

# PADRÕES GRASP

Dr<sup>a</sup>. Alana Morais

O QUE HOUE NA AULA PASSADA?



# O QUE É UM PADRÃO?

Maneira testada e documentada de alcançar um objetivo qualquer

- Padrões são comuns em várias áreas da engenharia

Design Patterns, ou Padrões de Projeto\*

- Padrões para alcançar objetivos na engenharia de software usando classes e métodos em linguagens orientadas a objeto
- Inspirado em "A Pattern Language" de Christopher Alexander, sobre padrões de arquitetura de cidades, casas e prédios

# ROTEIRO



General Responsibility  
Assignment Software Patterns!!!

# PADRÕES GRASP

Os padrões GRASP descrevem os princípios fundamentais para a atribuição de responsabilidades em projetos OO

- Responsabilidades:

- Fazer algo (a si mesmo/a outros objetos )
- Conhecer/lembrar de algo (dados encapsulados / objetos relacionados )

- Responsabilidade != método

- Métodos implementam responsabilidades
  - Objetos colaboram para cumprir responsabilidades

# RESPONSABILIDADES

Booch e Rumbaugh “Responsabilidade é um contrato ou obrigação de um tipo ou classe.”

Dois tipos de responsabilidades dos objetos:

- De conhecimento (*knowing*)
  - Estão relacionadas à distribuição das características do sistema entre as classes
- De realização (*doing*)
  - Estão relacionadas com a distribuição do comportamento do sistema entre as classes

Responsabilidades são atribuídas aos objetos durante o planejamento

# PADRÕES GRASP

- Fornecem uma abordagem sistemática para a atribuição de responsabilidades às classes do projeto
- Padrões de análise catalogados por Craig Larman.
- Indicam como atribuir responsabilidades a classes da melhor forma possível.
- Úteis na construção de
  - ✓ diagramas de interações (atividade, sequência)
  - ✓ diagramas de classes

# PADRÕES GRASP

## Padrões básicos:

- Information Expert
- Creator
- Low Coupling
- High Cohesion
- Controller

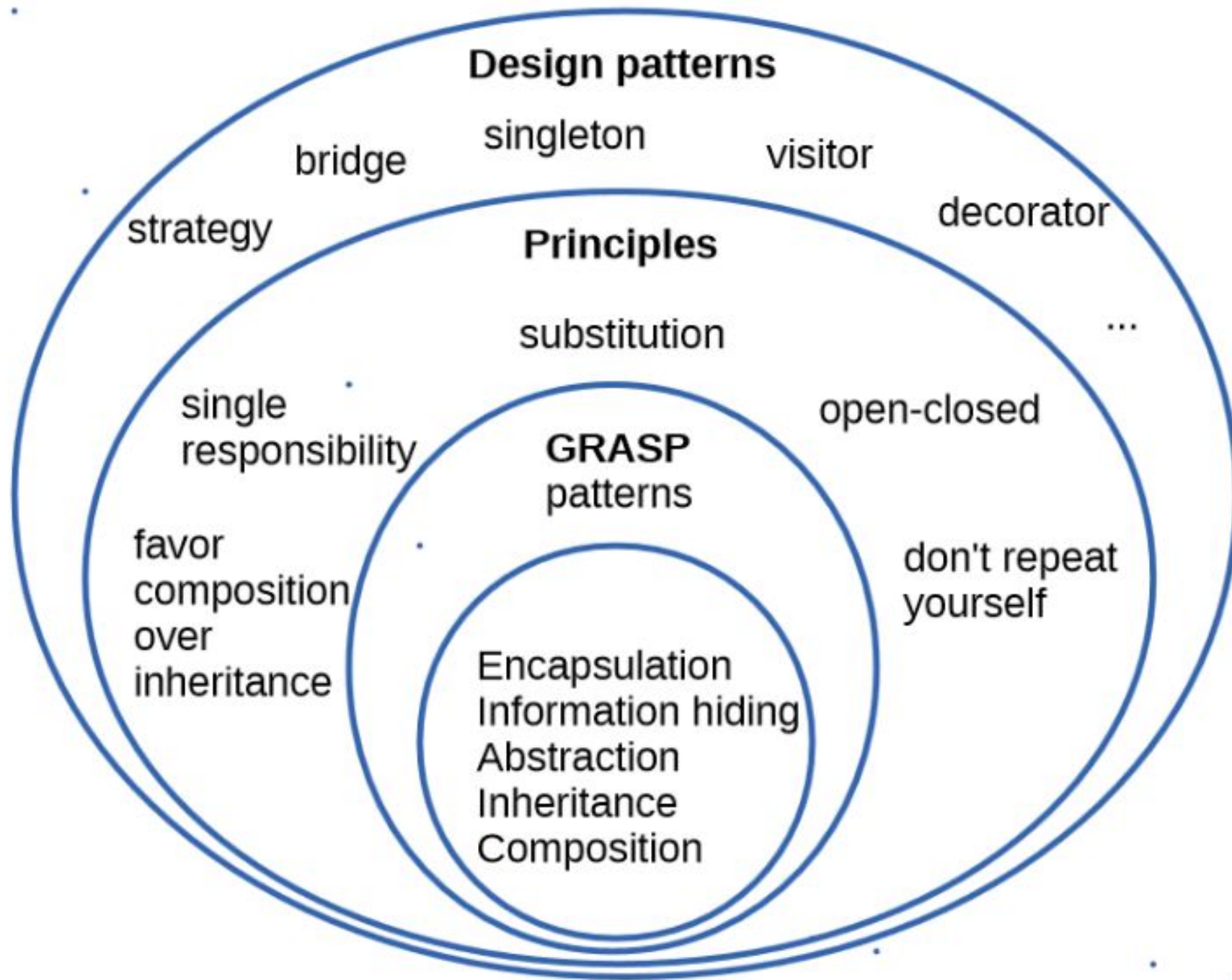
## Padrões avançados

- Polymorphism
- Pure Fabrication
- Indirection
- Protected Variations



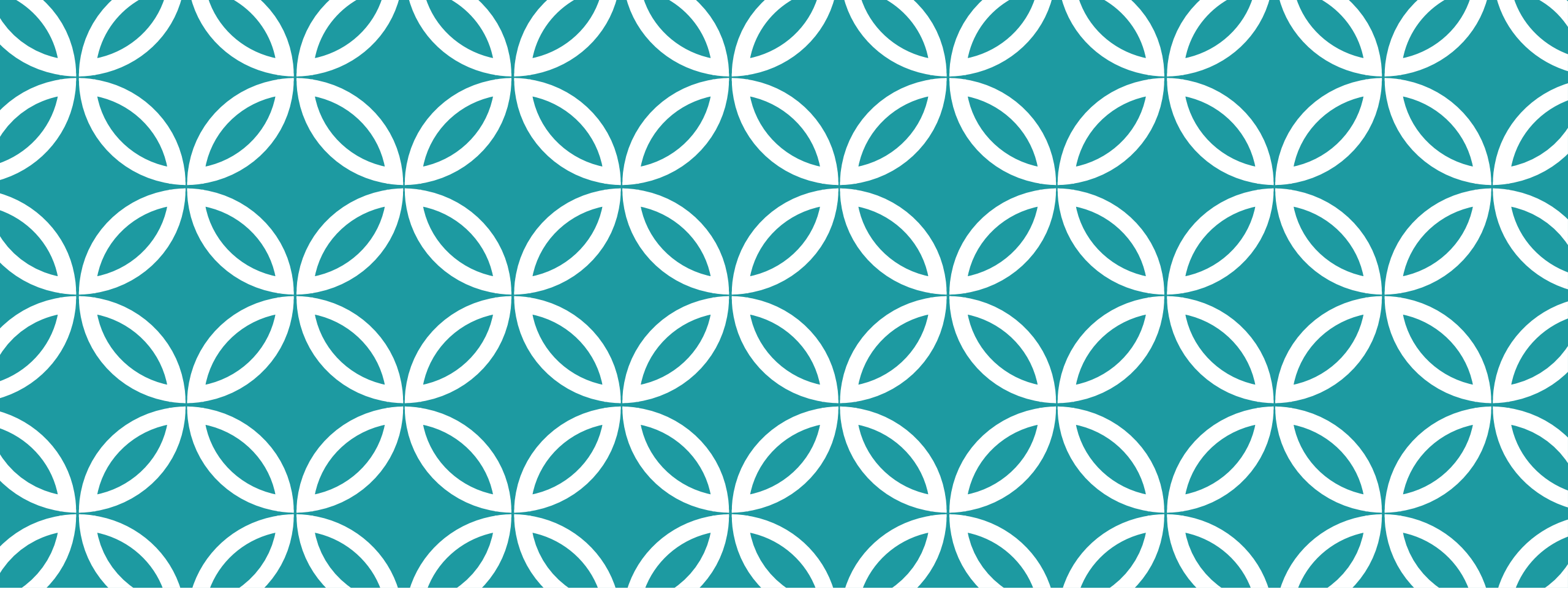
# PADRÕES GRASP

Padrão	Princípio
Information Expert	“Não se repita.”
Creator	Substituição Liskov
High Cohesion	Responsabilidade Simples.
Low Coupling	Informação escondida.
Controller	“Aberta para extensão e fechada para modificações.”
Polymorphism	“Programa uma interface, não uma implementação.”
Pure Fabrication	“À favor da composição ao invés da herança;”



# PADRÕES GRASP

(CESPE – Analista de Informática – MPU 2010) GRASP (*general responsibility assignment software patterns*) consiste em um conjunto de sete padrões básicos para atribuir responsabilidades em projeto orientado a objetos: *information expert, creator, controller, low coupling, high cohesion, polymorphism* e *pure fabrication*.



# INFORMATION EXPERT

○ Especialista

# INFORMATION EXPERT

**Problema:** dado um comportamento (responsabilidade) a qual classe essa responsabilidade deve ser alocada?

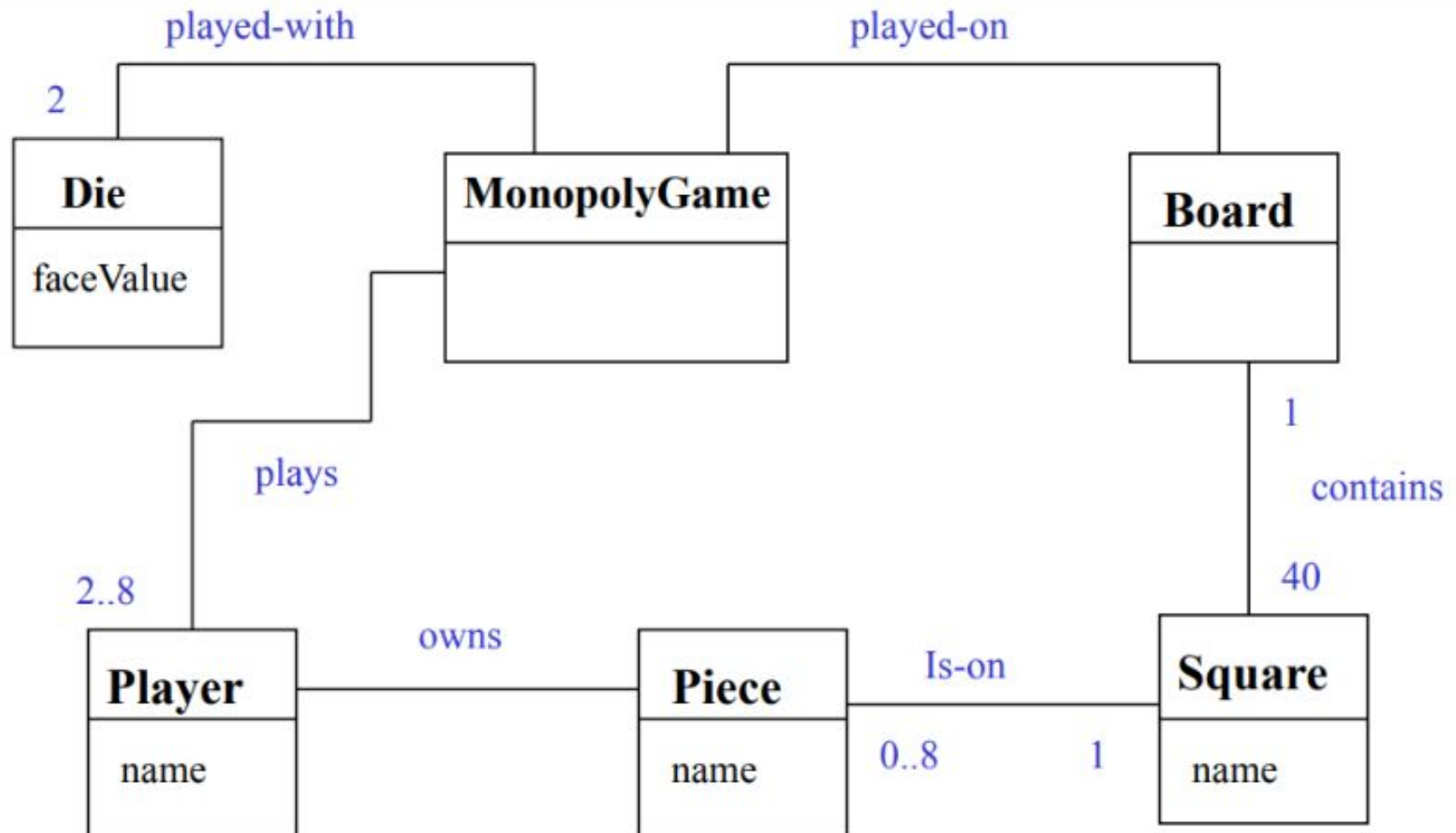
**Solução:** atribuir essa responsabilidade ao especialista da informação — a classe que tem a informação necessária para satisfazer a responsabilidade.

É o padrão mais usado para atribuir responsabilidades





# EXEMPLO – BANCO IMOBILIÁRIO



# INFORMATION EXPERT

## EXEMPLO: BANCO IMOBILIÁRIO

Quem deve localizar uma posição do tabuleiro dada a sua identidade?

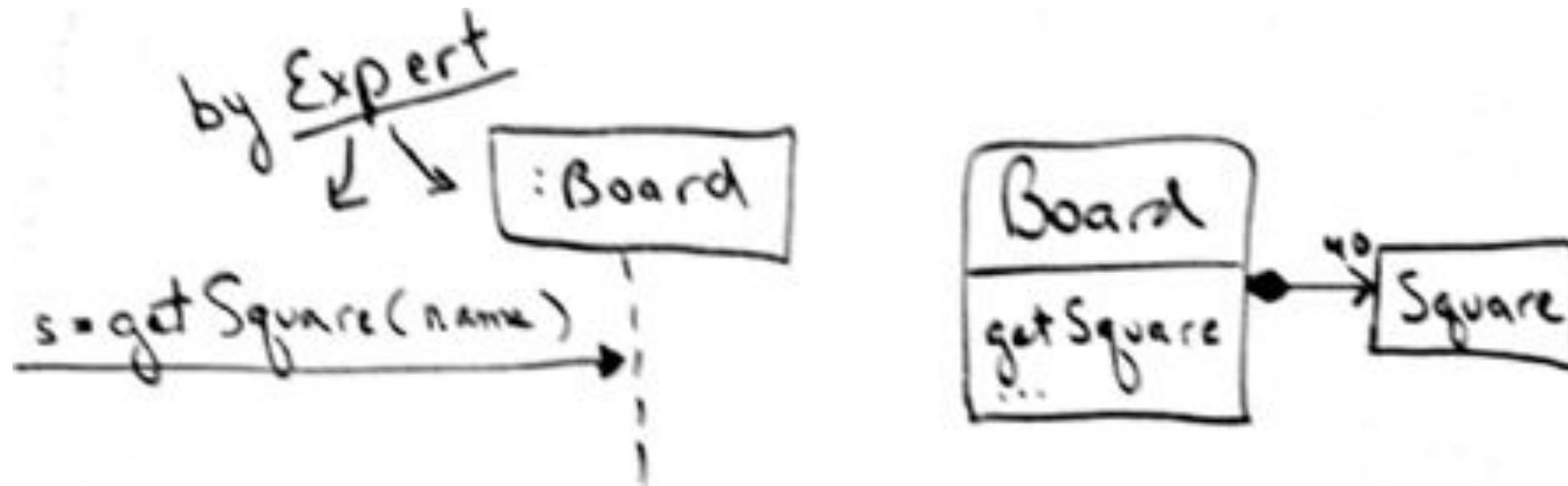




# INFORMATION EXPERT

## EXEMPLO: BANCO IMOBILIÁRIO

Quem deve localizar uma posição do tabuleiro dada a sua identidade?



# INFORMATION EXPERT

## EXEMPLO: O PONTO DE VENDA

Considere o exemplo de um sistema de vendas e pagamentos.

- Padrões GRASP são utilizados quase sempre durante a etapa de análise.

Sistema:

- Grava vendas e gera pagamentos

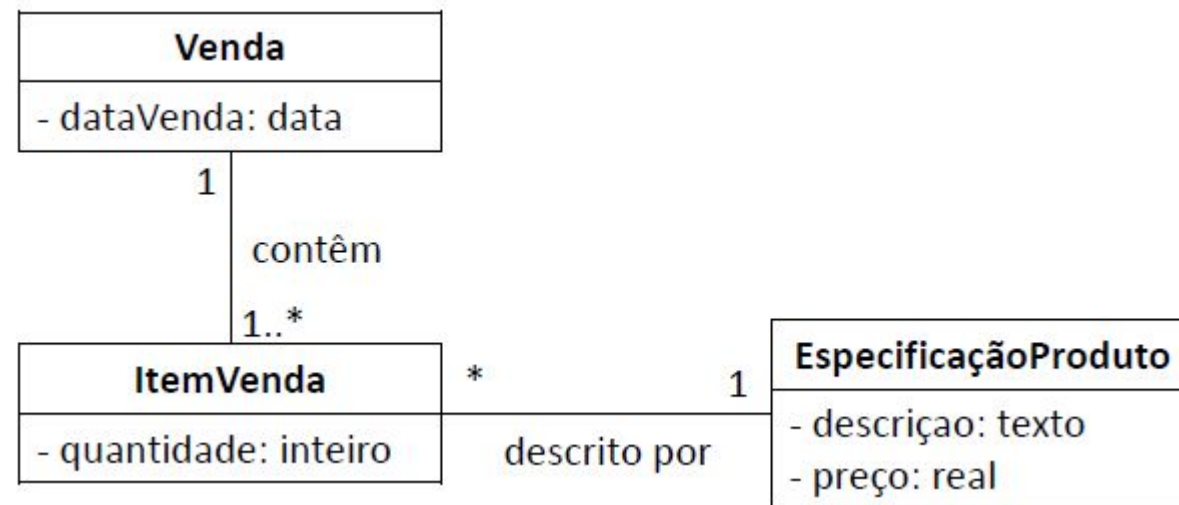


# INFORMATION EXPERT

## EXEMPLO: O PONTO DE VENDA

Caso de uso registrar venda, foi identificada a responsabilidade do sistema gerar o total da venda.

Quem deve ser responsável por conhecer o total da venda?



# INFORMATION EXPERT

## EXEMPLO: O PONTO DE VENDA

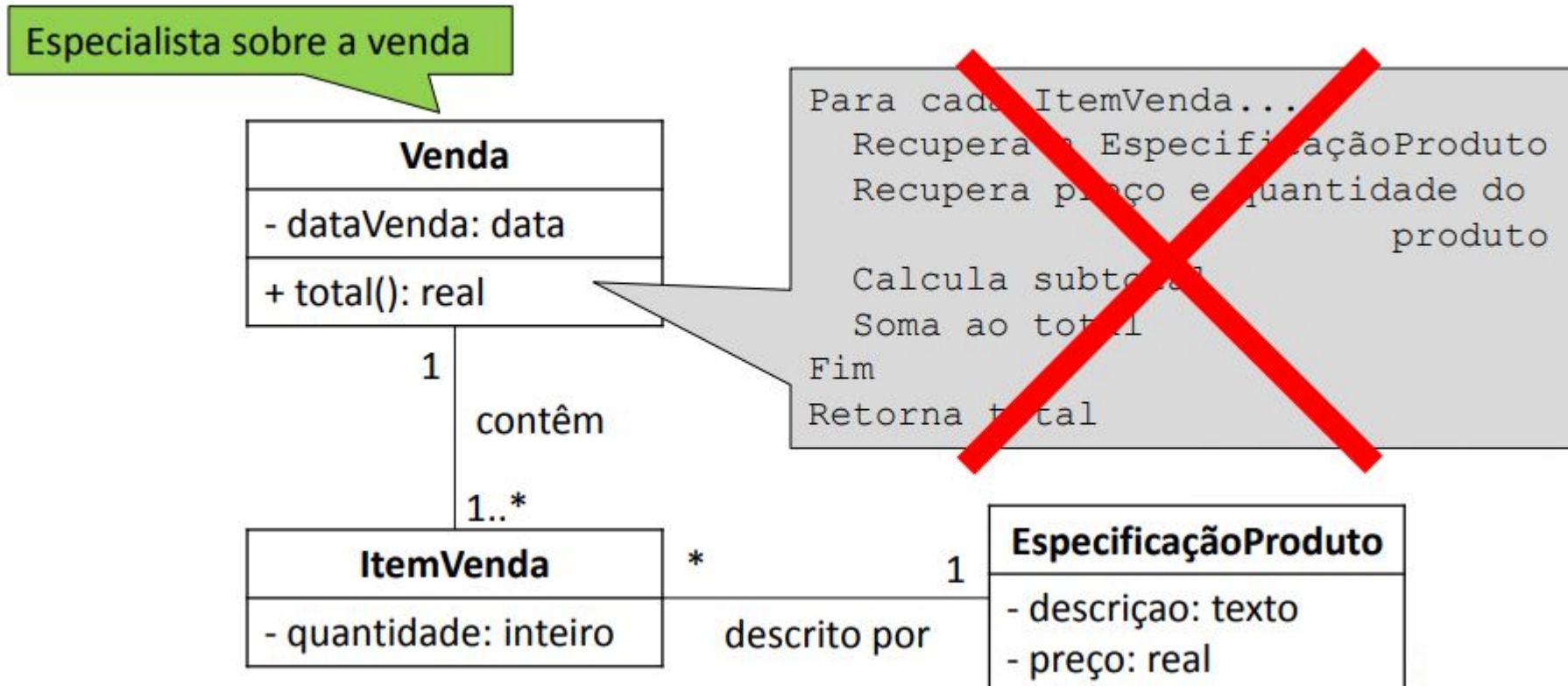
Caso de uso registrar venda, foi identificada a responsabilidade do sistema gerar o total da venda.

Quem deve ser responsável por conhecer o total da venda?



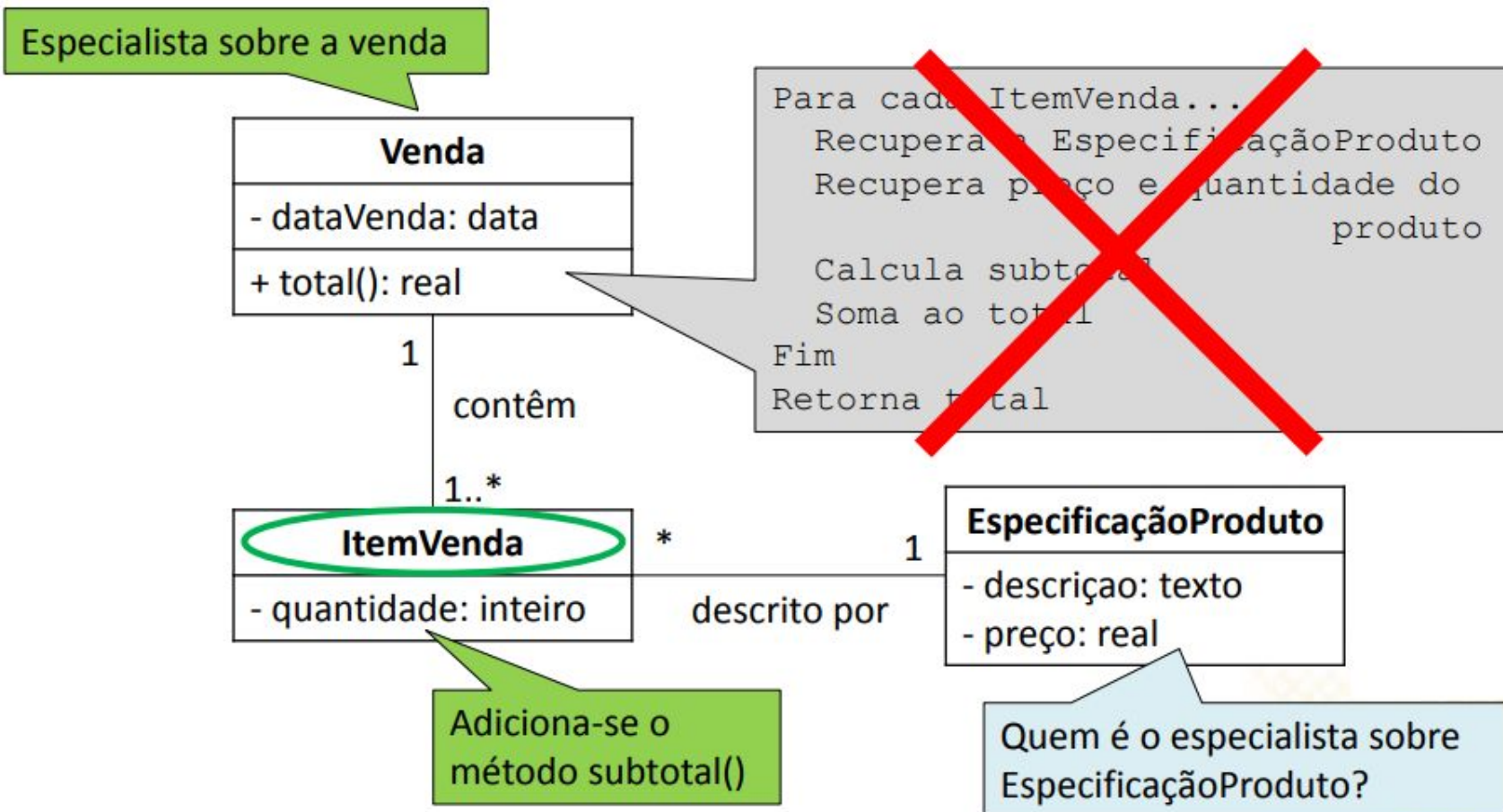
# INFORMATION EXPERT

## EXEMPLO: O PONTO DE VENDA



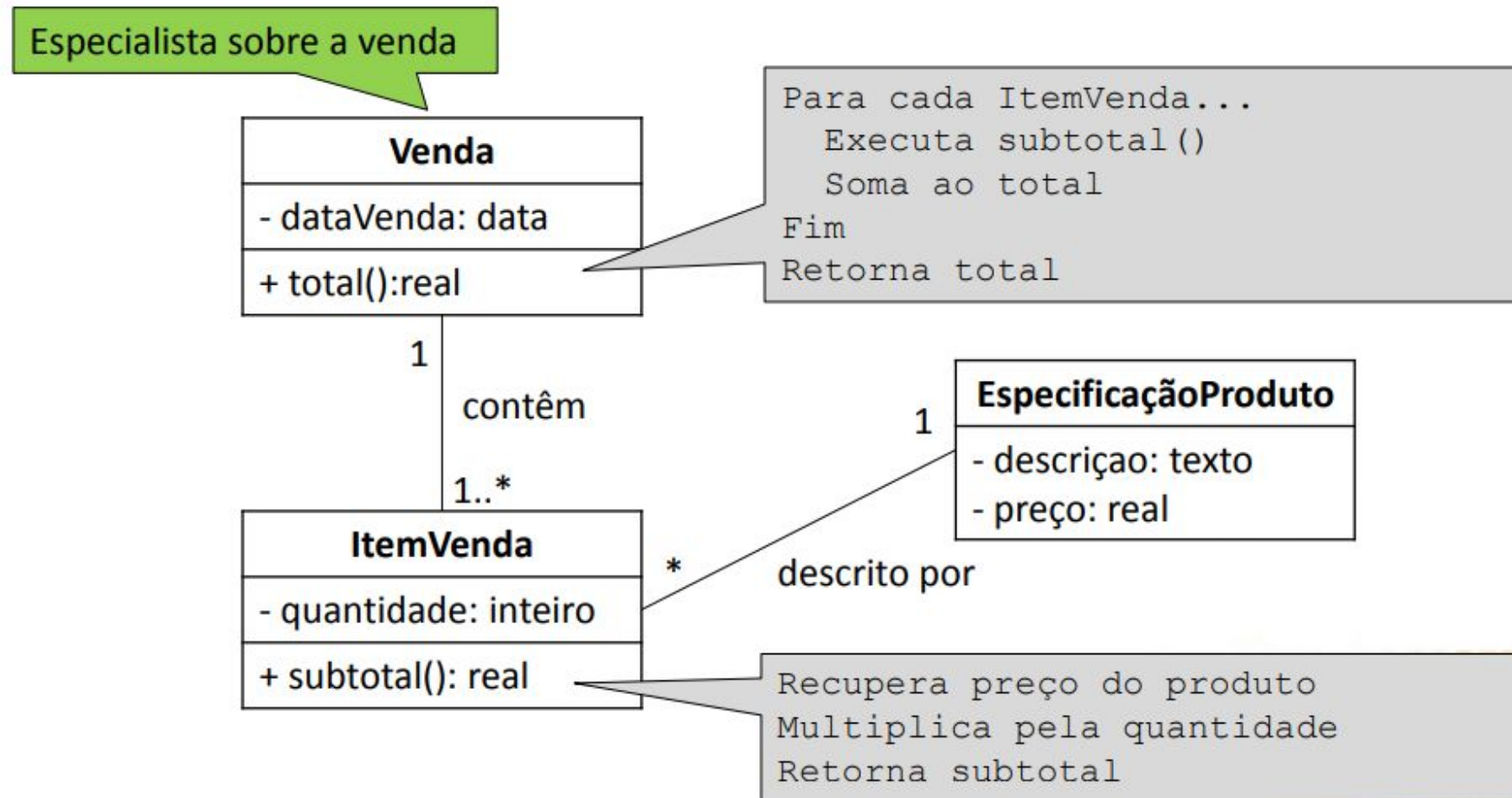
# INFORMATION EXPERT

## EXEMPLO: O PONTO DE VENDA



# INFORMATION EXPERT

## EXEMPLO: O PONTO DE VENDA



# INFORMATION EXPERT

## EXEMPLO: O PONTO DE VENDA

Simple demais!!!

- Mas errar gera problemas...

Imagine que apenas Venda fosse definida como especialista sobre os itens

O primeiro algoritmo seria válido, certo?



# INFORMATION EXPERT

## EXEMPLO: O PONTO DE VENDA

Um belo dia, o seu patrão lhe diz que o cálculo para cada item de venda foi alterado.

- .Quantidade  $> 100$
- .Desconto de até 30% no valor do item de venda
- .Detalhe: depende do item de venda!

Você acha esse cenário difícil de acontecer ?

- Claro que não é! É até bem trivial!!!

# INFORMATION EXPERT

## EXEMPLO: O PONTO DE VENDA

### Consideração óbvia de OO

- A porcentagem é um atributo da especificação do produto.

### Vamos implementar a solução Expert!!!

- Quantas classes foram modificadas???

# INFORMATION EXPERT

## **Consequências**

- Encapsulamento é mantido
- Acoplamento fraco, facilidade de manutenção
- Alta coesão – Objetos fazem tudo relacionado à sua própria informação

# INFORMATION EXPERT

- É intuitivo – mas você tem que entender a intuição
- No fim, vários especialistas “parciais” podem colaborar
- Com especialista, vários objetos inanimados tem ações (isso é OO)

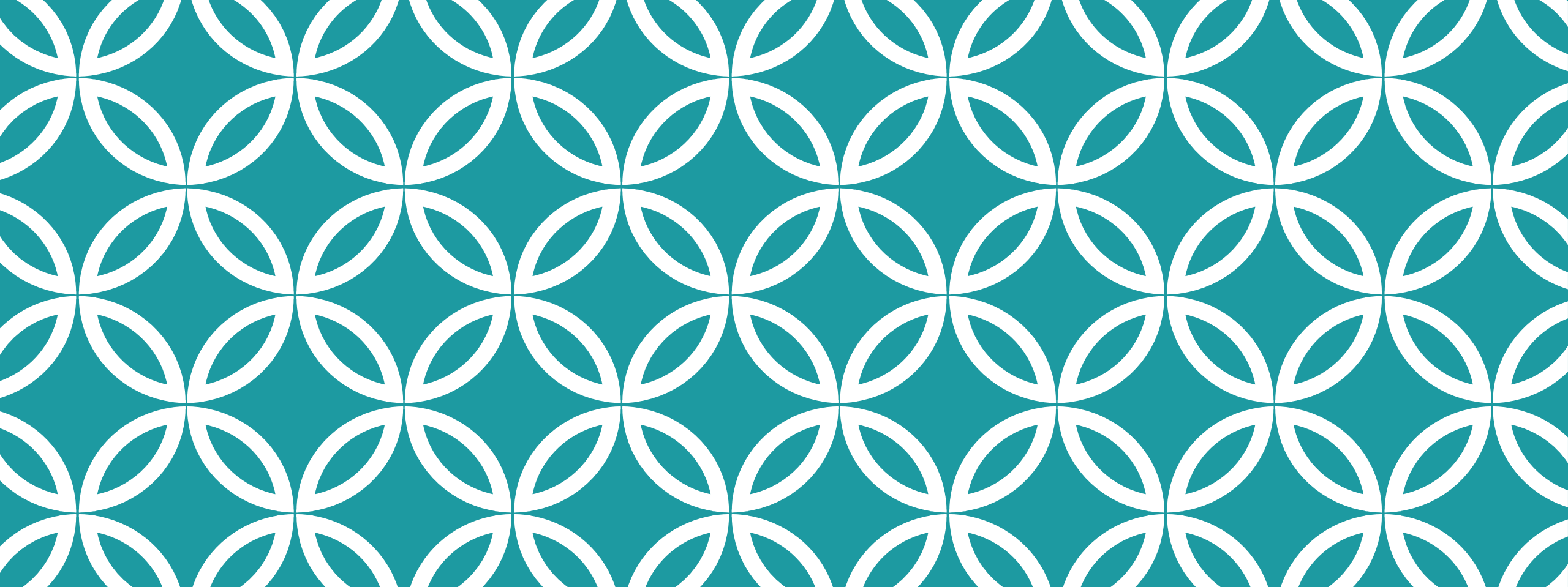
**Há contraindicações, claro!**

# EXERCÍCIO

Onde o padrão Expert está sendo furado no Projeto ExercícioE ?

Melhore o programa, aplicando o padrão Information Expert.





# CREATOR

O criador

# CREATOR

**Problema:** quem deve ser o responsável por criar (novas) instâncias de uma determinada classe? ”

**Solução:** Atribuir a uma classe a responsabilidade de criar a classe X se:

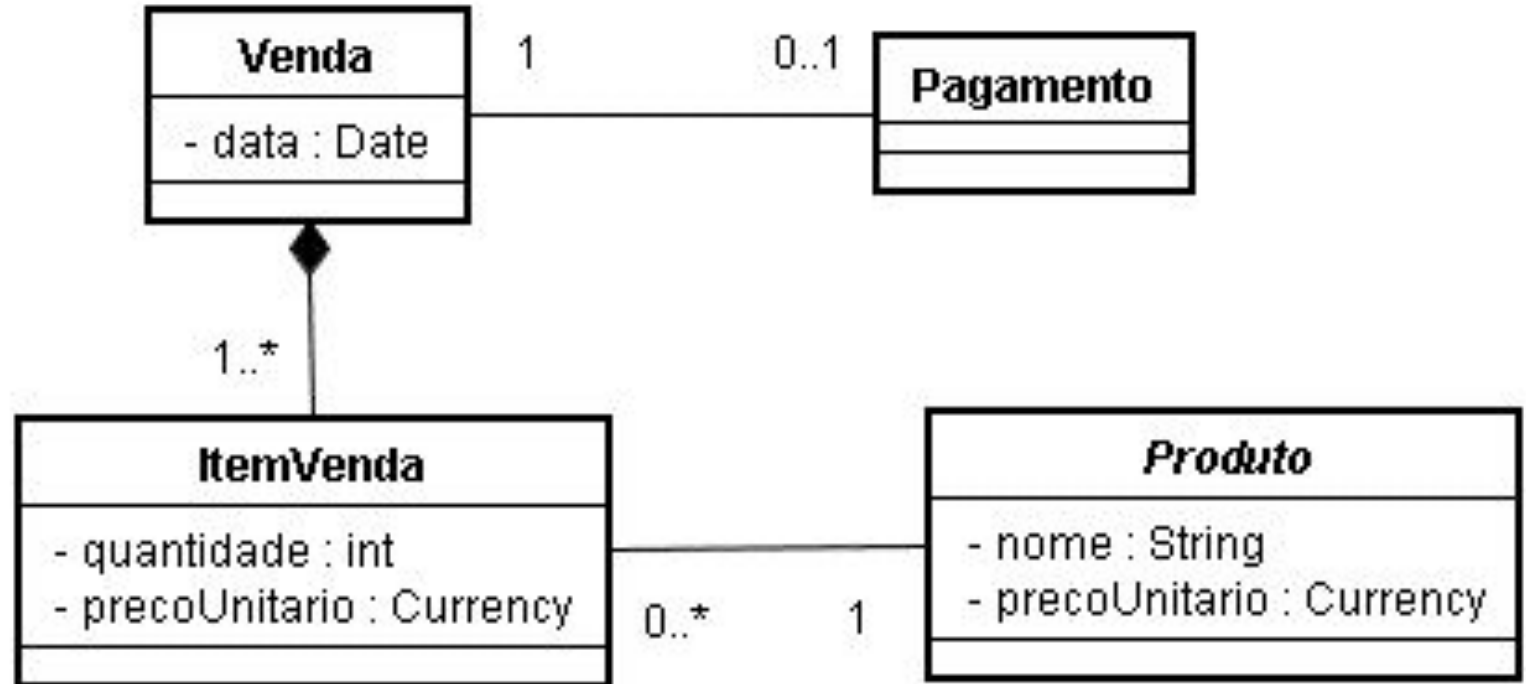
- contém objetos de X
- registra instâncias de X
- usa muitos objetos da classe X
- possui os dados de inicialização de X

# CREATOR

## EXEMPLO: O PONTO DE VENDA

Exemplo:

- Quem deve criar objetos ItemVenda?
- Quem deve criar objetos Pagamento?
- Quem deve criar objetos Venda?





# CREATOR

É mais adequado escolher criador que estará conectado ao objeto criado, de qualquer forma, depois da criação.

Exemplo de criador que possui os valores de inicialização

- Uma instância de Pagamento deve ser criada
- A instância deve receber o total da venda
- Quem tem essa informação? Venda
- Venda é um bom candidato para criar objetos da classe Pagamento

Creator é um caso particular de Expert

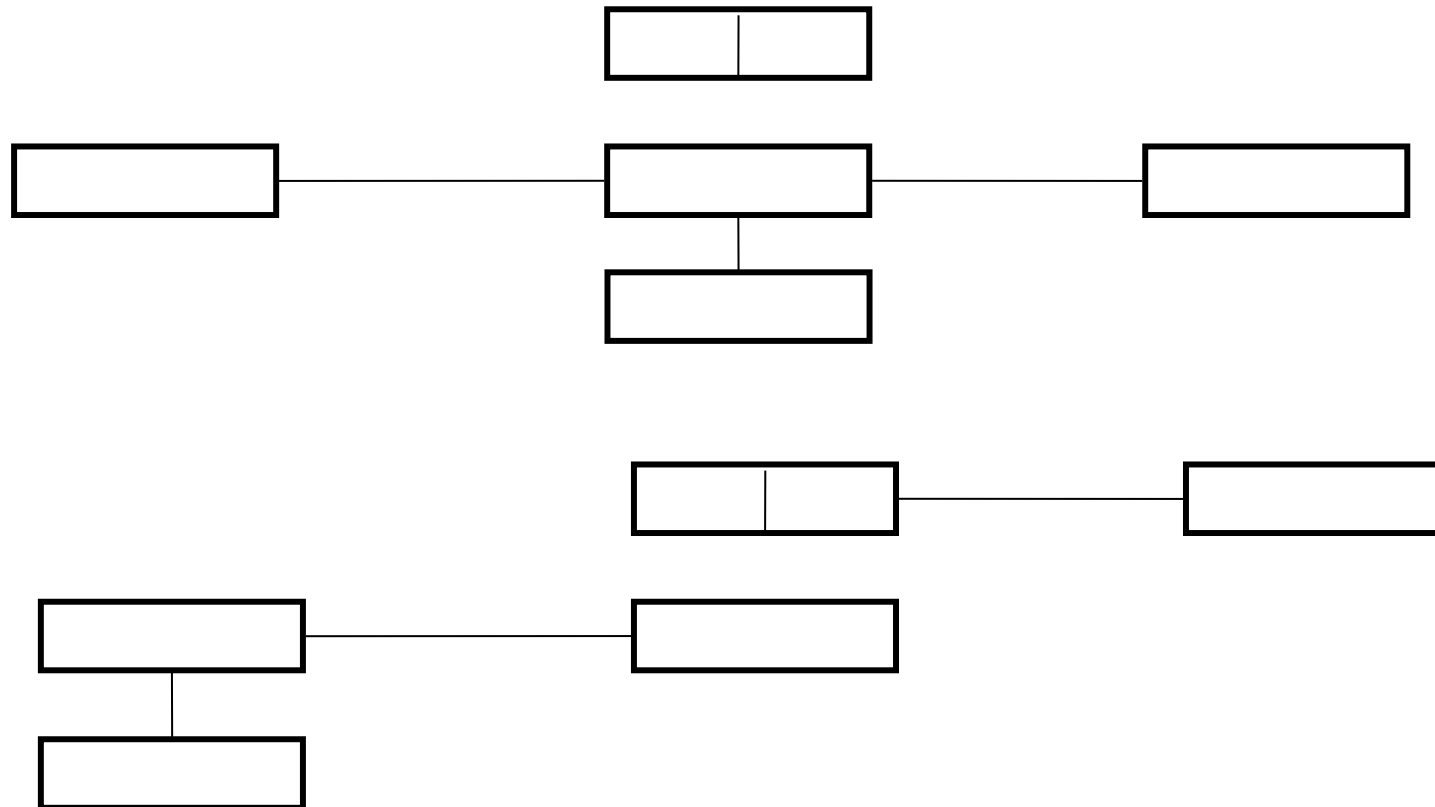
- Por quê?

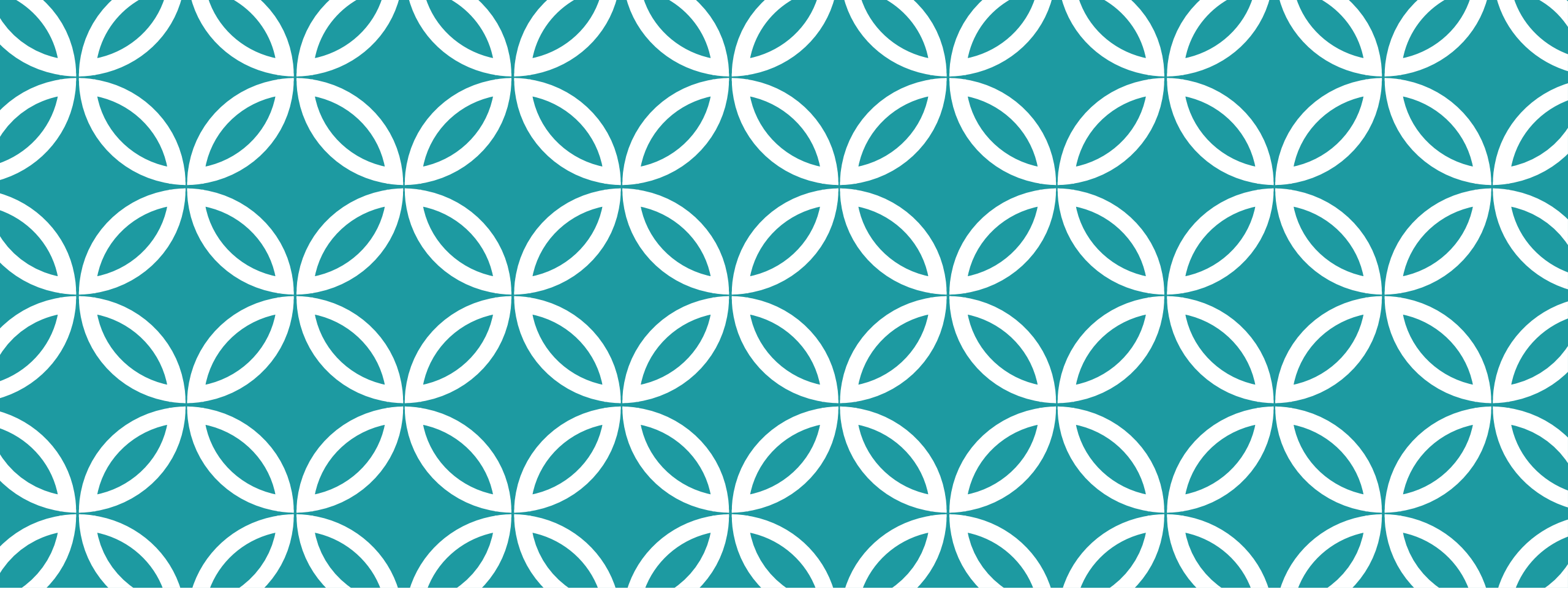
# CREATOR

## Consequência

- Baixo acoplamento
  - Venda já estaria acoplado de qualquer forma

TED 3: DIANTE DAS SOLUÇÕES ABAIXO, QUAL A MAIS ADEQUADA PARA UM PROJETO?





# LOW COUPLING

Acoplamento Fraco

# LOW COUPLING

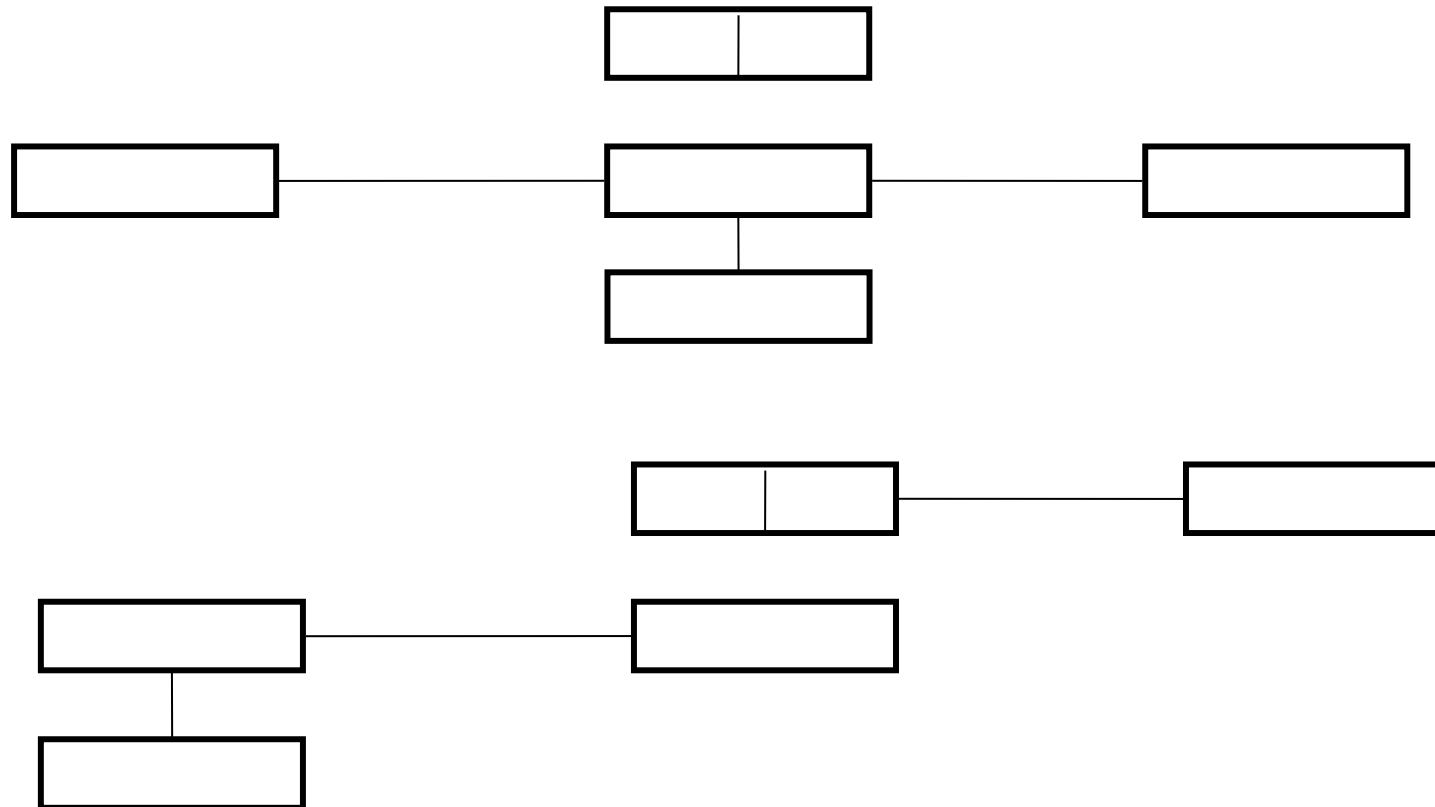
O acoplamento é uma medida de quão fortemente uma classe está conectada a outras classes, tem conhecimento das mesmas ou depende delas.

- Uma classe com baixo (fraco) acoplamento não depende de muitas outras.

Uma classe com acoplamento forte é:

- mais difícil de compreender isoladamente
- mais difícil de reutilizar (seu uso depende da reutilização das outras classes da qual ela depende)
- sensível a mudanças nas classes associadas.

# LOW COUPLING



# LOW COUPLING

No slide anterior:

- Que configuração de classes é melhor?
- Por quê?

Aspectos gerais:

- Qual a relação do conceito de acoplamento com os objetos de controle em um caso de uso?
- Quais propriedades de um produto de *software* estão relacionadas com esse conceito de acoplamento?

# LOW COUPLING

## Problema

- Como minimizar dependências e maximizar o reuso???

## Solução

- Atribuir responsabilidades visando minimizar o acoplamento

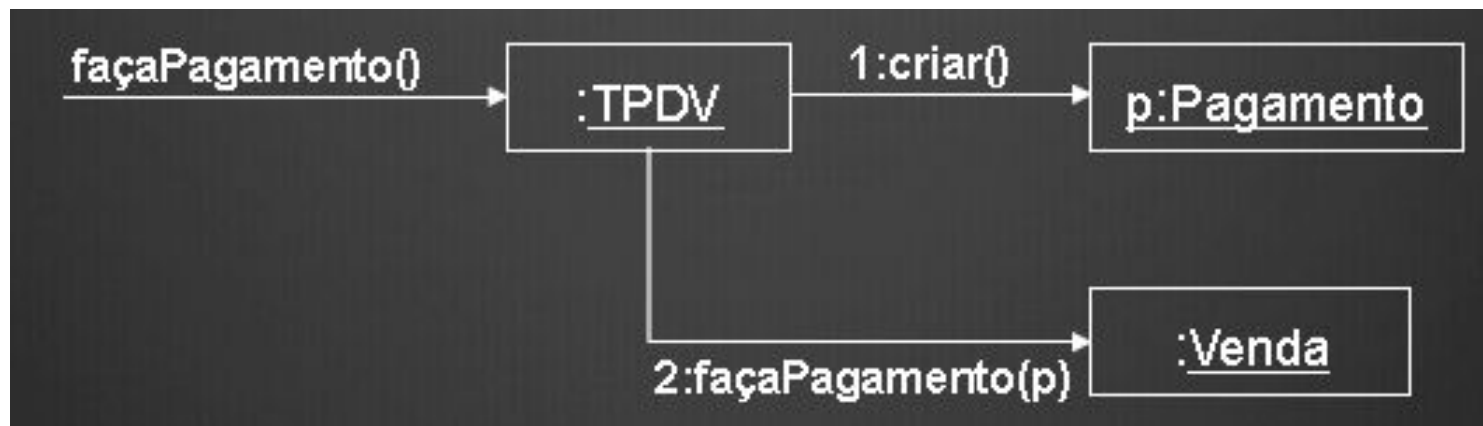


# LOW COUPLING

## EXEMPLO: O PONTO DE VENDA

Suponha a criação de um Pagamento associado à Venda

- Segundo o padrão Creator, TPDV deveria criar Pagamento e repassá-lo a Venda!



- Mas aí teremos três classes acopladas

# LOW COUPLING

## EXEMPLO: O PONTO DE VENDA

Então... em nome do baixo acoplamento... ignoramos o Creator!!!



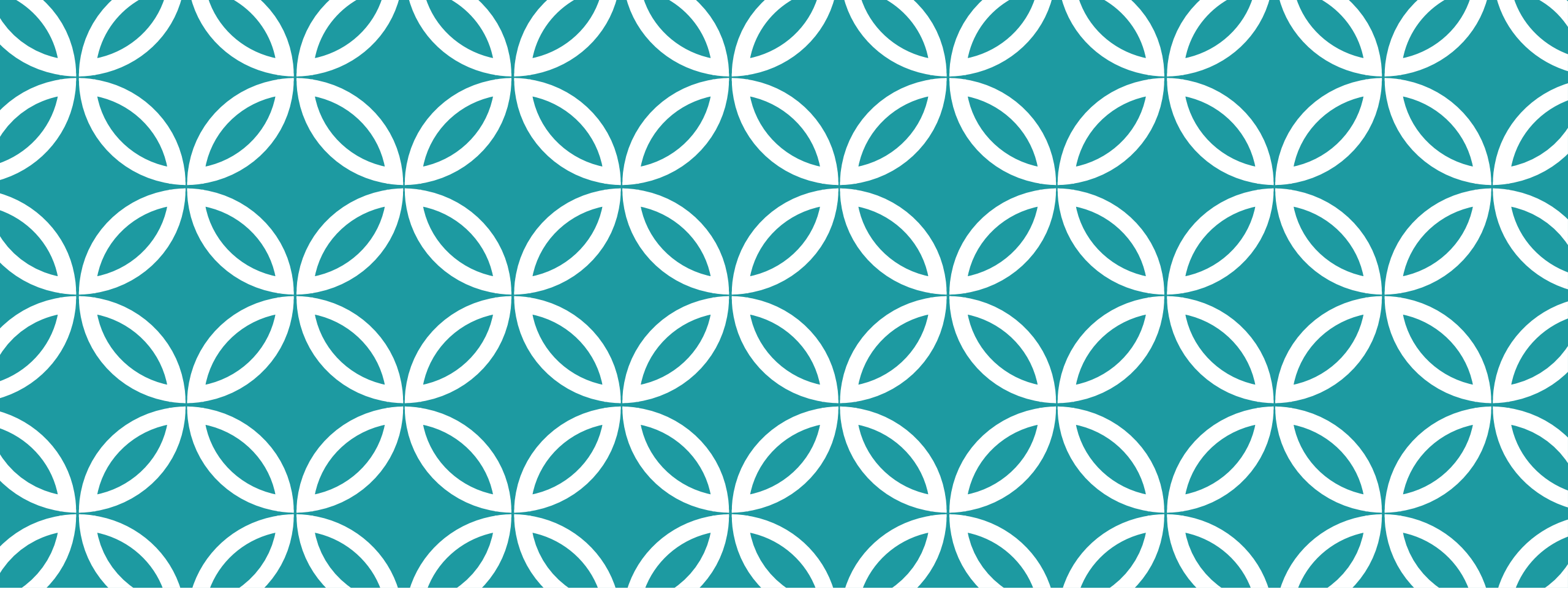
Agora temos acoplamento entre duas classes apenas

# LOW COUPLING

A maioria dos padrões visa o baixo acoplamento

## Consequências

- Uma classe fracamente acoplada não é afetada (ou pouco afetada) por mudanças em outras classes
- Simples de entender isoladamente
- Reuso mais fácil



# HIGH COHESION

Coesão Alta

# HIGH COHESION (COESÃO ALTA)

A coesão é uma medida do quão fortemente relacionadas e focalizadas são as responsabilidades de uma classe.

Uma classe com baixa coesão:

- faz muitas coisas não-relacionadas
- executa trabalho demais.

Classes não coesas são:

- difíceis de compreender
- difíceis de reutilizar
- difíceis de manter
- sensíveis a mudanças.

# HIGH COHESION (COESÃO ALTA)

## Problema

- Como gerenciar a complexidade?
  - Responsabilidades no lugar certo?
  - Funcionalidades implementadas pelas classes corretas?

## Solução

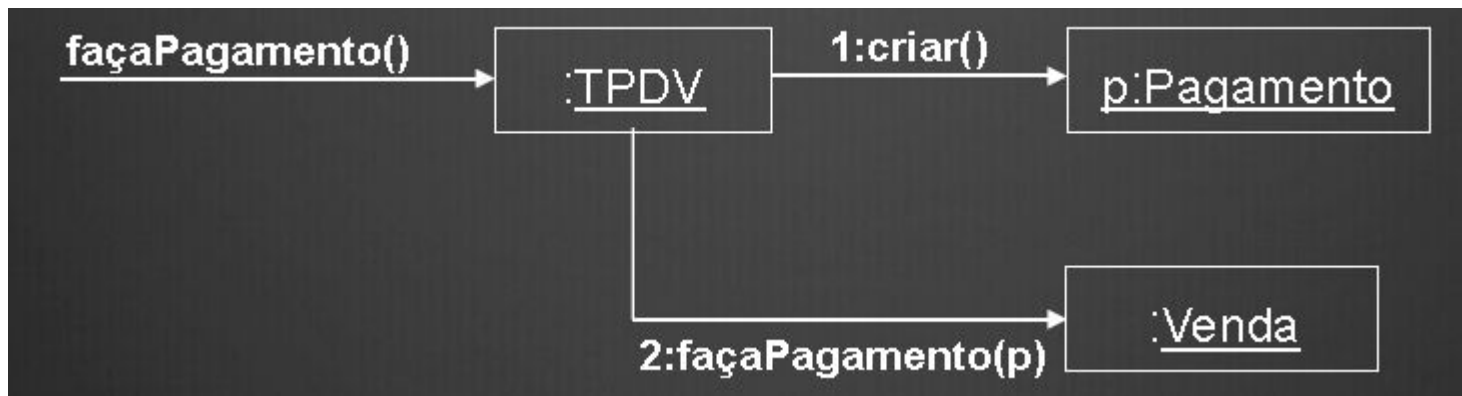
- Buscar uma alta coesão no projeto

# HIGH COHESION (COESÃO ALTA)

## EXEMPLO: O PONTO DE VENDA

De acordo com o Creator...

- TPDV deveria criar Pagamento



- Suponha que isto ocorra várias vezes (outras classes)
- TPDV acumula métodos não relacionados a ele
- Baixa coesão!!!

# HIGH COHESION (COESÃO ALTA)

## EXEMPLO: O PONTO DE VENDA

Com a delegação de `façaPagamento()`



□ Mantém-se a coesão em TPDV!!!



# HIGH COHESION (COESÃO ALTA)

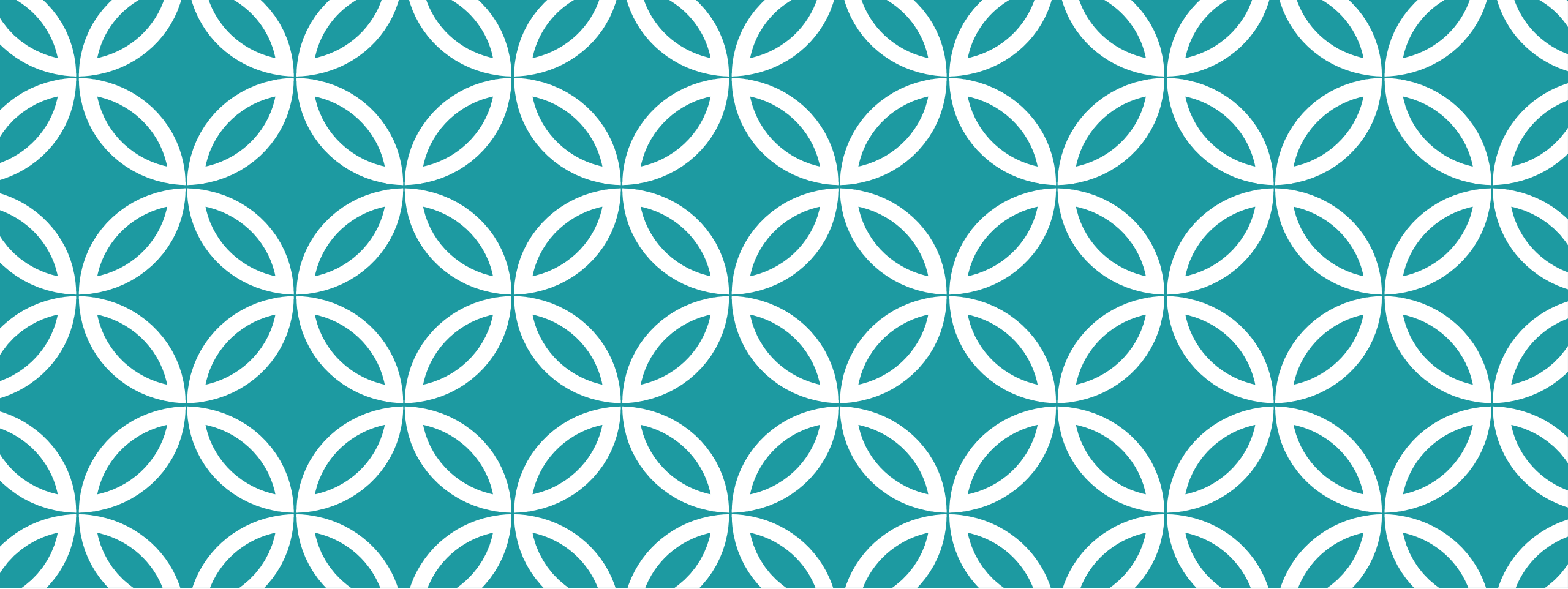
É extremamente importante assegurar que as responsabilidades atribuídas a cada classe sejam altamente relacionadas.

Em um bom projeto OO, cada classe não deve fazer muito trabalho.

- cada classe deve capturar apenas uma abstração.

Como perceber que a coesão de uma classe está baixa?

- Quando alguns atributos começam a depender de outros.
- Quando há subgrupos de atributos correlacionados na classe.



# CONTROLLER

Controladores

# CONTROLLER

**Problema:** quem deveria ser responsável por tratar um evento do sistema?

**Solução:** atribuir a responsabilidade do tratamento de um evento do sistema a uma classe que representa uma das seguintes escolhas:

- Representa o “sistema” todo (controlador fachada)
- Representa um tratador oficial de todos os eventos de sistema de um caso de uso (controlador de caso de uso)

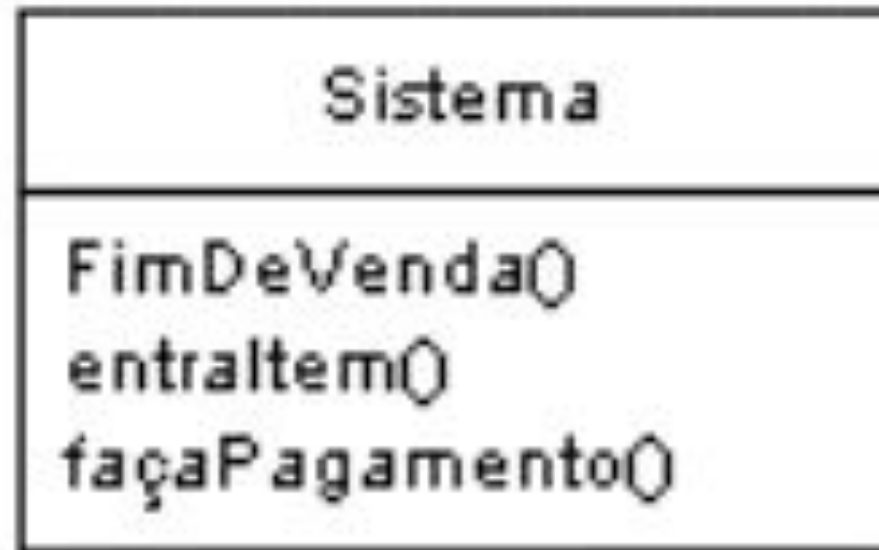
# CONTROLLER

## Use um controlador

- Um controlador é um objeto que não é de interface GUI responsável pelo tratamento de eventos do sistema
- Um controlador define métodos para as operações do sistema

# CONTROLLER

Um sistema contendo operações “de sistema” associados com eventos do sistema.



# CONTROLLER

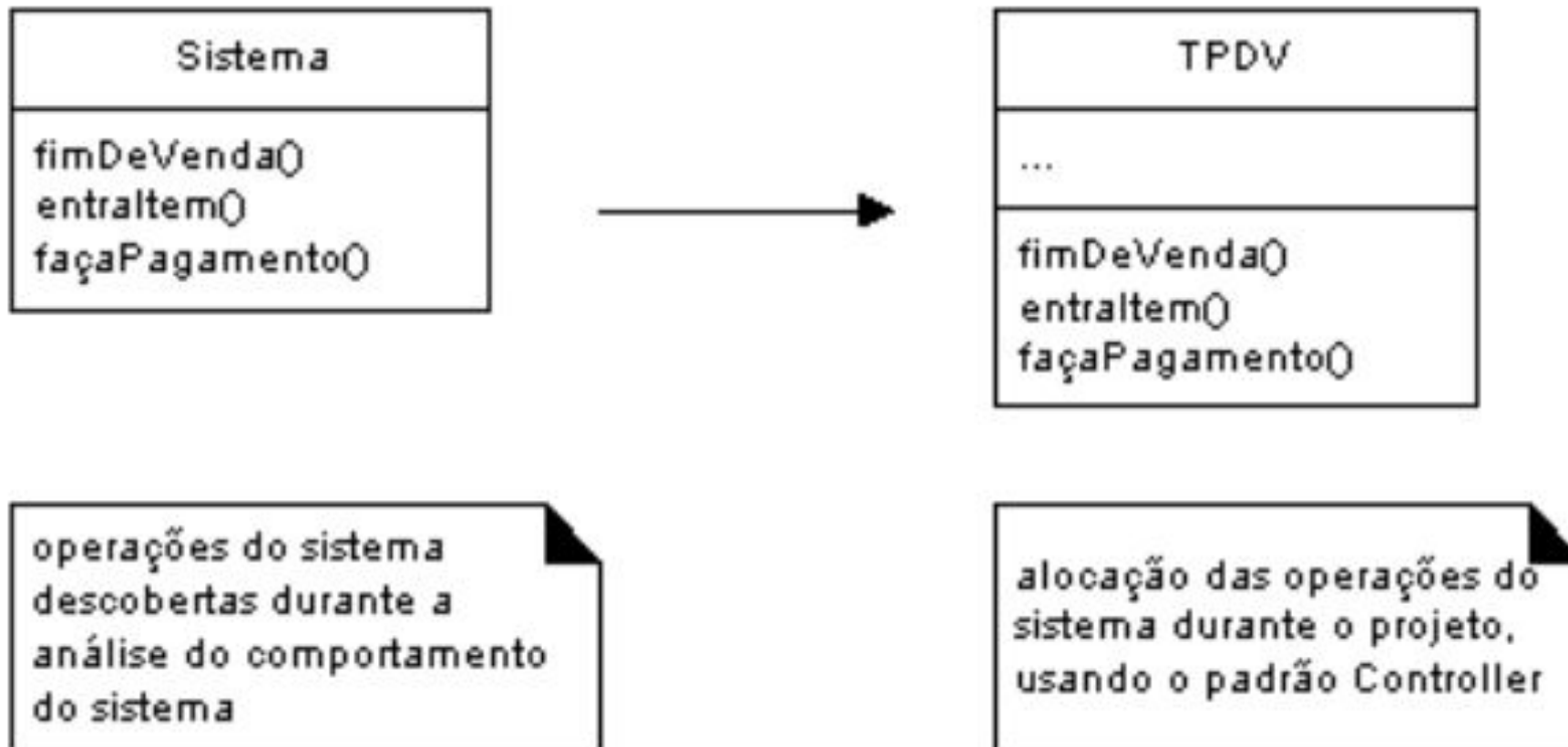
Qual classe deveria ser responsável pela recepção e tratamento deste evento do sistema?

É um controlador.

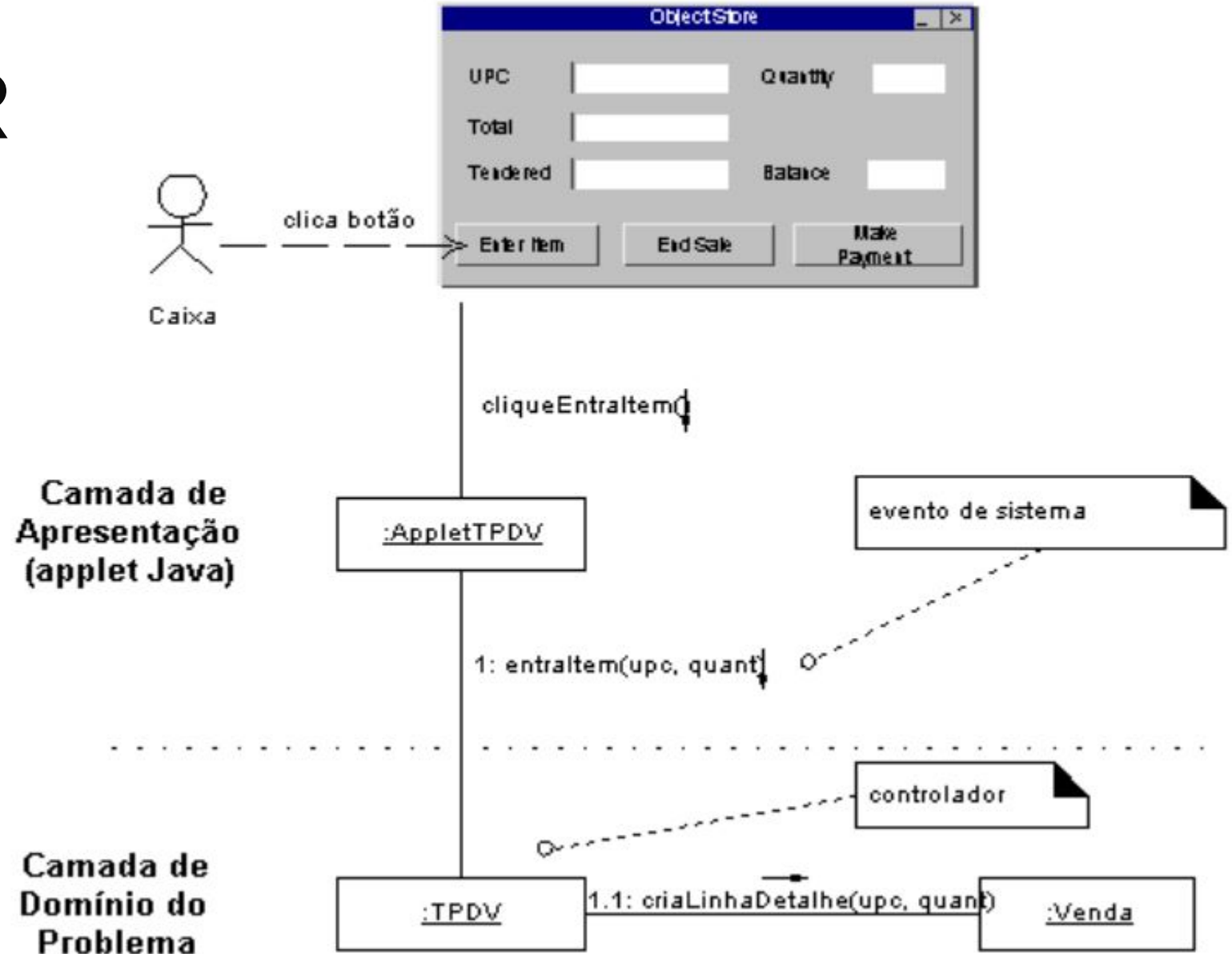
entraItem(upc, quantidade)

???

# CONTROLLER



# CONTROLLER

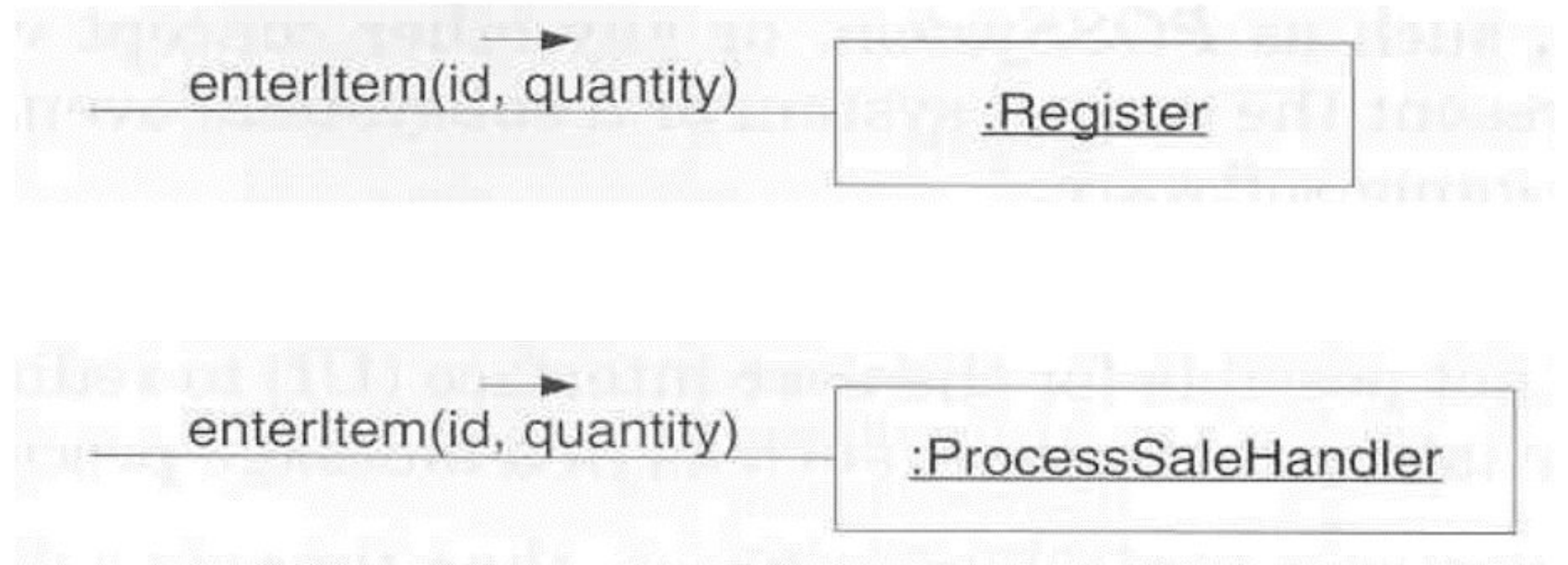




# CONTROLLER

A primeira solução representa o sistema inteiro

A segunda solução representa o destinatário ou *handler* de todos os eventos de um caso de uso



# CONTROLLER

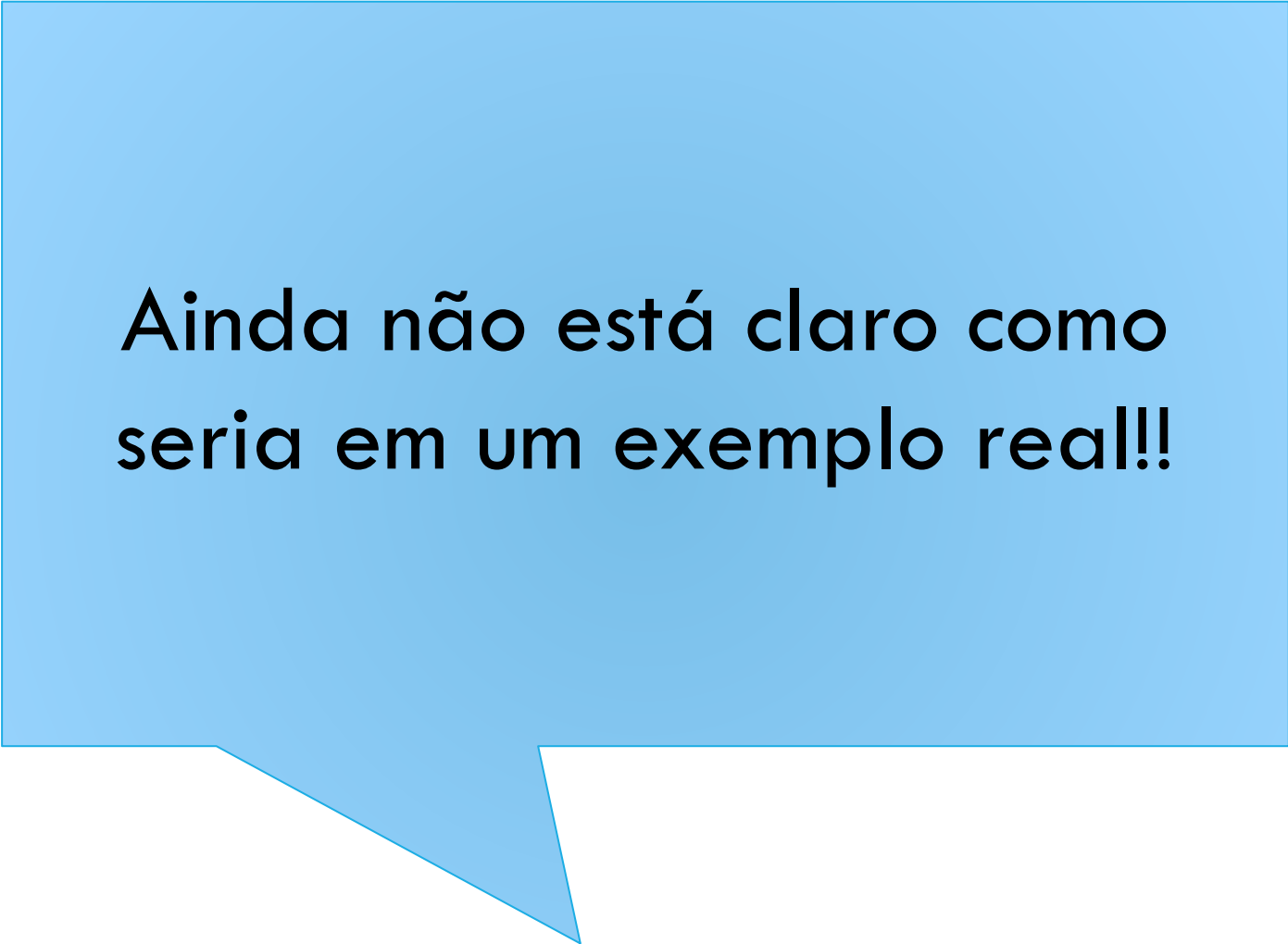
## Benefícios

- Diminui a sensibilidade da camada de apresentação em relação à lógica de domínio;
- Oportunidade para controlar o estado do caso de uso;
- Aumento potencial para reutilização
- Pode raciocinar / controlar o estado de um caso de uso, por exemplo, não fechar a venda até que o pagamento seja aceito.

# GRASP: CONTROLLER

## Problemas

- Controlador sobrecarrega muitas operações do sistema
- Controlador não consegue delegar tarefas
- Controlador tem muitos atributos
- É necessário balancear a quantidade de controladores:
  - O problema mais comum é ter poucos controladores (controladores sobrecarregados).



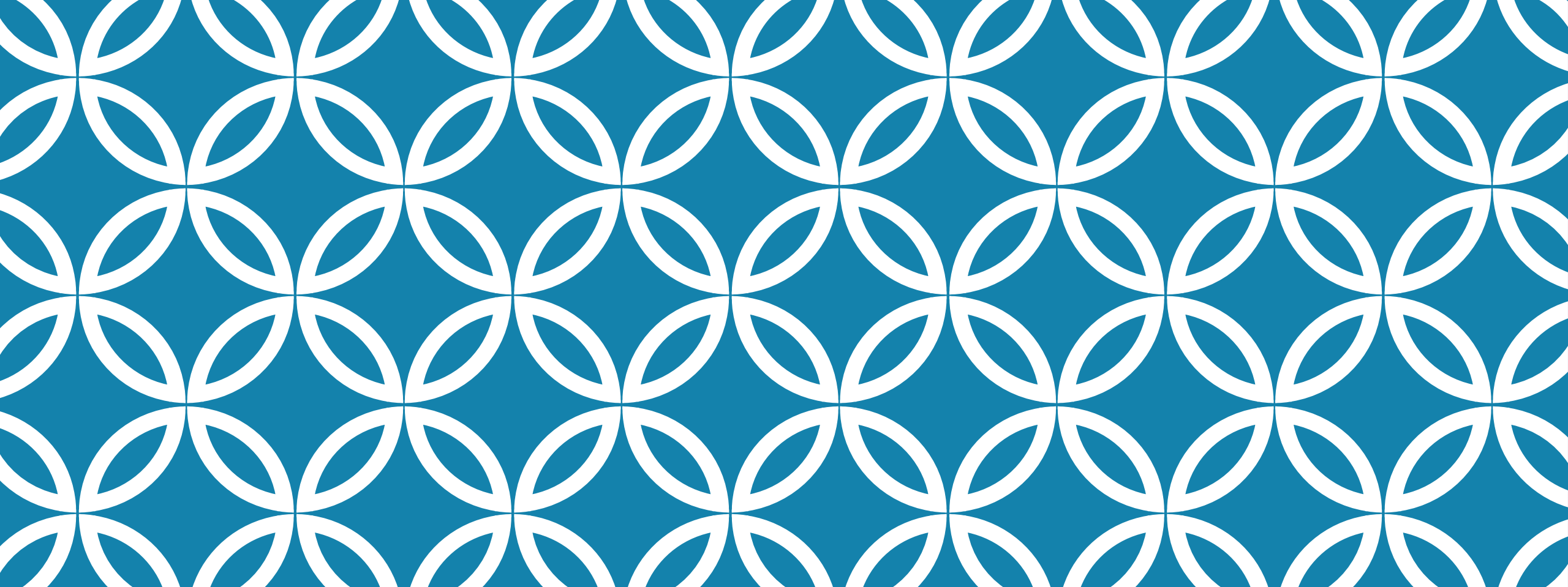
Ainda não está claro como  
seria em um exemplo real!!

# EXERCÍCIO CONTROLLER

Imagine que você está desenvolvendo um sistema de pontuação de um jogo tiro em primeira pessoa. Neste jogo, há três tipos de oponentes principais: Soldados da guerrilha, espiões e exército inimigo. Cada um destes oponentes tem suas habilidades específicas e forças diferenciada de acordo com a tabela.

Tipo de Inimigo	Pontos por destruí-lo	Habilidades
Soldados da guerrilha	100 pontos	Inteligência: 5, Força: 8, Estratégia: 8
Espiões	200 pontos	Inteligência: 10, Força: 7, Estratégia: 10
Exército inimigo	50 pontos	Inteligência: 2, Força: 7, Estratégia: 10

Qual seria a arquitetura de classes, baseada no padrão Controller, considerando que o projeto possui as seguintes classes: Inimigos (e suas sub-classes se necessário), Pontuação e PontuaçãoController.



# DÚVIDAS

[alanamm.prof@gmail.com](mailto:alanamm.prof@gmail.com)