípio básico de atribuição de responsabilidades?
- Solução
  - Atribuir uma responsabilidade ao especialista na informação
    - a classe que tem a informação necessária para satisfazer a responsabilidade

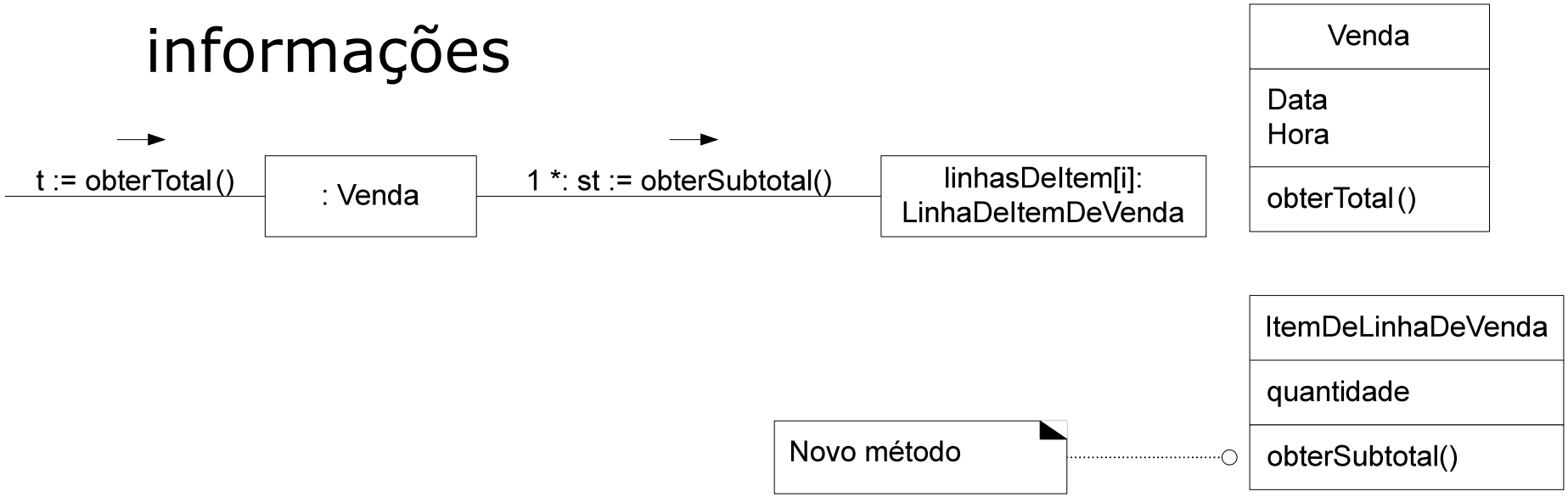
Exemplo - Especialista

- Que informação é necessária para determinar o total geral?
  - conhecer todas as instâncias de *LinhaDeItemDeVenda* de uma venda e a soma de seus subtotais
- Quem deve ser o responsável por conhecer o total geral de uma venda?
  - Venda pois conhece a informação necessária, é a especialista na informação



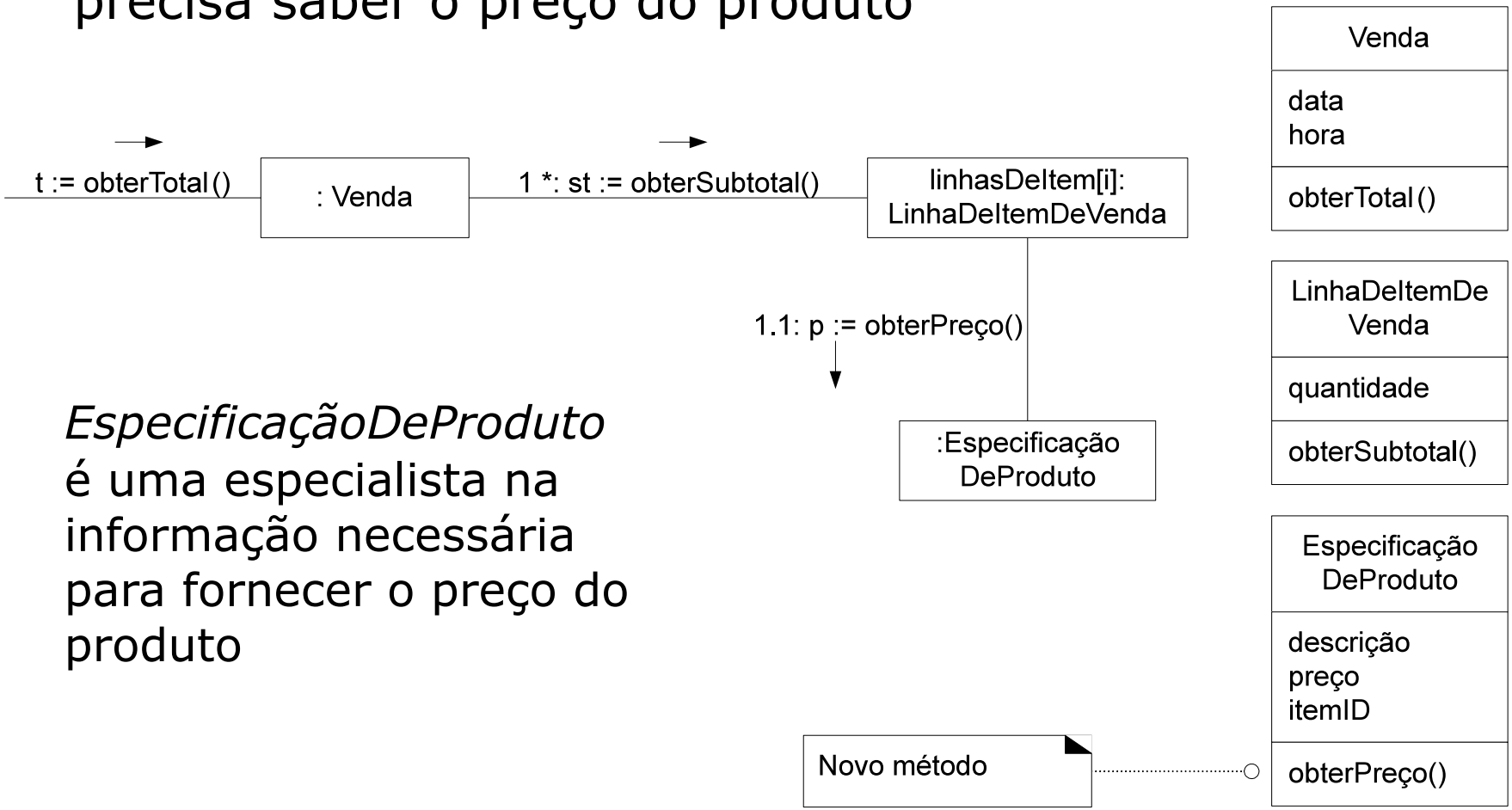
Exemplo - Especialista

- Que informação é necessária para determinar o subtotal da *LinhaDeItemDeVenda*?
  - *LinhaDeItemDeVenda.quantidade*
  - *EspecificaçãoDeProduto.preço*
- *LinhaDeItemDeVenda* deve determinar o subtotal pois conhece ambas informações



Exemplo - Especialista

- Para satisfazer a responsabilidade de conhecer e informar seu subtotal, uma *LinhaDeItemDeVenda* precisa saber o preço do produto



EspecificaçãoDeProduto é uma especialista na informação necessária para fornecer o preço do produto

Exemplo - Especialista

- Conclusão
  - Para satisfazer a responsabilidade de conhecer e informar o total da venda, três responsabilidades foram atribuídas para três classes de objetos

Classe de Projeto	**Responsabilidade**
Venda	Sabe o total da venda
LinhaDeItemDeVenda	Sabe o subtotal da linha de item
EspecificaçãoDeProduto	Sabe o preço do produto

Especialista

- Contra-indicações
  - viola a separação dos principais interesses
    - por exemplo: lógica e controle
- Benefícios
  - encapsulamento é mantido
  - comportamento distribuído
- Nomes alternativos
  - colocar as responsabilidades com os dados, quem sabe faz, fazê-lo eu mesmo, colocar serviços com os atributos com os quais eles trabalham

Como aplicar o especialista

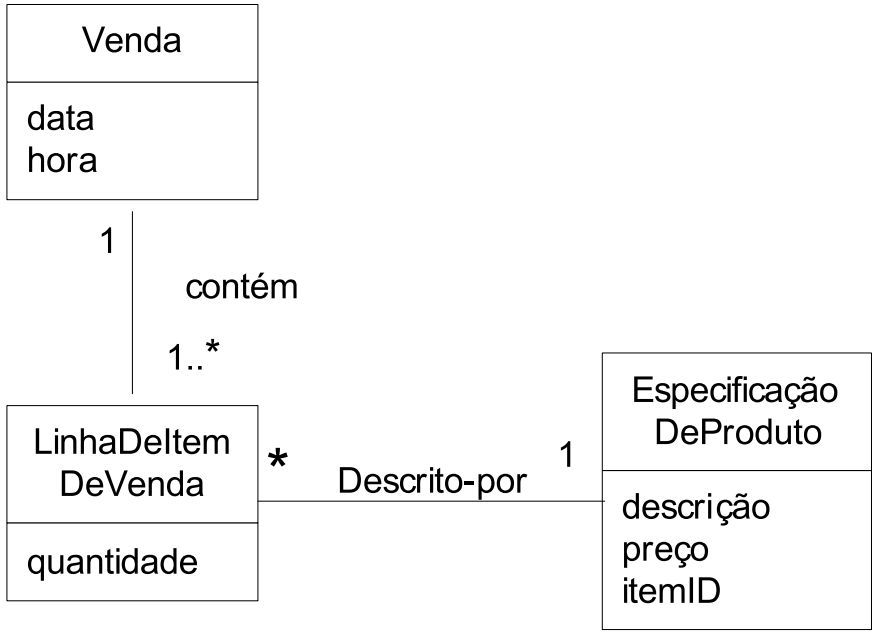
1. Criar um diagrama de interação seguindo os passos abaixo.
2. Partindo dos eventos de um caso de uso, fazer a pergunta "de quem é a responsabilidade por realizar isto?". Procurar no modelo de classes de projeto e no modelo de classes de domínio.
   - Em primeira instância, a responsabilidade seria da classe que tem as informações necessárias para tal.
3. Definir quais colaborações são necessárias (informações de outros objetos) para cumprir a responsabilidade do objeto considerado.
4. Para cada objeto responsável pelas colaborações (que têm a responsabilidade de realizá-las), repetir o passo acima.

Criador

- Problema
  - Quem deve ser responsável pela criação de uma nova instância de uma classe?
- Solução
  - Atribuir à classe B a responsabilidade de criar uma instância da classe A se:
    - B *agrega* objetos de A
    - B *contém* objetos de A
    - B *registra* objetos de A
    - B *usa* objetos de A
    - B *tem os dados de inicialização* para criar instâncias de A
      - B é um especialista na criação de A

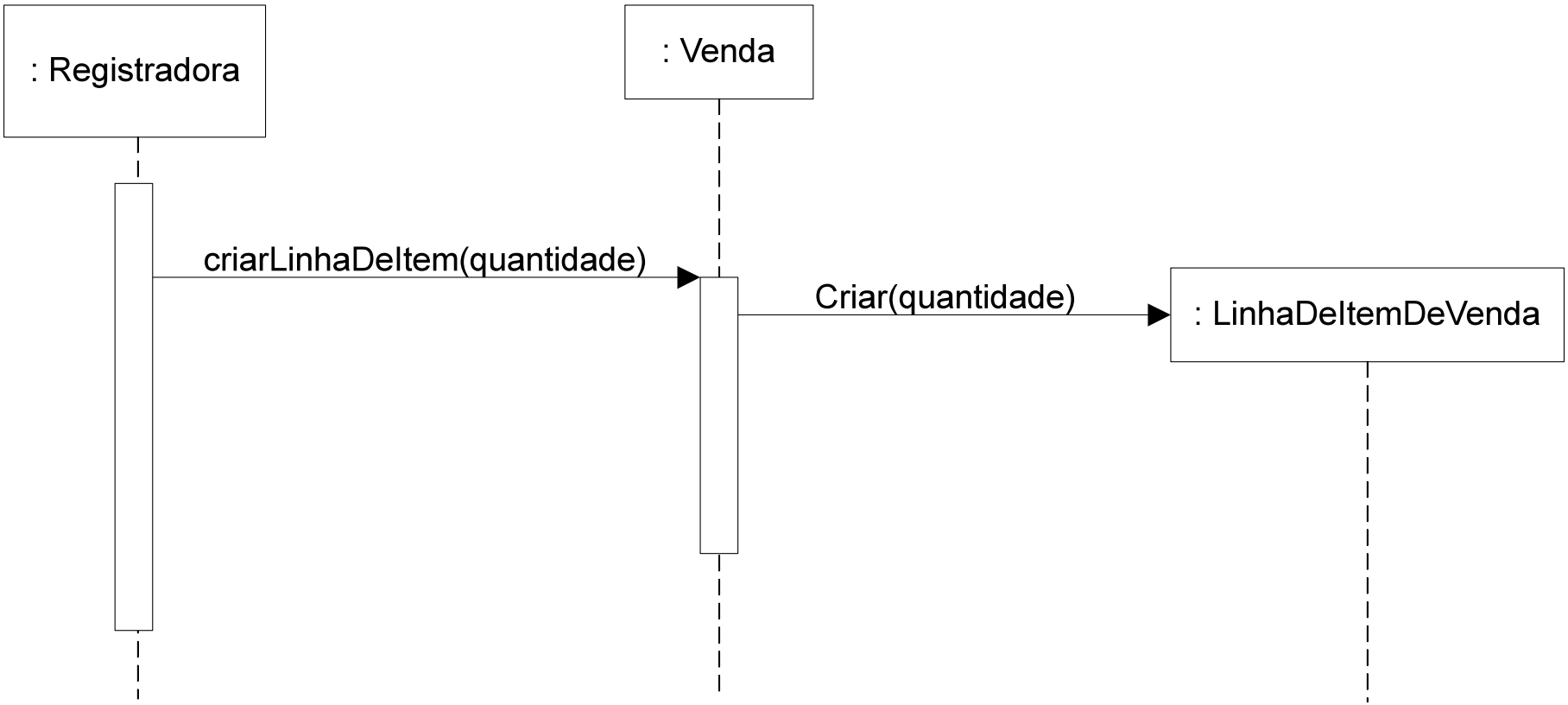
Exemplo - Criador

- Quem deve ser responsável por criar uma instância de *LinhaDeItemDeVenda*?
  - Segundo o padrão Criador, *Venda* deve ser responsável, pois contém ou agrega muitos objetos *LinhaDeItemDeVenda*



Exemplo - Criador

- Projeto das interações



Criador

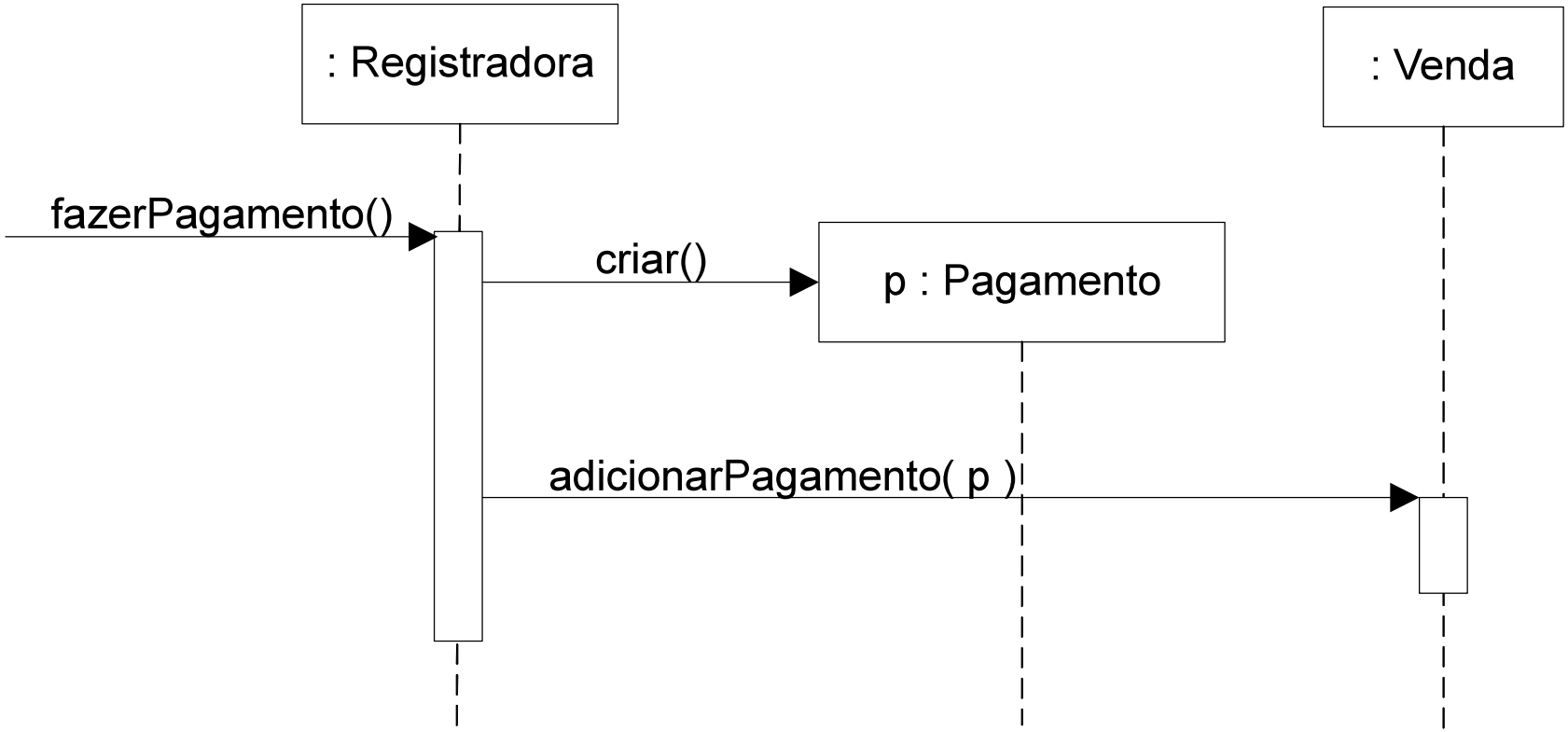
- contra-indicação
  - Não é indicado se a criação de um objeto for uma tarefa complexa
    - delegar a criação a uma classe auxiliar chamada Fábrica
- Benefícios
  - Acoplamento fraco
    - não aumenta o acoplamento pois provavelmente a classe criada já é visível à classe criadora devido às associações existentes

Acoplamento Fraco

- Problema
  - Como favorecer a baixa dependência, o pequeno impacto à mudanças e aumentar a reutilização?
- Solução
  - Atribuir a responsabilidade de modo que o acoplamento (medida de dependência entre classes) permaneça fraco.
- Exemplo
  - Quem deve ser responsável por criar um Pagamento e associá-lo à Venda?

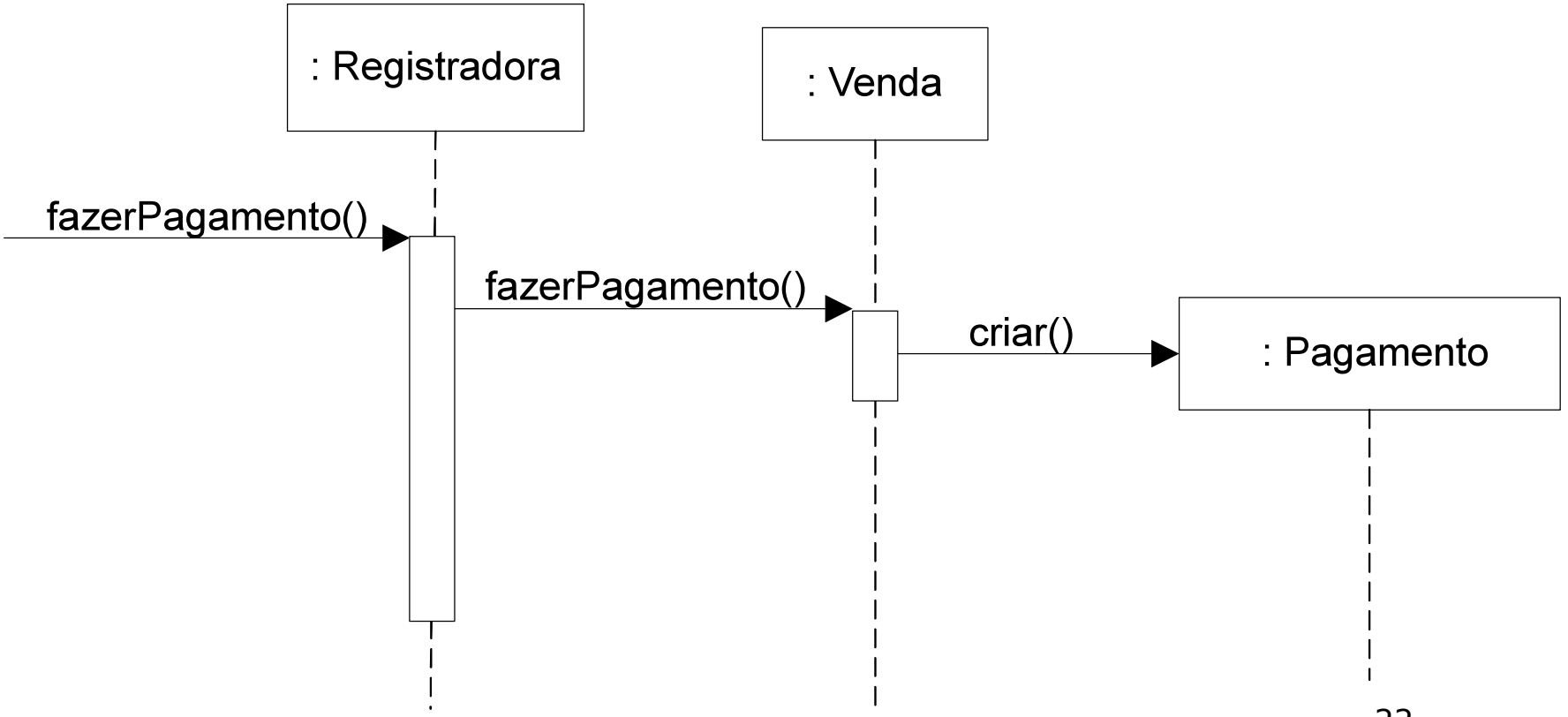
Exemplo: Acoplamento Fraco

- Pelo padrão Criador, a classe *Registradora* seria indicada



Exemplo: Acoplamento Fraco

- Para reduzir o acoplamento, *Venda* deve criar *Pagamento*



Acoplamento em OO

- O *TipoX* tem um atributo (membro de dados ou variável de instância) que referencia uma instância do *TipoY* ou o próprio *TipoY*.
- Um objeto do *TipoX* chama os serviços de um objeto do *TipoY*.
- O *TipoX* tem um método que referencia uma instância do *TipoY*, ou o próprio *TipoY*, de alguma forma. Isso normalmente inclui um parâmetro ou variável local de *TipoY*, ou então o objeto retornado por uma mensagem pode ser uma instância do *TipoY*.
- O *TipoX* é uma subclasse direta ou indireta do *TipoY*.
- O *TipoY* é uma interface e o *TipoX* implementa essa interface.

Acoplamento Fraco

Benefícios

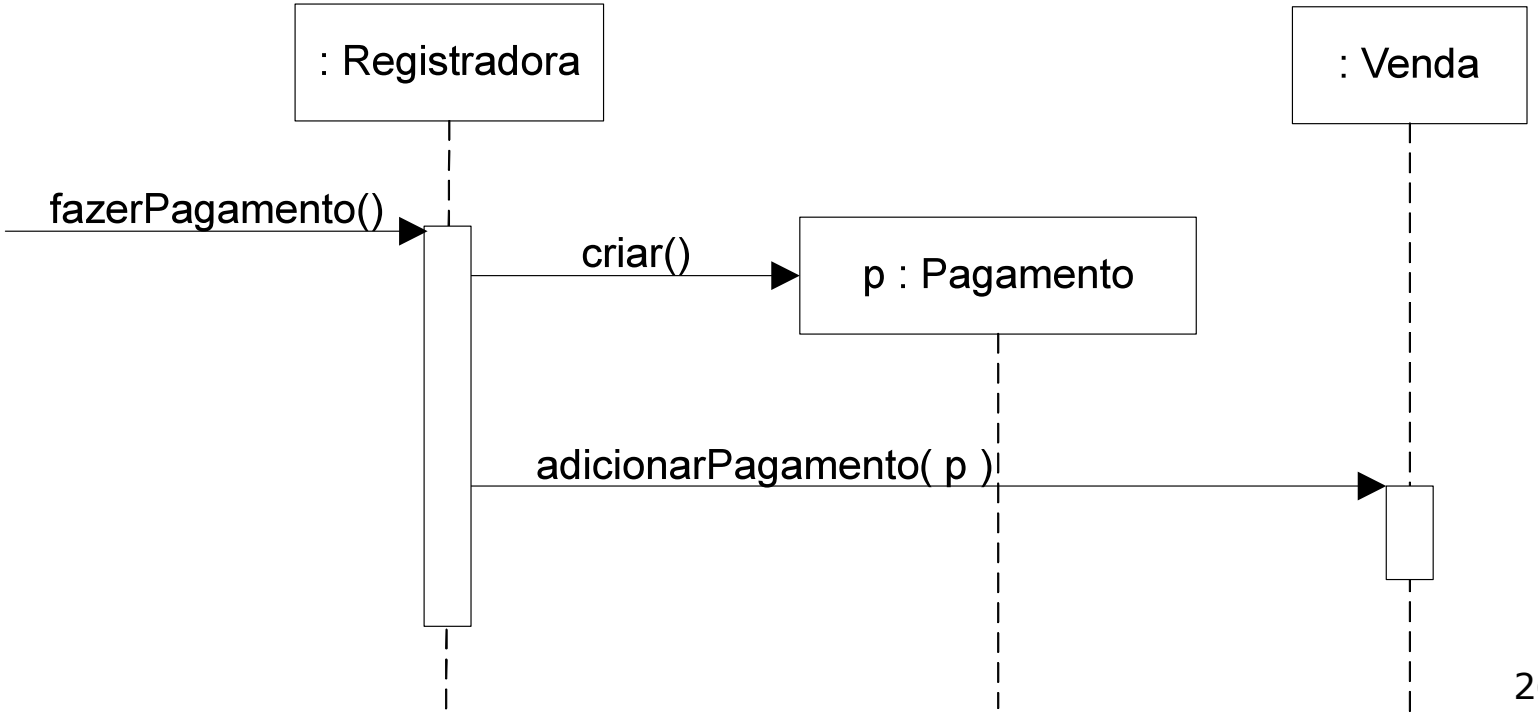
- Responsabilidade de uma classe é pouco afetada por mudanças em outros componentes.
- A responsabilidade de uma classe é mais simples de entender isoladamente.
- Aumenta a chance de reutilização das classes.

Coesão Alta

- Problema
  - Como manter a complexidade (das classes) em um nível "controlável"?
- Solução
  - Atribuir a responsabilidade de modo que a coesão permaneça alta
    - coesão = medida do relacionamento entre as responsabilidades de uma classe
- Exemplo
  - Quem deve ser responsável por criar um Pagamento e associá-lo à Venda?

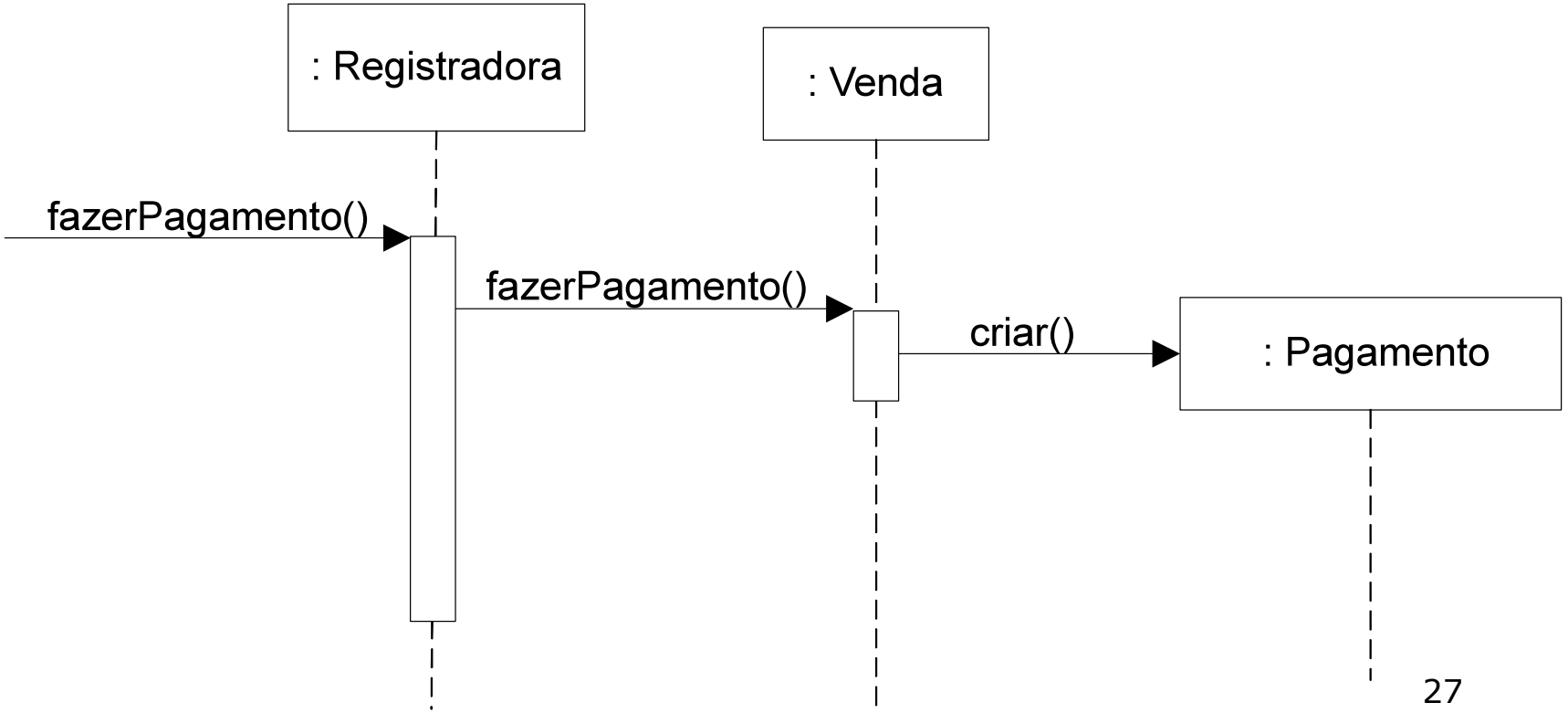
Exemplo – Coesão Alta

- Pelo padrão Criador, seria *Registradora*. Mas se *Registradora* for responsável pela maioria das operações do sistema, ela vai ficar cada vez mais sobrecarredada e incoesa.



Exemplo – Coesão Alta

- A criação de *Pagamento* deve ser delegada a *Venda* para favorecer uma coesão alta e um acoplamento fraco



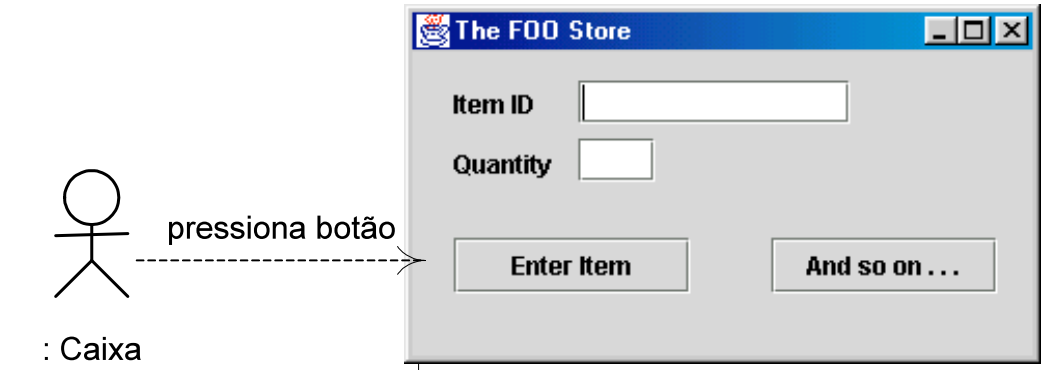
Coesão Alta

- Benefícios
  - Mais clareza e compreensão no projeto
  - Simplificação de manutenção
  - Favorecimento do acoplamento fraco
  - Aumento do potencial de reutilização

Controlador

- Problema
  - Quem deve ser responsável por tratar um evento do sistema?
- Solução
  - Atribuir a responsabilidade de tratar um evento do sistema a uma classe "controladora" que represente:
    - o sistema como um todo (*façade controller*)
    - o negócio ou organização com um todo (*façade controller*)
    - uma coisa ou papel de uma pessoa do mundo real envolvida diretamente com a tarefa (*role controller*)
    - um "tratador" (*handler*) artificial para todos os eventos de um caso de uso (*use-case controller*)

Exemplo - Controlador



actionPerformed(actionEvent)

Camada de Interface

:JFrameVenda

entrarItem(itemID, qty)

mensagem de evento do sistema

Camada de Domínio

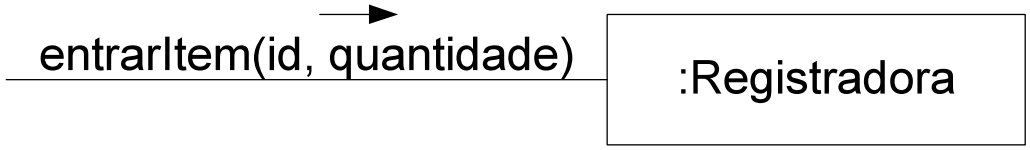
: ???

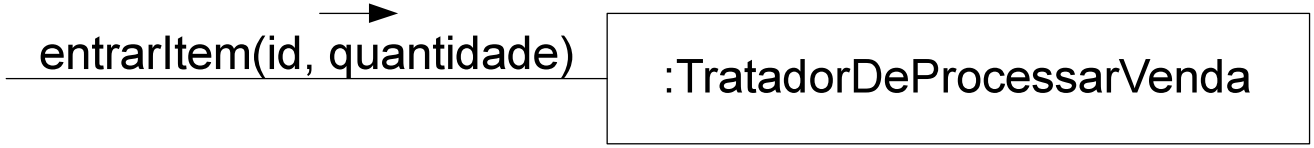
Que classe de objeto deve ser responsável por receber a mensagem de evento do sistema?
Essa classe é muitas vezes chamada de controlador ou coordenador. Ela normalmente não faz o trabalho, mas o delega a outros objetos.
O controlador é um tipo de "fachada" entre a camada de domínio e a camada de interface.

Quem deve ser responsável pelos eventos do sistema, tais como *entrarItem*, *terminarVenda*, *IniciarNovaVenda()*, *fazerPagamento()*

Exemplo - Controlador

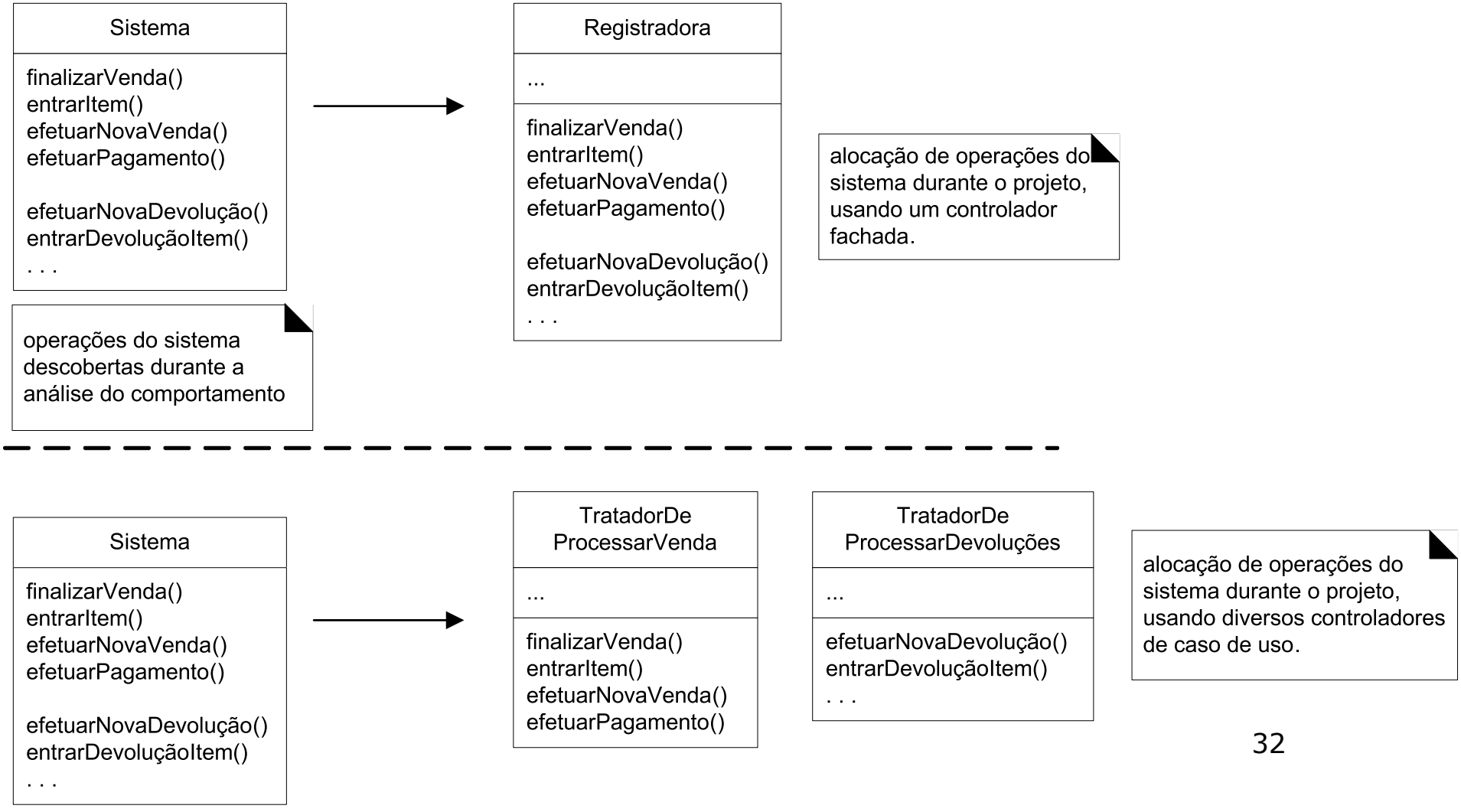
- De acordo com o padrão Controlador
  - *Registradora* pode ser um controlador representando o sistema todo (*façade Controller*)
  - *TratadorDeProcessarVenda* pode ser um controlador representando um receptor ou tratador de todos os eventos de sistema de um caso de uso (*use-case controller*)





Exemplo - Controlador

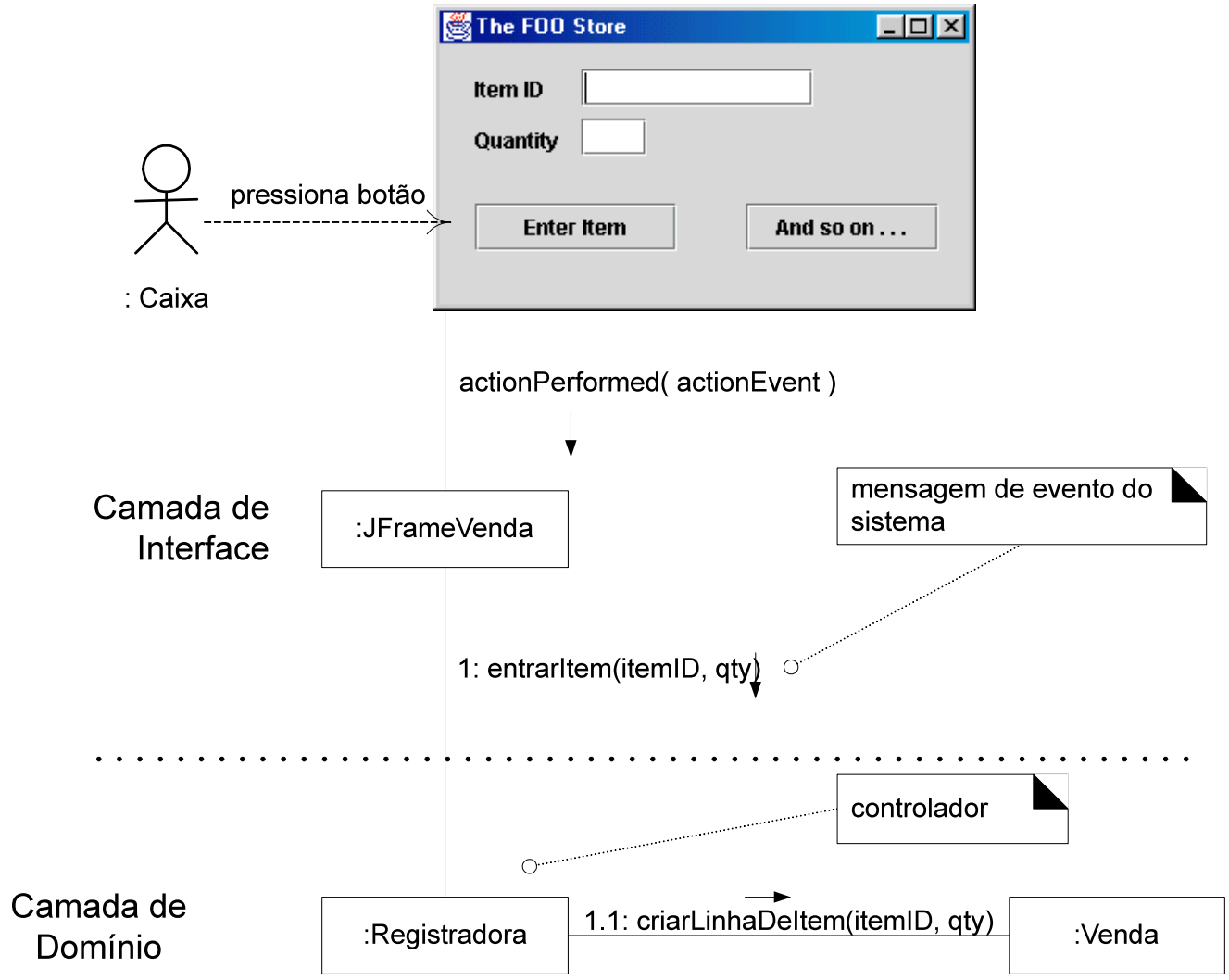
- Alocação de operações do sistema



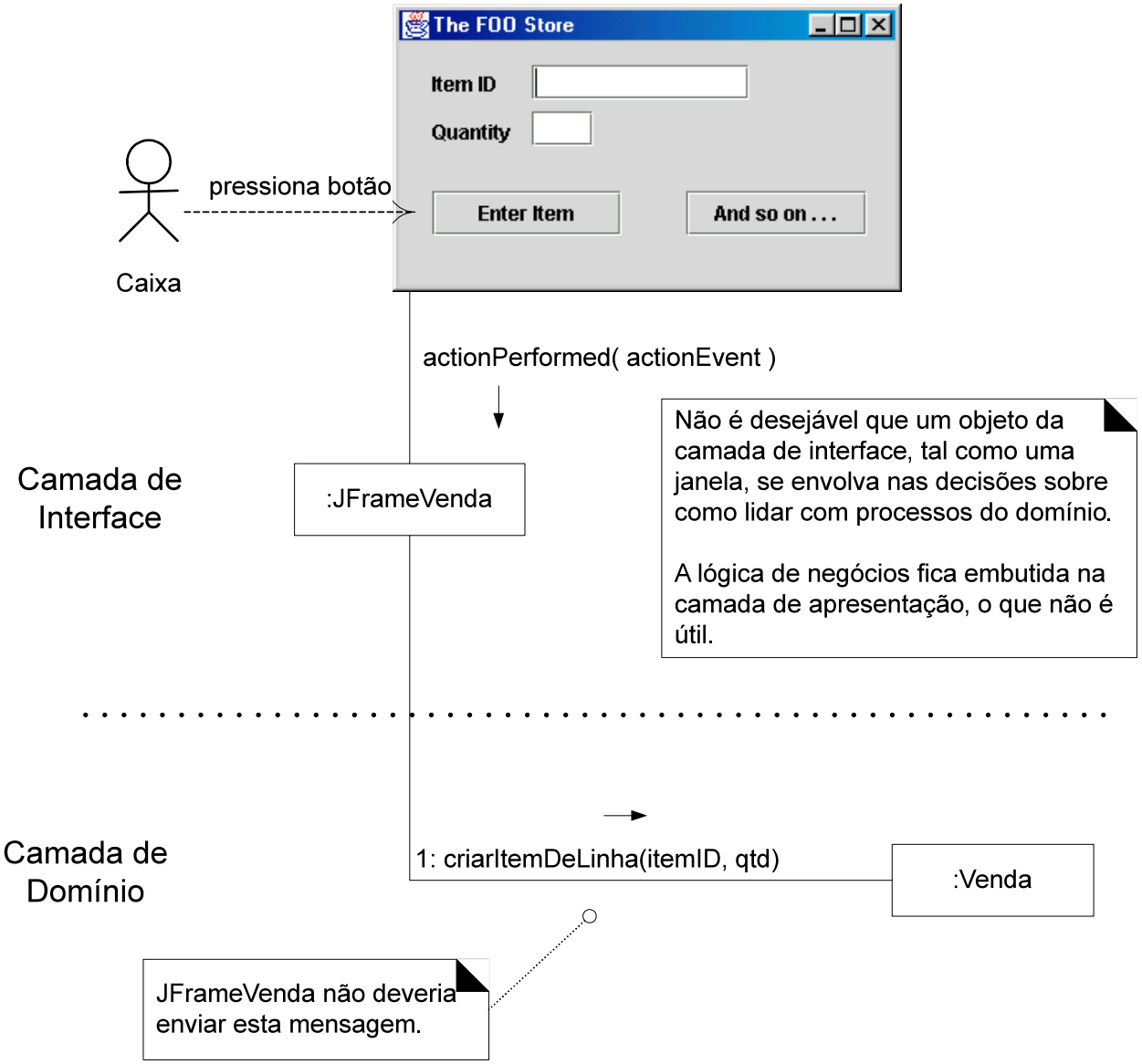
Exemplo - Controlador

- Qual solução é melhor?
  - *Controlador Fachada*
    - são adequados quando não existem muitos eventos de sistema
      - muitos eventos podem levar a um controlador inchado, de baixa coesão e alto acoplamento
  - *Controlador de casos de uso*
    - são adequados quando
      - o sistema possui muitos eventos com diferentes processos
      - é necessário conhecer o estado de um caso de uso para identificar eventos fora de sequência

Exemplo - controlador fachada



Contra-exemplo



Benefícios - Controlador

- aumento das possibilidades de reutilização e de interfaces *plugáveis*
  - garante que a lógica da aplicação não seja tratada na camada de interface
- conhecer o estado do caso de uso
  - garantir que as operações do sistema ocorram em uma sequência válida.
    - Exemplo: terminarVenda deve preceder fazer fazerPagamento

Controladores inchados

- Sinais
  - Uma única classe recebe muitos ou todos eventos de sistema
  - o controlador executa muitas das tarefas necessárias para atender ao evento de sistema, sem delegar o trabalho
  - o controlador tem muitos atributos, e mantém informações sobre o sistema ou domínio que devem ser distribuídas para outros objetos
- Soluções
  - acrescentar mais controladores
  - projetar o controlador de forma que ele delegue o atendimento das responsabilidades de cada operação de sistema a outros objetos

Exemplos de código

- Exemplo do código de controlador usando Java swing - pág 324
- Exemplo de código de controlador web usando Java Struts - pág 325

Regra da substituição

- ***Regra da substituição***: seja a classe A uma generalização de outra B. Não pode haver diferenças entre utilizar instâncias de B ou de A, do ponto de vista dos usuários de A.
  - Ou seja, é inadequado o uso de generalização onde nem todas as propriedades da superclasse fazem sentido para a subclasse.

Herança de operações e polimorfismo

- Uma subclasse herda todas as propriedades de sua superclasse que tenham visibilidade pública ou protegida.
- Entretanto, pode ser que o comportamento de alguma operação herdada seja diferente para a subclasse.
  - Nesse caso, a subclasse deve redefinir o comportamento da operação.
  - A assinatura da operação pode ser reutilizada.
  - A implementação da operação (***método***) é diferente.

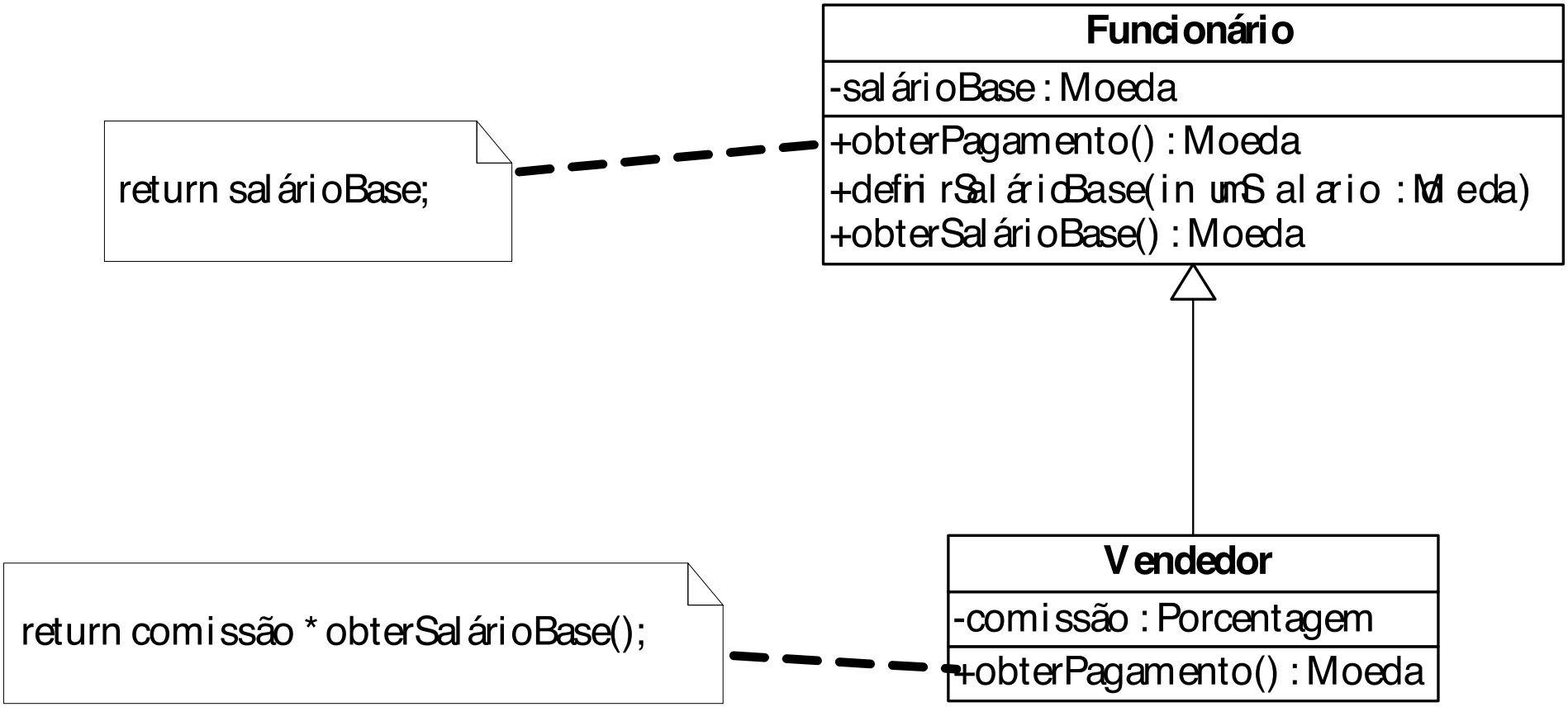
Operações polimórficas

- ***Operações polimórficas*** são operações de mesma assinatura definidas em diversos níveis de uma hierarquia de generalização e que possuem comportamento diferente.
  - assinatura é repetida na(s) subclasse(s) para enfatizar a redefinição de implementação.
- Operações polimórficas implementam o ***princípio do polimorfismo*** no qual duas ou mais classes respondem a mesma mensagem de formas diferentes.
- Objetivo: garantir que as subclasses tenham uma interface em comum.

Operações polimórficas

- Operações polimórficas facilitam a implementação.
  - Se duas ou mais subclasses implementam a mesma operação polimórfica, a mensagem a ser passada é a mesma para todas elas.
  - O remetente da mensagem não precisa saber qual a verdadeira classe de cada objeto, pois eles aceitam a mesma mensagem.
  - A diferença é que o *método* que implementa a operação é diferente em cada classe.

Exemplo (Operações polimórficas)

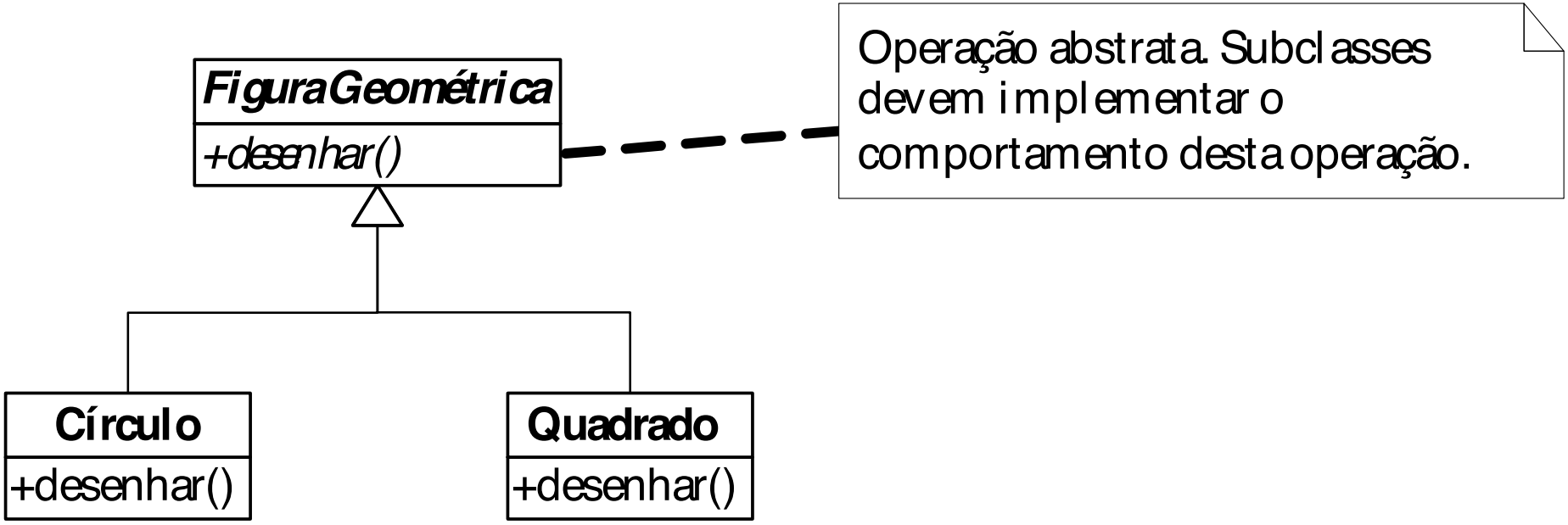


Operações abstratas e polimorfismo

- Em termos de operações, uma classe é abstrata quando ela possui pelo menos uma ***operação abstrata***.
- Uma operação abstrata não possui implementação.
  - Uma classe pode possuir tanto operações abstratas quanto operações concretas.
  - Uma classe que possui pelo menos uma operação abstrata é abstrata.
- Uma subclasse que herda uma operação abstrata e não fornece uma implementação é ela própria abstrata.

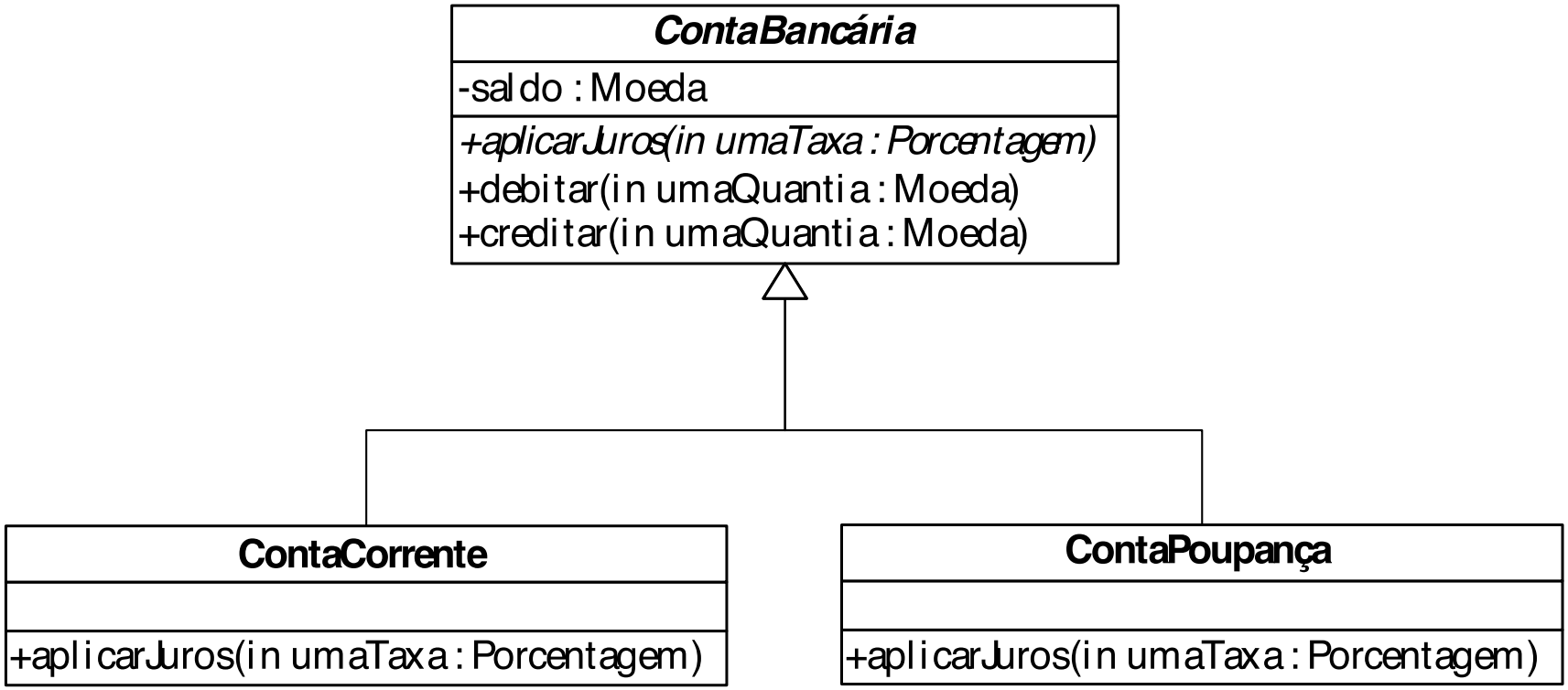
Operações abstratas e polimorfismo

- As classes Círculo e Quadrado são concretas, pois fornecem implementação para a operação abstrata herdada.



Operações abstratas e polimorfismo

- Classes **ContaCorrente** e **ContaPoupança** redefinem a operação **aplicarJuros**.

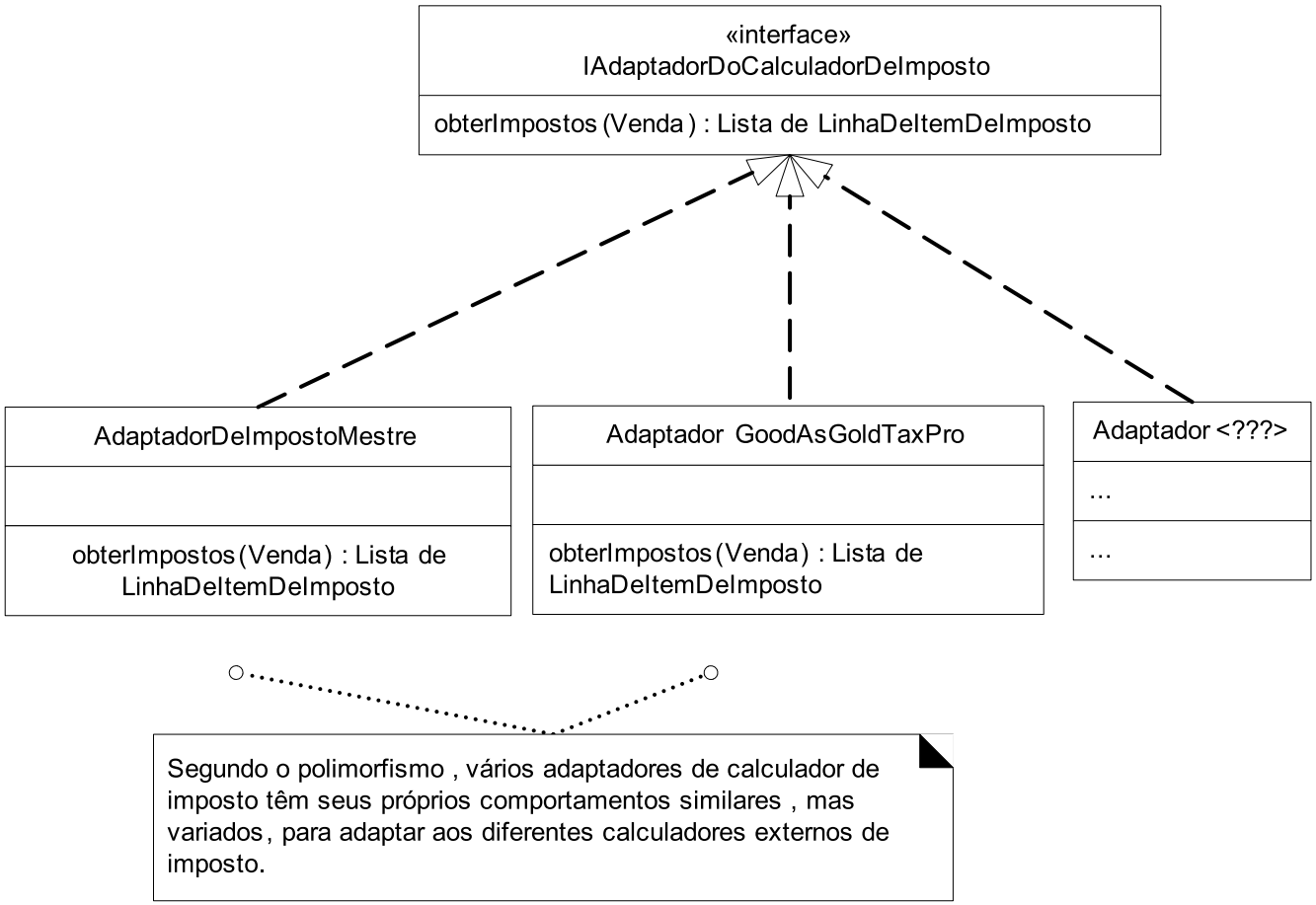


Padrão GRASP: Polimorfismo

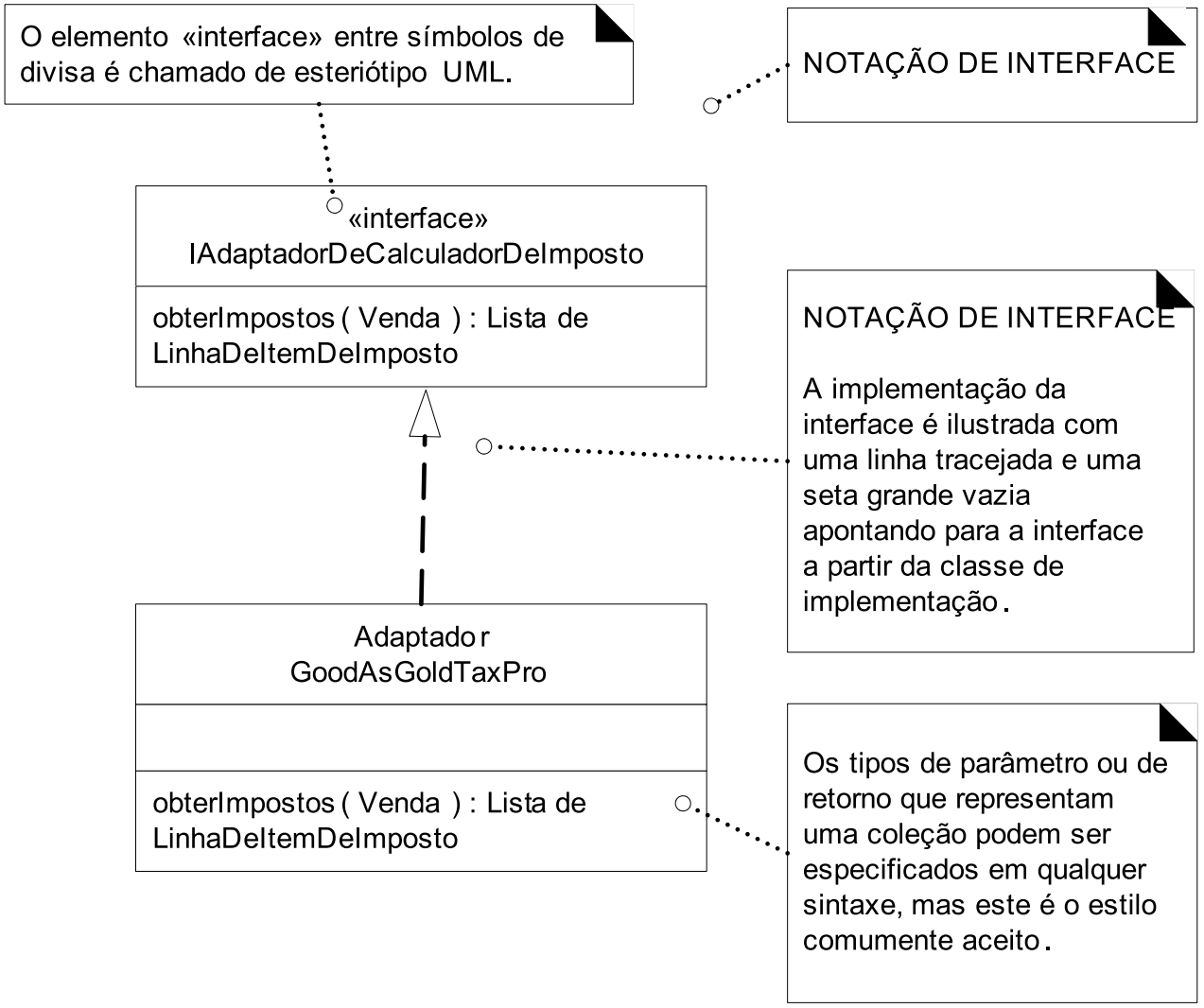
- Problema:
  - Como tratar alternativas com base no tipo?
  - Como criar componentes de software interligáveis?
- Solução:
  - Quando alternativas ou comportamentos relacionados variam segundo o tipo (classe), deve-se atribuir a responsabilidade pelo comportamento (usando operações polimórficas) aos tipos para os quais o comportamento varia.

Exemplo

- Polimorfismo na adaptação de diferentes calculadores externos de imposto



Notação UML para interfaces



Vantagens e contra-indicações

- Vantagens
  - as extensões exigidas para as novas variações são fáceis de adicionar
  - novas implementações podem ser introduzidas sem afetar os clientes
- Contra-indicações
  - não usar polimorfismo para adicionar uma flexibilidade para uma possível futura variação
    - o esforço pode não compensar

Padrão GRASP Invenção Pura

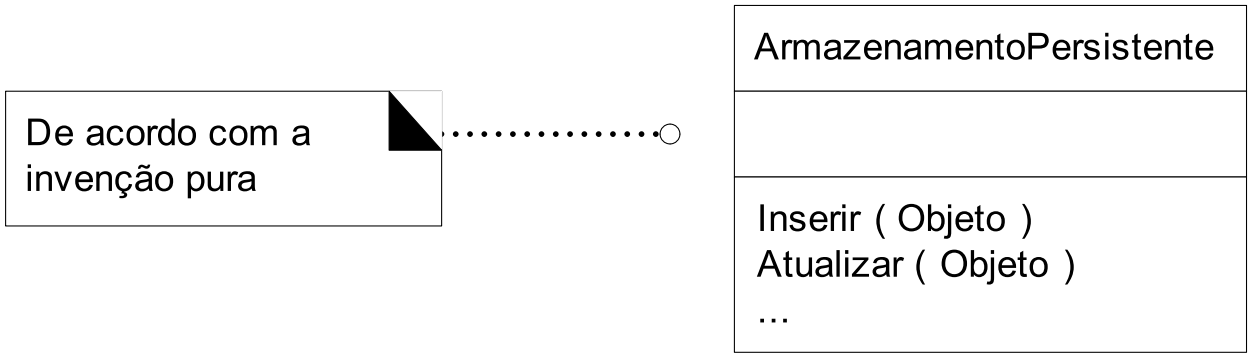
- Problema:
  - Às vezes, durante o projeto é preciso atribuir responsabilidades que não são encaixam naturalmente em nenhuma das classes conceituais
- Solução:
  - Criar uma classe artificial que não representa nenhuma entidade no domínio do problema

Exemplo: problema

- Suporte para salvar as intâncias de *Venda* em um banco de dados relacional
  - Segundo o especialista, atribuir esta responsabilidade à *Venda* é justificável
    - Venda tem os dados que precisam ser salvos
  - Porém, a tarefa exige inúmeras operações de suporte relacionadas ao banco de dados
    - *Venda* pode se tornar não coesa
    - *Venda* precisa estar acoplada à interface do banco de dados
    - Outras classes precisam do mesmo suporte ao serviço de salvar objetos em um banco de dados

Exemplo: solução

- Criar uma nova classe que seja responsável unicamente por salvar objetos em algum tipo de meio de armazenamento persistente



Vantagens e contra-indicações

- Vantagens
  - coesão alta é favorecida
  - potencial de reutilização pode aumentar devido à presença de classes de *Invenção Pura* refinadas
- Contra-indicações
  - uso extremo – funções simples se tornam objetos

Padrão GRASP Indireção

- Problema:
  - Reduzir o acoplamento direto com objetos que podem sofrer alterações
- Solução:
  - Usar um objeto intermediário para ser o mediador entre componentes para que eles não sejam diretamente acoplados
- Vantagem:
  - Acoplamento mais fraco entre os componentes

Exemplo:

- Indireção por meio do adaptador