


**Welcome to the jungle!**

**Existem dois tipos de pessoas...**



facebook.com/BugginhoDeveloper

**As que seguem os padrões de arquitetura  
e as que só querem fazer funcionar**

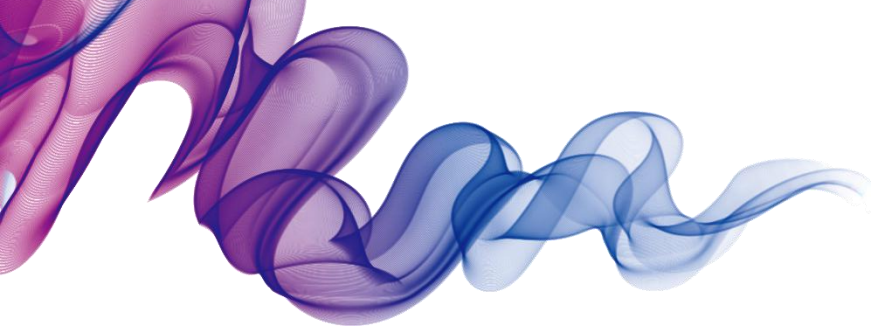


# Programação Orientada a Objetos



# **Afinal, o que são Padrões de Projeto (Design Patterns)?**



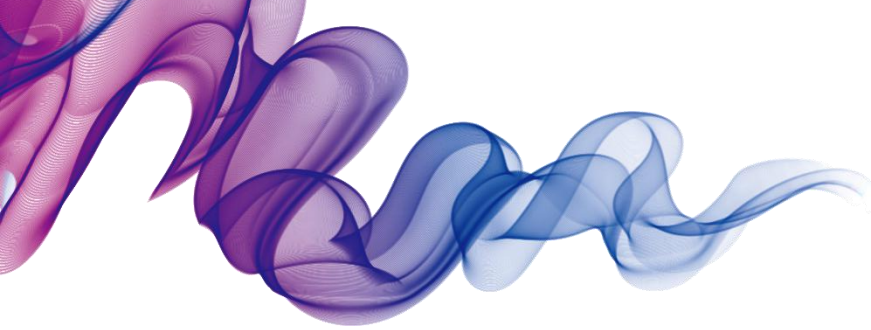


- Inicialmente descrito por Christopher Alexander (1977-1979)

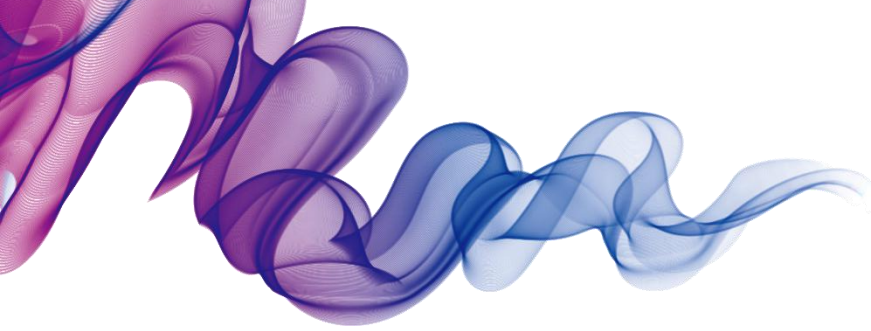
*Notes on the Synthesis of Form, The Timeless Way of Building e A Pattern Language.*

## Com as principais características:

- **Encapsulamento:** um padrão encapsula um problema ou solução bem definida. Ele deve ser independente, específico e formulado de maneira a ficar claro onde ele se aplica.
- **Generalidade:** todo padrão deve permitir a construção de outras realizações a partir deste padrão.
- **Equilíbrio:** quando um padrão é utilizado em uma aplicação, o equilíbrio dá a razão, relacionada com cada uma das restrições envolvidas, para cada passo do projeto. Uma análise racional que envolva uma abstração de dados empíricos, uma observação da aplicação de padrões em artefatos tradicionais, uma série convincente de exemplos e uma análise de soluções ruins ou fracassadas pode ser a forma de encontrar este equilíbrio.
- **Abstração:** os padrões representam abstrações da [experiência empírica](#) ou do conhecimento cotidiano.
- **Abertura:** um padrão deve permitir a sua extensão para níveis mais baixos de detalhe.
- **Combinatoriedade:** os padrões são relacionados hierarquicamente. Padrões de alto nível podem ser compostos ou relacionados com padrões que endereçam problemas de nível mais baixo.



- Alexander também definiu como deveria ser uma descrição de um Padrão:
  - **Nome:** uma descrição da solução, mais do que do problema ou do contexto.
  - **Exemplo:** uma ou mais figuras, diagramas ou descrições que ilustrem um protótipo de aplicação.
  - **Contexto:** a descrição das situações sob as quais o padrão se aplica.
  - **Problema:** uma descrição das forças e restrições envolvidos e como elas interagem.
  - **Solução:** relacionamentos estáticos e regras dinâmicas descrevendo como construir artefatos de acordo com o padrão, frequentemente citando variações e formas de ajustar a solução segundo as circunstâncias. Inclui referências a outras soluções e o relacionamento com outros padrões de nível mais baixo ou mais alto.



- Os primeiros padrões foram formalmente descritos em 1987 com BECK, Kent e CUNNINGHAN, Ward na conferência OOPSLA, na linguagem Smalltalk.

# *Padrões de Projeto / O que é ?*

- Um padrão descreve
  - **Problema:** que ocorre repetidamente
  - **Solução:** para esse problema de forma que se possa reutilizar a solução



# *Padrões de Projeto / O que é ?*

Por que usar Padrões de Projeto?

- Aprender com a **experiência** dos outros
- O **jargão** facilita a comunicação de princípios
- Melhora a **qualidade** do software
- Descreve **abstrações** de software
- Ajuda a **documentar** a arquitetura
- Captura as partes **essenciais** de forma compacta





# *Padrões de Projeto / O que é ?*

No entanto ...

- não apresentam uma solução exata
- não resolvem todos os problemas de *design*
- não é exclusivo de orientação a objetos



“é importante notar que um bom design não acontece por acidente, pois ele exige trabalho duro e evolução. De forma complementar, também não quer dizer que usando um padrão necessariamente se tem um bom design.”

“Apenas saber como aplicar padrões não é o suficiente para um bom design. Entender quando e onde aplica-los é tão importante quanto, senão mais importante.”

YODER



“Em 1998, foi feita a afirmação que a arquitetura que realmente predominava na prática era a ***Big Ball of Mud***. E mesmo com muito trabalho e dedicação ainda é possível acabar com um ***B.B.M.*** se você não está comprometido a manter o código sempre limpo.”

“É importante ressaltar que não é só porque uma linguagem orientada a objetos está sendo utilizada que se estará modelando de forma orientada a objetos. Se as suas classes não representam abstrações do sistema e são apenas repositórios de funções, você na verdade está programando segundo o paradigma estruturado.”

GUERRA, Eduardo

- Conceitos Fundamentais:
- **Herança:** O maior desafio é identificar os pontos onde essa abstração deve ser utilizada e que será aproveitada de forma adequada pelo sistema;
- **Encapsulamento:** “a ignorância é uma benção.” Deve haver uma separação entre o comportamento interno de uma classe com a interface disponibilizada para os clientes;



- **Polimorfismo:** consequência de usar Herança e interfaces. Em outras palavras qualquer objeto pode ser atribuído para uma variável do tipo de uma das suas superclasses ou para o tipo de suas interfaces;
- **Interface ou Classe Abstrata ?**

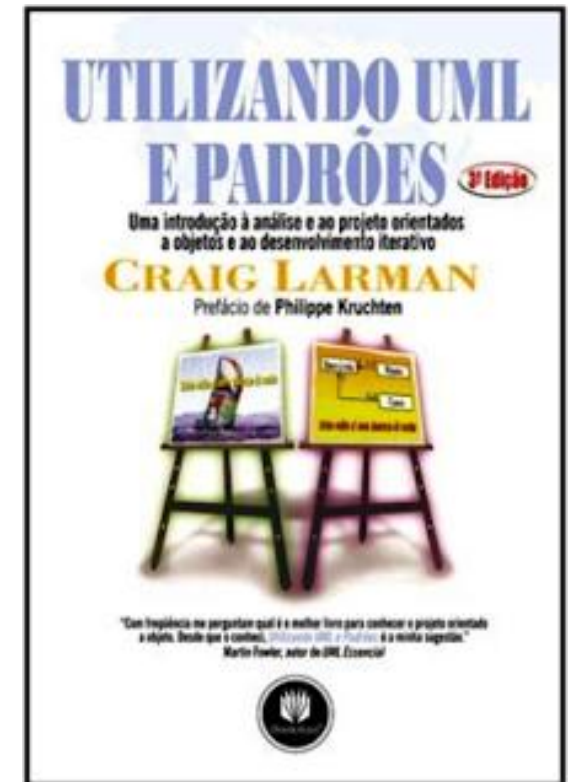
Como se seleciona um padrão ?

- Entenda as **soluções**
- Estude o **relacionamento** entre os padrões
- Estude as **similaridades** entre os padrões
- Conheça as principais causas de **retrabalho**
- Considere o que pode **mudar**



# Padrões GRASP

- Descrita inicialmente por Alexander, mas tendo como sua principal obra Utilizando UML e Padrões, de LARMAN, Craig.



- **GRASP - General Responsibility Assignment Software Patterns (or Principles)**
  - refletem práticas mais pontuais da aplicação de técnicas OO
  - ocorrem na implementação de vários padrões GoF





# *Padrões de Projeto / GRASP*

## ***Responsabilidade***

*“Contrato ou obrigação de um tipo ou classe”*

*[Booch e Rumbaugh]*

Dois tipos de responsabilidades dos objetos:

- De conhecimento (**knowing**):
  - sobre dados privativos e encapsulados; sobre objetos relacionados; sobre coisas que pode calcular ou derivar.
- De realização (**doing**):
  - fazer alguma coisa em si mesmo; iniciar uma ação em outro objeto; controlar e coordenar atividades em outros objetos.

# *Padrões de Projeto / GRASP*





# *Padrões de Projeto / GRASP*

- Ou seja, responsabilidades são obrigações de um tipo ou de uma classe.
- **Obrigações de fazer algo**
  - Fazer algo a si mesmo
  - Iniciar ações em outros objetos
  - Controlar ou coordenar atividades em outros objetos
- **Obrigações de conhecer algo**
  - Conhecer dados encapsulados
  - Conhecer objetos relacionados
  - Conhecer coisas que ele pode calcular
- **Exemplos**
  - Uma Venda tem a responsabilidade de criar linha de detalhe (fazer algo)
  - Uma Venda tem a responsabilidade de saber sua data (conhecer algo)

## Atribuição de Responsabilidades

- Atribuição a objetos durante o design (projeto)
  - Responsabilidades    Classes e Métodos
- tradução depende da granularidade da responsabilidade
- Responsabilidades do tipo *doing*
  - Realizadas por um único método ou uma coleção de métodos trabalhando em conjunto
- Responsabilidades do tipo *knowing*
  - inferidas a partir do modelo conceitual (são os atributos e relacionamentos)



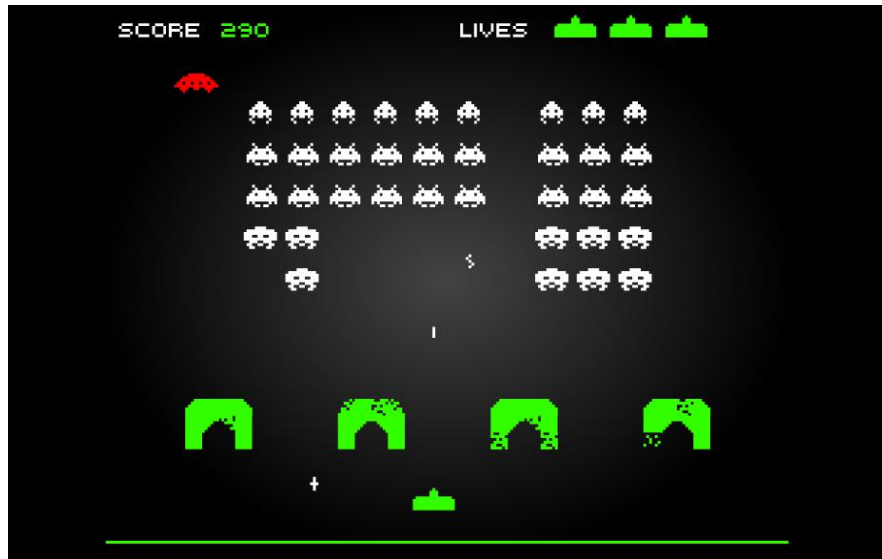
# *Padrões de Projeto / GRASP*

## Granularidade

- Uma responsabilidade pode envolver um único método (ou poucos)
  - Exemplo: Criar uma linha de detalhe
- Uma responsabilidade pode envolver dezenas de classes e métodos
  - Exemplo: Responsabilidade de fornecer acesso a um BD
- Uma responsabilidade não é igual a um método
  - Mas métodos são usados para implementar responsabilidades



# Padrões de Projeto / GRASP



Quem é **responsável** por:

- Manter as coordenadas dos alienígenas?
- Manter as coordenadas da nave?
- Criar novas naves no início das fases?
- Controlar pontuação?
- Controlar movimento dos personagens?
- Lançar tiros?



# *Padrões de Projeto / GRASP*

## **Padrões Básicos**

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

## **Padrões Avançados**

- Polymorphism
- Pure Fabrication
- Indirection
- Protected Variations

## **Information Expert – Especialista de informação**

### **Problema**

- No design, quando são definidas interações entre objetos precisamos de um princípio para atribuir responsabilidades a classes

### **Solução**

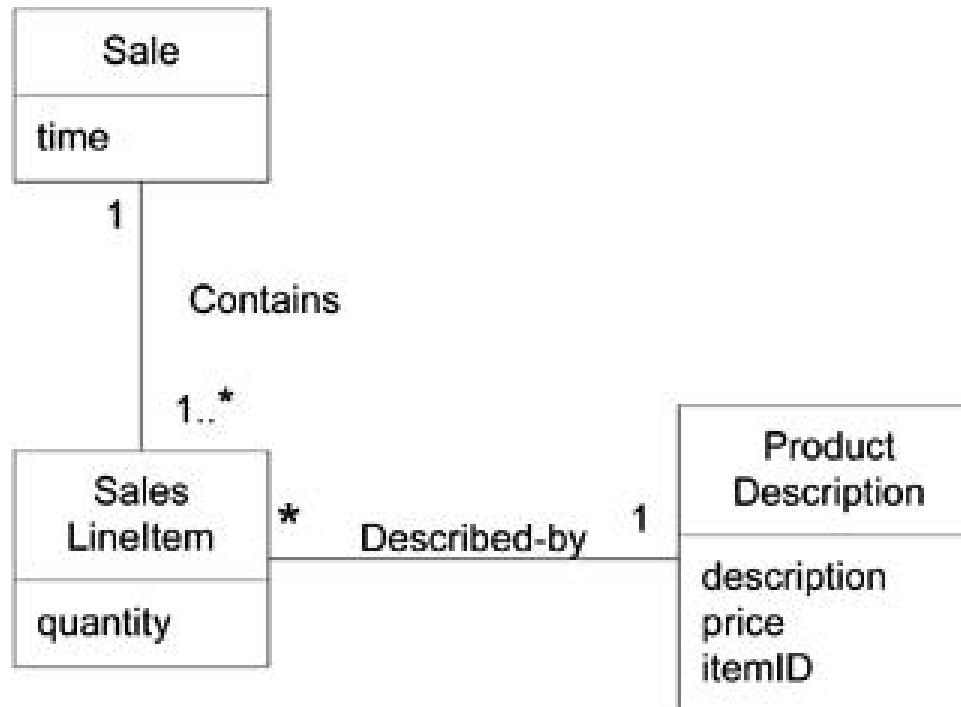
- Atribuir uma responsabilidade ao especialista de informação: classe que possui a informação necessária para cumpri-la
- Comece a atribuição de responsabilidades ao declarar claramente a responsabilidade

## **Information Expert – Especialista de informação**

Vamos usar como um exemplo um:

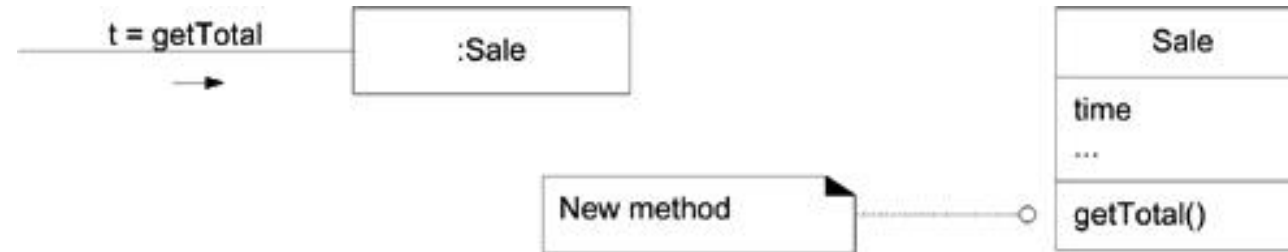
- Sistema de POS
- Utilizado tipicamente em lojas (comércio)
- Grava vendas (Sale) e gera pagamentos (Payment)

## Information Expert – Especialista de informação

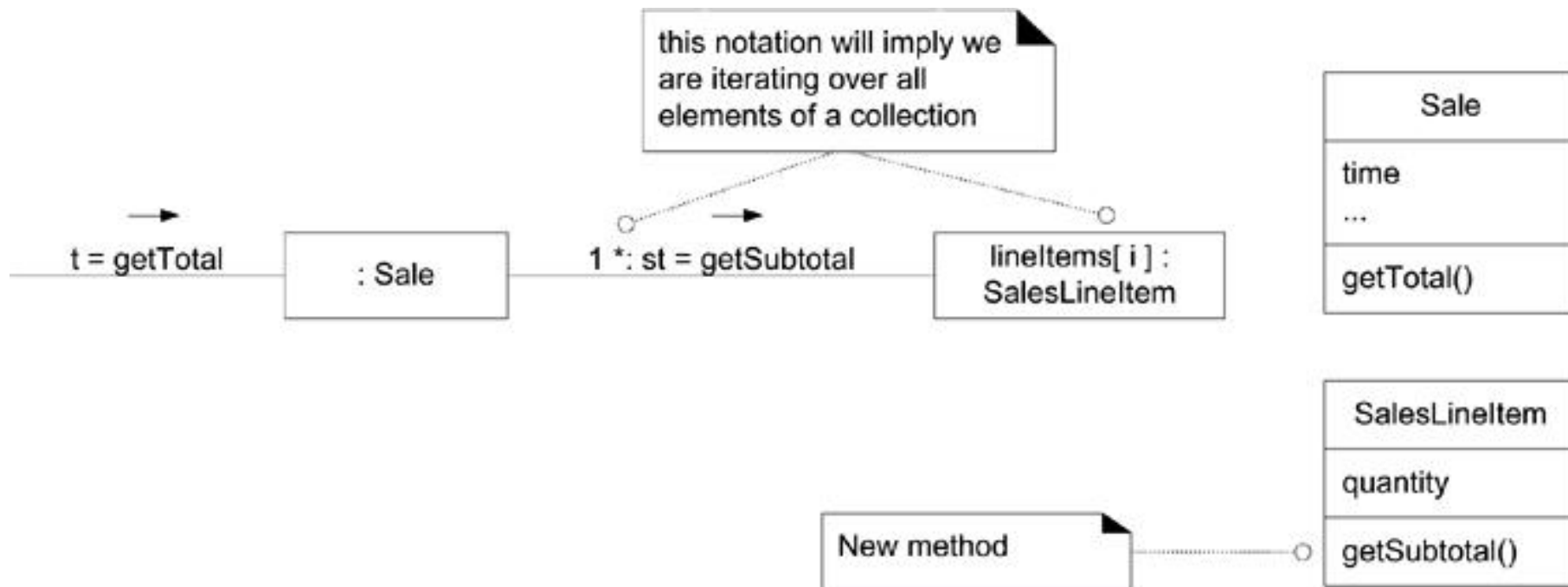




## Information Expert – Especialista de informação

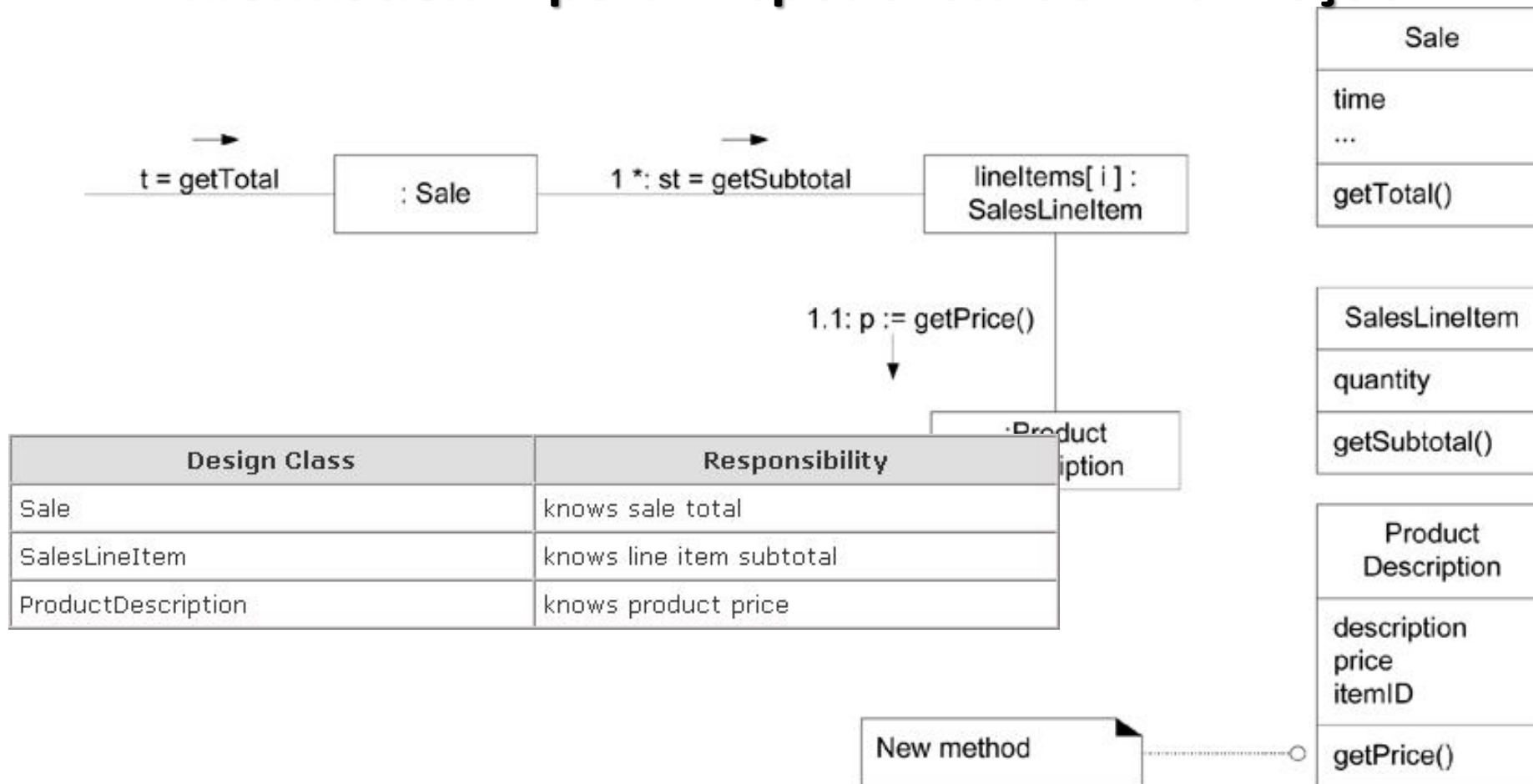


## Information Expert – Especialista de informação



# Padrões de Projeto / GRASP

## Information Expert – Especialista de informação



## **Creator**

### **Problema:**

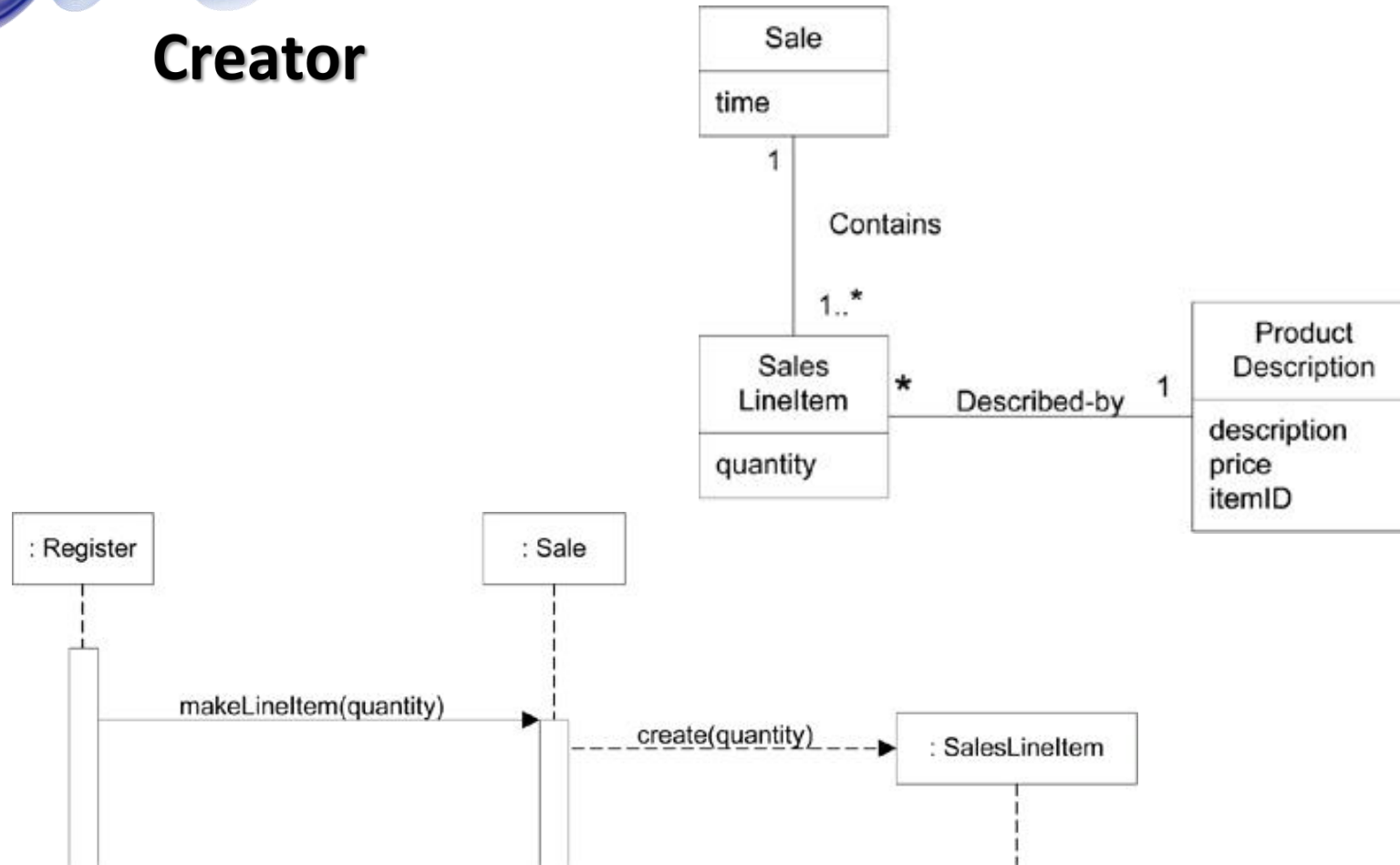
- Que classe deve ser responsável pela criação de uma nova instância de uma classe?

### **Solução:**

- Atribua a B a responsabilidade de criar A se:
  - B agrega A objetos;
  - B contém A objetos;
  - B guarda instâncias de A objetos;
  - B faz uso de A objetos;
  - B possui dados para inicialização de A.

# Padrões de Projeto / GRASP

Creator





## **Controller**

**Problema:** Quem deve ser responsável pelo controlar os eventos do sistema?

**Solução:** uma classe que represente

- O sistema como um todo
- Todo o negócio/organização
- Algo no mundo real envolvido na tarefa

## **Controller – Facade Controller**

- Centraliza acesso ao sistema em um controlador
- **Prós**
  - Centralizado
  - Baixo acoplamento na interface
  - Simplicidade
- **Contras**
  - Centralizado
  - *Controller* tem acoplamento alto
  - Coesão baixa

Lê-se “fessad contrôler”

## **Controller – Role Controller**

- Divide acesso ao sistema de acordo com o papel dos usuários ou sistemas externos
- **Prós**
  - Coesão melhora
  - Descentralizado
  - Reduz acoplamento do Controller
- **Contras**
  - Pode ser mal balanceado

## **Controller – Use Case Controller**

- Divide acesso ao sistema de acordo os casos de uso do sistema
- **Prós**
  - Coesão
- **Contras**
  - Acoplamento aumenta
  - Explosão de classes

## **Low Coupling – Baixo Acoplamento**

**Problema:** Como suportar baixa dependência, baixo impacto devido a mudanças e reuso constante?

**Solução:** Atribuir uma responsabilidade para que o acoplamento mantenha-se fraco.

O acoplamento (dependência entre classes) deve ser mantido o mais baixo possível.

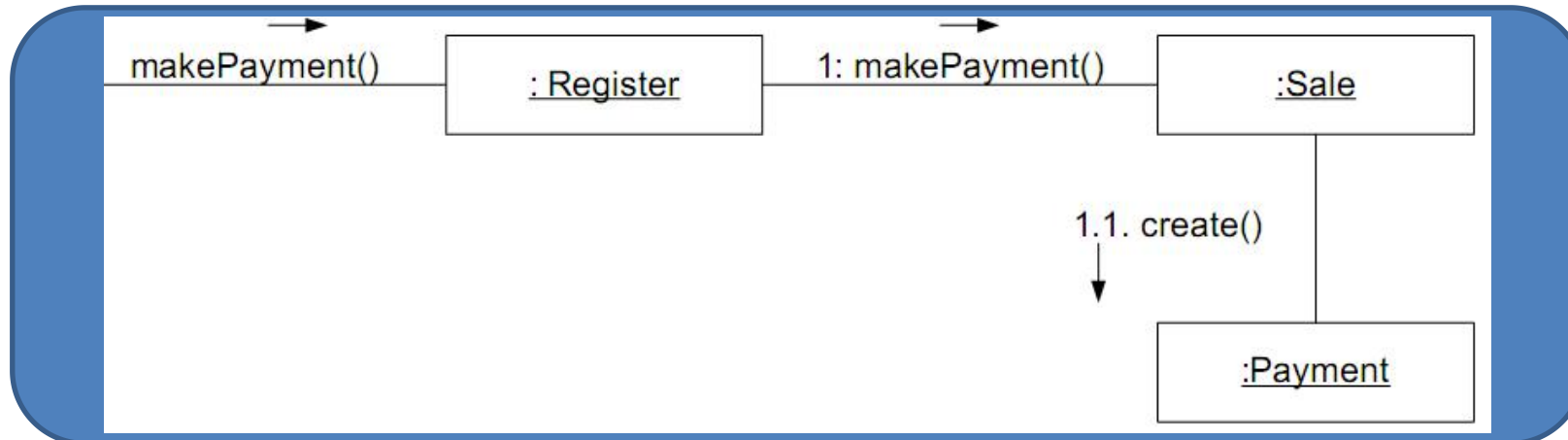
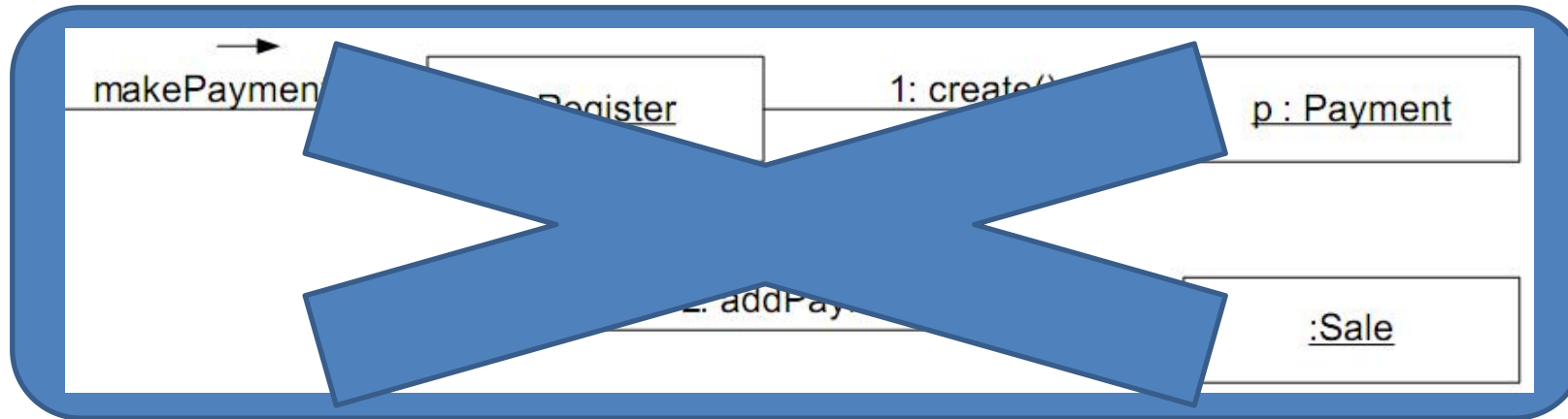
## **Low Coupling – Baixo Acoplamento**

- É uma medida de quanto um elemento está conectado a, ou depende de outros elementos.
- Uma classe com acoplamento forte depende de muitas outras classes.
  - Por isto, acoplamento forte é indesejável
- O acoplamento está associado à coesão:
  - Quanto maior a coesão, menor o acoplamento e vice-versa.



# Padrões de Projeto / GRASP

## Low Coupling – Baixo Acoplamento



## Low Coupling – Baixo Acoplamento

*Discussão:*

- *Classes que, por natureza, são genéricas e que têm alta probabilidade de reutilização deveriam ter acoplamento baixo*
- *O caso extremo do baixo acoplamento é o não acoplamento: contraria o princípio da orientação a objetos: objetos conectados, trocando mensagens entre si.*
- *O acoplamento alto não é o problema em si. O problema é o acoplamento a classes que, de alguma forma são instáveis: sua interface, sua implementação ou sua mera presença.*

## **High Cohesion – Alta Coesão**

### **Problema:**

- Como manter a complexidade sob controle?
- Classes que fazem muitas tarefas não relacionadas são:
  - mais difíceis de entender,
  - mais difíceis de manter e de reusar,
  - além de serem mais vulneráveis à mudança.

### **Solução**

- Atribuir uma responsabilidade para que a coesão se mantenha alta.

Lê-se “rai corrígiom”

## High Cohesion – Alta Coesão

### *Coesão coincidental:*

- o pior tipo de coesão, há nenhuma ou pouca relação construtiva entre os elementos de um módulo, em outras palavras é uma classe inchada, com um punhado de métodos, todos executando tarefas diferentes, sem nenhuma relação com a classe que os implementa.

```
class Angu {  
    public static int acharPadrão(String texto, String padrão)  
        { // ... }  
  
    public static int média(Vector números)  
        { // ... }  
  
    public static outputStream abreArquivo(string nomeArquivo)  
        { // ... }  
}  
class Xpto extends Angu {  
    // quer aproveitar código de Angu ...  
}
```

## High Cohesion – Alta Coesão

### *Coesão lógica:*

- melhor do que a coincidental mas não menos pior em um projeto, semelhante ao acoplamento de controle, onde um módulo faz um conjunto de funções relacionadas e uma das quais é escolhida através de um parâmetro para controlá-lo.

```
public void faça(int flag) {  
    switch(flag) {  
        case ON: // coisas para tratar de ON break;  
        case OFF: // coisas para tratar de OFF break;  
        case FECHAR: // coisas para tratar de FECHAR break;  
        case COR: // coisas para tratar de COR break;  
    }  
}
```

## High Cohesion – Alta Coesão

### *Coesão temporal:*

- Elementos estão agrupados no mesmo módulo porque são processados no mesmo intervalo de tempo
- Exemplos comuns:
  - Método de inicialização que provê valores defaults para um monte de coisas diferentes
  - Método de finalização que limpa as coisas antes de terminar

```
public void inicializaDados() {  
    font = "times";  
    windowSize = "200,400";  
    xpto.nome = "desligado";  
    xpto.tamanho = 12;  
    xpto.localização = "/usr/local/lib/java";  
}
```

```
class Xpto {  
  
    public Xpto() {  
        this.nome = "desligado";  
        this.tamanho = 12;  
        this.localização =  
            "/usr/local/lib/java";  
    }  
}
```



## **High Cohesion – Alta Coesão**

### ***Coesão procedural:***

- o módulo só tem sentido sobre a aplicação associada, sem ela, há dificuldade em entendê-lo, basicamente é a coesão relacionada aos procedimentos executados pelos elementos do módulo.

## **High Cohesion – Alta Coesão**

### ***Coesão de comunicação:***

- um módulo tem coesão de comunicação se os seus elementos usam a mesma entrada ou a mesma saída.

### ***Coesão sequencial:***

- a saída de um elemento é a entrada de outro e a solução é decompor em módulos menores, isso nós já vimos em tópicos passados, chamado também de acoplamento de dados.

## **High Cohesion – Alta Coesão**

### **Coesão Funcional**

- Uma medida de quão relacionadas ou focadas estão as responsabilidades de um elemento.
- Um módulo funcionalmente coeso contém todos os elementos e apenas aqueles necessários para realizar uma única tarefa bem definida.
- Exemplo: uma classe Cão é coesa se:
  - tem operações relacionadas ao Cão (morder, correr, comer, latir)
  - e apenas ao Cão (não terá por exemplo, validar, listarCaes, etc)
- Alta coesão promove design modular

## **High Cohesion – Alta Coesão**

Um módulo não será funcionalmente coeso se, para descrever sua função for necessário um, ou mais, dos seguintes itens:

- - Frase composta;
  - Mais do que uma frase;
  - Palavras que indiciam uma ligação temporal;
  - Falta de um objetivo específico simples;
  - Verbo ambíguo.

## **High Cohesion – Alta Coesão**

*Como um princípio básico, uma classe com alta coesão:*

- *tem um número relativamente pequeno de métodos,*
- *a funcionalidade desses métodos é altamente relacionada, e*
- *não faz trabalho de mais.*



*O princípio da separação Modelo-Vista pode ser enunciado em duas partes:*

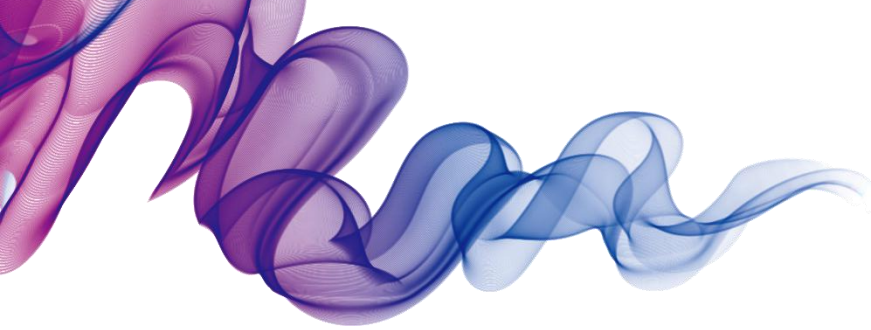
- Não conecte diretamente objetos pertencentes à interface com o usuário (a vista) com objetos não pertencentes à interface com o usuário (IU).*
- Não coloque lógica da aplicação (tal como o cálculo de impostos) nos métodos dos objetos da IU*





*A motivação para a separação Modelo-Vista inclui:*

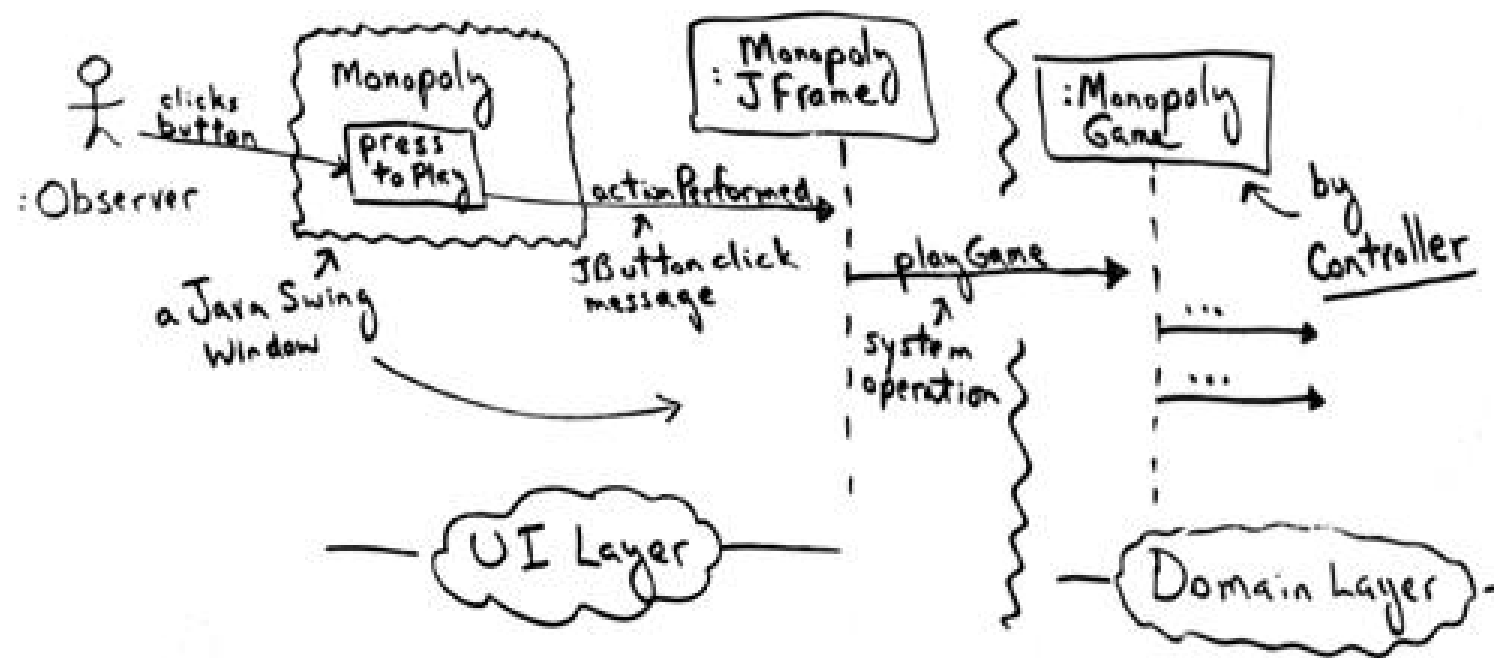
- *Suportar a criação de classes de negócio coesas, com foco nos processos do domínio ao invés de na interface com o usuário.*
- *Permitir o desenvolvimento separado das camadas de apresentação e negócio.*
- *Minimizar o impacto na camada de negócio das alterações nos requisitos da interface com o usuário.*



*A motivação para a separação Modelo-Vista inclui:*

- *Permitir que novas vistas sejam facilmente conectadas aos objetos de negócio existentes, sem afetar a camada de negócios.*
- *Permitir a existência de múltiplas vistas simultâneas para uma mesma camada de negócios (por exemplo, a visualização de dados de vendas na forma tabular ou através de um gráfico de pizzas)*

*O GRASP com o objeto Controlador responde a uma questão básica no projeto de sistemas OO: Como conectar a camada de apresentação à camada da lógica do negócio?*



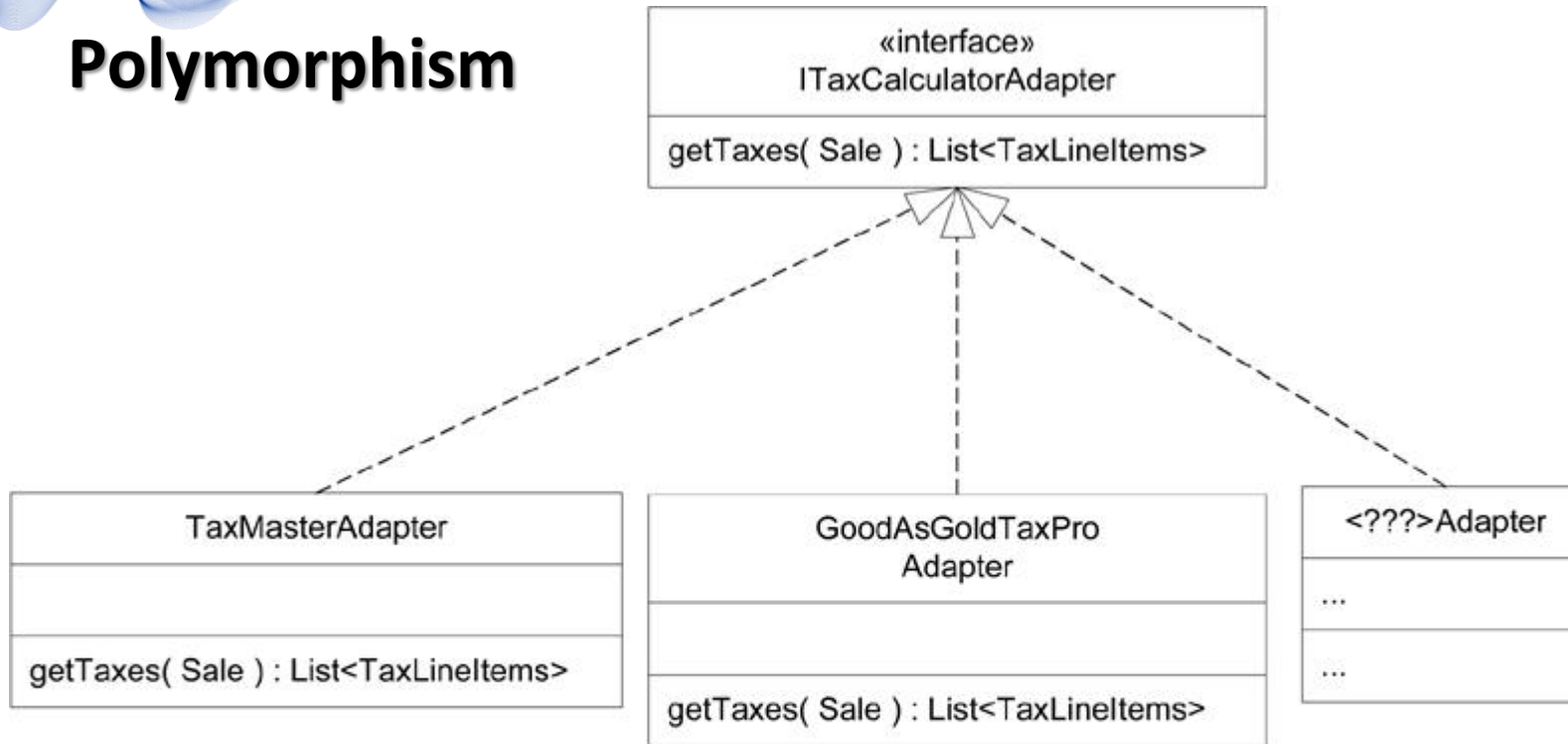
## **Polymorphism– Polimorfismo**

**Problema:** *Como tratar alternativas baseadas no tipo? Como criar componentes de software "plugáveis"?*

**Solução:** *Quando alternativas ou comportamentos relacionados variam com o tipo (classe), atribua as responsabilidades aos tipos usando operações polimórficas.*

# Padrões de Projeto / GRASP

## Polymorphism



By Polymorphism, multiple tax calculator adapters have their own similar, but varying behavior for adapting to different external tax calculators.

## **Pure Fabrication – Pura Invenção**

**Problema:** *Que objeto deve ter a responsabilidade quando você não quer violar "Alta Coesão" e "Baixo Acoplamento", mas as soluções oferecidas pelo "Especialista" não são apropriadas?*

**Solução:** *Atribua um conjunto coeso de responsabilidades a uma classe artificial que não representa um conceito no domínio da aplicação, uma classe fictícia que possibilite alta coesão, baixo acoplamento e o reuso.*

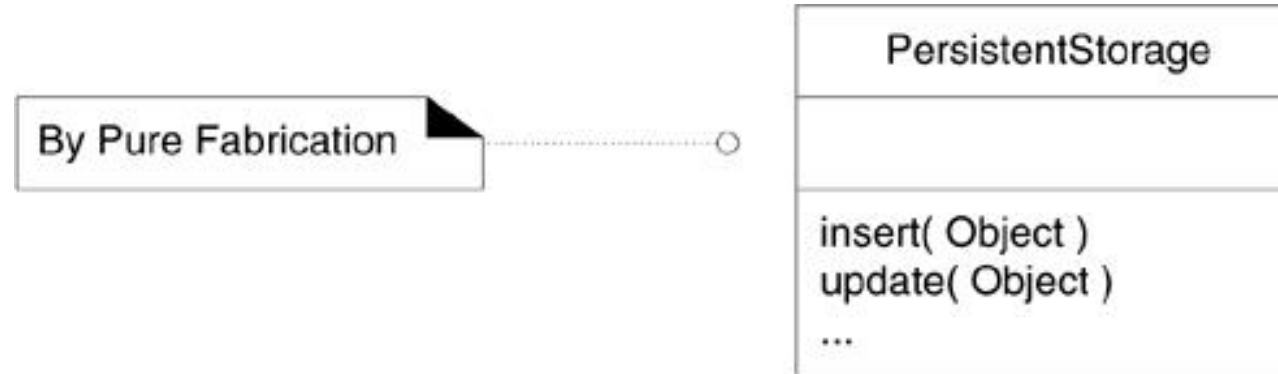


## **Pure Fabrication – Pura Invenção**

*O especialista nos diz para atribuir a responsabilidade à classe Venda, uma vez que ela conhece os dados da venda. Considere no entanto as seguintes implicações:*

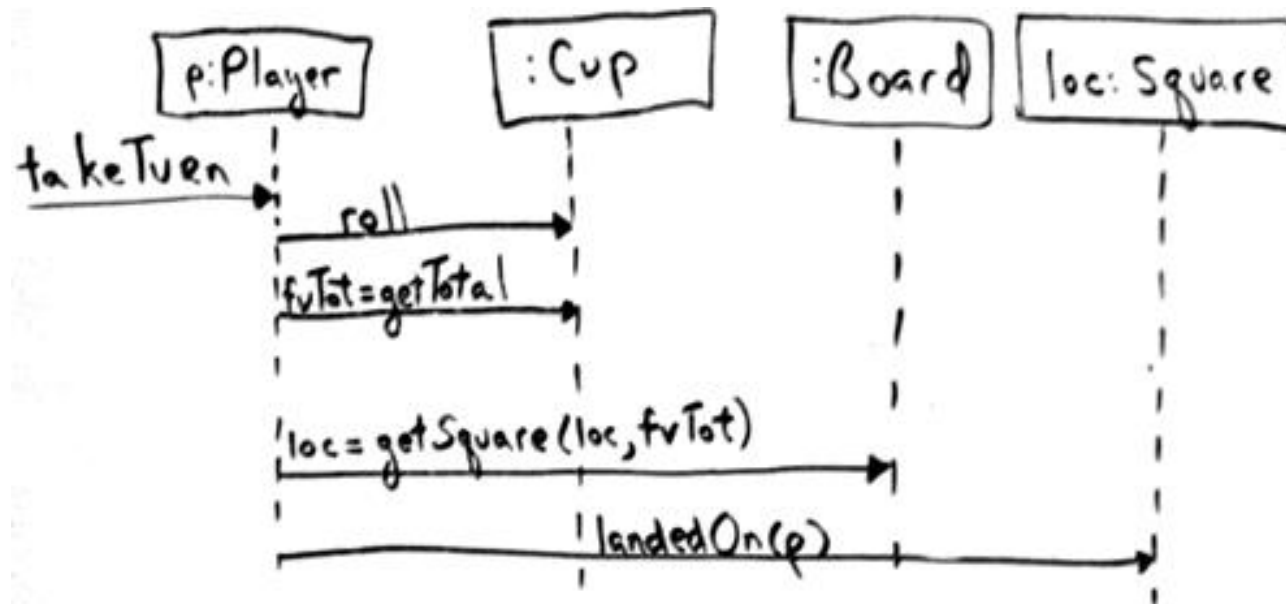
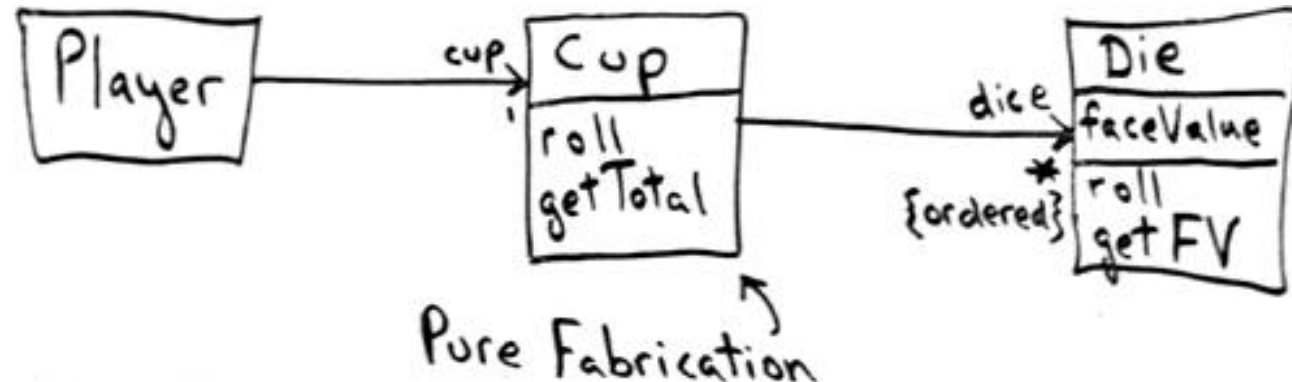
- Salvar um objeto no Banco de Dados implica em uma série de operações não relacionadas ao conceito de venda*
- A classe venda tem de ser associada à interface do banco de dados relacional (JDBC, por exemplo)*
- Várias outras classes no projeto terão de fazer a mesma coisa.*

## **Pure Fabrication – Pura Invenção**



# Padrões de Projeto / GRASP

## Pure Fabrication – Pura Invenção



## **Indirection – Indireção**

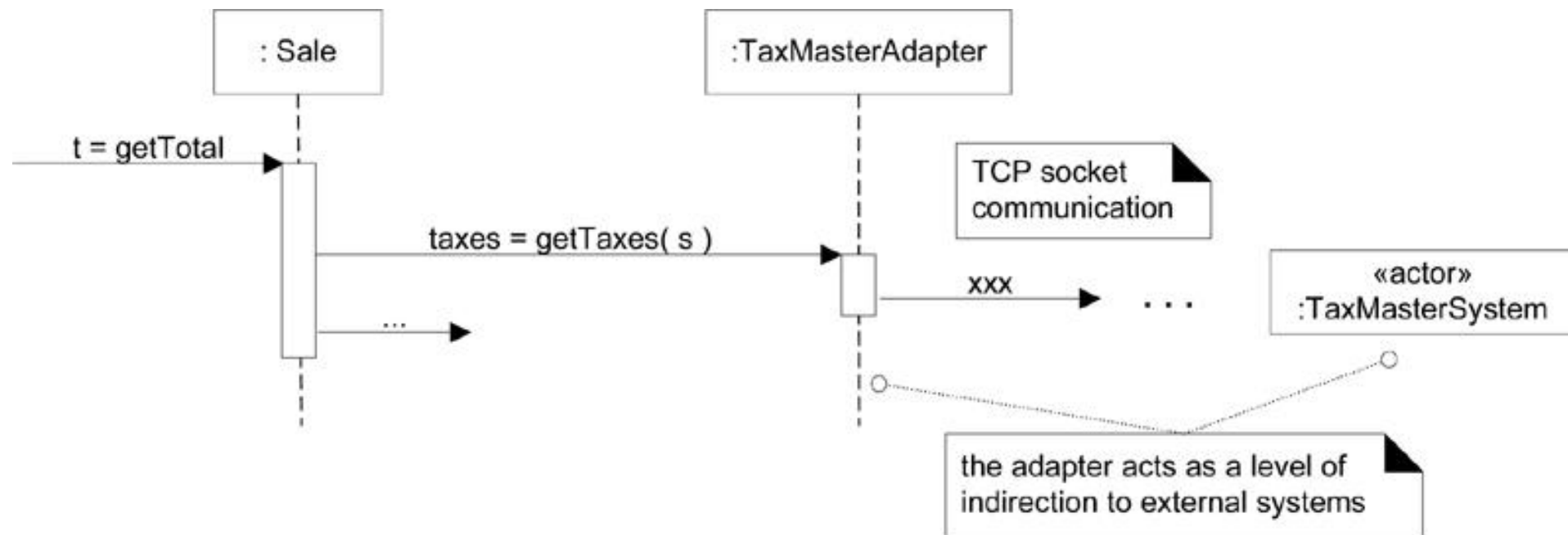
**Problema:** *Onde colocar uma responsabilidade de modo a evitar o acoplamento direto entre duas ou mais classes? Como desacoplar objetos de modo a possibilitar o baixo acoplamento e manter alta a possibilidade de reuso?*

**Solução:** *Atribua a responsabilidade a um objeto intermediário que faça a mediação entre componentes ou serviços de modo que eles não sejam diretamente acoplados.*

# Padrões de Projeto / GRASP

## Indirection – Indireção

*Exemplo: Indireção através de um adaptador*



## **Indirection – Indireção**

*"A maior parte dos problemas em Ciência da Computação pode ser resolvida por um nível adicional de indireção"*

*Velho provérbio com especial relevância para sistemas orientados a objetos*

*"A maior parte dos problemas de desempenho pode ser resolvida removendo-se algumas camadas de indireção"*



## *Padrões de Projeto / Um exemplo*

Imagine um sistema de estacionamento que tenha diversos critérios para o cálculo do valor a ser cobrado. Veículos de passeio calcula-se R\$ 2,00 a hora, mas se o tempo for maior que 12h cobra-se uma diária, e caso o período for maior que 15 dias cobra-se um mês. Há regras diferentes para caminhões, dependendo do número de eixos e para veículos de passageiros.





Por hoje é  
só  
Pessoal!