RECONHECER O PROPÓSITO DOS PADRÕES GRASP E AS SITUAÇÕES NAS QUAIS ELES PODEM SER APLICADOS

PADRÕES GRASP

GRASP é o acrônimo para o termo em inglês ***General Responsibility Assignment Software Patterns***, termo definido por Craig Larman no livro intitulado ***Applying UML and patterns***, que define padrões gerais para a atribuição de responsabilidades em software.

Os padrões GoF tratam de problemas específicos em projeto de software.

Os padrões GRASP podem ser vistos como princípios gerais de projeto de software orientado a objetos aplicáveis a diversos problemas específicos.

Neste módulo, você vai aprender seis padrões GRASP:

- Especialista
- Criador
- Baixo acoplamento
- Alta coesão
- Controlador
- Polimorfismo

Especialista

- **Problema:**

Quando estamos elaborando a solução técnica de um projeto utilizando objetos, a nossa principal tarefa consiste em definir as responsabilidades de cada classe e as interações necessárias entre os objetos dessas classes, para cumprir as funcionalidades esperadas com uma estrutura fácil de entender, manter e estender.

- **Solução:**

Este padrão trata do princípio geral de atribuição de responsabilidades aos objetos de um sistema: atribua a responsabilidade ao especialista, isto é, ao módulo que traga o conhecimento necessário para realizá-la.

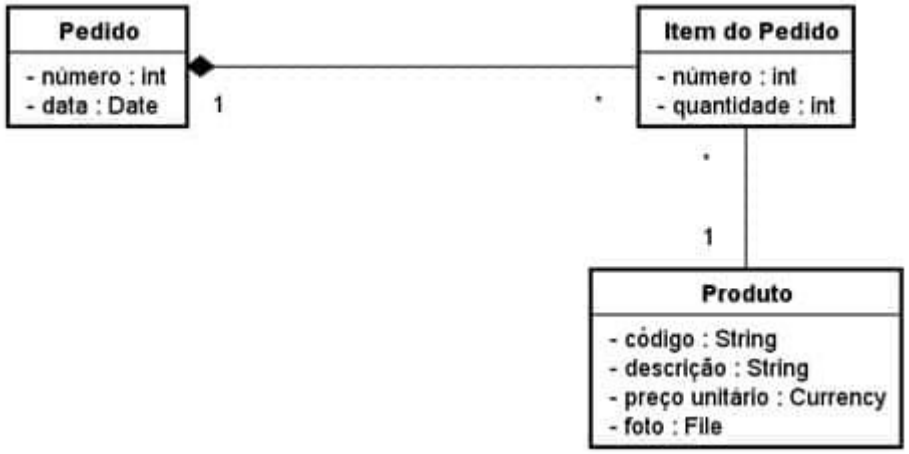
Comentário

Atribuir a responsabilidade ao especialista é uma heurística intuitiva que utilizamos no nosso cotidiano. A quem você atribuiria a responsabilidade de trocar a parte elétrica da sua casa? Possivelmente a alguém com o conhecimento necessário para realizar essa atividade (um especialista), ou seja, um eletricista.

Suponha que você esteja desenvolvendo um site de vendas de produtos pela internet.

A figura a seguir apresenta um modelo simplificado de classes desse domínio. Digamos que o site deva apresentar o pedido do cliente e o valor total do pedido.

Como você organizaria as responsabilidades entre as classes, para fornecer essa informação?



Padrão Especialista – classes de domínio.

O valor total do pedido pode ser definido como a soma do valor de cada um de seus itens.

Segundo o padrão **Especialista**, a responsabilidade deve ficar com o detentor da informação. Nesse caso, quem conhece todos os itens que compõem um pedido? O próprio pedido, não é mesmo? Então, vamos definir a operação **obterValorTotal** na classe Pedido.

E onde ficaria o cálculo do preço de um item do pedido?

Quais informações são necessárias para esse cálculo?

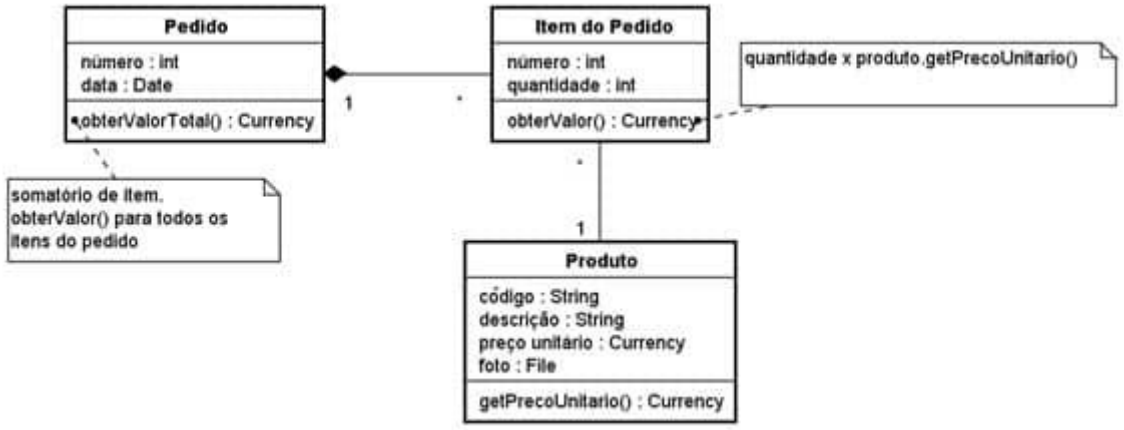
A quantidade e o preço do produto. Quem conhece essas informações?

A classe Item do Pedido conhece a quantidade e o produto associado. Vamos definir, então, uma operação **obterValor** na classe Item do Pedido.

E o preço do produto?

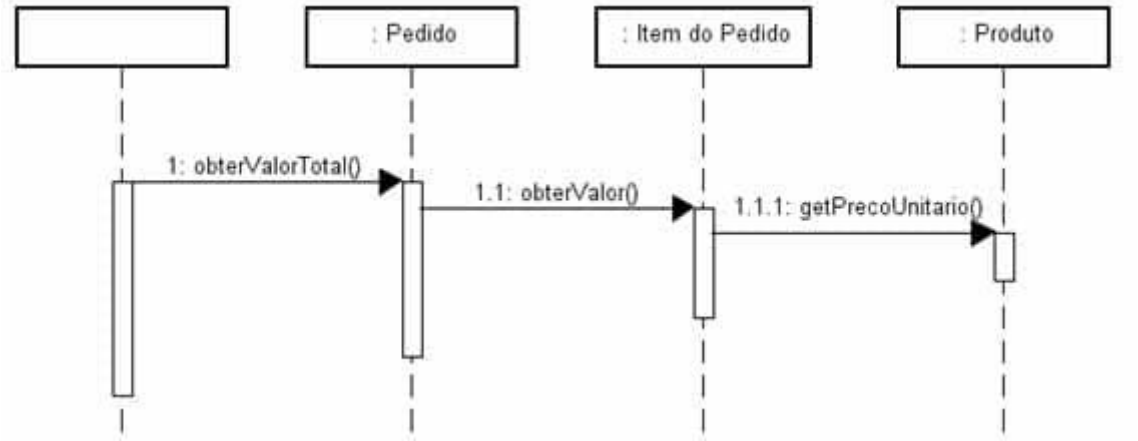
Basta o objeto item do pedido pedir essa informação para o produto, por meio da operação de acesso **getPrecoUnitario**.

A figura seguinte apresenta o diagrama de classes com a alocação de responsabilidades resultante.



Padrão Especialista – alocação de responsabilidades.

O diagrama de sequência da próxima figura ilustra a colaboração definida para implementar a obtenção do valor total de um pedido.



Padrão Especialista – colaboração (opção 1).

- **Consequências:**

Quando o padrão Especialista não é seguido, é comum encontrarmos uma solução deficiente (antipadrão) conhecida como "***God Class***" – que consiste em definir apenas operações de acesso (conhecidas como ***getters*** e ***setters***) nas classes de domínio – e concentrar toda a lógica de determinada funcionalidade do sistema em uma única classe, usualmente definida na forma de uma classe de controle ou de serviço. Essa classe implementa procedimentos utilizando as operações de acesso das diversas classes de domínio, que, nesse estilo de solução, são conhecidas como classes "idiotas".

Existem, entretanto, situações em que a utilização desse padrão pode comprometer conceitos como coesão e acoplamento.

Exemplo

Qual classe deveria ser responsável por implementar o armazenamento dos dados de um Pedido no banco de dados?

Pelo princípio do Especialista, deveria ser a própria classe Pedido, uma vez que ela detém todas as informações que serão armazenadas. Porém, essa solução acoplaria a classe de negócio com conceitos relativos à tecnologia de armazenamento (e.g. SQL, NoSQL, arquivos etc.), ferindo o princípio fundamental da coesão, pois a classe Pedido ficaria sujeita a dois tipos de mudança: mudanças no negócio e mudanças na tecnologia de armazenamento, o que é absolutamente inadequado.

Criador

- **Problema**

A instanciação de objetos é uma das instruções mais presentes em um programa orientado a objetos. Embora um comando simples, como *new ClasseX* em Java, resolva a questão, a instanciação indiscriminada – e sem critérios bem definidos – de objetos por todo o sistema tende a gerar uma estrutura pouco flexível, difícil de modificar e com alto acoplamento entre os módulos.

Portanto, a pergunta que este padrão tenta responder é:

Quem deve ser responsável pela instanciação de um objeto de determinada classe?

- **Solução**

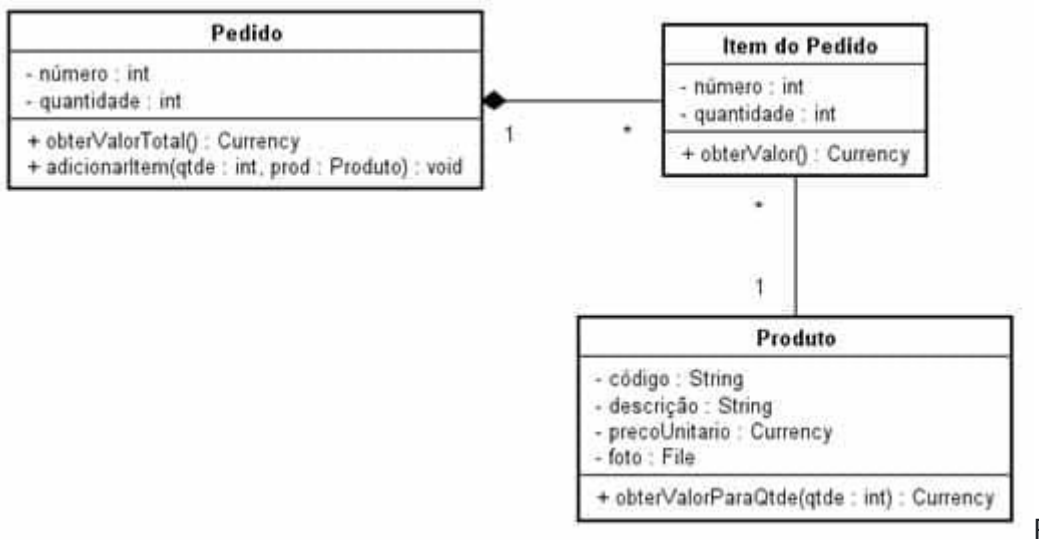
Coloque em uma classe X a responsabilidade de criar uma instância da classe Y se X for um agregado formado por instâncias de Y, ou se X possuir os dados de inicialização necessários para a criação de uma instância de Y.

No exemplo do site de vendas de produtos pela internet, considerando o modelo de classes da figura ***Especialista – classes de domínio*****, qual classe deveria ser responsável pela criação das instâncias de Item do Pedido?**

Resposta

Uma abordagem comum, mas inadequada, é instanciar o item em uma classe de serviço e apenas acumulá-lo no Pedido. Entretanto, quando se trata de uma relação entre um agregado e suas partes, a responsabilidade pela criação das partes deve ser alocada ao agregado, responsável por todo o ciclo de vida das suas partes (criação e destruição).

A figura a seguir apresenta o diagrama de classes após definirmos a operação **adicionarItem** na classe Pedido. Perceba que as operações vão sendo definidas nas diversas classes à medida que estabelecemos os mecanismos de colaboração para cada funcionalidade do sistema.



Padrão Creator – classes de domínio.

- **Consequências**

O padrão Criador é especialmente indicado para a criação de instâncias que formam parte de um agregado, por promover uma solução de menor acoplamento.

Por outro lado, o padrão Criador não é apropriado em algumas situações especiais, como, por exemplo, a criação condicional de uma instância dentro de uma família de classes similares.

Baixo acoplamento

- **Problema**

Acoplamento corresponde ao grau de dependência de um módulo em relação a outros módulos do sistema. Um módulo com alto acoplamento depende de vários outros módulos e tipicamente apresenta problemas como propagação de mudanças pelas relações de dependência, dificuldade de entendê-lo isoladamente e dificuldade de reusá-lo em outro contexto, por exigir a presença dos diversos módulos que formam a sua cadeia de dependências.

Outra questão importante com relação ao acoplamento é a natureza das dependências.

Se uma classe A depende de uma classe B, dizemos que A depende de uma implementação concreta presente em B.

Por outro lado, se uma classe A depende de uma interface I, dizemos que A depende de uma abstração, pois A poderia trabalhar com diferentes implementações concretas de I, sem depender de nenhuma específica.

De forma geral, sistemas mais flexíveis são construídos quando fazemos implementações (classes) dependerem de abstrações (interfaces), especialmente quando a interface abstrair diferentes possibilidades de implementação, seja por envolver diferentes soluções tecnológicas (ex.: soluções de armazenamento e recuperação de dados), seja por envolver diferentes questões de negócio (ex.:

diferentes regras de negócio, diferentes fornecedores de uma mesma solução de pagamento etc.).

Portanto, a pergunta que este padrão tenta responder é:

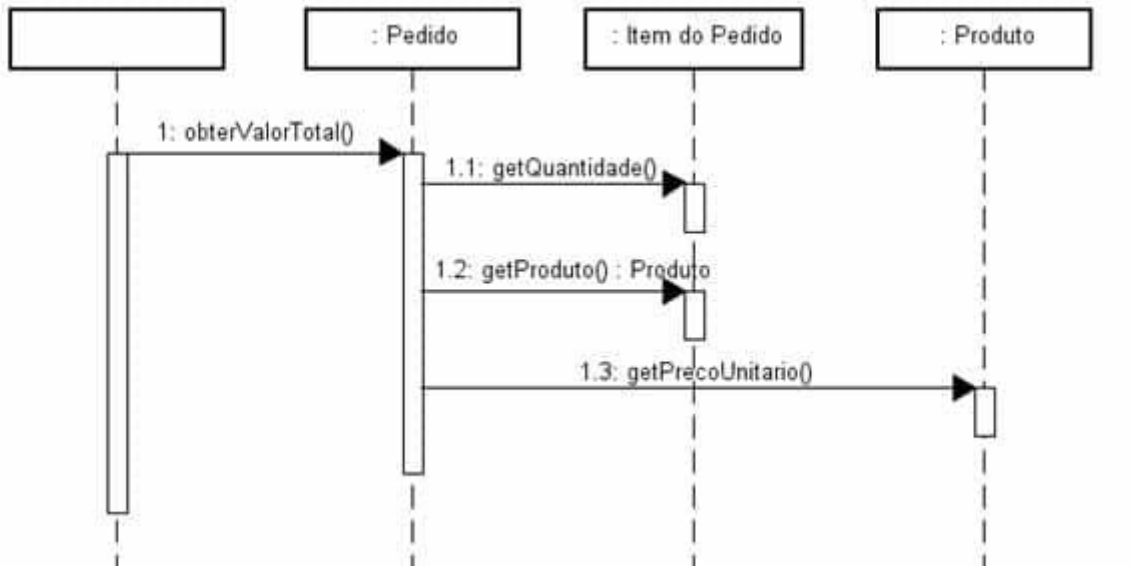
Como definir relações de dependência entre as classes de um sistema de forma a manter o acoplamento baixo, minimizar o impacto de mudanças e facilitar o reuso?

- **Solução**

Distribuir as responsabilidades de forma a gerar um baixo acoplamento entre os módulos.

Para você entender o conceito de baixo acoplamento, vamos apresentar um contraexemplo, isto é, uma situação em que foi criado um acoplamento maior do que o desejado. A figura seguinte apresenta outra solução para o problema mostrado no padrão Especialista, na qual fazemos o cálculo do preço do item do pedido dentro da operação obterValorTotal da classe Pedido. Para isso, percorremos todos os itens do pedido, obtemos a quantidade e o produto de cada item, o preço unitário do produto e multiplicamos a quantidade pelo preço unitário.

Essa solução gerou um acoplamento entre Pedido e Produto que não está presente na solução dada pelo padrão Especialista.



Padrão Baixo Acoplamento – exemplo de alto acoplamento.

- **Consequências**

Acoplamento é um princípio fundamental da estruturação de software que deve ser considerado em qualquer decisão de projeto de software. Portanto, avalie com cuidado se cada acoplamento definido no projeto é realmente necessário ou se existem alternativas que levariam a um menor acoplamento, ou, ainda, se alguma abstração poderia ser criada para não gerar dependência com uma implementação específica.

Atenção

Cuidado para não gerar soluções excessivamente complexas em que não haja motivação real para criar soluções mais flexíveis. Em geral, manter as classes de domínio isoladas e não dependentes de tecnologia (ex.: persistência, **GUI**, integração entre sistemas) é uma política geral de acoplamento que deve ser seguida, recomendada por diversas proposições de arquitetura.

Alta coesão

- **Problema**

Coesão é uma forma de avaliar se as responsabilidades de um módulo estão fortemente relacionadas e têm o mesmo propósito. Buscamos criar módulos com alta coesão, isto é, elementos com responsabilidades focadas e com um tamanho aceitável.

Comentário

Módulos ou classes com baixa coesão realizam muitas operações pouco correlacionadas, gerando sistemas de difícil entendimento, reuso, manutenção e muito sensíveis às mudanças.

Portanto, a pergunta que este padrão tenta responder é:

Como definir as responsabilidades dos módulos de forma que a complexidade resultante seja gerenciável?

- **Solução**

A solução consiste em definir módulos de alta coesão. Mas como se mede a coesão?

Coesão está ligada ao critério utilizado para reunir um conjunto de elementos em um mesmo módulo.

Coesão está ligada ao critério utilizado para reunir um conjunto de elementos em um mesmo módulo.

Coesão de um método de uma classe

Um método reúne um conjunto de instruções.

Coesão de uma classe

Uma classe reúne um conjunto de atributos e operações.

Coesão de um pacote

Um pacote reúne um conjunto de classes e interfaces.

Coesão de um subsistema

Um subsistema reúne um conjunto de pacotes.

A coesão de um módulo, seja ele classe, pacote ou subsistema, pode ser classificada de acordo com o critério utilizado para reunir o conjunto dos elementos que o compõem. Devemos estruturar os módulos de forma que eles apresentem coesão funcional, isto é, os elementos são agrupados porque, juntos, cumprem um único propósito bem definido.

As classes do pacote java.io da linguagem Java, por exemplo, estão reunidas por serem responsáveis pela entrada e saída de um programa. Nesse pacote, encontramos classes com responsabilidades bem específicas, como:

Arraste para os lados.

FileOutputStream
(para escrita de arquivos binários)
FileInputStream
(para leitura de arquivos binários)
FileReader
(para leitura de arquivos texto)
FileWriter
(para escrita de arquivos texto)

- **Consequências**

Coesão e acoplamento são princípios fundamentais em projetos de software. A base da modularidade de um software está na definição de módulos com alta coesão e baixo acoplamento.

Comentário

Sistemas construídos com módulos apresentando alta coesão tendem a ser mais flexíveis, mais fáceis de serem entendidos e evoluídos, além de proporcionarem mais possibilidades de reutilização e de um projeto com baixo acoplamento.

Entretanto, em sistemas distribuídos é preciso balancear a elaboração de módulos com responsabilidades específicas com o princípio fundamental de sistemas distribuídos, que consiste em minimizar as chamadas entre processos.

Controlador

- **Problema**

Um sistema interage com elementos externos, também conhecidos como atores. Muitos elementos externos geram eventos que devem ser capturados pelo sistema, processados e produzir alguma resposta, interna ou externa.

Por exemplo, quando o cliente solicita o fechamento de um pedido na nossa loja online, esse evento precisa ser capturado e processado pelo sistema. Este padrão procura resolver o seguinte problema:

A quem devemos atribuir a responsabilidade de tratar os eventos que correspondam a requisições de operações para o sistema?

- **Solução**

Atribuir a responsabilidade de receber um evento do sistema e coordenar a produção da sua resposta a uma classe que represente uma das seguintes opções:

Opção 1

Uma classe correspondente ao sistema ou a um subsistema específico, solução conhecida pelo nome ***Controlador Fachada***. Normalmente é utilizada em sistemas com poucos eventos.

Opção 2

Uma classe correspondente a um caso de uso onde o evento ocorra. Normalmente essa classe tem o nome formado pelo nome do caso de uso com um sufixo Processador, Controlador, Sessao ou algo similar.

Essa classe deve reunir o tratamento de todos os eventos que o sistema receba no contexto deste caso de uso. Esta solução evita a concentração das responsabilidades de tratamento de eventos de diferentes funcionalidades em um único *Controlador Fachada*, evitando a criação de um módulo com baixa coesão.

Atenção

Note que esta classe não cumpre responsabilidades de interface com o usuário. Em um sistema de *home banking*, por exemplo, o usuário informa todos os dados de uma transação de transferência em um componente de interface com o usuário e, ao pressionar o botão transferir, esse componente delega a requisição para o controlador realizar o processamento lógico da transferência. Assim, o mesmo controlador pode atender a solicitações realizadas por diferentes interfaces com o usuário (web, dispositivo móvel, totem 24 horas).

- **Consequências**

Normalmente, um módulo Controlador, ao receber uma requisição, coordena e controla os elementos que são responsáveis pela produção da resposta.

Em uma orquestra, um maestro comanda o momento em que cada músico deve entrar em ação, mas ele mesmo não toca nenhum instrumento.

Da mesma forma, um módulo Controlador é o grande orquestrador de um conjunto de objetos, cada qual com sua responsabilidade específica na produção da resposta ao evento.

Um problema que pode ocorrer com este padrão é alocar ao Controlador responsabilidades além da orquestração, como se o maestro, além de comandar os músicos, ficasse responsável também por tocar piano, flauta e outros instrumentos. Essa concentração de responsabilidades no Controlador gera um módulo grande, complexo e que ninguém se sente confortável em evoluir.

Polimorfismo

- **Problema**

Como evitar construções condicionais complexas no código?

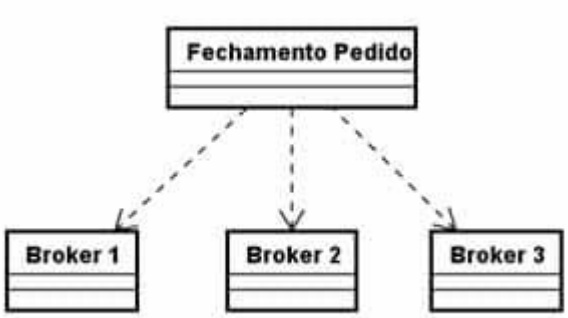
Suponha que você esteja implementando a parte de pagamento em cartão de uma loja virtual. Para realizar um pagamento interagindo diretamente com uma administradora de cartão, temos que passar por um processo longo e complexo de homologação junto a ela. Imagine realizar esse processo com cada administradora!

Para simplificar o problema, existem diferentes ***brokers*** de pagamento que já são homologados com as diversas administradoras e fornecem uma **API** para integração com a nossa aplicação. Cada ***broker*** tem uma política de preços e volume de transações e, eventualmente, podem surgir novos ***broker*** com políticas mais atrativas no mercado.

Agora imagine que fornecemos uma solução de software de loja virtual para diferentes lojas, que podem demandar diferentes ***brokers*** de pagamento em função das suas exigências de segurança, preço e volume de transações. Isso significa que o nosso software tem que ser capaz de funcionar com diferentes ***brokers***, cada um com a sua API.

Sem polimorfismo, esse problema poderia ser resolvido com uma solução baseada em ***if-then-else*** ou ***switch-case***, sendo cada alternativa de ***brokers*** mapeada para um comando case no switch ou em uma condição no if-then-else (figura a seguir).

Pense como ficaria esse código se houvesse vinte ***brokers*** diferentes.




```
FechamentoPedido

if (brokerConfigurado == BROKER_1)
    broker1.efetuarPagamento();
else if (brokerConfigurado == BROKER_2)
    broker2.efetuarPagamento();
else if (brokerConfigurado == BROKER_3)
    broker3.efetuarPagamento();
```

Solução sem polimorfismo.

O problema que esse padrão resolve é o seguinte:

Como escrever uma solução genérica para alternativas baseadas no tipo de um elemento, criando módulos plugáveis?

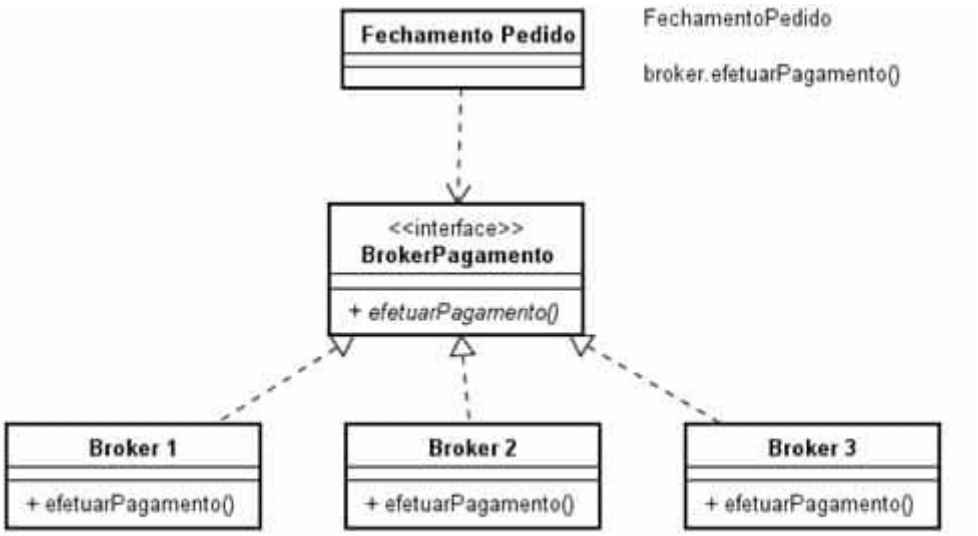
No nosso exemplo, as alternativas são todos os diferentes tipos de *brokers* com as suas respectivas APIs.

- **Solução**

Percebeu como a solução baseada em estruturas condicionais do tipo *if-then-else* ou *switch-case*, além de serem mais complexas, criam um acoplamento do módulo chamador com cada implementação específica? Note como a classe Fechamento Pedido depende diretamente de todas as implementações de *broker* de pagamento.

A solução via polimorfismo consiste em criar uma interface genérica para a qual podem existir diversas implementações específicas (veja na figura seguinte).

A estrutura condicional é substituída por uma única chamada utilizando essa interface genérica. O chamador, portanto, não precisa saber quem está do outro lado da interface concretamente provendo a implementação. Essa capacidade de um elemento (a interface genérica) poder assumir diferentes formas concretas (broker1, broker2 ou broker3, no nosso exemplo) é conhecida como polimorfismo.



Solução com polimorfismo.

- **Consequências**

Polimorfismo é um princípio fundamental em projetos de software orientados a objetos que nos ajuda a resolver, de forma sintética, elegante e flexível, o problema de lidar com variantes de implementação de uma mesma operação conceitual.

Dica

Esse princípio geral é utilizado para a definição de diversos padrões GoF, tais como: Adapter, Command, Composite, Proxy, State e Strategy.

Entretanto, é preciso ter cuidado para não implementar estruturas genéricas em situações em que não haja possibilidade de variação. Uma solução genérica é mais flexível, mas é preciso estar atento para não investir esforço na produção de soluções genéricas para problemas que sejam específicos por natureza, isto é, que não apresentem variantes de implementação.