



General Responsibility Assignment Software Patterns

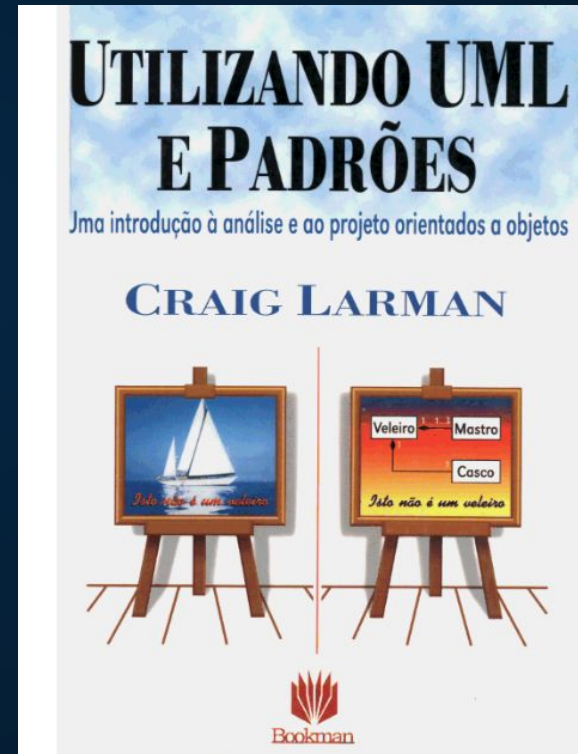
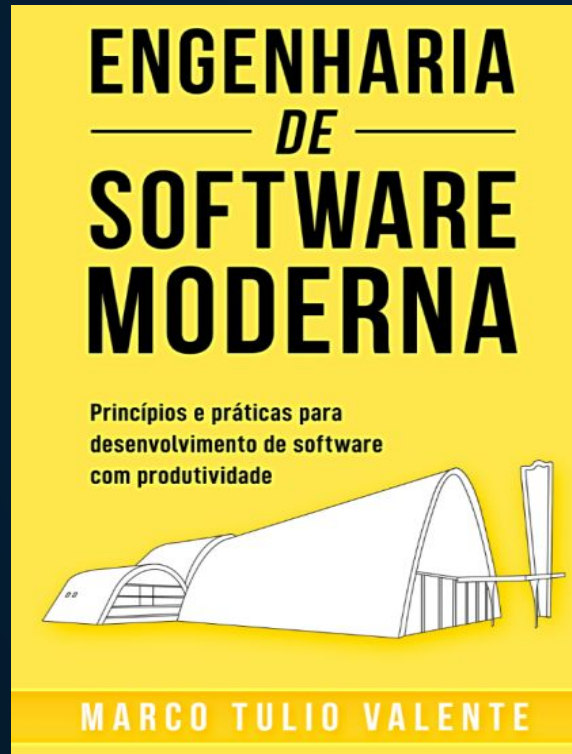



Olá

Eu sou **Willian Brito**

- ❖ Desenvolvedor FullStack na Msystem Software.
- ❖ Formado em Análise e Desenvolvimento de Sistemas.
- ❖ Pós Graduado em Segurança Cibernética.
- ❖ Certificação SYCP (Solyd Certified Pentester) v2018.

ESTE CONTEÚDO É BASEADO NESTAS OBRAS





“Os padrões **GRASP**, consistem em uma série de **princípios** baseados em conceitos para **atribuição de responsabilidades** a classes e objetos na construção de bons softwares usando programação orientada a objetos.”

O COMEÇO

Este **conjunto de princípios** definido como **GRASP (Padrões de Princípios Gerais para Atribuição de Responsabilidades)**, foram publicados originalmente no livro *“Applying UML and Patterns — An Introduction to Object-Oriented Analysis and Design and the Unified Process”* (obra traduzida em português com o título *“Utilizando UML e Padrões — Uma introdução à análise e ao projeto orientados a objetos e ao desenvolvimento iterativo”*) escrito pelo **Craig Larman em 1997**, contribuindo para a codificação de princípios de design de software, esse é um livro muito popular que contribuiu para a adoção generalizada do desenvolvimento orientado a objetos.

OBJETIVO

O conceito de **responsabilidade** deve ser compreendido como as **obrigações** que um objeto possui quando se leva em conta o seu papel dentro de um determinado **contexto**. Além disso, é preciso considerar ainda as prováveis colaborações entre diferentes objetos.

A implementação de soluções OO considerando os conceitos de responsabilidade, papéis e colaborações fazem parte de uma abordagem conhecida como **Responsibility-Driven Design** ou simplesmente **RDD**.

OBJETIVO

Responsibility-Driven Design é uma técnica de desenvolvimento OO proposta no início dos anos 1990, como resultado do trabalho dos pesquisadores **Rebecca Wirfs-Brock** e **Brian Wilkerson**. Este paradigma está fundamentado nas ideias de responsabilidades, papéis e colaborações.

Partindo de práticas consagradas no desenvolvimento de aplicações orientados a objetos, procuram definir quais as obrigações dos diferentes tipos de objetos em uma aplicação, o **GRASP** disponibiliza uma **catálogo de recomendações** que procuram favorecer um conjunto de práticas visando um software bem estruturado.

OBJETIVO

A responsabilidade é definida como um contrato ou obrigação de uma classe que está relacionada ao comportamento. Existem 2 tipos de responsabilidades:

Saber:

- O conhecimento dos dados privados que o objeto em questão encapsula.
- Saber a respeito de outros objetos relacionados.
- Saber sobre as coisas que pode derivar ou calcular.

Fazer:

- A execução de ações que condizem com o papel desempenhado por tal objeto.
- A criação de outros objetos dos quais a instância inicial depende.
- Controlando e coordenando atividades em outros objetos.

OBJETIVO

Como os outros **patterns** conhecidos, os diferentes padrões **GRASP** não devem ser encarados como soluções pré-definidas para problemas específicos. Estes patterns devem ser compreendidos como princípios que auxiliam os desenvolvedores a projetar de uma forma bem estrutura aplicações orientadas a objetos.

OBJETIVO

Esses padrões possuem a seguinte classificação:

Padrões Fundamentais:

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

Padrões Avançados:

- Polymorphism
- Pure Fabrication
- Indirection
- Demeter Law

CATÁLOGO

01

Expert

Determina quando devemos delegar a responsabilidade para um outro objeto que seja especialista naquele domínio.

04

Low Coupling

Determina que as classes não devem depender de objetos concretos e sim de abstrações, permitindo que haja mudanças sem impacto.

07

Pure Fabrication

É uma classe que não representa nenhum conceito no domínio, ela funciona como uma classe prestadora de serviços, e é projetada para que possamos ter um baixo acoplamento e alta coesão no sistema.

02

Creator

Determina qual classe deve ser responsável pela criação certos objetos.

05

High Cohesion

Este princípio determina que as classes devem se focar apenas na sua responsabilidade.

08

Indirection

Este princípio ajuda a manter o baixo acoplamento, através de delegação de responsabilidades através de uma classe mediadora.

03

Controller

Atribui a responsabilidade de lidar com os eventos do sistema para uma classe que representa a um cenário de caso de uso do sistema global.

06

Polymorphism

As responsabilidades devem ser atribuídas a abstrações e não a objetos concretos, permitindo que eles possam variar conforme a necessidade.

09

Demeter Law

Interagir apenas com objetos vizinhos imediatos e não com objetos mais distantes.



EXPERT

Information Expert (Especialista na Informação):
Determina quando devemos delegar a
responsabilidade para um outro objeto que seja
especialista naquele domínio.

EXPERT

O padrão **Expert**, é um padrão de design que busca alocar responsabilidades de uma maneira que o conhecimento necessário para realizar determinada tarefa esteja contido na classe mais apropriada para executá-la. Esse padrão está relacionado ao princípio de design conhecido como o princípio do especialista (ou o princípio de menor conhecimento), que sugere que uma classe deve conter o conhecimento necessário para realizar suas próprias operações, em vez de depender fortemente de outras classes.

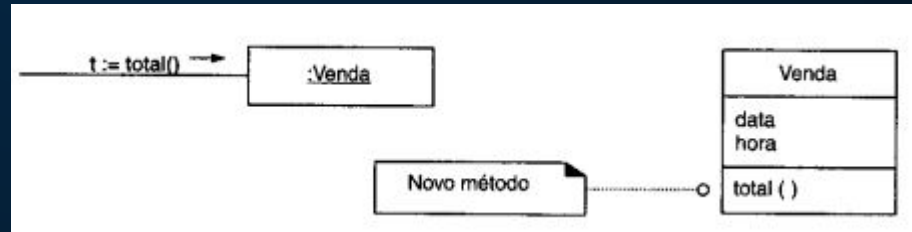
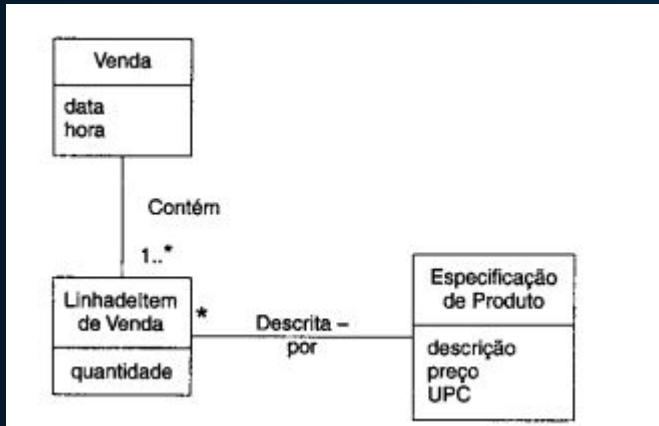
A ideia central do padrão Expert é delegar uma responsabilidade para a classe que possui o maior conhecimento necessário para cumprir essa tarefa específica. Isso significa que, em vez de dispersar o conhecimento relacionado a uma operação por várias classes, procura-se centralizá-lo naquela que naturalmente possui as informações ou métodos mais pertinentes para a realização da ação.

EXPERT

Por exemplo, considere um sistema de ponto de venda (PDV). O padrão Expert sugere que a classe responsável por uma determinada operação (como calcular o preço total de uma venda) deve possuir o conhecimento necessário para realizar essa tarefa. Nesse caso, a classe **Venda** poderia ser a especialista, contendo métodos para **adicionar itens**, **calcular o total** e **aplicar descontos**. Isso evita a necessidade de espalhar esse conhecimento por várias classes e promove um design mais coeso e fácil de manter.

EXPERT

Que informação é necessária para determinar o total de um item da venda? São necessárias “**VendaItem.quantidade**” e “**Produto.preco**”. O VendaItem conhece a quantidade e o Produto associado, portanto, de acordo com o padrão **Expert**, VendaItem deveria determinar o subtotal, ela é o **especialista** na informação.

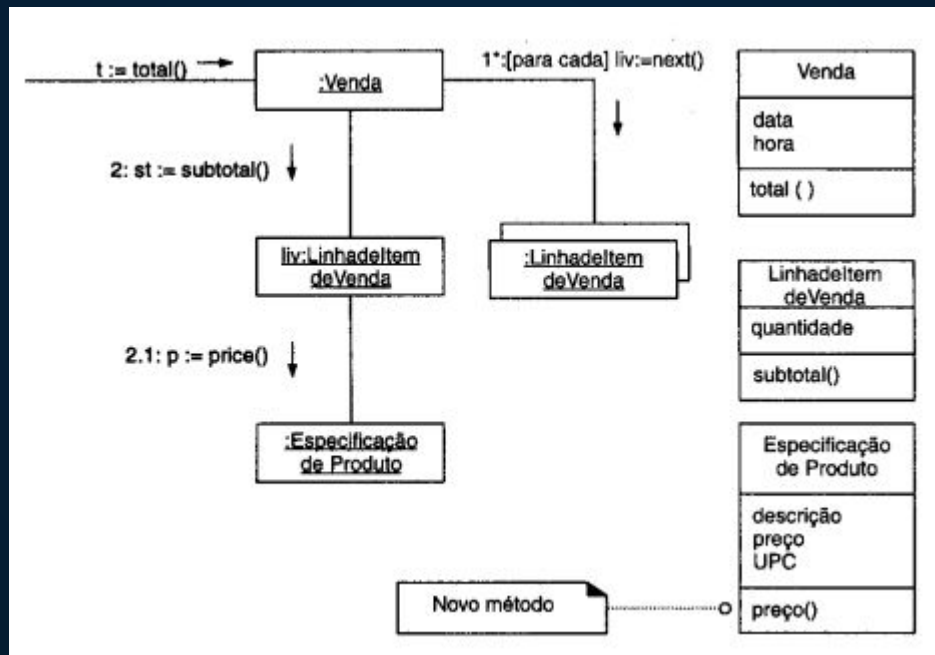


EXPERT

Em termos de design de classes, isso significa que Venda precisa delegar a busca pelo subtotal para cada uma das instâncias de VendaItem e somar os resultados.

De maneira a satisfazer a responsabilidade de conhecer e informar seu subtotal, um item da venda necessita conhecer o preço do Produto. O Produto é um expert da informação necessária para fornecer este preço, portanto deve ser enviada enviada uma mensagem para ele, perguntando o seu preço.

EXPERT



EXPERT

Concluindo, de forma a satisfazer a responsabilidade de conhecer e informar o total da venda, três responsabilidades foram atribuídas para três classes de objetos.

Classe	Responsabilidade
Venda	conhece o total da venda
LinhadeItemdeVenda	conhece o subtotal da Linha de Item
EspecificaçãodeProduto	conhece o preço do produto

EXPERT

Essa abordagem oferece benefícios como:

Coesão Melhorada: O conhecimento e a lógica relacionados a uma determinada responsabilidade são mantidos juntos, o que facilita a compreensão e a manutenção do código.

Baixo Acoplamento: Ao evitar a dispersão do conhecimento entre várias classes, reduz-se a dependência entre diferentes partes do sistema, o que torna o código mais flexível e menos propenso a erros.

Manutenibilidade Aprimorada: Como o conhecimento está centralizado na classe apropriada, as modificações necessárias para uma determinada funcionalidade são mais localizadas e menos propensas a afetar outras partes do sistema.

EXPERT

Em suma, o padrão **Expert** visa promover um design mais coeso, onde as responsabilidades são atribuídas às classes que possuem o conhecimento necessário para realizar as operações, contribuindo para sistemas mais robustos e fáceis de manter.



CREATOR

Creator (Criador): Determina qual classe deve ser responsável pela criação certos objetos.

CREATOR

Esse padrão visa delegar a responsabilidade de **criação de objetos** para classes específicas, permitindo uma criação mais flexível e desacoplada de objetos no sistema.

O padrão **Creator** guia a atribuição de responsabilidades relacionadas com a criação de objetos, uma tarefa muito comum em sistemas orientado a objetos. A intenção básica do padrão é encontrar um criador que necessita ser conectado ao objeto em qualquer evento, garantindo um acoplamento fraco.

CREATOR

Atribua à classe **B** a responsabilidade de criar uma instância da classe **A** se uma das seguintes condições for verdadeira:

- **B** agrega objetos **A**.
- **B** contém objetos **A**.
- **B** registra instâncias de objetos **A**.
- **B** usa de maneira muito próxima objetos **A**.
- **B** tem os dados de inicialização que serão passados para objetos **A**, quando de sua criação (assim, B é um Expert com relação à criação de A).

B é um criador de objetos **A**. Se mais de uma opção for aplicável, prefira uma classe B que agrega ou contém a classe A.

CREATOR

Na aplicação do ponto de vendas, quem deveria ser responsável pela criação de uma instância `VendaItem`? De acordo com o padrão Creator, deveríamos procurar uma classe que agrega, contém (e assim por diante) instâncias de `VendaItem`.

Uma vez que `Venda` contém muitos objetos de `VendaItem`, o padrão Creator sugere que `Venda` é um bom candidato para ter a responsabilidade de criação de instâncias de `VendaItem`.



CREATOR

Padrões Relacionados

- Acoplamento Fraco
- Todo-Parte (descreve um padrão para definir objetos agregados que suporta o encapsulamento dos seus componentes)

Benefícios

- Acoplamento Fraco

CREATOR

Com essa abordagem conseguimos o **Fraco Acoplamento** o que implica em menores dependências para a manutenção e maiores oportunidades de reutilização. O acoplamento, provavelmente não é aumentado porque é provável que a classe “*criada*” já seja visível para a classe “*criadora*”, devido às associações existentes que motivaram sua escolha anterior.



CONTROLLER

Controller (Controlador): Atribui a responsabilidade de lidar com os eventos do sistema para uma classe que representa a um cenário de caso de uso do sistema global.

CONTROLLER

O **objetivo** deste padrão é atribuir uma responsabilidade do **tratamento de eventos** do sistema a uma **classe**.

De acordo com o padrão Controller uma classe deve ter essa responsabilidade, quando ela representa as seguintes escolhas:

- Representa o “sistema” todo (controlador fachada).
- Representa todo o negócio ou organização (controlador fachada).
- Representa algo no mundo real que é ativo (por exemplo, o papel de uma pessoa) e que pode estar envolvido na tarefa (controlador do papel).
- Representa um tratador artificial de todos os eventos de sistema de um caso de uso, geralmente chamado controlador do caso de uso exemplo “<Nome do Caso de Uso>Handler”.

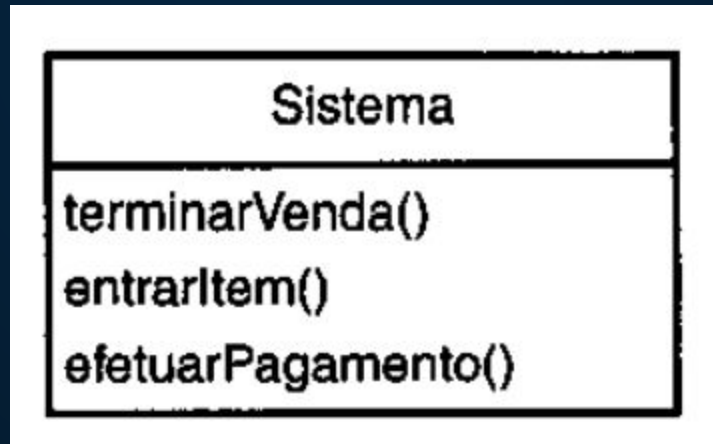
CONTROLLER

Quem deveria ser responsável por tratar um evento de sistema? Um evento de sistema é um evento de sistema de alto nível gerado por um ator externo, ele é um evento de entrada externo. Eles estão associados a operações do sistema, operações que o sistema realiza em resposta a eventos do sistema.

Por exemplo, quando um caixa usando um sistema de terminais de ponto de vendas pressiona o botão “Terminar Venda”, ele está gerando um evento de sistema indicando que “a venda terminou”. Similarmente, quando um escritor que está usando um processador de texto pressiona o botão de “verificação ortográfica”, ele está gerando um evento de sistema indicando “executar uma verificação ortográfica”.

CONTROLLER

Um Controller (controlador) é um objeto de interface, não de usuário responsável por tratar um evento de sistema. Um Controller define o método para a operação de sistema. Na aplicação ponto de vendas, há várias operações de sistema, conforme a imagem abaixo.



CONTROLLER

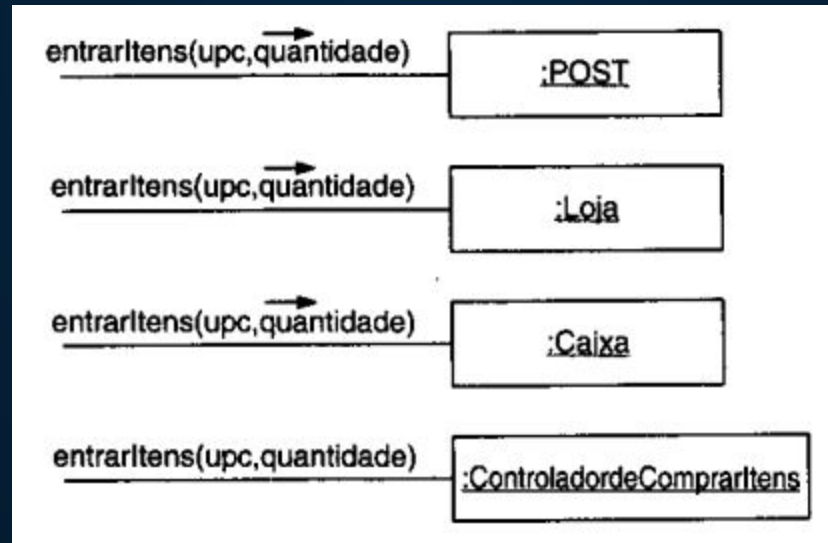
Quem deveria ser o controlador para os eventos de sistema, tais como **entrarItem** e **terminarVenda** ?

De acordo com o padrão **Controller**, aqui estão as opções:

representa o "sistema" todo	<i>POST</i>
representa todo o negócio ou organização	<i>Loja</i>
representa algo no mundo real que é ativo (por exemplo, o papel desempenhado por uma pessoa) e que pode estar envolvido na tarefa	<i>Caixa</i>
representa um tratador ("handler") artificial de todas as operações em um caso de uso.	<i>Controlador de ComprarItens</i>

CONTROLLER

Em termos de diagramas de colaboração, isso significa que será usado um dos exemplos da figura abaixo:



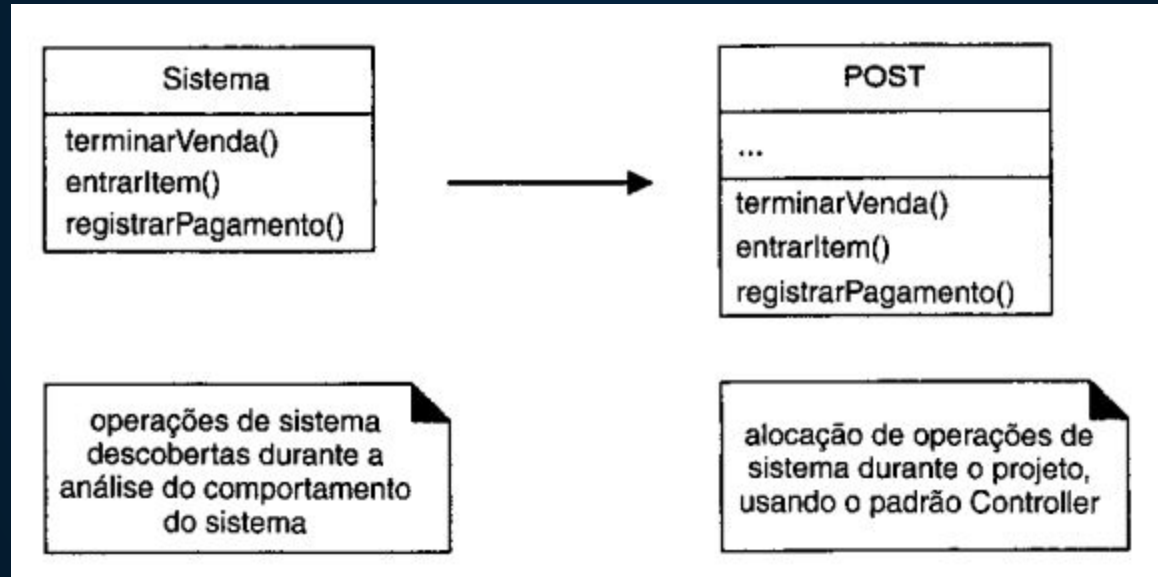
CONTROLLER

A maioria dos sistemas recebe eventos de entrada externos, envolvendo tipicamente uma interface gráfica de usuário (GUI) operada por uma pessoa.

Outros meios de entrada incluem mensagens externas, tais como em uma chamada “processando chave de telecomunicações”, ou sinais provenientes de sensores como em sistemas de controle de processos.

Em todos os casos, se for usado um projeto orientado a objetos, devem ser escolhidos **“controllers”** para **tratar** esses **eventos de entrada**. O padrão Controller fornece a orientação para escolhas adequadas e geralmente aceitas.

CONTROLLER



CONTROLLER

A mesma classe controladora deveria ser usada para todos os eventos de sistema de um caso de uso, de maneira que seja possível manter informações sobre o estado do caso de uso. Essa informação é útil, por exemplo, para identificar eventos de sistema fora de sequência (tal como uma operação registrarPagamento antes de uma operação terminar Venda). Diferentes controladores podem ser usados para diferentes casos de uso.

Um erro comum no design de controladores é dar-lhes muita responsabilidade. Normalmente, um controlador deveria delegar a outros objetos o trabalho que necessita ser feito enquanto ele coordena a atividade.

CONTROLLER

Controladores Inchados: Se for mal projetada, uma classe controladora terá coesão baixa falta de foco e tratamento de muitas áreas de responsabilidade, a isso chamamos controlador inchado. Os sinais de inchamento incluem:

- Existe somente uma única classe controladora, recebendo todos os eventos de sistema e existem muitos destes eventos. Isso, às vezes, acontece quando é escolhido um “controlador de papéis” ou um “controlador fachada”.
- O mesmo controlador executa muitas das tarefas necessárias para atender o evento de sistema, sem delegar o trabalho. Usualmente, isso envolve uma violação dos padrões Expert e Coesão Alta.
- Um controlador tem muitos atributos e mantém informações significativas sobre o sistema ou o domínio, as quais deveriam ter sido distribuídas entre outros objetos, ou duplica informações presentes em algum outro lugar.

CONTROLLER

Há várias soluções para um controlador inchado, entre elas:

- Acrescentar mais controladores: um sistema não deve ter somente um. Além de “controladores fachada”, use “controladores de papel” ou “controladores de caso de uso”.
- Projete o controlador de forma que ele delegue basicamente o atendimento de responsabilidade de cada operação de sistema a outros objetos.

CONTROLLER

Padrões relacionados:

- **Command:** Em um sistema de tratamento de mensagens, cada mensagem pode ser representada e tratada por um objeto Command separado.
- **Façade:** Escolher um objeto que representa todo o sistema ou organização para ser um controlador é um tipo de Facade.
- **Forward-Receiver:** Este é um padrão da Siemens, útil para sistemas de tratamento de mensagens.
- **Pure Fabrication:** Este é um outro padrão **GRASP**. Uma Pure Fabrication é uma classe artificial, e não um conceito do domínio. Um controlador de caso de uso é um tipo de Pure Fabrication.

CONTROLLER

O padrão **Controller** aborda a organização da lógica de controle em um sistema. Esse padrão atribui a responsabilidade de lidar com os eventos do sistema para uma classe que representa a um cenário de caso de uso do sistema global.

O Controller se concentra na designação de uma classe (ou conjunto de classes) que é responsável por receber e gerenciar solicitações de entrada do usuário ou de outros sistemas, bem como coordenar as ações e interações necessárias para lidar com essas solicitações. O objetivo é separar a lógica de controle da interface do usuário e das outras partes do sistema.



LOW COUPLING

Low Coupling (Baixo Acoplamento): Determina que as classes não devem depender de objetos concretos e sim de abstrações, para permitir que haja mudanças sem impacto.

LOW COUPLING

O princípio do **Baixo Acoplamento** se refere à redução da dependência entre módulos ou classes, permitindo que as partes do sistema sejam mais independentes e modificáveis sem afetar drasticamente outras partes.

O baixo acoplamento busca diminuir as conexões diretas entre diferentes componentes do sistema, de modo que uma mudança em uma parte do sistema não cause efeitos colaterais significativos em outras partes.

O **objetivo** deste padrão é **atribuir uma responsabilidade** de maneira que **acoplamento** permaneça **fraco**.

LOW COUPLING

O Acoplamento é o **nível de dependência** entre duas classes. Apesar de parecer simples, o conceito possui algumas nuances, as quais derivam da existência de dois tipos de acoplamento entre classes: **acoplamento aceitável** e **acoplamento ruim**.

Dizemos que existe um **acoplamento aceitável** de uma classe A para uma classe B quando:

- A classe **A** usa apenas métodos públicos da classe **B**.
- A interface provida por **B** é estável do ponto de vista sintático e semântico. Isto é, as assinaturas dos métodos públicos de **B** não mudam com frequência; e o mesmo acontece com o comportamento externo de tais métodos. Por isso, são raras as mudanças em **B** que terão impacto na classe **A**.

LOW COUPLING

Por outro lado, existe um **acoplamento ruim** de uma classe A para uma classe B quando mudanças em B podem facilmente impactar A. Isso ocorre principalmente nas seguintes situações:

- Quando a classe **A** realiza um acesso direto a um arquivo ou banco de dados da classe **B**.
- Quando as classes **A** e **B** compartilham uma variável ou estrutura de dados global. Por exemplo, a classe **B** altera o valor de uma variável global que a classe **A** usa no seu código.
- Quando a interface da classe **B** não é estável. Por exemplo, os métodos públicos de **B** são renomeados com frequência.

LOW COUPLING

Em essência, o que caracteriza o acoplamento ruim é o fato de que a dependência entre as classes não é mediada por uma interface estável. Por exemplo, quando uma classe altera o valor de uma variável global, ela não tem consciência do impacto dessa mudança em outras partes do sistema. Por outro lado, quando uma classe altera sua interface, ela está ciente de que isso vai ter impacto nos clientes, pois a função de uma interface é exatamente anunciar os serviços que uma classe oferece para o resto do sistema.

LOW COUPLING

Problemas com acoplamento forte:

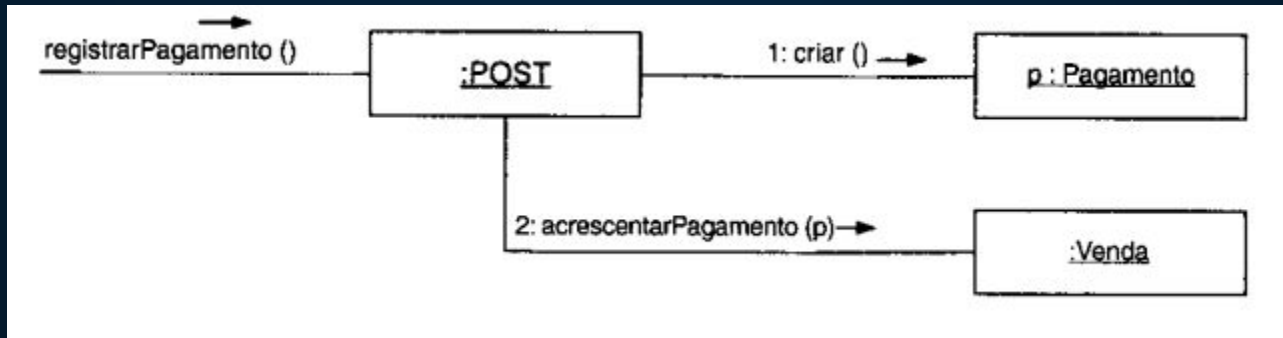
- Mudanças em classes relacionadas forçam mudanças locais.
- Mais difícil de compreender isoladamente.
- Mais difícil de reutilizar, porque o seu uso requer a presença adicional das classes das quais ela depende.

Benefícios com acoplamento fraco:

- Não afetado por mudanças em outros componentes.
- Simples de entender isoladamente.
- Facilidade para reutilizar.

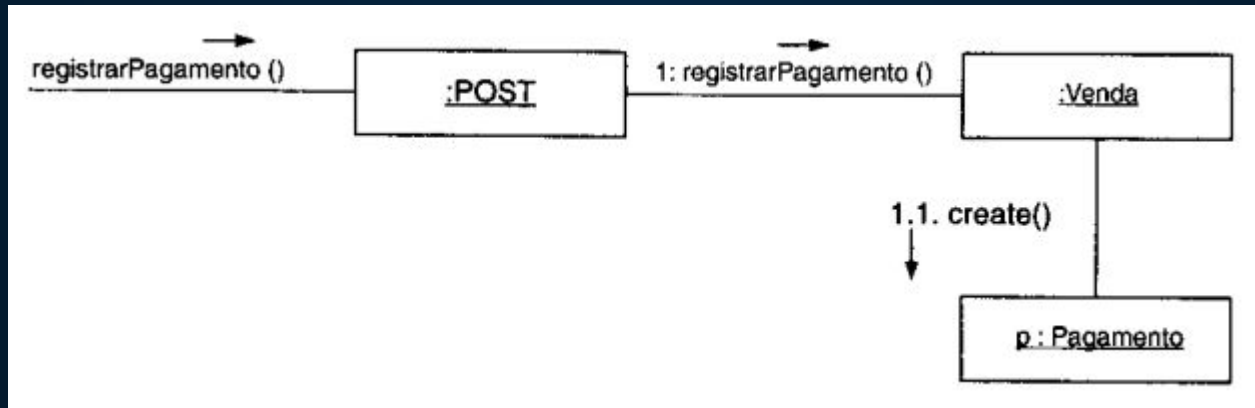
LOW COUPLING

Considere o seguinte diagrama de classe de uma aplicação de terminal de ponto de venda. Suponha que necessitamos criar uma instância de Pagamento e associá-la a Venda. Uma vez que um POST registra um pagamento no domínio do mundo real, o padrão Creator sugere POST como um candidato para criar um pagamento. A instância de POST poderia, então, enviar uma mensagem acrescentar pagamento a venda, passando junto um novo pagamento como parâmetro.



LOW COUPLING

Esta atribuição de responsabilidade acopla POST a conhecimento da classe Pagamento. Uma solução alternativa para criar pagamento é associá-lo à venda.



LOW COUPLING

Qual das soluções de projeto, baseada na atribuição das responsabilidades, resulta em Acoplamento Fraco? Em ambos os casos, iremos supor que a **Venda** deve, finalmente, ser acoplada ao conhecimento de um **Pagamento**. A primeira imagem, no qual **POST** cria o Pagamento, acrescenta um acoplamento de POST a Pagamento, enquanto na segunda imagem, no qual a Venda efetua a criação de um Pagamento, não aumenta o acoplamento. Sob o ponto de vista exclusivo do acoplamento, A segunda opção é preferível porque é mantido um acoplamento geral mais fraco.

Este é um exemplo no qual dois padrões **Acoplamento Fraco** e **Creator**, podem sugerir diferentes soluções. Na prática, o nível de acoplamento sozinho não pode ser considerado isoladamente de outros princípios, tais como **Expert** e **Coesão Alta**. Apesar disso, ele é um outro fator a ser considerado na melhoria de um projeto.

LOW COUPLING

O Acoplamento Fraco estimula a atribuição de uma responsabilidade de maneira que a sua localização não aumente o acoplamento até um nível que conduza aos resultados negativos que o acoplamento forte pode produzir.

O Acoplamento Fraco apóia o projeto de classes que são mais independentes. Isso reduz o impacto de mudanças e promove maior capacidade de reutilização, o que aumenta as oportunidades de se obter uma produtividade mais alta. Ele não pode ser considerado isoladamente de outros padrões, tais como o padrão Expert e o padrão Coesão Alta, mas necessita ser incluído como um dentre vários princípios de projetos que influenciam uma escolha na atribuição de uma responsabilidade.

LOW COUPLING

O caso extremo de Acoplamento Fraco é quando não existe ou existe muito pouco acoplamento entre as classes. Isso não é desejável porque uma metáfora central da tecnologia de objetos é um sistema composto de objetos conectados que se comunicam através de mensagens. Se o Acoplamento Fraco é aplicado em excesso, leva a um projeto fraco, porque conduz a uns poucos objetos ativos, sem coesão, inchados e complexos, que efetuam todo o trabalho.

Ao mesmo tempo, existirão muitos objetos praticamente passivos, com acoplamento zero, que funcionam como simples repositórios de dados. Algum grau moderado de acoplamento entre classes é normal e necessário, de forma a criar um sistema orientado a objetos, no qual as tarefas são executadas por uma colaboração entre objetos conectados.



HIGH COHESION

High Cohesion (Alta Coesão): este princípio determina que as classes devem se focar apenas na sua responsabilidade.

HIGH COHESION

O **objetivo** deste padrão é atribuir **uma responsabilidade** de forma que a **coesão** permaneça **alta**.

A implementação de qualquer classe deve ser coesa, isto é, toda classe deve implementar uma única funcionalidade ou serviço. Especificamente, todos os métodos e atributos de uma classe devem estar voltados para a implementação do mesmo serviço. Uma outra forma de explicar coesão é afirmando que toda classe deve ter uma única responsabilidade no sistema. Ou, ainda, afirmando que deve existir um único motivo para modificar uma classe.

HIGH COHESION

Separação de interesses (separation of concerns) é uma outra propriedade desejável em projetos de software, a qual é semelhante ao conceito de coesão. Ela defende que uma classe deve implementar apenas um interesse (concern).

Nesse contexto, o termo interesse se refere a qualquer funcionalidade, requisito ou responsabilidade da classe. Portanto, as seguintes recomendações são equivalentes:

1. Uma classe deve ter uma única responsabilidade;
2. Uma classe deve implementar um único interesse;
3. Uma classe deve ser coesa.

HIGH COHESION

Alta Coesão tem as seguintes vantagens:

- Facilita a implementação de uma classe, bem como o seu entendimento e manutenção.
- Facilita a alocação de um único responsável por manter uma classe.
- Facilita o reuso e teste de uma classe, pois é mais simples reusar e testar uma classe coesa do que uma classe com várias responsabilidades.

HIGH COHESION

Classes de coesão baixa representam, geralmente, uma abstração de grande granularidade ou, então, assumiram responsabilidades que deveriam ter sido delegadas a outros objetos.

Baixa Coesão tem os seguintes problemas:

- Difíceis de compreender.
- Difíceis de reutilizar.
- Difíceis de manter.
- São constantemente afetadas pelas mudanças.

HIGH COHESION

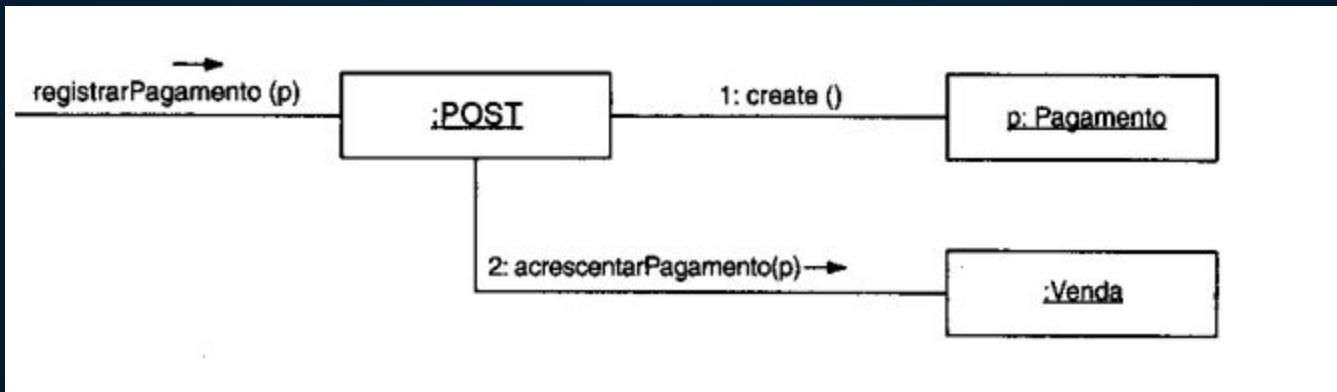
O mesmo exemplo de problema usado no padrão **Acoplamento Fraco** pode ser usado para análise da **Coesão Alta**.

Suponha que necessitamos criar uma instância de Pagamento (em dinheiro) e associá-lo à Venda. Qual classe deveria ser responsável por isso?

Uma vez que POST registra um Pagamento no domínio do mundo real, o padrão Create sugere POST como um candidato para criar Pagamento. A instância de POST deveria, então, manda uma mensagem acrescentar Pagamento para a Venda, passando como um parâmetro o novo Pagamento, conforme mostrado na figura a seguir.

HIGH COHESION

Esta atribuição de responsabilidades coloca a responsabilidade de criar um pagamento em POST. POST está assumindo parte da responsabilidade de executar a operação de sistema registrarPagamento.



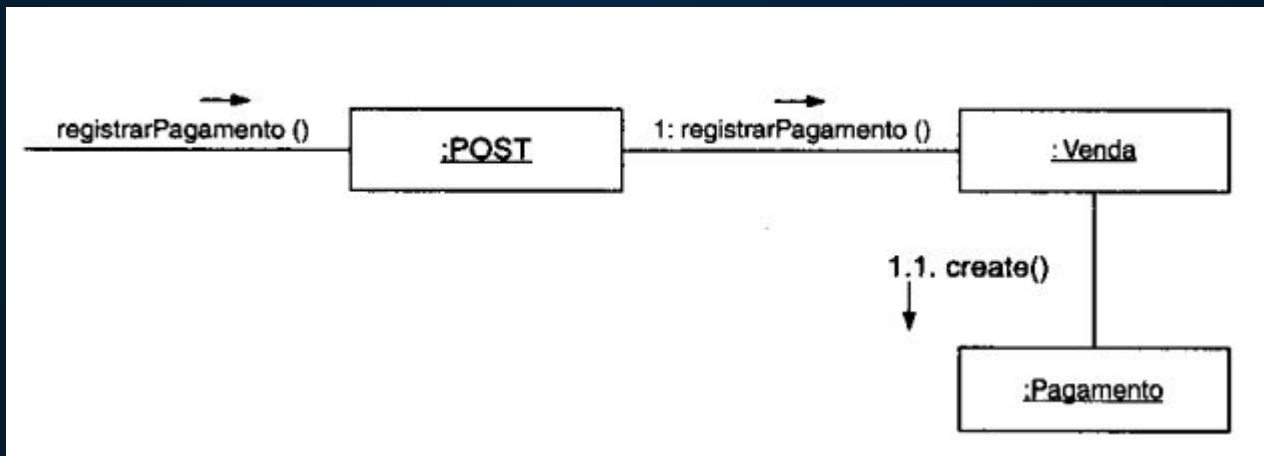
HIGH COHESION

Neste exemplo isolado, isso é aceitável, porém se continuarmos a fazer a classe POST responsável por realizar algum trabalho ou a maior parte dele, o qual está relacionado com cada vez mais operações do sistema, ela se tornará progressivamente carregada com tarefas e perderá sua coesão.

Imagine que existissem cinquenta operações do sistema, todas recebidas por POST. Se ele fizesse o trabalho relacionado com cada uma, se tornaria um objeto “inchado”, sem coesão. O problema não está no fato de que esta única tarefa de criação de Pagamento, sozinha, torna POST sem coesão. Mas como parte de um quadro de atribuição de responsabilidades mais amplo, ele pode sugerir uma tendência para uma coesão baixa.

HIGH COHESION

Ao contrário, como mostrado na figura abaixo, o segundo projeto delega a criação do pagamento para a Venda, o que favorece uma coesão mais alta em POST. Uma vez que o segundo projeto favorece tanto uma coesão alta como um acoplamento fraco, ele é mais desejável.



HIGH COHESION

Como regra prática, uma classe com coesão alta tem um número relativamente pequeno de métodos, com funcionalidades altamente relacionadas e não executa muito trabalho. Se a tarefa for grande, ela colabora com outros objetos para compartilhar o esforço.

Uma classe com coesão alta é vantajosa porque é relativamente fácil de manter, de compreender e de reutilizar. O alto grau de funcionalidades relacionadas, combinado com um pequeno número de operações, também simplifica a manutenção e as melhorias. A granularidade fina de funcionalidades altamente relacionadas também suporta um aumento do potencial de reutilização.

HIGH COHESION

O padrão de Coesão Alta, como muitos conceitos na orientação a objetos, encontra uma analogia no mundo real. É amplamente observado que quando uma pessoa assume múltiplas responsabilidades não relacionadas, especialmente aquelas que poderiam ser mais apropriadamente delegadas, acaba gerando ineficiências no dia a dia de trabalho.

Essa situação é comum em alguns gerentes que não desenvolveram a habilidade de delegar tarefas. Esses indivíduos acabam sofrendo com uma baixa coesão, onde as responsabilidades não estão claramente alinhadas, afetando a eficácia e eficiência de suas atividades diárias.



POLYMORPHISM

Polymorphism (Polimorfismo): As responsabilidades devem ser atribuídas a abstrações e não a objetos concretos, permitindo que eles possam variar conforme a necessidade.

POLYMORPHISM

O **objetivo** deste padrão é atribuir **uma responsabilidade** usando operações **polimórficas** para evitar o uso de condições lógicas (if-else) no código para executar alternativas que variam com base no tipo.

Como tratar alternativas com base no tipo? Como criar componentes de software “conectáveis”?

Alternativas com base no tipo. A variação condicional é um tema fundamental em programas. Se um programa é projetado, usando a lógica condicional dos comandos “if-else” ou “case”, então quando uma nova variação surge, ela irá requerer a modificação da lógica desses comandos. Esta abordagem torna difícil a extensão de um programa para atender novas variantes, porque as mudanças tendem a ser necessárias em vários lugares onde quer que exista a lógica dos comandos condicionais.

POLYMORPHISM

Componentes de Software “conectáveis” . Vendo os componentes como relacionamentos cliente-servidor, como podemos substituir um componente servidor por outro, sem afetar o cliente?

Na aplicação do ponto de vendas, quem deveria ser responsável pela autorização de diferentes tipos de pagamentos?

Uma vez que o comportamento relativo à autorização varia com o tipo de pagamento em dinheiro, cartão de crédito ou cheque, por Polimorfismo, deveríamos atribuir a responsabilidade pela autorização para cada tipo de pagamento, implementada com uma operação polimórfica autorizar.

POLYMORPHISM

A implementação de cada operação autorizar será diferente: **PagamentoComCredito** se comunicará com um serviço de autorização de crédito e assim por diante.

Em vez de operar externamente sobre um pagamento para autorizá-lo, um pagamento autoriza a si próprio. Se o Expert pode ser visto como o mais importante padrão tático básico, o polimorfismo é o mais importante padrão estratégico básico no projeto orientado a objetos.

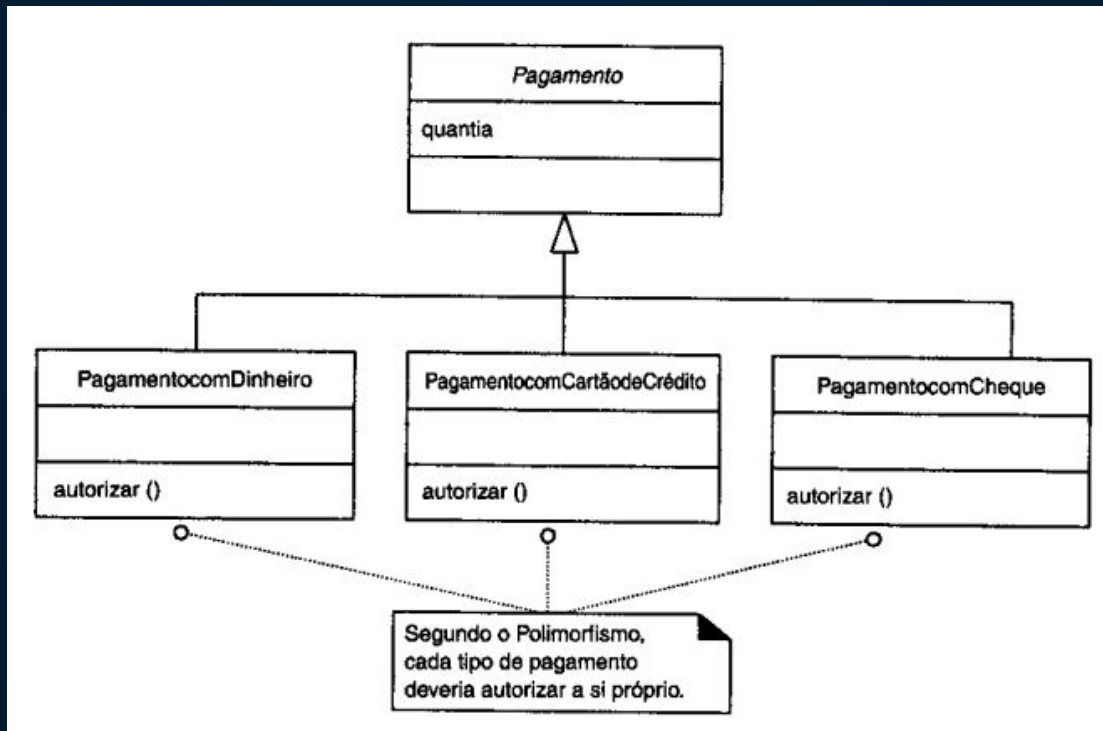
Um projeto baseado na atribuição de responsabilidades segundo **Polimorfismo** pode ser facilmente estendido para tratar novas variantes. Por exemplo, o acréscimo de uma nova classe **PagamentoComDebito**, com sua própria operação polimórfica autorizar, terá um impacto pequeno sobre o projeto existente no que diz respeito ao modo como as autorizações são tratadas.

POLYMORPHISM

Olhando os objetos em relacionamentos cliente-servidor, os objetos clientes necessitarão pouca ou nenhuma modificação, quando for introduzido um novo tipo de objeto servidor, na medida em que o novo servidor suporte as operações polimórficas esperadas pelo cliente. As futuras extensões, necessárias para novas variantes não previstas, são fáceis de acrescentar.

O Polimorfismo permite adicionar novos comportamentos sem modificar o código existente, além do código ficar mais legível, já que os diferentes comportamentos são encapsuladas em classes separadas. Isso facilita a compreensão do código e a manutenção futura.

POLYMORPHISM



POLYMORPHISM

O **Polimorfismo** torna o código mais flexível, fácil de entender e manter, o que é fundamental em muitos sistemas de software. É especialmente útil quando há uma necessidade de variação de comportamento em tempo de execução ou quando se espera mudanças frequentes nos algoritmos utilizados.



PURE FABRICATION

Pure Fabrication (Pura Fabricação): é uma classe que não representa nenhum conceito no domínio do problema, ela apenas funciona como uma classe prestadora de serviços, e é projetada para que possamos ter um baixo acoplamento e alta coesão no sistema.

PURE FABRICATION

o **objetivo** do **Pure Fabrication** é atribuir um conjunto de responsabilidades altamente coeso a uma classe artificial que não representa nada no domínio do problema, algo construído especialmente, para suportar coesão alta, acoplamento fraco ou reutilização.

Tal classe é uma criação da imaginação. Idealmente, as responsabilidades atribuídas a esta invenção suportam uma coesão alta e um acoplamento fraco, de maneira que o projeto desta invenção seja muito limpo ou puro, daí a chamarmos invenção pura.

A quem atribuir responsabilidades quando você está desesperado e não quer violar os princípios da Coesão Alta e do Acoplamento Fraco?

PURE FABRICATION

Projetos orientados a objetos são caracterizados por implementarem como classes de software as representações de conceitos no domínio do problema, no mundo real, por exemplo, uma classe **Venda** e uma classe **Cliente**. Contudo, existem muitas situações nas quais a atribuição de responsabilidades somente a classes do domínio leva a problemas em termos de coesão ou de acoplamento inadequados, ou a um baixo potencial de reutilização.

Por exemplo, suponha que necessitemos de suporte para salvar instâncias de Venda em um banco de dados relacional. Segundo o padrão Expert, há alguma justificativa em atribuir esta responsabilidade à própria classe Venda.

PURE FABRICATION

Porém, analise as seguintes implicações:

- A tarefa requer um número relativamente grande de operações voltadas para o banco de dados, nenhuma delas relacionadas com o conceito de “vender”, dessa forma, a classe Venda fica com baixa coesão.
- A classe Venda tem que ser acoplada com a interface do banco de dados relacional (geralmente fornecida pelo vendedor da ferramenta de desenvolvimento), de modo que o seu acoplamento aumenta muito. E, ainda por cima, o acoplamento não é nem mesmo relacionado com outro objeto do domínio, porém com uma interface de banco de dados e suas características.
- Salvar objetos em um banco de dados relacional é uma tarefa muito geral para a qual muitas classes necessitam de suporte. A colocação destas responsabilidades na classe Venda sugere que teremos uma má reutilização, ou muitas duplicações de códigos, em outras classes, fazendo a mesma coisa.

PURE FABRICATION

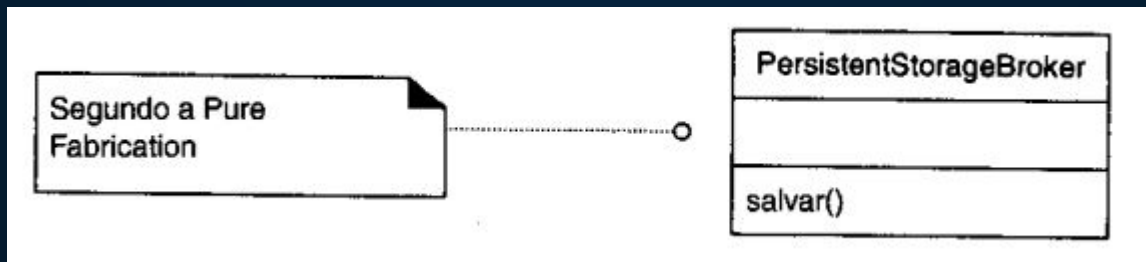
Assim, mesmo se, de acordo com o padrão Expert, Venda seja um candidato lógico para salvar a si próprio em um banco de dados, esta solução conduz a um projeto com coesão baixa, acoplamento forte e baixo potencial de reutilização, exatamente o tipo de situação desesperadora que pede que criemos algo especial.

Uma solução razoável é criar uma nova classe que é unicamente responsável por salvar objetos em algum tipo de meio de armazenamento persistente, tal como um banco de dados relacional, chamamos a esta solução **PersistentStorageBroker**'. ("IntermediárioDeArmazenamentoPersistente"). Esta classe é uma Pure Fabrication, uma criação da imaginação.

PURE FABRICATION

Ela soluciona os seguintes problemas de projeto:

- A Venda permanece bem-projetada, apresentando coesão alta e acoplamento fraco.
- A classe PersistentStorageBroker é ela própria relativamente coesa, tendo como finalidade única o armazenamento de objetos em um meio de armazenamento persistente.
- A classe PersistentStorageBroker é um objeto muito genérico e reutilizável.



PURE FABRICATION

A criação de uma “invenção pura”, neste exemplo, é exatamente o tipo de situação na qual o seu uso é necessário eliminar uma solução ruim de projeto, com coesão e acoplamento inadequados, por uma boa solução de projeto com uma potencialidade maior de reutilização.

Note que, como acontece com todos os padrões GRASP, a ênfase está em onde as responsabilidades devem ser colocadas. Neste exemplo, as responsabilidades foram movidas da classe Venda para uma Invenção Pura.

Uma Invenção Pura deveria ser projetada tendo em mente uma alta potencialidade de reutilização, garantindo-se que suas responsabilidades sejam pequenas e coesas. Estas classes tendem a ter um conjunto de responsabilidades de granularidade fina.

PURE FABRICATION

Uma Invenção Pura é geralmente particionada com base em funcionalidades relacionadas, sendo assim um tipo de objeto centrado na função.

Uma Invenção Pura é geralmente considerada como parte da Camada de Serviços de Alto Nível Orientado a Objetos em uma arquitetura.

Muitos padrões existentes para o projeto orientado a objetos são exemplos de **Pure Fabrications**, como: **Adapter**, **Observer**, **Visitor** e assim por diante.

PURE FABRICATION

O conceito de Pure Fabrication é baseado no princípio de separação de responsabilidades. Ele sugere a criação de classes ou componentes que não têm um papel natural no domínio do problema, mas são criados para simplificar o design, centralizar lógicas complexas ou fornecer funcionalidades reutilizáveis.

Essas classes podem ser criadas para:

- **Encapsular lógica complexa:** Quando certas operações são complexas o suficiente para justificar a criação de uma classe dedicada para lidar com elas. Isso ajuda a manter o código mais organizado e legível, separando a complexidade em um local específico.

PURE FABRICATION

- **Promover reutilização:** Se uma funcionalidade específica é necessária em várias partes do sistema, criar uma classe de "fabricação pura" para encapsular essa funcionalidade pode facilitar a reutilização do código.
- **Centralizar operações ou lógicas que não se encaixam naturalmente em outras classes:** Certas operações podem não ter uma classe apropriada para residir, e a criação de uma classe puramente para lidar com essas operações pode simplificar o design geral.

PURE FABRICATION

Padrões Relacionados

- Acoplamento Fraco
- Alta Coesão
- Invenções Puras
- Adapter
- Observer
- Visitor

Benefícios

- Alta Coesão
- Reutilização

PURE FABRICATION

Ao aplicar o conceito de **Pure Fabrication**, os desenvolvedores podem separar lógicas complexas ou repetitivas em classes separadas, melhorando a coesão e o encapsulamento, tornando o sistema mais fácil de entender, manter e evoluir ao longo do tempo.



INDIRECTION

Indirection (Indireção): este princípio ajuda a manter o baixo acoplamento, através de delegação de responsabilidades através de uma classe mediadora.

INDIRECTION

O objetivo **Indirection (Indireção)** é atribuir a responsabilidade a um objeto intermediário que faz a mediação entre outros componentes ou serviços, de modo que eles não estejam diretamente acoplados.

O intermediário cria uma indireção para os outros componentes ou serviços.

- Quem irá evitar o acoplamento direto?
- Como desacoplar objetos de maneira a suportar Acoplamento Fraco e manter alto o potencial de reutilização?

INDIRECTION

O exemplo de uma **Invenção Pura** de desacoplar **Venda** dos serviços de banco de dados relacionais, através da introdução de uma classe **PersistentStorageBroker**, também é um exemplo de atribuição de responsabilidades para suportar Indireção. O PersistentStorageBroker atua como um intermediário entre a Venda e o banco de dados.

Modem, suponha que:

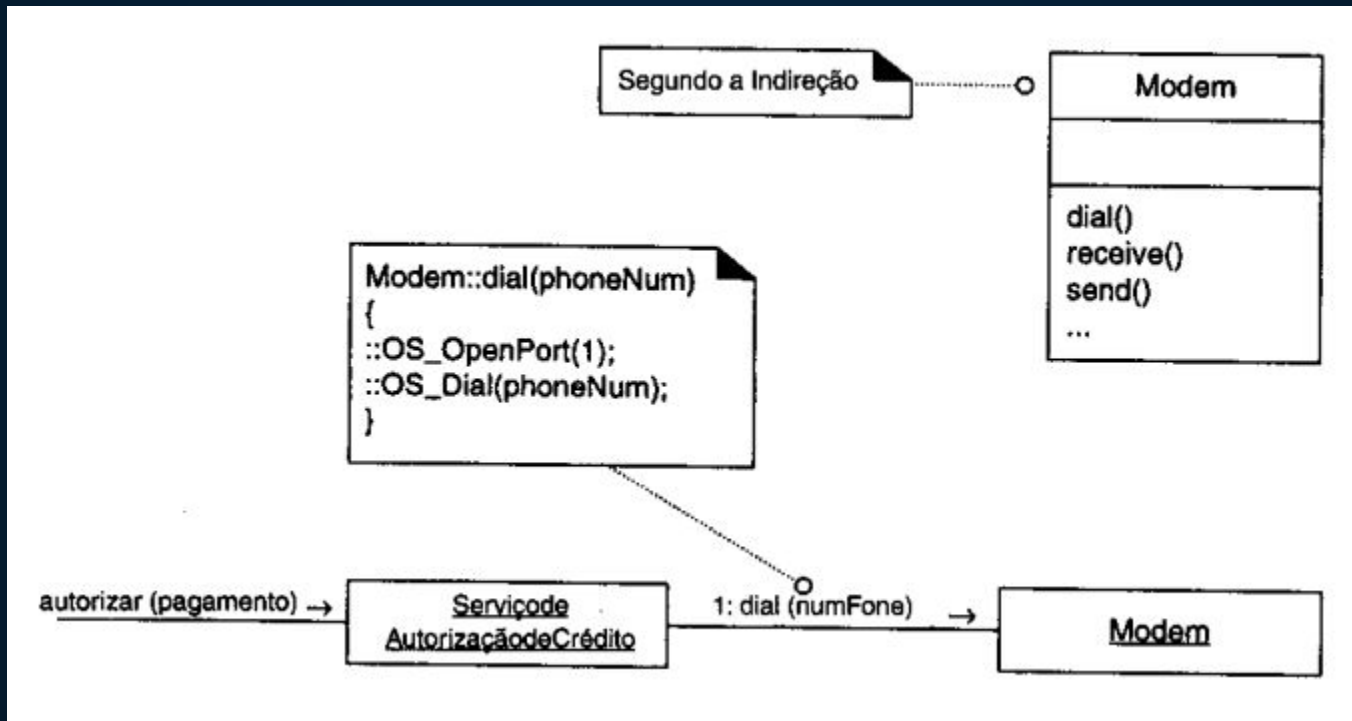
- Uma aplicação de terminal de ponto de vendas necessite manipular um modem de forma a poder transmitir solicitações de pagamentos com crédito.
- O sistema operacional forneça uma função de baixo nível chamada API para fazer isso.
- Uma classe chamada ServiçoDeAutorizaçãodeCrédito seja responsável por “falar” com o modem.

INDIRECTION

Se o ServiçoDeAutorizacaoDeCredito invoca as chamadas de baixo nível às funções da API diretamente, ela fica altamente acoplada à API de um particular sistema operacional e seus comportamentos. Se for necessário transportar a classe para um outro sistema operacional (para utilização na mesma aplicação ou em uma aplicação diferente), então ela necessitará modificações.

Adicione uma classe Modem intermediária entre o ServiçoDeAutorizacaoDeCredito e a API do modem. Ela será a responsável pela tradução de solicitações abstratas para o modem em chamadas para a API, criando uma Indireção entre o ServiçoDeAutorizacaoDeCredito e a API do modem. (Uma classe como Modem, que representa e interfaceia um dispositivo eletromecânico, também é conhecida como um device proxy um “representante” de dispositivo).

INDIRECTION



INDIRECTION

Resumindo, a ideia de **Indireção**, quando aplicada como um conceito de design, pode estar relacionada à mediação ou à criação de camadas intermediárias entre componentes ou serviços. Essa abordagem é frequentemente usada para promover o desacoplamento, a flexibilidade e a facilitação de mudanças futuras no sistema.

Se considerarmos a indireção como um padrão que facilita a mediação entre componentes ou serviços, isso geralmente envolve:

- **Desacoplamento:** Ao introduzir uma camada intermediária (ou um serviço intermediário), é possível reduzir a dependência direta entre os componentes ou serviços. Isso permite que eles se comuniquem por meio dessa indireção, sem precisar conhecer detalhes internos uns dos outros

INDIRECTION

- **Facilidade de Manutenção e Evolução:** A presença dessa camada intermediária facilita a manutenção e a evolução do sistema. Se os componentes ou serviços mudarem, a camada de indireção pode ser ajustada para lidar com essas mudanças, mantendo os outros componentes inalterados.
- **Aprimoramento da Flexibilidade:** A indireção permite introduzir lógica adicional, validações, transformações de dados, entre outras operações, sem afetar diretamente os componentes ou serviços originais.
- **Mediação e Coordenação:** Essa camada intermediária pode atuar como um mediador entre diferentes componentes ou serviços, coordenando suas interações, aplicando regras específicas ou traduzindo entre diferentes interfaces.

INDIRECTION

Padrões Relacionados

- Acoplamento Fraco
- Mediator
- Invenções Puras

Benefícios

- Acoplamento Fraco

INDIRECTION

A **Indireção** pode ser enfatizada como uma técnica para criar abstrações intermediárias que facilitam a comunicação e a interação entre diferentes partes do sistema, proporcionando uma melhor organização e flexibilidade à medida que o sistema evolui.



DEMETER LAW

Demeter Law (Lei de Demeter): Interagir apenas com objetos vizinhos imediatos e não com objetos mais distantes.

DEMETER LAW

O Padrão **Demeter Law** tem como objetivo atribuir responsabilidade a um objeto diretamente referenciado por um cliente, de forma a colaborar com um objeto indiretamente referenciado, evitando que o cliente necessite ter conhecimento do objeto indiretamente referenciado.

Resumindo é um princípio de programação orientada a objetos que afirma que um objeto deve ter informações limitadas sobre outros objetos.

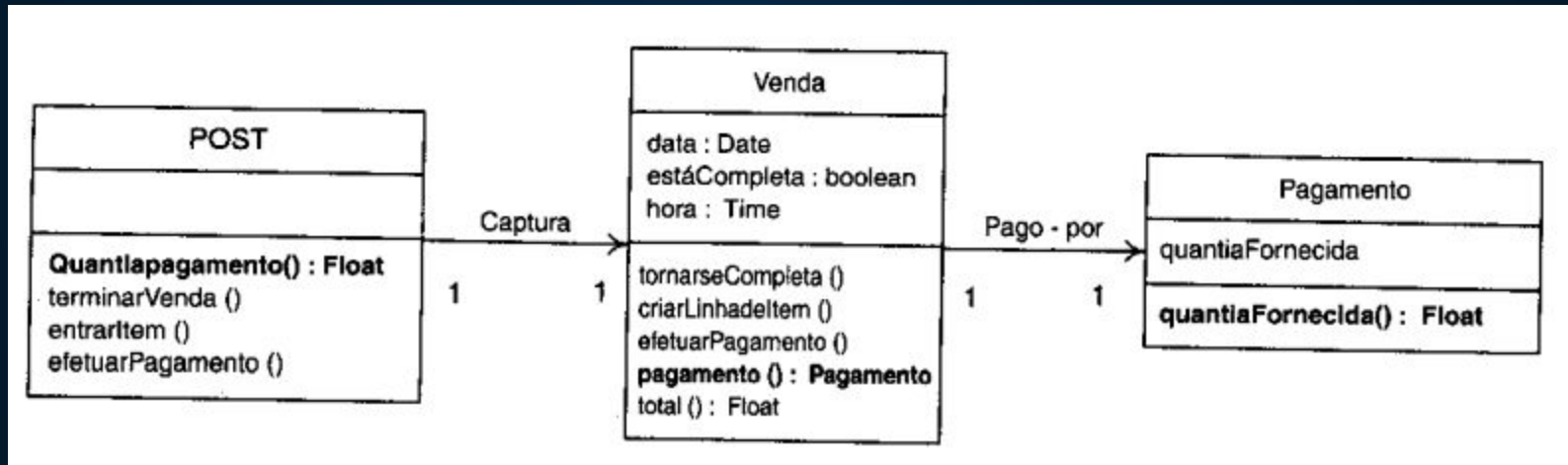
DEMETER LAW

O padrão, também conhecido como **Don't Talk to Strangers (Não fale com estranhos)**, coloca restrições sobre a quais objetos você deve se comunicar a partir de um método.

- de sua própria classe (caso 1)
- de objetos passados como parâmetros (caso 2)
- de objetos criados pelo próprio método (caso 3)
- de atributos da classe do método (caso 4)

DEMETER LAW

Em uma aplicação de ponto de vendas, suponha que uma instância de **POST** tenha um atributo que referencia uma **Venda**, a qual tem um atributo que referencia um **Pagamento**.



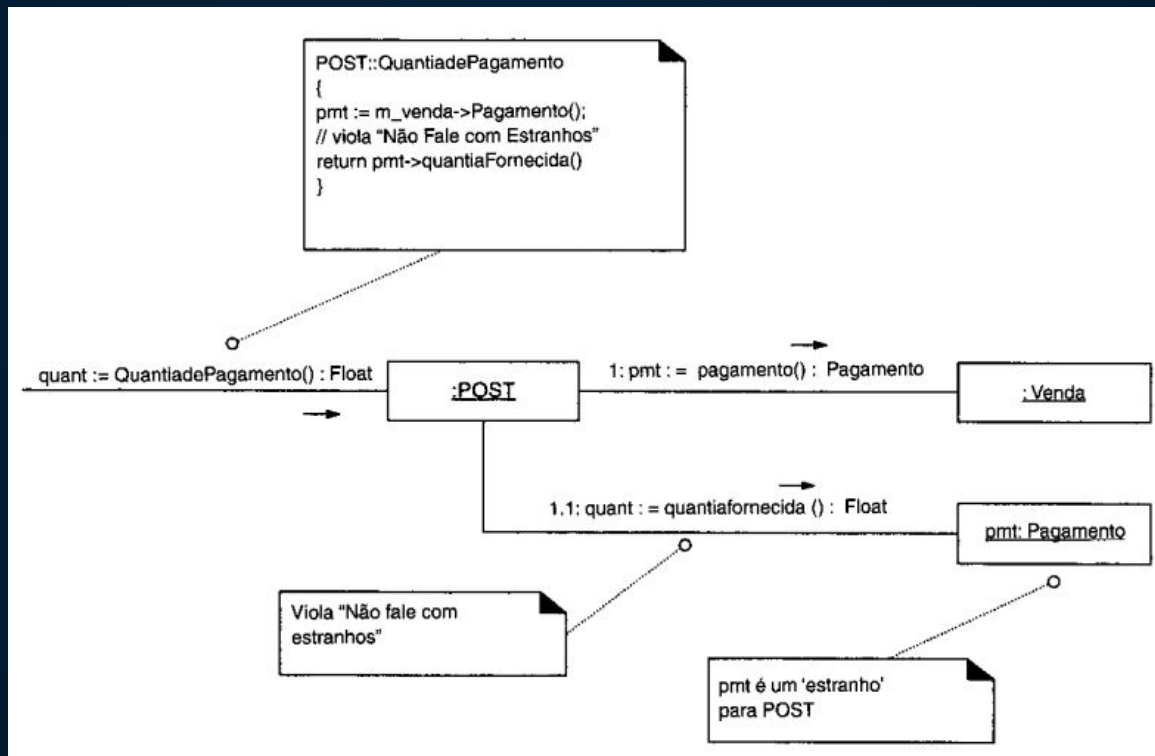
DEMETER LAW

Além disso, suponha que:

- As instâncias de POST suportem a operação `QuantiaDoPagamento`, a qual retorna a quantia fornecida para o pagamento.
- As instâncias de Venda suportem a operação `pagamento`, a qual retorna a instância de Pagamento associada à Venda.

Uma solução para retornar a quantia do pagamento é:

DEMETER LAW



DEMETER LAW

A solução, mostrada na imagem anterior é uma **violação** do princípio “Lei de Demeter” porque a instância de **POST** está enviando uma mensagem para um objeto indireto o **Pagamento** não é um dos quatro candidatos “familiares”.

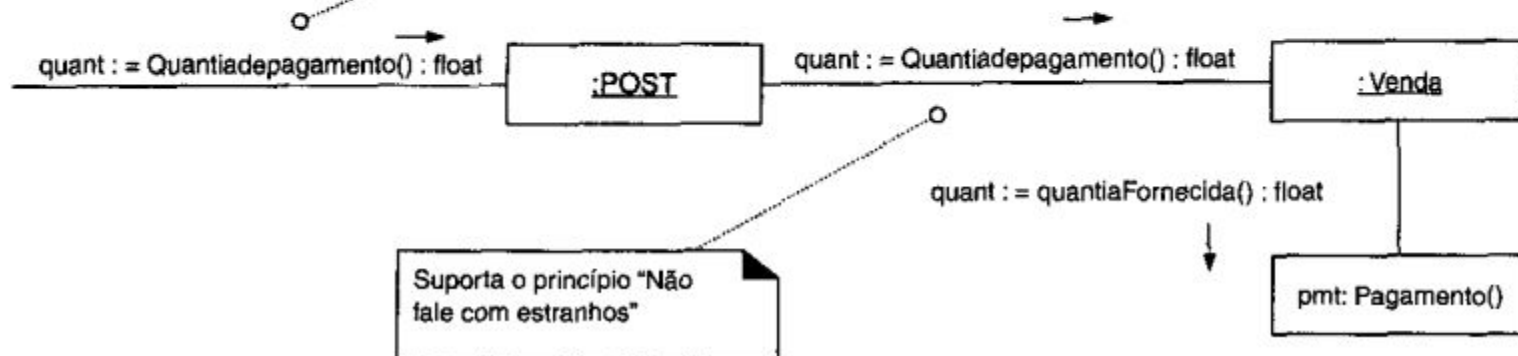
A solução, como sugere o padrão, é atribuir a responsabilidade para o objeto direto neste caso, a **Venda** para que este retorne a quantia do pagamento para o POST. Isso é conhecido como promoção da interface, que é a solução geral para suportar este princípio. Portanto, uma operação **QuantiaDePagamento** é adicionada à Venda, de maneira que o POST não tenha que falar com um estranho.



```

POST::QuantidadePagamento()
{
    return m_venda->QuantidadePagamento()
}

```



DEMETER LAW

O princípio “Lei de Demeter” ou “Não Fale com Estranhos” se preocupa em evitar a obtenção de visibilidade temporária para objetos indiretos, objetos que são conhecidos por outros objetos, mas não pelo cliente. A desvantagem de obter a visibilidade de estranhos é que a solução fica, então, acoplada ao conhecimento da estrutura interna de outros objetos. Isso conduz a um acoplamento forte, o que torna o projeto menos robusto e com maior probabilidade de exigir mudanças, se os relacionamentos estruturais indiretos mudarem.

Por exemplo, suponha que seja utilizada a solução que viola a Lei de Demeter, e que, então, a definição de classe Venda seja mudada, de maneira que ela não mais mantenha referência a um Pagamento. Nesse caso, o método POST pagamentoQuantia da classe POST não é mais válido, exigindo manutenção.

DEMETER LAW

Neste pequeno exemplo, isso não parece ser um problema custoso, porém em um sistema com muitas centenas, se não milhares de classes que estão sendo simultaneamente desenvolvidas por muitos desenvolvedores, isso é um problema.

Projetar uma solução que depende do conhecimento de relacionamentos estruturais indiretos torna um sistema frágil.

Agora, se usarmos a solução que aplica o princípio da Lei de Demeter, então os métodos de POST não são afetados por quaisquer mudanças na representação interna e nas conexões da Venda. Tudo que o POST sabe é que uma Venda pode lhe responder o total do seu pagamento, porém ele não tem conhecimento de como isso é obtido.

DEMETER LAW

Rompendo a regra

A primeira coisa a saber com relação às leis de software é que elas são criadas para serem rompidas. Assim, embora “Não fale com estranhos” (A Lei de Demeter) seja um conselho bem fundamentado, existem situações nas quais é razoável ignorá-lo.

Uma situação comum razoável para violá-lo é quando houver algum tipo de “broker” (intermediário) ou “servidor de objeto” (usualmente, uma Invenção Pura), o qual é responsável por retornar outros objetos através de uma busca baseada no valor de uma chave. É considerada aceitável a obtenção de visibilidade para um objeto X através de um “broker” (“intermediário”), passando-se, então, a enviar mensagens diretamente a X, mesmo que isso viole o princípio “Não fale com estranhos”.

DEMETER LAW

Padrões Relacionados

- Acoplamento Fraco
- Indireção
- Chain of Responsibility

Benefícios

- Acoplamento Fraco

DEMETER LAW

A ideia da Lei de Demeter é "**Não fale com estranhos, converse apenas com seus amigos**", levando isso em consideração temos na classe POST o método `QuantiaPagamento()` que trabalha com o valor total do pagamento apenas porque ele conhece a classe `Venda` por isso pode pedir seu valor, com isso mantemos um dos propósitos do encapsulamento da Orientação a Objetos. Não conhecemos as propriedades e nem os cálculos que ocorrem dentro do método `QuantidadeDePagamento` da classe `Venda`, mas sabemos que ele irá nos retornar o total do pagamento.

Então se você deparar com código realizando chamadas de métodos encadeados como: **`a.getB().getC().Metodo()`**. Fique atento e verifique se não seria o caso de aplicar a Lei de Demeter para reduzir o acoplamento e manter o encapsulamento.

CONCLUSÃO

Todos esses padrões servem para a resolução de problemas comuns típicos de desenvolvimento de software orientado a objeto. Tais técnicas apenas documentam e normatizam as práticas já consolidadas, testadas e conhecidas no mercado.

Práticas de engenharia de software são relativamente desconhecidas ou pouco aplicadas pelos desenvolvedores e há uma suposição de que é tudo complicado ou muito bonito no papel, mas se você realmente entender como eles funcionam e quando aplicá-los, pode-se perceber que é fácil implementar e obter seus benefícios.



Obrigado!

Alguma pergunta?

Você pode me encontrar em:

- ❖ willian_brito00@hotmail.com
- ❖ linkedin.com/in/willian-ferreira-brito
- ❖ github.com/willian-brito