Padrões GRASP

Marum Simão Filho

marumsimao@gmail.com

Agenda

- Responsabilidades
- Princípios e Padrões
- Padrões GRASP Básicos
- Padrões GRASP Avançados

Responsabilidades

- A **qualidade** de um projeto orientado a objetos está fortemente relacionada com a **distribuição de responsabilidades.**
- Segundo Booch e Rumbaugh, “Responsabilidade é um contrato ou obrigação de um tipo ou classe.”
- Responsabilidades são atribuídas aos objetos durante o *design.*

Responsabilidades

- As responsabilidades de um projeto podem ser divididas em “conhecer” e “fazer”.
  - As responsabilidades do tipo “conhecer” (*knowing*) estão relacionadas à **distribuição das características** do sistema entre as classes**:**
    - Sobre dados privativos e encapsulados; sobre objetos relacionados; sobre coisas que pode calcular ou derivar.
  - As responsabilidades do tipo “fazer” (*doing*) estão relacionadas com a **distribuição do comportamento** do sistema entre as classes:
    - Fazer alguma coisa em si mesmo; iniciar uma ação em outro objeto; controlar e coordenar atividades em outros objetos.

Responsabilidades

- A **principal característica** da atribuição de responsabilidades está em **não sobrecarregar os objetos** 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.

Responsabilidades

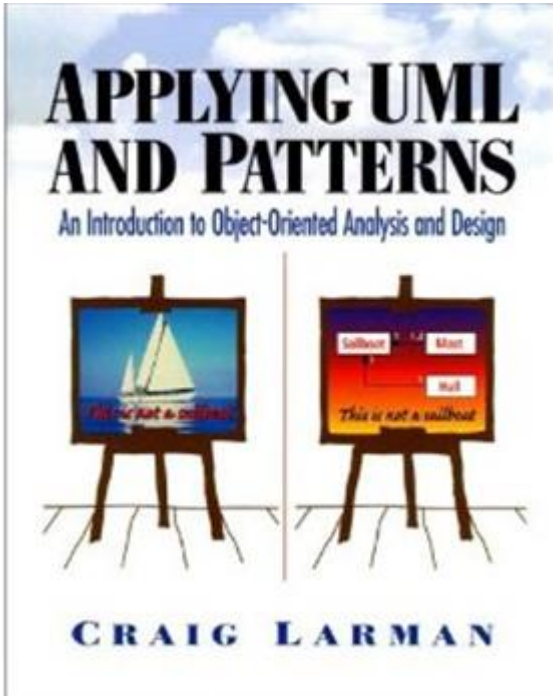
- A tradução de responsabilidades em classes e métodos depende da granularidade da responsabilidade.
- **Métodos** são implementados para cumprir **responsabilidades**.
  - Uma responsabilidade pode ser cumprida por um único método ou uma coleção de métodos trabalhando em conjunto.
- Responsabilidades do tipo *knowing* geralmente são inferidas a partir do modelo conceitual (são os atributos e relacionamentos).

Princípios X Padrões

- Os **princípios de projeto** fornecem os fundamentos necessários para o entendimento **do 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**.

Padrões GRASP

- Introduzidos por **Craig Larman** em seu livro "**Applying UML and Patterns**".
- **GRASP: G**eneral **R**esponsibility and **A**ssignment **S**oftware **P**atterns
  - Padrões de Software Gerais para Atribuição de Responsabilidade
- Descrevem os princípios fundamentais da atribuição de responsabilidades a objetos, expressos na forma de padrões.
- Ajudam a compreender melhor a utilização de vários conceitos do paradigma OO em projetos mais complexos.



Padrões GRASP

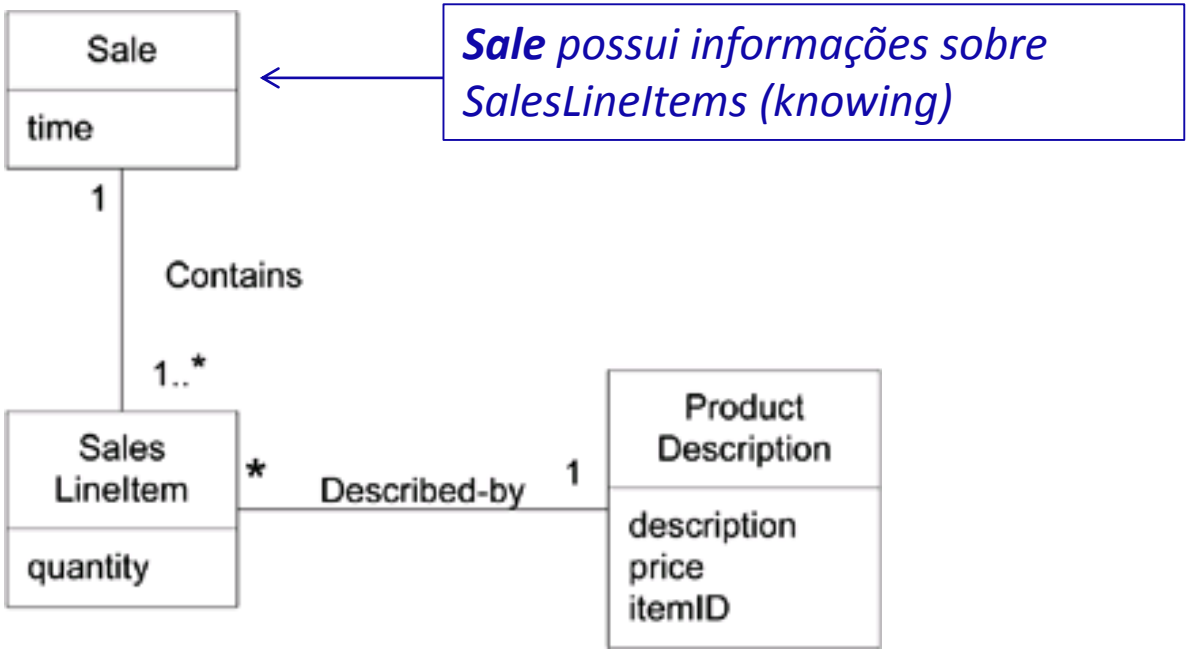
- Padrões Básicos
  - *Information Expert*
  - *Creator*
  - *High Cohesion*
  - *Low Coupling*
  - *Controller*
- Padrões Avançados
  - *Polymorphism*
  - *Pure Fabrication*
  - *Indirection*
  - *Protected Variations*

Information Expert (Especialista da Informação)

- Problema:
  - Precisa-se de um princípio geral para atribuir responsabilidades a objetos.
  - Durante o design, quando são definidas interações entre objetos, **como selecionamos quais responsabilidades devem estar em quais classes?**
- Solução:
  - **Atribuir uma responsabilidade ao especialista da informação**: a classe que possui a informação necessária para satisfazer a responsabilidade.
  - Comece a atribuição de responsabilidades ao declarar claramente a responsabilidade.

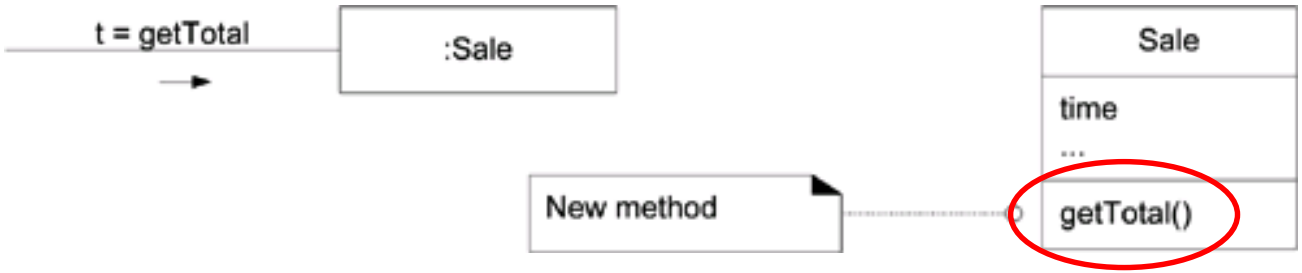
Information Expert (Especialista da Informação)

- Exemplo:
  - Na situação abaixo, qual classe deveria ser a responsável por calcular o *total geral* de uma venda (*Sale*)?



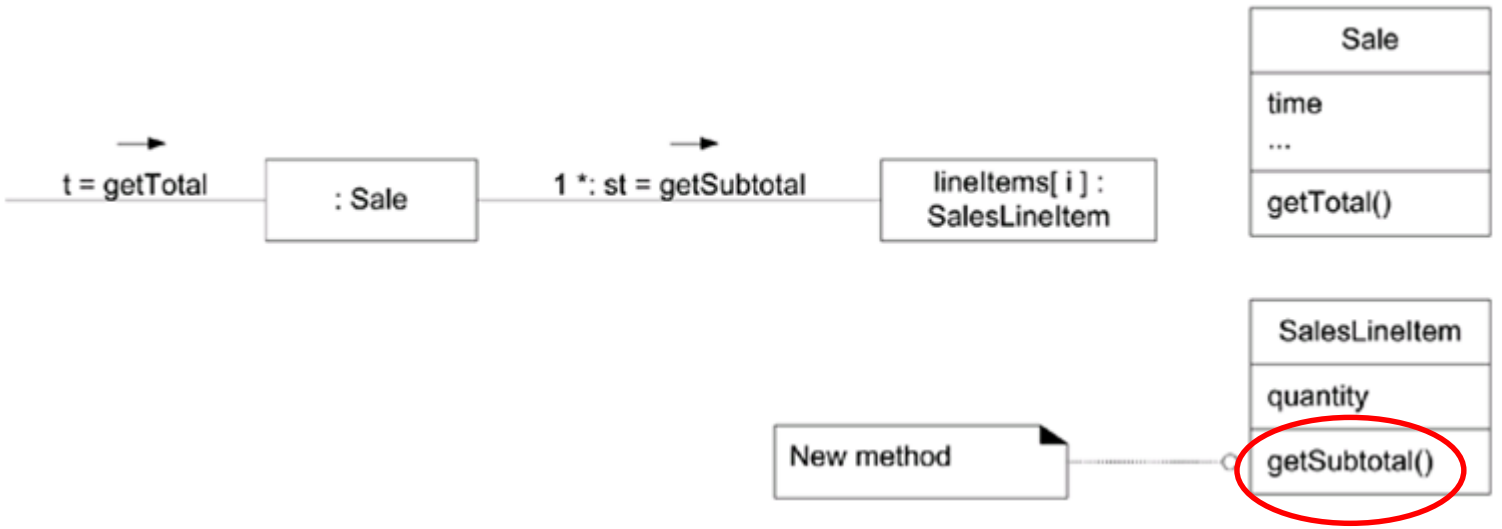
Information Expert (Especialista da Informação)

- Exemplo:
  - Um novo método é criado para a classe ***Sale***.
  - A nova responsabilidade é representada por uma operação no diagrama de interação.



Information Expert (Especialista da Informação)

- Exemplo:
  - Mas como a classe *Sale* vai calcular o valor total?
    - Somando os **subtotais**. Mas quem gera os **subtotais**?
  - A responsabilidade para cada subtotal é atribuída a ***SalesLineItem*** (item de linha do pedido).

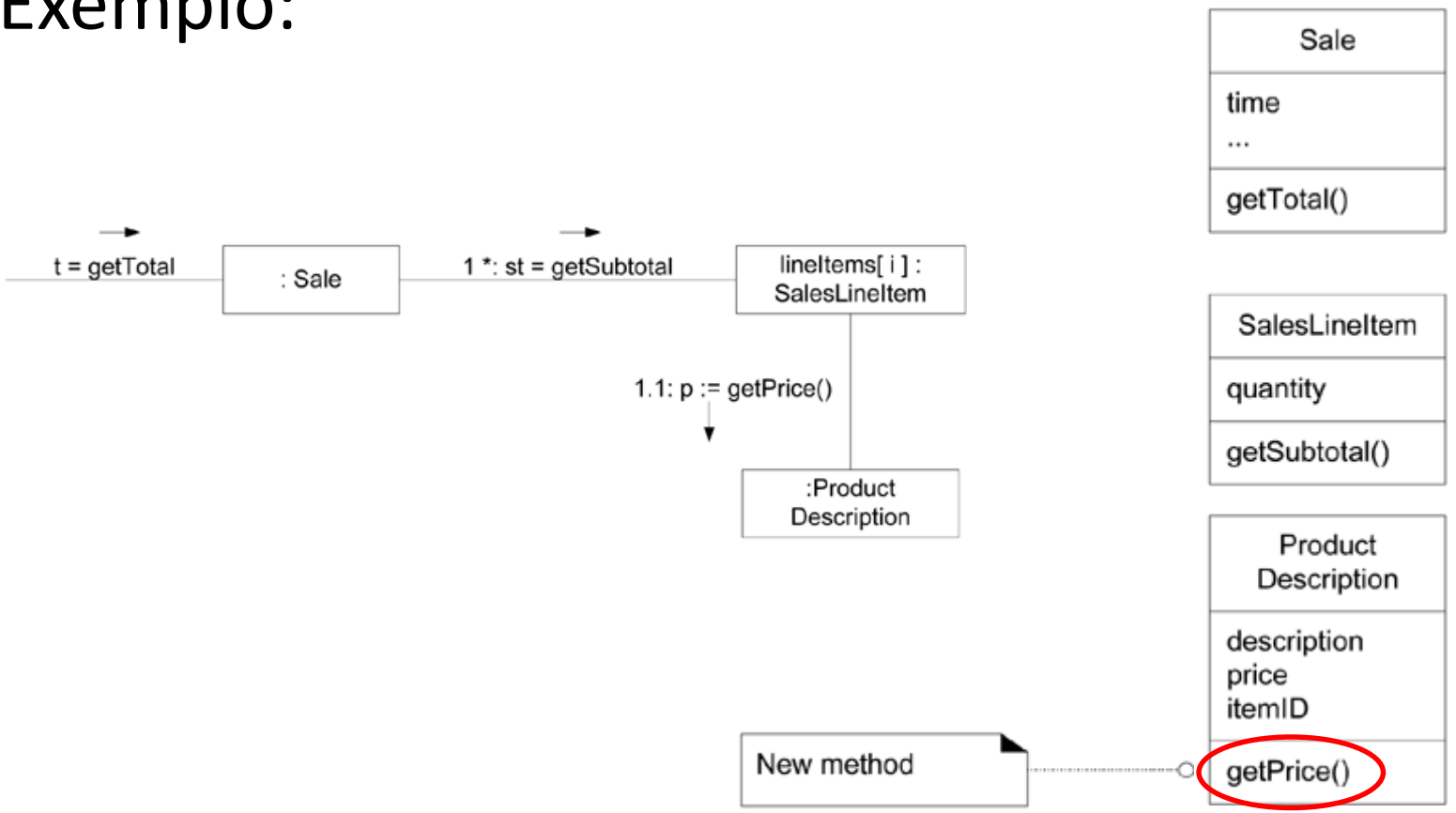


Information Expert (Especialista da Informação)

- Exemplo:
  - O **subtotal** depende do **preço** do produto.
  - Mas quem deve ser responsável por fornecer o preço de cada item de venda?
  - *ProductSpecification* é o especialista que conhece o preço, portanto a responsabilidade é dele.

Information Expert (Especialista da Informação)

- Exemplo:



Information Expert (Especialista da Informação)

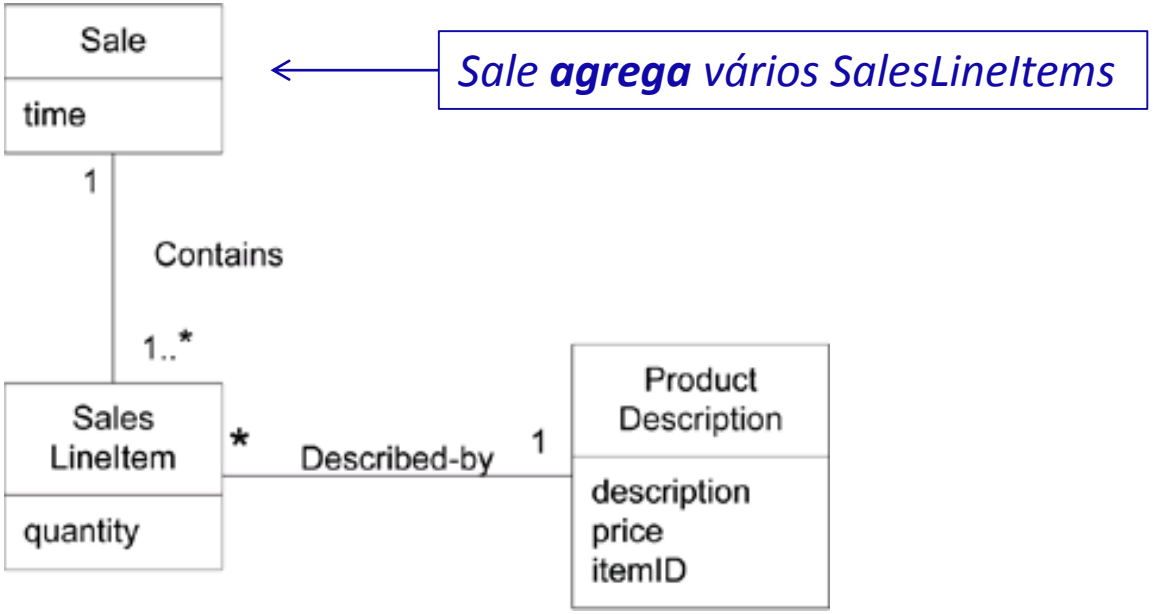
- Benefícios:
  - Conduz a projetos onde o objeto de software faz o que o objeto real faria.
  - Mantém o encapsulamento e não aumenta o acoplamento, pois utiliza informações próprias.
  - Distribui o comportamento uniformemente entre as classes do sistema, aumentando a coesão das mesmas.
- Contraindicações:
  - Em algumas situações, a solução sugerida pelo especialista pode ser indesejada. Por exemplo, quem deve persistir uma venda no banco de dados?

Creator (Criador)

- Problema:
  - A criação de objetos é uma atividade comum em sistemas OO.
  - **Quem deveria ser responsável pela criação de uma nova instância de uma classe?**
- Solução:
  - Atribua à classe A a responsabilidade de criar instâncias da classe B se:
    - A contém objetos de B
    - A agrega objetos de B
    - A registra objetos de B
    - A usa objetos de B
    - A tem os dados necessários para a inicialização de objetos de B.

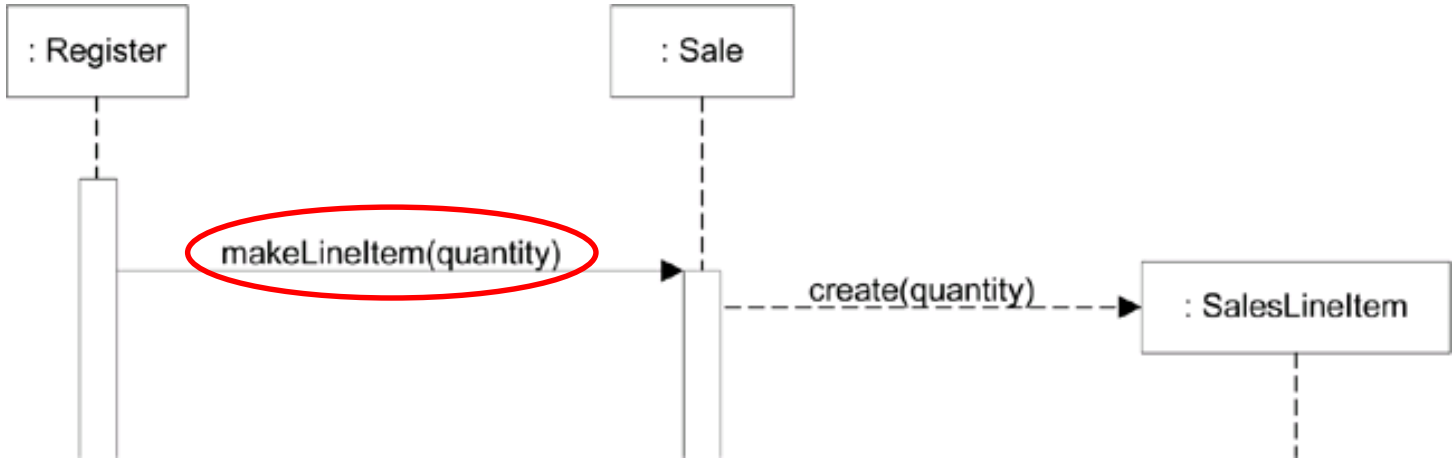
Creator (Criador)

- Exemplo:
  - Que classe deve ser responsável por criar uma instância do objeto *SalesLineItem* abaixo?



Creator (Criador)

- Exemplo:
  - A nova responsabilidade é traduzida em uma operação em um diagrama de sequência.
  - Um novo método é criado na classe para expressar isto.



Creator (Criador)

- Benefícios:
  - Não aumenta o acoplamento, pois a visibilidade entre as classes envolvidas já existia.

- Contraindicações:
  - Criação de objetos complexos (há padrões mais elaborados para isso).

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,
    - mais vulneráveis à mudança.
- Solução:
  - Atribuir uma responsabilidade para que a coesão se mantenha alta.

High Cohesion (Alta Coesão)

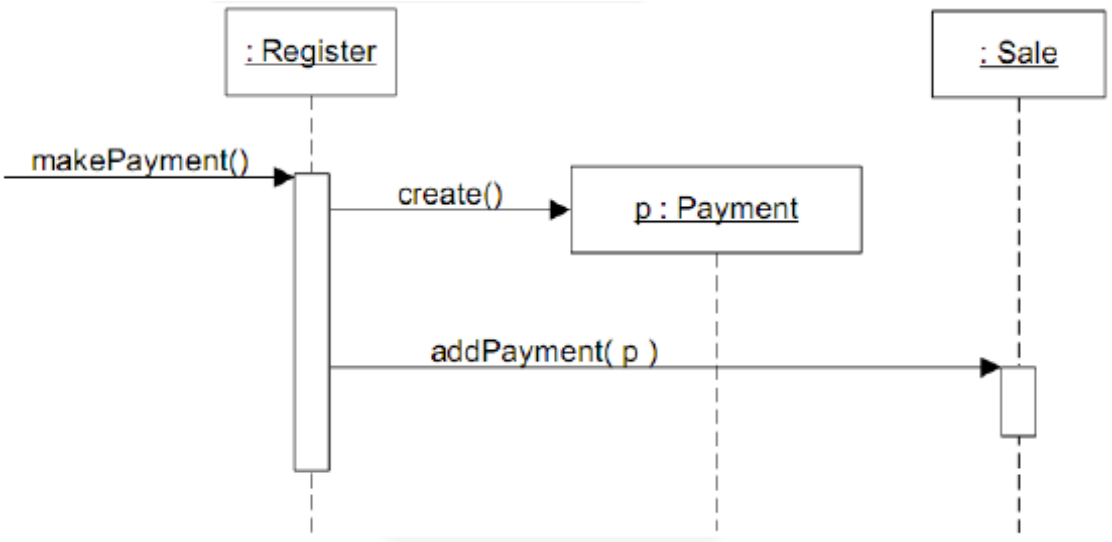
- Coesão é o que mesmo?
  - Uma medida de quão **relacionadas** ou **focadas** estão as responsabilidades de um elemento.
- Exemplo:
  - Uma classe Cão é coesa se tem operações relacionadas ao Cão (morder, correr, comer, latir)
  - e apenas ao Cão (não terá, por exemplo, validar, imprimirCao, listarCaes).
- Alta coesão promove o ***design* modular**.

High Cohesion (Alta Coesão)

- Cenários
  - **Coesão muito baixa**: uma única classe é responsável por muitas coisas em áreas funcionais diferentes.
  - **Coesão baixa**: uma classe é a única com responsabilidades de uma tarefa complexa.
  - **Coesão alta**: a classe tem responsabilidades moderadas em uma área funcional e colabora com outras classes para cumprir as tarefas.

High Cohesion (Alta Coesão)

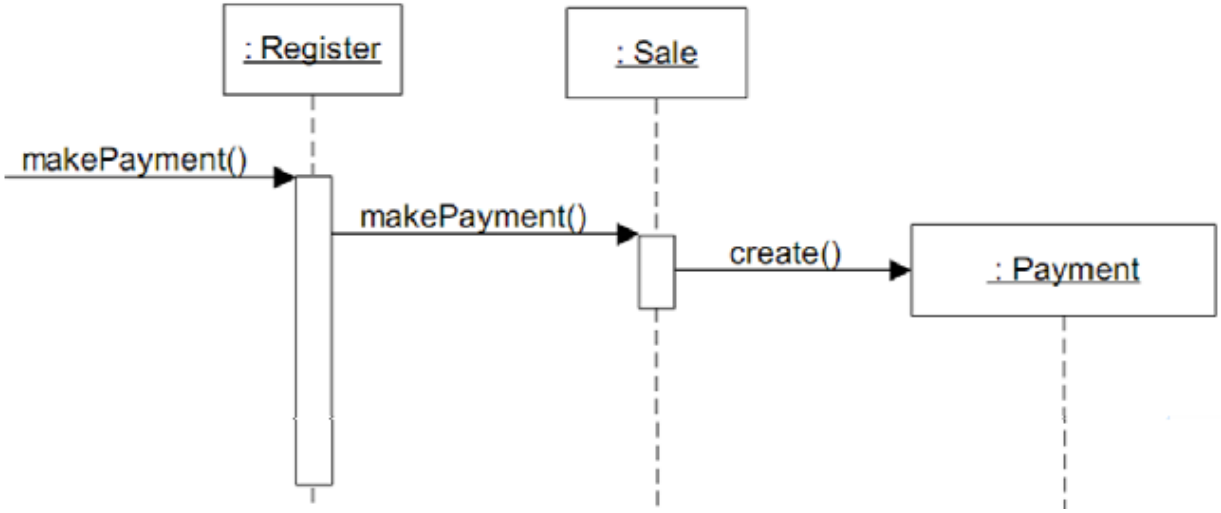
- Exemplo:
  - Que classe é responsável por criar um pagamento (*Payment*) e associá-lo a uma venda (*Sale*)?



Fazer um pagamento não é responsabilidade de Registrar. *Register* assumiu responsabilidade por uma coisa que é parte de *Sale*.

High Cohesion (Alta Coesão)

- Exemplo:
  - Numa solução melhorada, *Register* delega a responsabilidade a *Sale*, aumentando a coesão de *Register*.

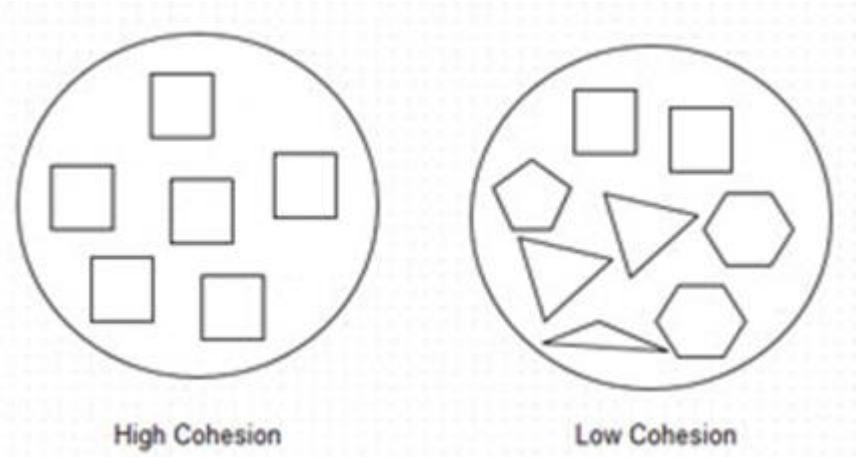


A criação do processo de pagamento agora é responsabilidade da venda e não do registro. Faz mais sentido pois o pagamento é parte de *Sale*.

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 executa muito trabalho.

High Cohesion (Alta Coesão)



High Cohesion (Alta Coesão)

- Benefícios:
  - Clareza e facilidade de compreensão do projeto;
  - Simplificação das atividades de manutenção;
  - Favorecimento do baixo acoplamento;
  - Facilidade de reutilização, uma vez que a classe é bem específica.
- Alguns casos de coesão moderada trazem benefícios, como, por exemplo, o padrão *Facade.*

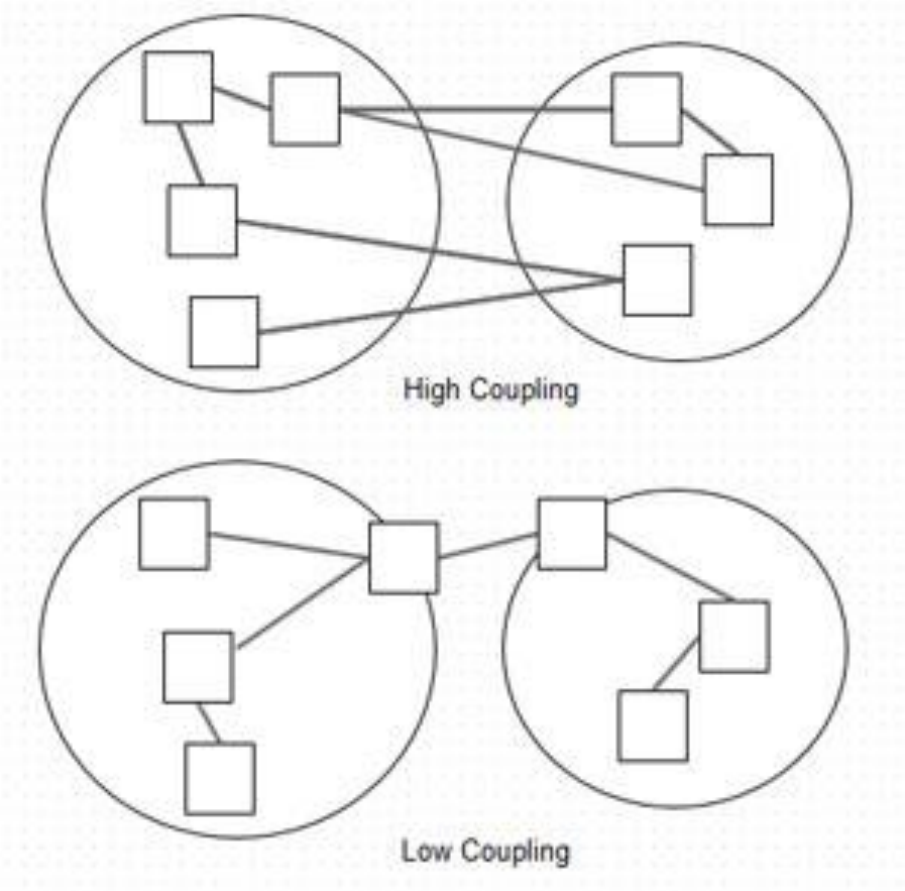
Low Coupling (Baixo Acomplamento)

- Problema:
  - Como prover baixa dependência entre classes, reduzir o impacto de mudanças e obter alta reutilização?
- Solução:
  - Atribuir as responsabilidades de modo que o **acoplamento** entre classes permaneça **baixo**.

Low Coupling (Baixo Acomplamento)

- Acoplamento é o que mesmo?
  - 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: tais classes podem ser indesejáveis.
  - O acoplamento está associado à coesão: quanto maior a coesão, menor acoplamento, e vice-versa.

Low Coupling (Baixo Acomplamento)

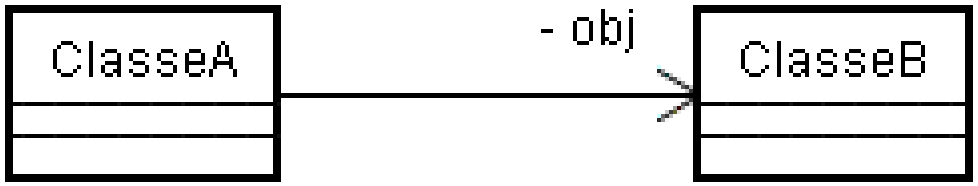


Low Coupling (Baixo Acomplamento)

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

Low Coupling (Baixo Acomplamento)

- A ClasseA tem um atributo do tipo ClasseB

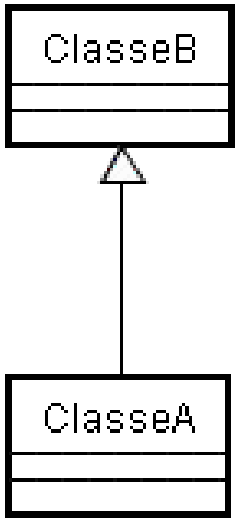



```
public class ClasseA {
    ClasseB obj;

    // ...
}
```

Low Coupling (Baixo Acomplamento)

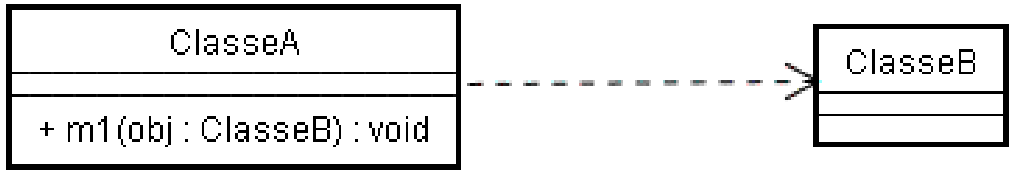
- A ClasseA é uma subclasse de ClasseB




```
public class ClasseA extends ClasseB    {
    // ...
}
```

Low Coupling (Baixo Acomplamento)

- A ClasseA tem um método que, de alguma forma, referencia uma instância de ClasseB, seja através de um **parâmetro**, **variável local** ou **retorno**.

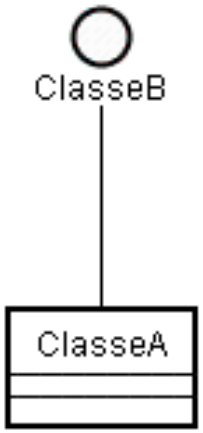



```
public class ClasseA {
    public void m1(ClasseB obj)    {
        // ...
    }
}
```

```
public class ClasseA {
    public void m1()    {
        ClasseB obj = new ClasseB();
        // ...
    }
}
```

Low Coupling (Baixo Acomplamento)

- A ClasseB é uma interface e a ClasseA implementa esta interface



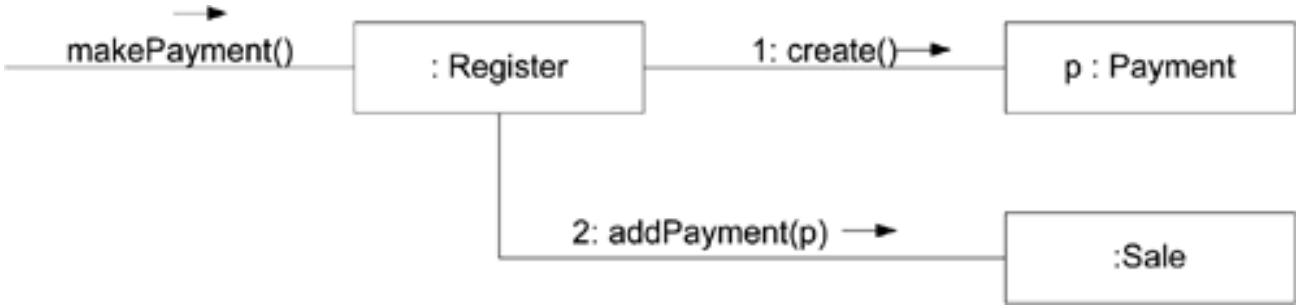

```
public class ClasseA implements ClasseB     {
    // ...
}
```

Low Coupling (Baixo Acomplamento)

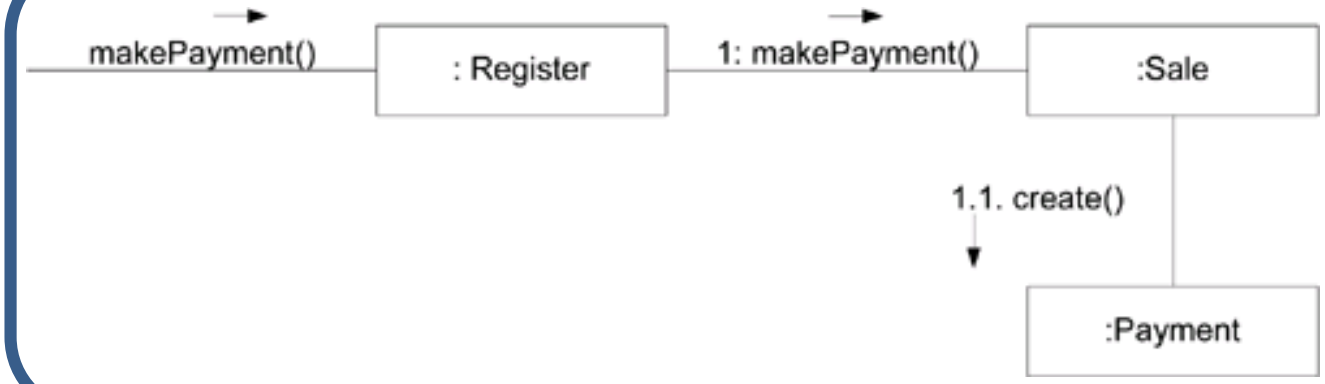
- Exemplo:
  - Suponha que temos que criar um objeto **pagamento** e associá-lo à **venda**. Que classe deve ser responsável por isso?

Low Coupling (Baixo Acomplamento)

- Exemplo:
  - Qual das soluções provê o menor acoplamento?



1ª alternativa



2ª alternativa

Low Coupling (Baixo Acomplamento)

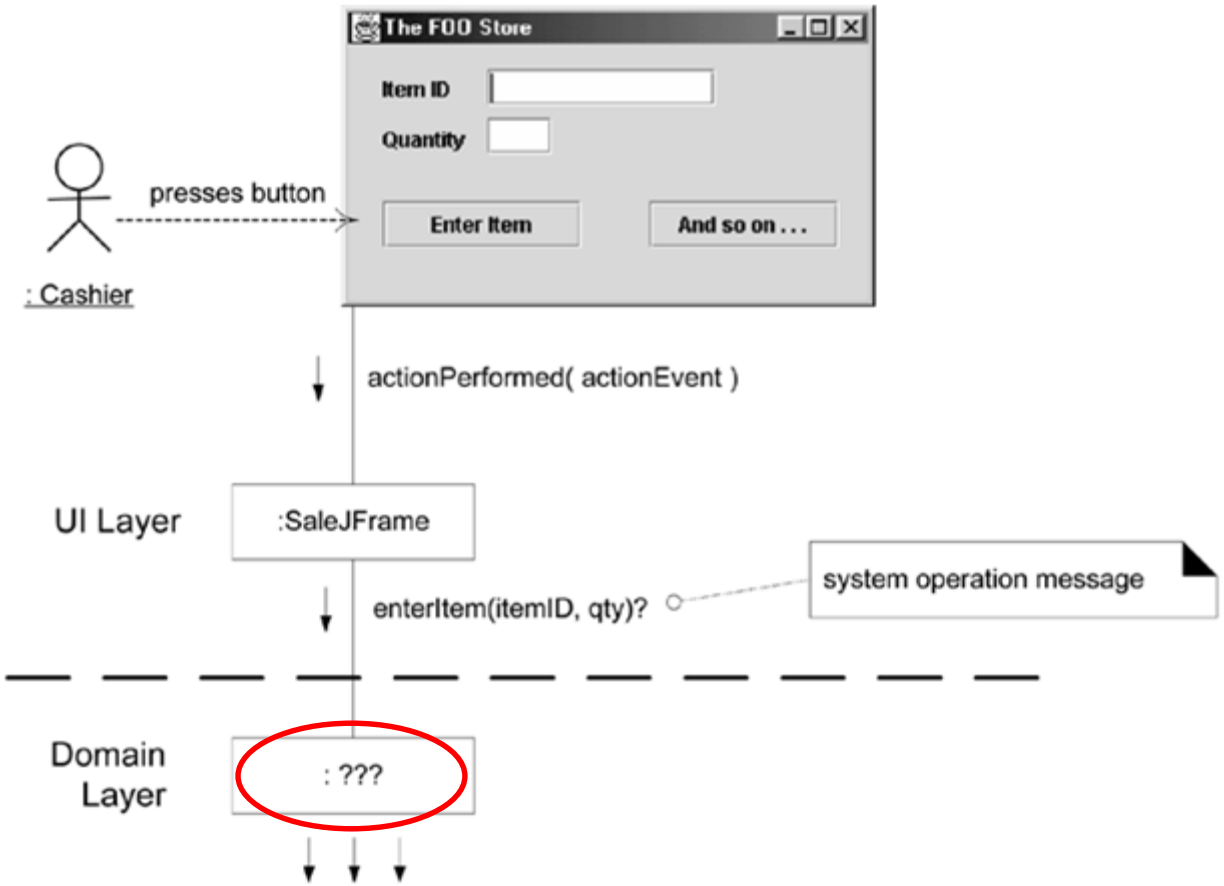
- Classes que, por natureza, são genéricas e que têm alta probabilidade de reutilização deveriam ter acoplamento baixo.
- O caso extremo do baixo acoplamento é o não acoplamento: contraria o princípio da orientação a objetos, qual seja, objetos conectados, trocando mensagens entre si.
- O acoplamento 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.

Controller (Controlador)

- Problema:
  - Quem deve ser o responsável por tratar um evento de uma interface de entrada?
- Solução:
  - Atribuir responsabilidades para receber ou tratar um evento do sistema para:
    - uma classe que representa todo o sistema ou subsistema (*facade controller*);
    - uma classe que representa um cenário de caso de uso (controlador de caso de uso ou de sessão).

Controller (Controlador)

- Exemplo:



Controller (Controlador)

- Exemplo:
  - Utilize uma classe com papel de controlador, seja do sistema ou do caso de uso.
  - Essa classe normalmente não executa trabalho algum, apenas delega para outras classes executarem.

Controller (Controlador)

- 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 mais indicado.

Controller (Controlador)

- Benefícios:
  - Diminui a sensibilidade da camada de apresentação em relação à lógica de negócio.
  - Facilita a reutilização de componentes específicos de negócio.

Padrões GRASP

- Padrões Básicos
  - *Information Expert*
  - *Creator*
  - *High Cohesion*
  - *Low Coupling*
  - *Controller*
- Padrões Avançados
  - *Polymorphism*
  - *Pure Fabrication*
  - *Indirection*
  - *Protected Variations*

Polymorphism (Polimorfismo)

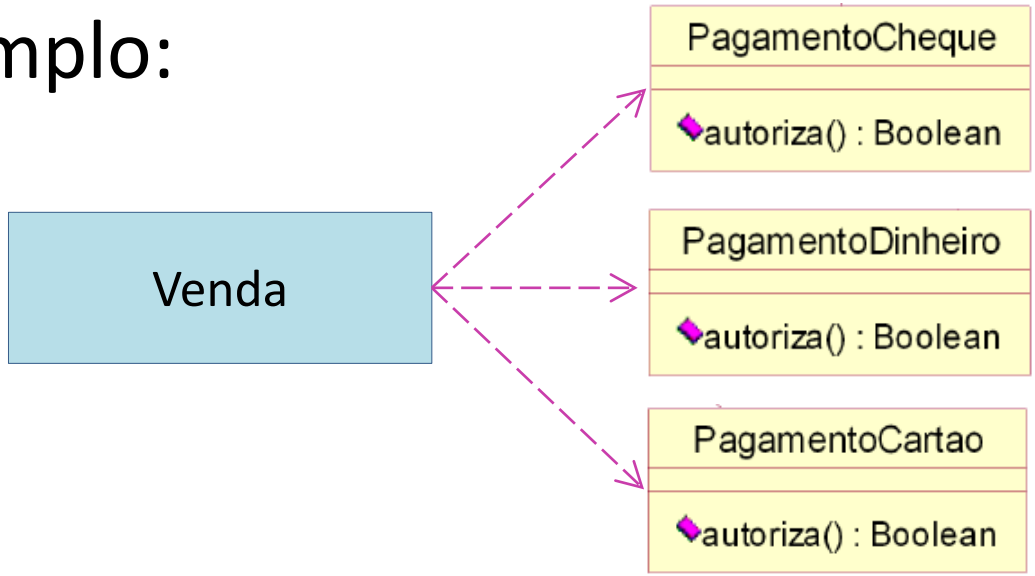
- Problema:
  - Como lidar com alternativas baseadas no tipo?
  - Como criar componentes de software plugáveis?
  - Deseja-se evitar variação condicional (if-then-else): pouco extensível.
  - Deseja-se substituir um componente por outro sem afetar o cliente.

Polymorphism (Polimorfismo)

- Solução
  - Não use lógica condicional para realizar alternativas diferentes baseadas em tipo.
  - Atribua responsabilidades ao comportamento usando operações polimórficas.
  - Refatore!

Polymorphism (Polimorfismo)

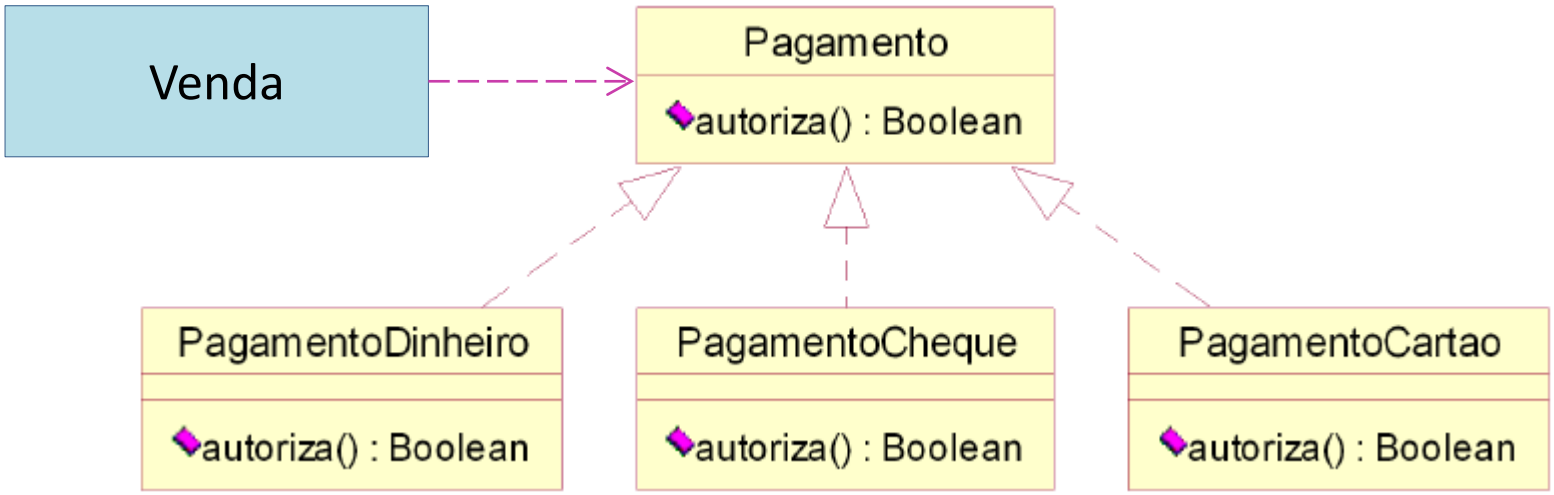
- Exemplo:




```
class Venda {
    void pagar (int tipo) {
      if ( tipo == 1 )
          (new PagamentoDinheiro() ).autoriza();
      else if ( tipo == 2 )
          (new PagamentoCheque() ).autoriza();
      else if ( tipo == 3 )
          (new PagamentoCartao() ).autoriza();
      ...
    }
}
```

Polymorphism (Polimorfismo)

- Exemplo:




```
class Venda {
    void pagar (Pagamento pagamento) {
          pagamento.autoriza();
          ...
    }
}
```

Pure Fabrication (Pura Invenção)

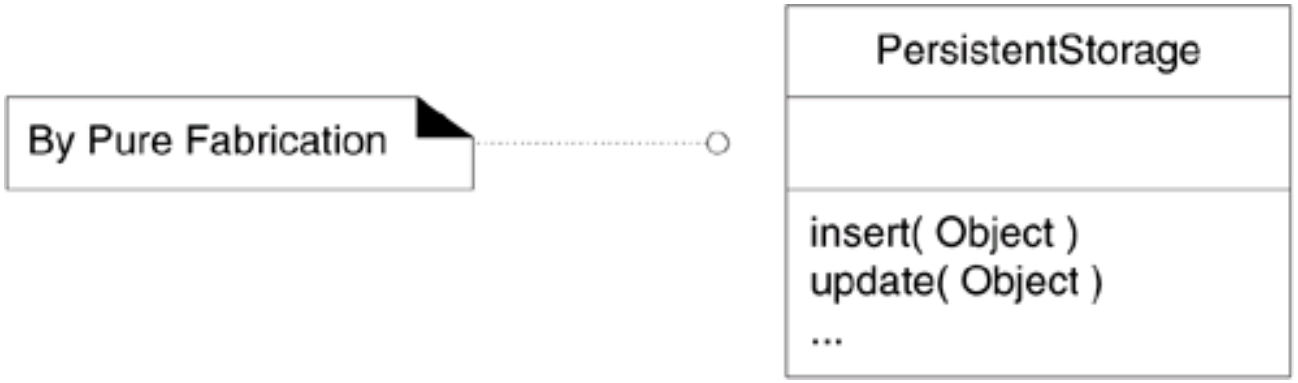
- Problema:
  - Que objeto deve ter a responsabilidade, quando você não quer violar *High Cohesion* e *Low Coupling*, mas as soluções oferecidas por *Expert* não são adequadas?
  - Atribuir responsabilidades apenas para classes do domínio conceitual pode levar a situações de maior acoplamento e menos coesão.

Pure Fabrication (Pura Invenção)

- Solução
  - Atribuir um conjunto altamente coesivo de responsabilidades a uma classe **artificial** que não representa um conceito do domínio do problema.

Pure Fabrication (Pura Invenção)

- Exemplo:
  - Apesar de *Sale* ser a candidata lógica para ser a *Expert* para salvar a si mesma em um banco de dados, isto levaria o projeto a ter alto acoplamento, baixa coesão e baixo reuso.
  - Uma solução seria criar uma classe responsável somente por isto.



Protected Variations
(Variações Protegidas)

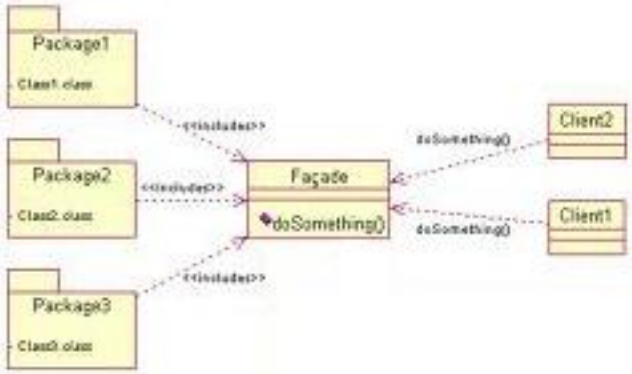
- Problema:
  - Como projetar objetos, subsistemas e sistemas para que as variações ou instabilidades nesses elementos não tenham um impacto indesejável nos outros elementos?

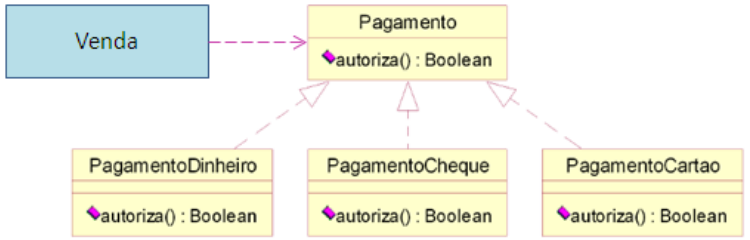
Protected Variations
(Variações Protegidas)

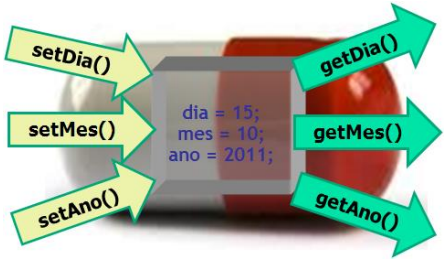
- Solução:
  - Identificar pontos de variação ou instabilidade potenciais.
  - Atribuir responsabilidades para criar uma interface estável em volta desses pontos.
  - Evite enviar mensagens a objetos muito distantes.
  - Encapsulamento, interfaces, polimorfismo, indireção, padrões, máquinas virtuais e *brokers* são motivados por este princípio.

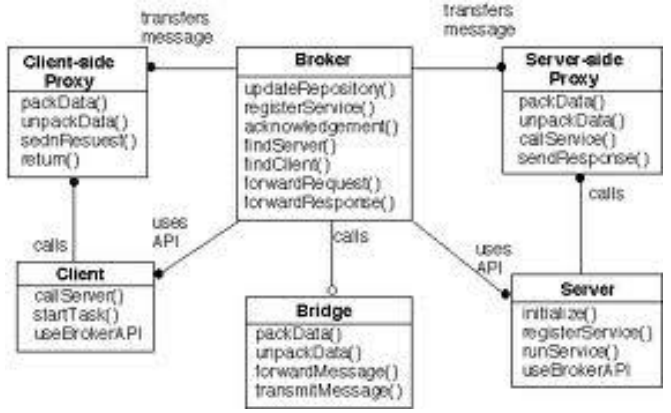
Protected Variations
(Variações Protegidas)

- Exemplos:









Indirection (Indireção)

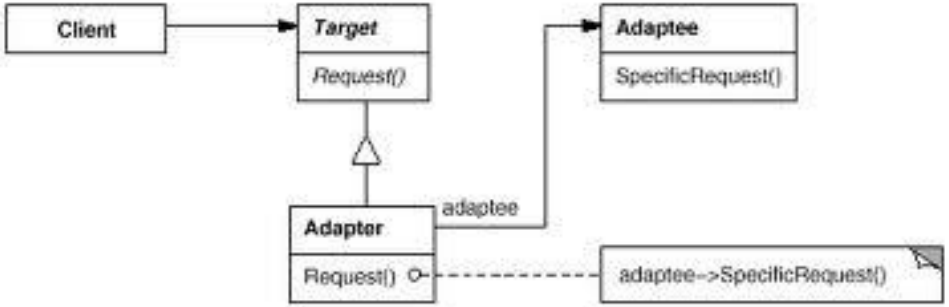
- Problema:
  - Onde atribuir uma responsabilidade para evitar acoplamento direto entre duas ou mais coisas?
  - Como desacoplar objetos para que seja possível suportar baixo acoplamento e manter elevado o potencial de reuso?

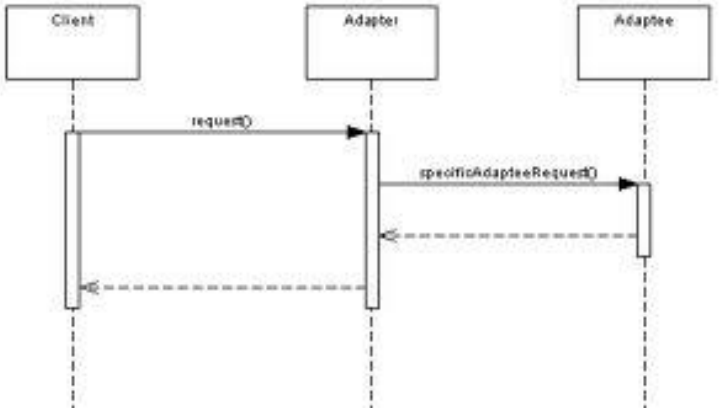
Indirection (Indireção)

- Solução:
  - Atribua a responsabilidade a um **objeto intermediário** para mediar as mensagens entre outros componentes ou serviços para que não sejam diretamente acoplados.
  - O objeto intermediário cria uma camada de **indireção** entre os dois componentes, que não mais dependem um do outro: agora ambos dependem da indireção.

Indirection (Indireção)

- Exemplo:
  - A comunicação entre duas classes pode ser intermediada por um **adaptador**, que se torna responsável pela **indireção**.





"A maioria dos problemas em ciência da computação podem ser resolvidos por um outro nível de indireção." Dijkstra






Referências

- **Utilizando UML e Padrões – Uma Introdução à Análise e ao Projeto Orientados a Objetos**. Craig Larman. Bookman.
- **Designing Object-Oriented Software**. Wirfs-Brock, Wilkerson e Wiener. Prentice Hall.
- **Padrões Para Atribuir Responsabilidades.** Jacques Philippe Sauvé. http://jacques.dsc.ufcg.edu.br/cursos/map/html/pat/expert.htm.
- **Padrões GRASP**. Leonardo Gresta Paulino Murta. Instituto de Computação – UFF.
- **Padrões de Design com Aplicações em Java**. Helder da Rocha. www.argonavis.com.br.

Padrões GRASP

Obrigado!!!

Marum Simão Filho
marumsimao@gmail.com