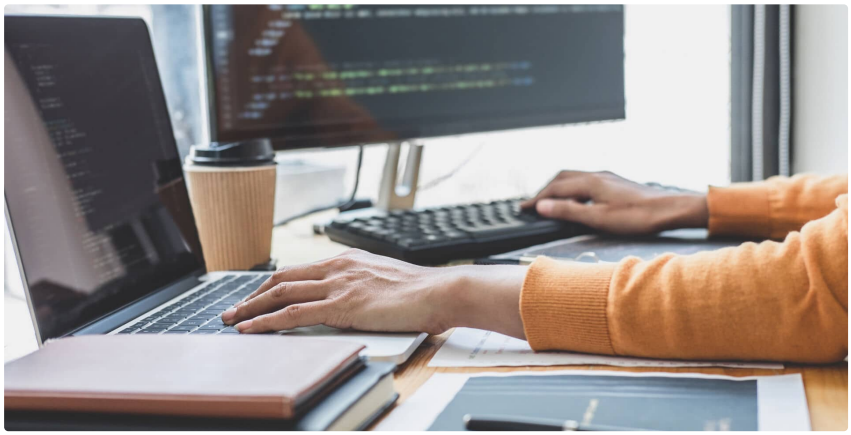


Padrões GRASP

Prof. Alexandre Luis Correa

Descrição

Padrões GRASP: Especialista na Informação, Criador, Acoplamento Baixo, Coesão Alta, Controlador, Polimorfismo, Invenção Pura, Indireção e Variações Protegidas.

Propósito

Compreender os padrões GRASP e identificar oportunidades para a sua aplicação são habilidades importantes para um projetista de software, pois eles ajudam a produzir sistemas flexíveis e reutilizáveis, contribuindo para que a evolução desses sistemas possa ser realizada em prazo e custo aceitáveis.

Preparação

Antes de iniciar a leitura deste conteúdo, é recomendado instalar em seu computador um programa que lhe permita elaborar modelos sob a forma de diagramas da UML (linguagem unificada de modelagem). Nossa sugestão é o Free Student License for Astah UML, que será usado nos exemplos deste estudo. Acesse os **arquivos Astah com diagramas UML** utilizados neste conteúdo.

Além disso, recomendamos a instalação de um ambiente de programação em Java: o Apache Netbeans. Porém, antes de instalar o Netbeans, é necessário ter instalado o JDK (Java Development Kit) referente à edição Java SE (Standard Edition), que pode ser encontrado no site da Oracle Technology Network.

Objetivos

Módulo 1

Padrões Especialista na Informação e Criador

Reconhecer o propósito e as situações de aplicação dos padrões Especialista na Informação e Criador.

Módulo 2

Padrões Coesão Alta e Controlador

Reconhecer o propósito e as situações de aplicação dos padrões Coesão Alta e Controlador.

Módulo 3

Padrões Acoplamento Baixo e Polimorfismo

Reconhecer o propósito e as situações de aplicação dos padrões Acoplamento Baixo e Polimorfismo.

Módulo 4

Padrões Invenção Pura, Indireção e Variações Protegidas

Reconhecer o propósito e as situações de aplicação dos padrões Invenção Pura,

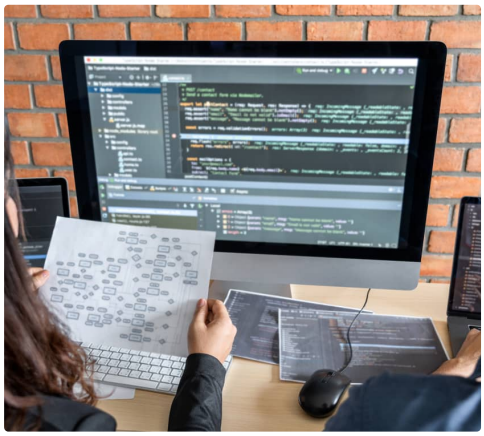
Indireção e Variações Protegidas.

Introdução

GRASP é o acrônimo para o termo em inglês *general responsibility assignment software patterns* proposto por Craig Larman no livro *Applying UML and patterns*, que define as diretrizes para a atribuição de responsabilidades em software. Os padrões GRASP podem ser vistos como os princípios gerais de um projeto de software orientado a objetos que são aplicáveis na solução de diversos problemas específicos.

A distribuição de responsabilidades pelos módulos do sistema é uma das tarefas mais importantes no desenvolvimento orientado a objetos. Diagramas UML são veículos que nos permitem expressar e discutir decisões sobre as responsabilidades de cada módulo.

No entanto, as decisões tomadas é que realmente são importantes. Com isso, seguir padrões e princípios bem estabelecidos aumenta nossas chances de tomar decisões que resultem em um software mais fácil de ser evoluído ao longo da sua existência. Neste conteúdo, você vai conhecer os padrões GRASP e entender os problemas de projeto de software que eles resolvem.



1 - Padrões Especialista na Informação e Criador

Ao final deste módulo, você será capaz de reconhecer o propósito e as situações de aplicação dos padrões Especialista na Informação e Criador.

O padrão Especialista na Informação

No desenvolvimento de um sistema orientado a objetos, é comum elaborar um modelo UML representando os principais conceitos e dados no escopo do sistema por meio de classes, atributos e associações. Entretanto, quando se elabora a solução técnica de um projeto, as principais questões a serem respondidas passam a ser:

- Quais responsabilidades cada classe deve possuir?
- Quais serão as interações necessárias entre os objetos dessas classes para o sistema realizar as funcionalidades esperadas?

Suponha que você esteja desenvolvendo um sistema de venda de produtos pela internet. A figura 1 apresenta um modelo simplificado de classes desse domínio.

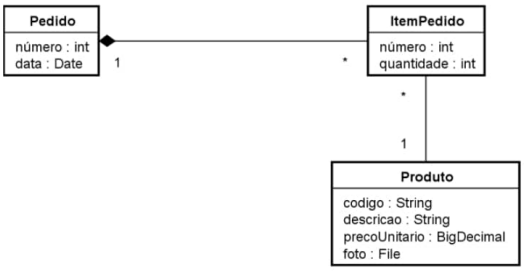


Figura 1 – Diagrama de classes.
Elaborado por: Alexandre Luis Correa

Digamos que o sistema deva listar os itens e o valor total do pedido do cliente, sendo que esse valor é uma informação calculada a partir dos produtos e das suas respectivas quantidades compradas. **Como você organizaria as responsabilidades entre as classes para fornecer essa informação?**

Solução do Especialista na Informação

Este padrão recomenda como princípio geral a atribuição correta de responsabilidade. Você pode estar se perguntando: mas o que isso significa?

> **Atribua a responsabilidade ao especialista, isto é, ao módulo que possua o conhecimento necessário para realizá-la.**

Atribuir a responsabilidade ao especialista é uma heurística intuitiva que utilizamos no nosso cotidiano. Suponha que você precise trocar o encanamento da sua casa. A quem você atribuiria essa responsabilidade? Provavelmente você recorrerá a um especialista para realizar tal atividade.

Atenção

A transposição dessa heurística do nosso cotidiano para o mundo de objetos em software é conhecida por heurística antropomórfica, ou seja, imaginamos os objetos como pessoas que possuem determinado conhecimento e que podem utilizá-lo para realizar algumas tarefas. Como projetista de software, você assume o papel de diretor desse universo de objetos, podendo definir quantas classes quiser – cada qual com as responsabilidades que você achar melhor.

Voltemos para o nosso problema do sistema de venda de produtos: o valor total do pedido pode ser definido como a soma do valor de cada um de seus itens. Segundo o padrão Especialista na Informação, a responsabilidade tem de ficar com o detentor da informação.

Nesse caso, quem conhece todos os itens que compõem um pedido? O próprio pedido, não é mesmo? Então que tal definir uma operação obterValorTotal na classe Pedido? Mas onde ficaria o cálculo do preço de cada item do pedido? Na própria classe Pedido? Pensemos um pouco mais nessa questão:

Quais informações são necessárias para esse cálculo?

A quantidade e o preço do produto, já que o valor de um item é a multiplicação desses dois valores.

Quem conhece essas informações?

A classe ItemPedido conhece a quantidade e o produto associado.

Vamos definir, portanto, uma operação obterValor na classe ItemPedido. E o preço do produto? Basta o objeto item do pedido solicitar essa informação ao produto (acessível pelo relacionamento entre eles) por meio da operação de acesso getPrecoUnitario.

O código a seguir apresenta a implementação das classes Pedido, ItemPedido e Produto, considerando essa distribuição de responsabilidades:

```
Java
```

Classes Pedido, ItemPedido e Produto.

O diagrama de sequência modela a colaboração entre os objetos que implementam o cálculo do valor total de um pedido, como exemplificada na figura 2.

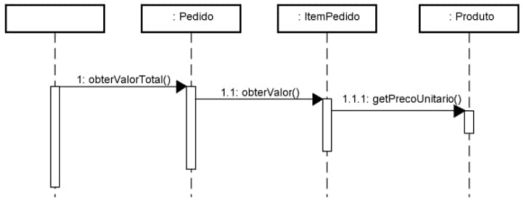


Figura 2 – Diagrama de sequência.
Elaborado por: Alexandre Luis Correa

Uma alternativa para a alocação da responsabilidade pelo cálculo do valor de um item do pedido seria considerar a classe Produto como a especialista em preço. No entanto, ela precisa de uma informação adicional (a quantidade) para cumprir essa responsabilidade.

Desse modo, podemos definir a operação obterValorParaQuantidade na classe Produto, fazendo com que a operação obterValor (definida em ItemPedido) passe a quantidade como argumento. Nesse caso, o cálculo do valor para determinada quantidade fica na classe Produto, que pode eventualmente aplicar políticas de desconto específicas conforme a quantidade e o tipo do produto, por exemplo. O código a seguir apresenta a implementação dessa alternativa de alocação de responsabilidades:

Java

Classe ItemPedido e classe Produto.

Consequências do Especialista na Informação

Normalmente, a realização de uma funcionalidade do sistema envolve a presença de diversos especialistas, pois cada classe possui uma parte das informações necessárias para resolver o problema. Dessa forma, será necessário estabelecer um mecanismo de colaboração entre os objetos – por intermédio da troca de mensagens – para realizar a função maior.

Quando o padrão Especialista na Informação não é seguido, é comum encontrar uma solução deficiente conhecida como *God Class*. Essa solução consiste em duas etapas:

1

Etapa 1

Definir, nas classes de domínio, apenas operações de acesso aos seus atributos (operações conhecidas como *getters* e *setters*).

2

Etapa 2

Concentrar a lógica de determinada funcionalidade do sistema em uma única classe (usualmente definida na forma de uma classe de controle ou de um serviço) na qual se encontram algoritmos complexos utilizando as operações de acesso das diversas classes de domínio, as quais, nesse estilo de solução, são meras fornecedoras de dados.

Há, porém, situações em que a utilização desse padrão pode comprometer conceitos mais fundamentais, como, por exemplo, a coesão e o acoplamento. Uma delas ocorre quando existe algum aspecto tecnológico envolvido, como é o caso do armazenamento de dados ou da interface com usuário.

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

Resposta

Pelo padrão Especialista na Informação, deveria ser a própria classe Pedido, uma vez que ela possui as informações que serão armazenadas. Contudo, se implementarmos a solução de armazenamento na própria classe Pedido, a classe de negócio ficará acoplada com conceitos relativos à tecnologia de armazenamento (exemplos: SQL, NoSQL e arquivos). Isso fere o princípio fundamental da coesão, pois a classe Pedido ficaria sujeita a duas fontes de mudança: mudanças no negócio e na tecnologia de armazenamento utilizada, o que é claramente inadequado.

O padrão Criador

A instanciação de objetos é uma das instruções mais presentes em um programa orientado a objetos. Sempre que necessário, pode-se utilizar o operador *new* para criar um objeto em Java. Entretanto, a instanciação indiscriminada – e sem critérios bem definidos – de objetos em diferentes partes do código tende a gerar uma estrutura pouco flexível, difícil de

A criação indiscriminada de objetos pode prejudicar a estrutura do programa em desenvolvimento.

modificar e com alto acoplamento
entre os módulos.

> **Quando se cria um objeto da classe B em um método da classe A por meio de um comando "new B()", estabelece-se uma relação de dependência entre duas implementações.**

A dependência acontece porque, em Java, uma classe é uma implementação concreta de um conjunto de operações. Nesse exemplo, portanto, A é dependente de B.

Em várias situações, entretanto, a criação de uma relação de dependência entre duas ou mais classes torna o sistema inflexível, dificultando a sua evolução. Deve-se estruturar um projeto para fazer com que as implementações dependam de abstrações, especialmente em casos nos quais um módulo depende de um serviço que pode ter diferentes implementações. Dessa forma, a pergunta que esse padrão tenta responder é: **quem deve ser responsável pela instanciação de um objeto de determinada classe?**

Solução do Criador

O padrão Criador recomenda atribuir a uma classe X a responsabilidade de criar uma instância da classe Y se uma ou mais das seguintes condições for(em) verdadeira(s):

- X é um agregado formado por instâncias de Y.
- X contém instâncias de Y.
- X registra instâncias de Y.
- X possui os dados de inicialização necessários para a criação de uma instância de Y.

No exemplo do sistema de vendas de produtos pela internet, considerando o modelo de classes ilustrado na figura 3, qual classe deveria ser a responsável por criar uma instância da classe ItemPedido?

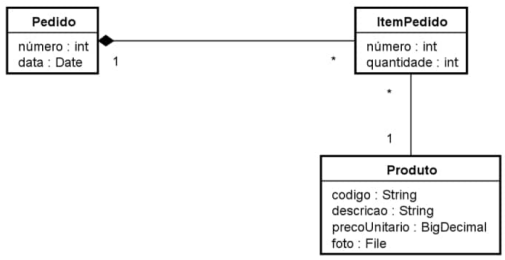


Figura 3 – Diagrama de classes.
Elaborado por: Alexandre Luis Correa

Uma abordagem comum – mas inadequada – é instanciar esse item em uma classe de serviço e apenas acumulá-lo no Pedido. Entretanto, quando se trata de um agregado, isto é, um objeto composto por vários itens, a responsabilidade pela criação dos itens, segundo o padrão Criador, deve ser alocada ao agregado, que é responsável por todo o ciclo de vida dos seus itens (criação e destruição).

Veja no código adiante que a classe Pedido contém uma lista de itens. Essa lista implementa o relacionamento de composição entre Pedido e ItemPedido, só podendo ser modificada dentro da classe Pedido, pois ela controla o ciclo de vida dos seus itens. A criação de um novo item do pedido é realizada pela operação adicionarItem que recebe os parâmetros quantidade e produto.

```
Java
```

Classe Pedido.

O diagrama ilustrado na figura 4 apresenta a interação entre os objetos nessa solução. Um objeto de serviço, por exemplo, solicita ao objeto Pedido que adicione um novo item correspondente ao produto e à quantidade de itens passados como argumentos da mensagem adicionarItem. O objeto Pedido, por sua vez, utiliza esses argumentos para instanciar um novo ItemPedido, acrescentando-o a uma lista de itens que implementa o relacionamento entre o pedido e os seus itens.

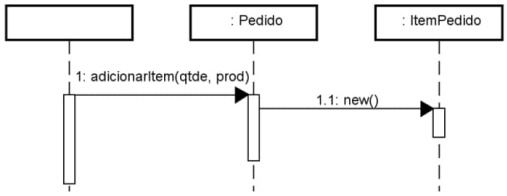


Figura 4 – Diagrama de sequência.
Elaborado por: Alexandre Luis Correa

Consequências do Criador

O padrão Criador é especialmente indicado para a criação de instâncias que formam parte de um agregado, pois o elemento que controla o ciclo de vida das suas partes é o próprio agregado, o qual, aliás, naturalmente já está relacionado com as suas partes. Esse padrão não é apropriado em algumas situações especiais, como é o caso da criação condicional de uma instância de uma família de classes similares.

Exemplo

Um sistema de vendas trabalha com diferentes soluções de pagamento. Por questões contratuais, a loja A opera com a solução de pagamento X; e a B, com a solução de pagamento Y. O problema passa a ser como criar a instância em conformidade com alguma parametrização externa feita para cada loja. Nesse caso, deve-se delegar a instanciação para uma classe auxiliar denominada Fábrica, aplicando o padrão de projeto GoF Abstract Factory.

De forma geral, quando a instanciação de objetos envolver cenários mais complexos, como o compartilhamento de objetos para racionalizar o uso de memória ou a criação de uma instância de uma família de classes similares condicionada ao valor de alguma configuração externa, será mais adequado aplicar padrões de projeto específicos, como os padrões GoF **Abstract Factory, Builder, Prototype** ou **Factory Method**.

Introdução aos padrões GRASP

No vídeo a seguir, apresentaremos uma visão geral dos padrões GRASP.

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.

Falta pouco para atingir seus objetivos.

Vamos praticar alguns conceitos?

Questão 1

Assinale a alternativa que expressa a intenção do padrão Especialista na Informação.

A — Padrão que recomenda alocar as operações pelas classes do sistema de acordo com o conhecimento que cada classe possui, sendo ele dado pelos seus atributos e relacionamentos.

B — Padrão que recomenda que cada classe seja desenvolvida pelo profissional com o maior conhecimento no assunto

relacionado à classe.

C — Padrão que recomenda a centralização da lógica de negócio em classes controladoras, ficando as classes de domínio como fornecedoras das informações via operações get/set, as quais, aliás, serão utilizadas por essas classes.

D — Padrão que recomenda a alocação de responsabilidades pelas classes de forma a aumentar o acoplamento entre os módulos do sistema.

E — Padrão que estabelece as diretrizes para a definição dos atributos das classes de um sistema.

Parabéns! A alternativa A está correta.

%0A%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%3Cp%20class%3D'c-paragraph'%3EO%20padr%C3%A3o%20Especialista%20na%20Informa%C3%A7%C3%A3o%20recomenda%20que%20a%20aloca%C3%A7%C3%

Questão 2

Marque a alternativa que expressa a intenção do padrão Criador.

A — O padrão Criador é indicado para situações nas quais é preciso fazer uso racional da memória, evitando a proliferação de réplicas de pequenos objetos por meio de uma solução baseada em compartilhamento.

B — O padrão Criador propõe uma solução para problemas nos quais é preciso instanciar uma classe de uma família de produtos conforme configuração realizada externamente.

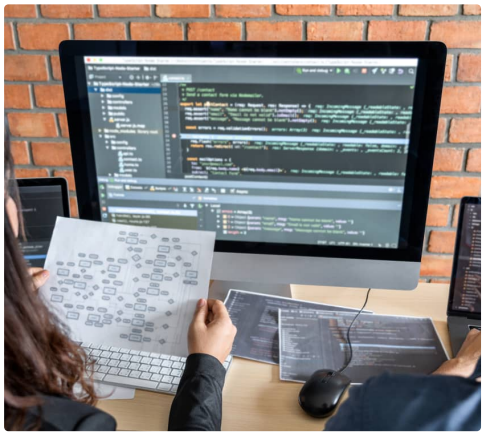
C — O padrão Criador oferece uma heurística para resolver problemas simples de instanciação de objetos, recomendando que um agregado seja responsável pela criação dos objetos que o compõem.

D — O padrão Criador propõe uma política de criação de módulos e pastas em um sistema de controle de versões.

E — O padrão Criador estabelece que as classes de um pacote devem ser instanciadas somente por classes de outros pacotes.

Parabéns! A alternativa C está correta.

%0A%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%3Cp%20class%3D'c-paragraph'%3EA%20alternativa%20A%20descreve%20a%20inten%C3%A7%C3%A3o%20do%20padr%C3%A3o%20de%20projeto%20Flyweight%



2 - Padrões Coesão Alta e Controlador

Ao final deste módulo, você será capaz de reconhecer o propósito e as situações de aplicação dos padrões Coesão Alta e Controlador.

O padrão Coesão Alta

Coesão é um conceito que nos permite avaliar se as responsabilidades de um módulo estão fortemente relacionadas e possuem o mesmo propósito. O objetivo é criar módulos com coesão alta, ou seja, módulos que tenham um propósito bem definido.

Módulos ou classes com coesão baixa realizam muitas operações pouco correlacionadas, gerando sistemas de difícil entendimento, reuso e manutenção, além de muito mais vulneráveis às mudanças. Portanto, a pergunta que esse padrão tenta responder é a seguinte: **como definir as responsabilidades dos módulos de forma que a complexidade do sistema resultante seja gerenciável, facilitando o seu entendimento e futuras evoluções?**

Solução da Coesão Alta

A solução proposta por esse padrão consiste em definir módulos de coesão alta.

Como se mede a coesão de um módulo?

O conceito de coesão está ligado ao critério utilizado para reunir um conjunto de elementos em um mesmo módulo. Note que esse conceito pode ser aplicado a módulos de diferentes níveis de granularidade:

Coesão de um método de uma classe

Um método reúne um conjunto de instruções. Pode-se avaliar se essas instruções formam um método com um propósito bem definido.

Coesão de uma classe

Uma classe reúne um conjunto de atributos e operações. Pode-se avaliar se esses atributos e essas operações formam uma classe com um propósito bem definido.

Coesão de um pacote

Um pacote reúne um conjunto de classes e interfaces. Pode-se avaliar se

essas classes formam um pacote com um propósito bem definido.

Coesão de um subsistema

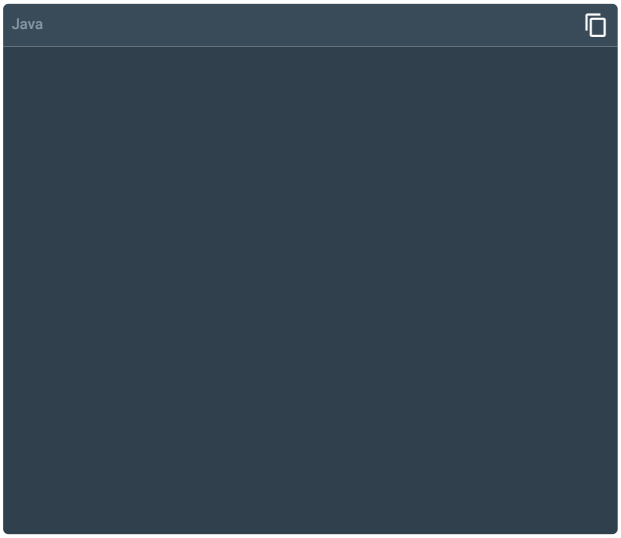
Um subsistema reúne um conjunto de pacotes.

A coesão de um módulo, seja ele uma classe, um pacote ou um subsistema, pode ser classificada de acordo com o critério utilizado para reunir o conjunto dos elementos que o compõem. Vamos conhecer agora esses critérios – do nível mais baixo para o mais alto de coesão.

Coesão coincidente

Quando os elementos estão agrupados em um módulo de forma arbitrária ou por conveniência, dizemos que esse módulo possui **coesão coincidente**. De forma esquemática, o exemplo mais à frente apresenta a classe Utils, que reúne operações, como a formatação de números, a conversão de medidas e o envio de arquivos via FTP. Esse módulo possui coesão baixa, pois muitas responsabilidades de diferentes naturezas estão reunidas em um único módulo.

Java



Classe Utils.

Atenção

O que normalmente motiva a criação desse tipo de módulo é a conveniência para os desenvolvedores de um projeto. O problema é que, se você quiser utilizar apenas uma dessas operações em um outro projeto, terá de trazer o módulo completo, incluindo todas as operações não utilizadas. Além disso, qualquer modificação feita em uma operação fará com que todo o módulo tenha de passar pelo ciclo de aprovação e de liberação.

Coesão lógica

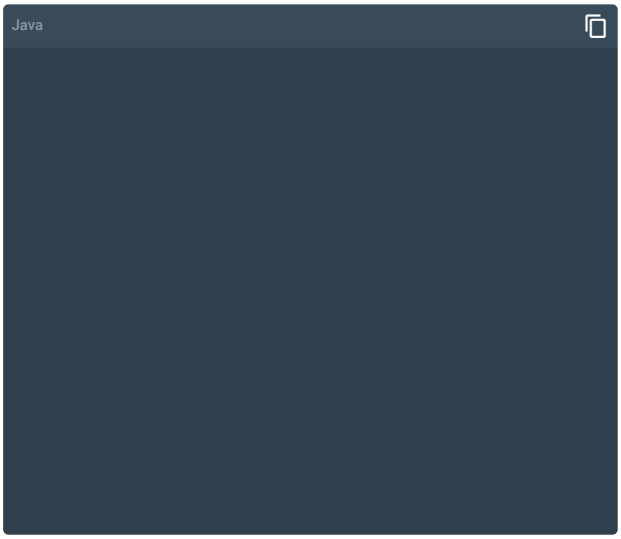
Em um módulo com coesão lógica, os elementos são agrupados por estarem logicamente relacionados ou por realizarem funções diferentes, ainda que tendo a mesma natureza. O código adiante apresenta a estrutura de implementação de uma classe que reúne operações de leitura de dados de produtos a partir de:

Um arquivo texto local.

Um arquivo obtido via FTP.

Um banco de dados relacional.

Esse é um exemplo de classe com coesão lógica, pois são reunidas em uma mesma classe todas as operações capazes de ler dados de produtos independentemente da fonte de dados disponível. Uma solução mais adequada seria separar a leitura de cada fonte em um módulo à parte.



Classe ProdutoRepository.

Coesão temporal

Um módulo com coesão temporal é aquele em que os seus elementos são agrupados por serem executados em determinado instante do tempo.

São colocadas no módulo Startup todas as operações executadas na inicialização do sistema, como, por exemplo:

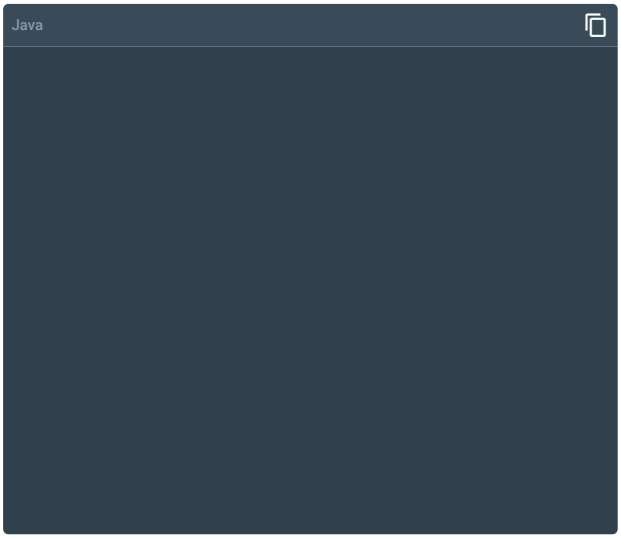
Inicialização do log do sistema

Inicialização da interface gráfica com o usuário

Inicialização das conexões com banco de dados

Inicialização das tarefas de segundo plano

Reúnem-se em um módulo, assim, as responsabilidades de diferentes naturezas por elas serem executadas em um momento específico, isto é, a inicialização do sistema. Uma solução mais adequada seria separar cada problema de inicialização (log, interface gráfica, banco de dados e assim por diante) em um módulo à parte. O código adiante ilustra um exemplo de módulo com coesão temporal.



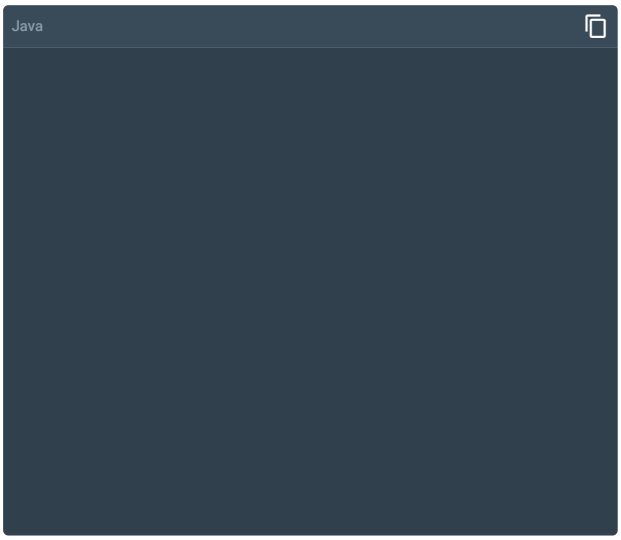
Classe Startup.

Coesão procedural

Um módulo com coesão procedural é aquele cujos elementos são agrupados por eles serem executados em determinada sequência utilizando diferentes conjuntos de dados. Normalmente, uma classe com essa coesão corresponde à implementação de uma God Class, ou seja, ela recebe uma requisição de execução de um serviço da aplicação e concentra nela própria o código de processamento dessa requisição em vez de dividir essa responsabilidade em operações menores de outras classes.

O código à frente apresenta um exemplo da estrutura de um módulo com coesão procedural.

Java



Classe ServicoPedido.

Nesse exemplo, a classe ServicoPedido reúne a implementação de todos os passos necessários para a realização do procedimento de confirmação do pedido, como, entre outros passos, a obtenção dos dados para pagamento, a aprovação do pagamento e o armazenamento do pedido no banco de dados.

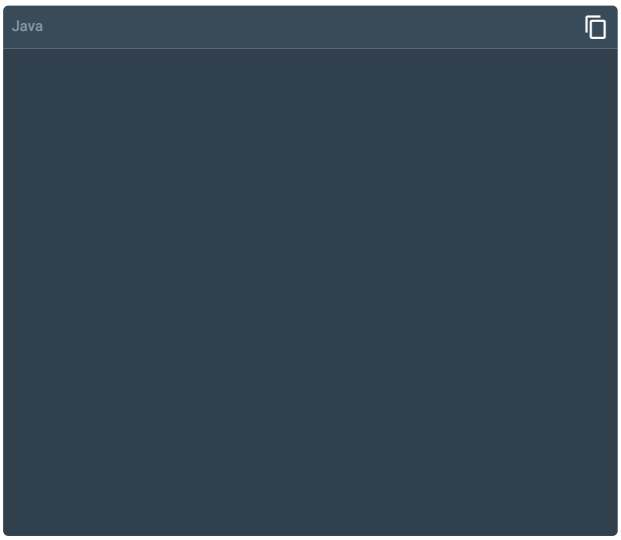
Atenção

Perceba que a classe ServicoPedido trata de assuntos completamente diferentes – ainda que necessários – para a confirmação do pedido. Esse estilo de construção deve ser evitado, pois dá origem a módulos de complexidade muito alta e com pouca flexibilidade.

Coesão de comunicação

Em um módulo com coesão de comunicação, os elementos são agrupados por eles realizarem funções diferentes utilizando o mesmo conjunto de dados. O código a seguir ilustra o esquema de um módulo com essa coesão.

Java



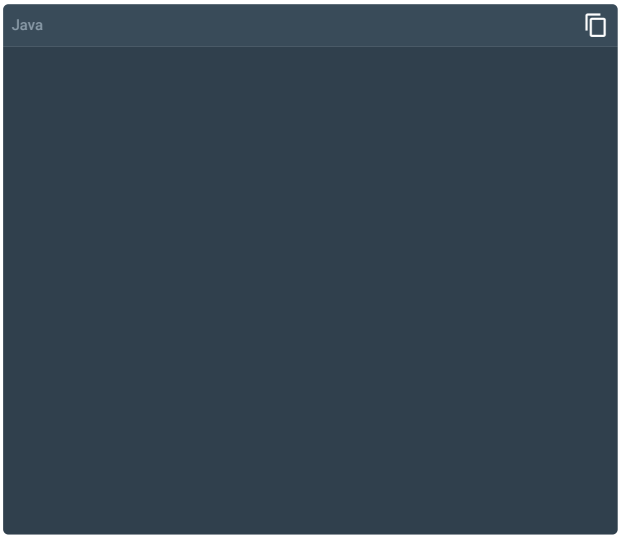
Classe Conta.

Observe que a classe Conta reúne operações de naturezas diversas, como obtenção do saldo, geração de extrato com envio para o cliente e armazenamento em banco de dados. O que elas têm em comum é o fato de trabalharem sobre o mesmo conjunto de dados, isto é, os dados da conta.

Coesão sequencial

Um módulo com coesão sequencial é aquele cujos elementos são agrupados por eles conterem todos os passos de execução de um procedimento sobre um mesmo conjunto original de dados, sendo que os resultados de um passo são utilizados como a entrada para o passo seguinte. A responsabilidade aqui é mais específica que a observada na coesão procedural, pois reúnem-se elementos que trabalham sobre o mesmo conjunto de dados, formando um *pipeline* de processamento.

O código adiante contém o esquema de um módulo com coesão sequencial, pois o procedimento transforma um bilhete recebido em formato string em um registro desse bilhete armazenado no banco de dados.



Classe ServicoBilhetagem.

Nesse exemplo, o procedimento é formado por três passos, sendo que o resultado de um passo é utilizado como a entrada para o passo seguinte. Além disso, os passos podem envolver o processamento de naturezas diversas.

Coesão funcional

Um módulo com coesão funcional reúne elementos que, juntos, cumprem um único propósito bem definido. As classes do pacote java.io da linguagem Java, por exemplo, estão nesse pacote pelo propósito de elas reunirem todas as responsabilidades de entrada e de saída.

Nesse pacote, pode-se encontrar classes com responsabilidades bem específicas. Entre muitas outras, destacam-se:

FileOutputStream

Para a escrita de arquivos binários.

FileInputStream

Para a leitura desses arquivos.

FileReader

Para a leitura de arquivos texto.

FileWriter

Para a escrita desses arquivos.

Dessa forma, o padrão Coesão Alta propõe a construção de módulos com coesão funcional. Cada módulo, portanto, deve reunir elementos que contribuam para que um único propósito bem definido seja atingido.

Consequências da Coesão Alta

Coesão é um dos princípios fundamentais em projetos de software. A base da modularidade de um software está na definição de módulos com coesão alta e acoplamento baixo. Sistemas construídos com módulos que apresentem uma coesão alta tendem a:

- Ser mais flexíveis.
- Ser mais fáceis de se entender e de se evoluir.
- Proporcionar maiores possibilidades de reutilização.
- Facilitar a elaboração de soluções de acoplamento baixo (enquanto a tendência dos módulos de coesão baixa seja a de gerar soluções de acoplamento alto).

No que tange à complexidade, os sistemas com coesão alta diferem dos sistemas com coesão baixa conforme vemos a seguir:

Coesão alta	Coesão baixa
A complexidade está distribuída por vários módulos, cada um contribuindo para resolver um pedaço específico do problema.	A complexidade fica centralizada em um pequeno número de módulos, os quais, muitas vezes, possuem centenas ou milhares de linhas de código.

O padrão Controlador

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

Exemplo

Quando o cliente solicita o fechamento de um pedido em uma loja on-line, esse evento precisa ser capturado e processado pelo sistema de vendas dela.

A quem devemos atribuir a responsabilidade de processar eventos que correspondam a requisições lógicas de execução de operações do sistema?

Essa é a pergunta que o padrão Controlador tenta responder.

Solução do Controlador

O padrão Controlador recomenda que a responsabilidade de receber um evento de sistema e de coordenar a produção da sua resposta precisa ser alocada a uma classe que represente uma das seguintes opções:

Opção 1

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

Opção 2

Uma classe correspondente ao caso de uso em que o evento ocorre. Nesse caso, essa classe pode ter o seu nome formado pelo nome do caso de uso e por um prefixo, como Processador, Controlador, Serviço ou algo similar. Ela ainda deve reunir o tratamento de todos os eventos que o sistema receba no contexto desse caso de uso. Tal solução é indicada como alternativa a concentrar as responsabilidades de tratamento de eventos de diferentes naturezas em um único Controlador Fachada, evitando, assim, a criação de um controlador com coesão baixa.

Um componente de interface captura os eventos de interface oriundos de teclado e mouse, por exemplo, e gera uma requisição para o controlador, que é o primeiro objeto, depois da camada de interface com o usuário, responsável por receber uma solicitação de sistema e coordenar a produção da respectiva resposta.

> O controlador não faz parte da interface com o usuário.

Em uma implementação Java Desktop, as classes do Java Swing geram as solicitações, enquanto, em uma implementação com JSP, os servlets é que as geram. Em uma aplicação web rich client, por sua vez, o código Javascript da interface com o usuário gera as solicitações para o controlador, o qual, portanto, fornece uma interface de alto nível para as diferentes formas de interação do usuário com o sistema.

Exemplo

Em um sistema de internet banking, o usuário informa todos os dados de uma transferência de valores para uma conta destino; ao pressionar o botão "transferir", o componente de interface com o usuário gera uma requisição para o controlador realizar o processamento lógico da transferência. Com isso, um mesmo controlador pode atender às solicitações realizadas por diferentes interfaces com o usuário (web, dispositivo móvel, totem 24 horas etc.).

Veja no diagrama representado na figura 5 que, quando o usuário clica no botão "transferir", após ter preenchido os dados da transferência, o componente UI (interface com o usuário) gera uma requisição para o controlador ServicoTransferencia, o qual, por sua vez, coordena a execução da transação interagindo com os objetos Conta e ContaRepository.

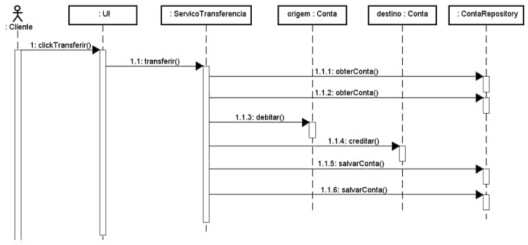


Figura 5 – Diagrama de sequência.
Elaborado por: Alexandre Luis Correa

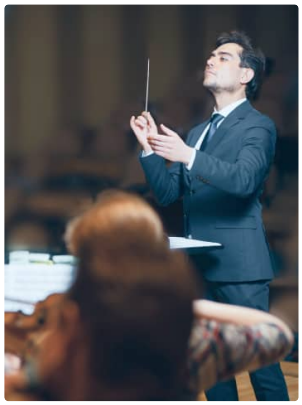
Atenção

O componente UI não realiza nenhuma operação ligada à lógica do negócio nem precisa conhecer os diversos componentes que compõem essa lógica. Ele, na verdade, apenas gera uma requisição para a classe controladora solicitando a execução da operação de sistema correspondente às interações realizadas pelo usuário com os componentes visuais.

Consequências do Controlador

Ao receber uma requisição, um módulo Controlador normalmente coordena e controla os elementos responsáveis pela produção da resposta. Imagine uma orquestra com um maestro e vários músicos. Ela conta com um maestro (controlador) que 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 esse padrão é alocar ao Controlador responsabilidades além da orquestração, como se o maestro, além de comandar os músicos, também ficasse responsável por tocar o piano, a flauta, o violino e outros instrumentos.



O controlador é o maestro que rege o conjunto de objetos.

> **A concentração de responsabilidades reduz a coesão do controlador.**

Uma importante consequência da utilização desse padrão é que os componentes de interface com o usuário não devem assumir a responsabilidade do tratamento de eventos lógicos de sistema. Eles devem apenas capturar as ações do usuário na interface e traduzi-las em um evento lógico de sistema a ser tratado por algum Controlador.

O Controlador Fachada (ou Controlador por Caso de Uso) corresponde à aplicação do padrão de projeto GoF Facade, pois essa classe de controle fornece uma interface de alto nível para a camada de interface com o usuário, isolando-a dos componentes internos da lógica do sistema.

Coesão de módulos de software

No vídeo a seguir, abordaremos a importância de se produzir módulos com coesão alta e os principais tipos de coesão.

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.

Falta pouco para atingir seus objetivos.

Vamos praticar alguns conceitos?

Questão 1

Assinale a alternativa correta sobre o padrão GRASP Coesão.

A — Buscamos produzir módulos com coesão baixa, isto é, poucos módulos devem concentrar as principais funcionalidades do sistema.

B — Buscamos produzir módulos com coesão alta, isto é, módulos que tenham um propósito bem definido.

C — Buscamos produzir módulos com coesão baixa, isto é, módulos que possuam um baixo grau de interdependência em relação a outros módulos.

D — Buscamos produzir módulos com coesão alta, isto é, poucos módulos devem concentrar as principais funcionalidades do sistema.

E — Buscamos produzir módulos com coesão baixa, isto é, módulos que tenham um propósito bem definido.

Parabéns! A alternativa B está correta.

%0A%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%3Cp%20class%3D'c-paragraph'%3EAs%20alternativas%20A%20e%20D%20est%C3%A3o%20incorretas%2C%20pois%20n%C3%A3o%20devemos%20concentrar%2

Questão 2

Aponte a alternativa correta sobre o padrão Controlador.

A O Controlador Fachada deve ser utilizado em sistemas com muitas funcionalidades e eventos.

B O controlador é uma parte da camada de interface com o usuário, sendo responsável por capturar os eventos gerados pelo usuário.

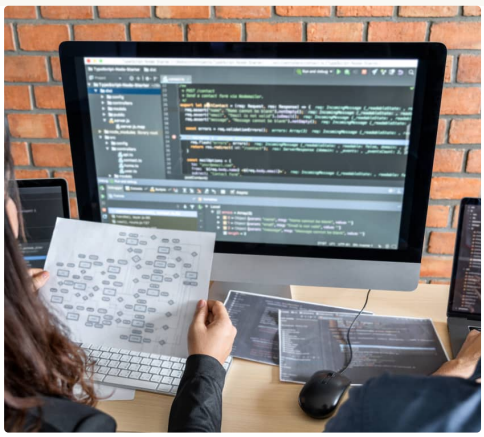
C O controlador é responsável pelo controle de acesso do usuário ao sistema.

D Em sistemas que possuam algoritmos complexos de cálculo, esses algoritmos devem ser implementados em classes do tipo Controlador.

E O controlador é um elemento de lógica de negócio responsável por coordenar a produção da resposta aos eventos lógicos gerada por componentes de interface do sistema com elementos externos.

Parabéns! A alternativa E está correta.

%0A%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%3Cp%20class%3D'c-paragraph'%3ENa%20alternativa%20A%2C%20o%20Controlador%20Fachada%20tem%20de%20ser%20utilizado%20em%20sistemas%20com%
se%20o%20uso%20de%20um%20controlador%20para%20cada%20caso%20de%20uso%3B%20na%20B%2C%20o%20controlador%20n%C3%A



3 - Padrões Acoplamento Baixo e Polimorfismo

Ao final deste módulo, você será capaz de reconhecer o propósito e as situações de aplicação dos padrões Acoplamento Baixo e Polimorfismo.

O padrão Acoplamento Baixo

Um acoplamento corresponde ao grau de dependência de um módulo em relação a outros do sistema. Um módulo com acoplamento alto depende de vários outros módulos e tipicamente apresenta problemas, como, por exemplo:

- Propagação de mudanças pelas relações de dependência, isto é, a mudança em um módulo causa um efeito cascata de mudanças nos módulos dependentes.
- Dificuldade para entender um módulo isoladamente.

- Dificuldade para reusar um módulo em outro contexto por exigir a presença dos diversos módulos que formam a sua cadeia de dependências.

Portanto, quando você estiver dividindo as responsabilidades pelos módulos de um software, pense sempre nos impactos que uma mudança pode provocar. Nesse sentido, podemos diferenciar os sistemas conforme a seguir:

Acoplamento baixo

As mudanças geram um impacto em poucas classes.

Acoplamento alto

As mudanças criam um efeito dominó que impacta muitas classes.

Atenção

Outra questão importante em relação ao acoplamento é a natureza das dependências. Se uma classe A depende de uma classe B, diz-se que A depende da implementação concreta presente em B. Por outro lado, se uma classe A depende de uma interface I, é dito que A depende de uma abstração, uma vez que A poderia trabalhar com diferentes implementações concretas de I sem depender diretamente de nenhuma implementação específica.

Em geral, sistemas mais flexíveis são construídos quando fazemos com que as implementações (classes) dependam de abstrações (interfaces). Isso é uma realidade especialmente nos casos em que a interface abstrai diferentes possibilidades de implementação, como:

Envolver diferentes soluções tecnológicas

Como, por exemplo, as soluções de armazenamento e recuperação de dados.

Contar com distintas questões de negócio

No caso, por exemplo, de diferentes regras de negócio e fornecedores de uma solução de pagamento on-line.

> Um sistema orientado a objetos é composto por vários objetos com responsabilidades específicas e bem definidas.

As relações de dependência são o desafio do padrão Acoplamento Baixo.

A complexidade do sistema emerge das relações de colaboração que estabelecemos entre os objetos. Quando projetamos um mecanismo de colaboração, naturalmente definimos relações de dependência e, portanto, geramos acoplamento entre os elementos envolvidos. Desse modo, o objetivo não é simplesmente minimizar dependências, e sim construir uma estrutura adequada e equilibrada delas. Cuida-se especialmente da natureza das dependências estabelecidas, ou seja, é dada uma preferência à presença das dependências em relação às abstrações em detrimento das dependências quanto às implementações específicas.

A pergunta que esse padrão tenta responder, portanto, é esta: **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 do Acoplamento Baixo

A solução proposta por esse padrão consiste em distribuir as responsabilidades a fim de gerar um acoplamento baixo entre os módulos.

> **Você sabe como se mede o grau de acoplamento entre os módulos?**

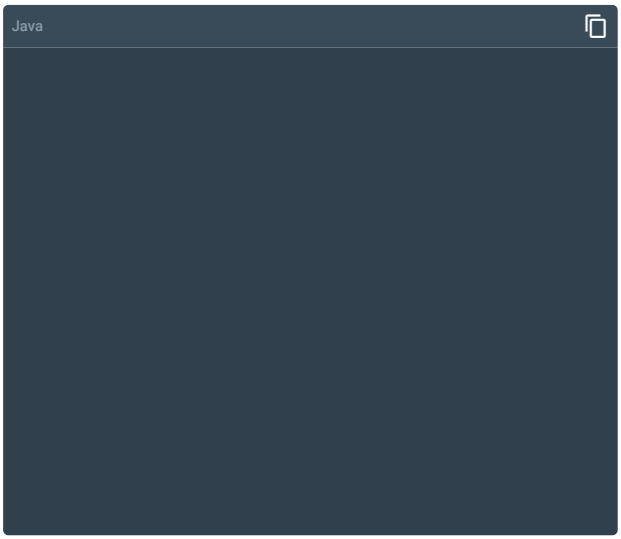
O grau de acoplamento está relacionado à forma com que uma relação de dependência é estabelecida entre dois módulos. A seguir, conheceremos tais formas do nível mais alto para o mais baixo de acoplamento.

Acoplamento de conteúdo

Ele ocorre quando um módulo utiliza aspectos de implementação de outro, ferindo o princípio a estabelecer que um módulo deve ocultar dos demais suas decisões de implementação, de forma que seja possível alterá-las sem se preocupar com os possíveis efeitos em outros módulos.

Veremos um exemplo de acoplamento de conteúdo. No código a seguir, a classe Data define seus atributos “dia”, “mês” e “ano” como públicos, enquanto a classe ModuloCliente acessa diretamente o atributo “mês” de uma instância de Data. Trata-se de um acoplamento de conteúdo, pois a forma de armazenamento da informação de Data está exposta e é diretamente utilizada pelos demais módulos.

Java



Classe Data e class ModuloCliente.

Você consegue visualizar o que aconteceria com os módulos que fazem o mesmo tipo de uso da classe Data caso resolvêssemos mudar a representação da data para texto ou para o número de dias julianos?

Acoplamento global

O acoplamento global entre dois módulos ocorre quando eles se comunicam por intermédio de recursos, como, por exemplo, variáveis globais. Ao contrário de C++, o Java não possui uma sintaxe para a definição de variáveis globais, mas é possível atingir um efeito similar ao se declarar em uma classe um atributo com os modificadores **public static**.

O código adiante exibe um exemplo de acoplamento global. A classe Globais define um conjunto de atributos que pode ser acessado globalmente.

Java

Classe Globais e class ModuloA.

Você consegue perceber que o módulo A é afetado por modificações realizadas pelo módulo B na propriedade **Globais.limiteParaSaque?** Embora A não utilize diretamente nenhum elemento de B, o comportamento de A é afetado pelas mudanças que B efetua em um recurso compartilhado por ambos.

Acoplamento externo

Ele ocorre quando módulos compartilham alguma parte do ambiente externo ao software, como, por exemplo, banco de dados, arquivos ou formato de dados. Para facilitar o entendimento, vamos explicar esse conceito por meio de duas situações hipotéticas:

1

Primeira situação

Imagine que você esteja desenvolvendo um sistema de vendas on-line no qual existem os módulos de Vendas e Estoque. Os dois módulos precisam acessar e atualizar dados referentes aos produtos vendidos na loja e, para isso, ambos utilizam a tabela Produto em um banco de dados relacional. Dessa forma, pode-se dizer que existe um acoplamento externo entre ambos. **Você consegue visualizar esse acoplamento?**

2

Segunda situação

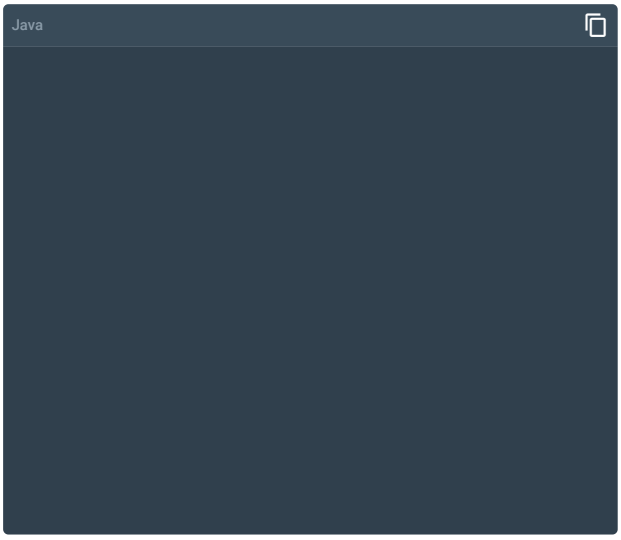
Suponha agora que você seja o responsável pelo módulo de Vendas e que outro desenvolvedor, responsável pelo de Estoque, resolva mudar a tabela Produto, criando alguns campos e modificando outros existentes. **Essas mudanças podem trazer impactos para o módulo Vendas? Possivelmente sim, não é mesmo?** Esses impactos são a evidência do acoplamento resultante do uso de recursos externos compartilhados.

Acoplamento de controle

Ocorre quando um módulo controla a lógica interna de outro por meio da passagem de alguma informação de controle. O código a seguir ilustra um exemplo de acoplamento de controle.

A operação obterProduto da classe ProdutoRepository pode recuperar dados de um produto a partir de um arquivo local, de um arquivo via FTP ou de um banco de dados relacional. Já a classe ModuloCliente chama a operação obterProduto,

passando uma informação de controle que indica de onde deve ser feita a leitura.



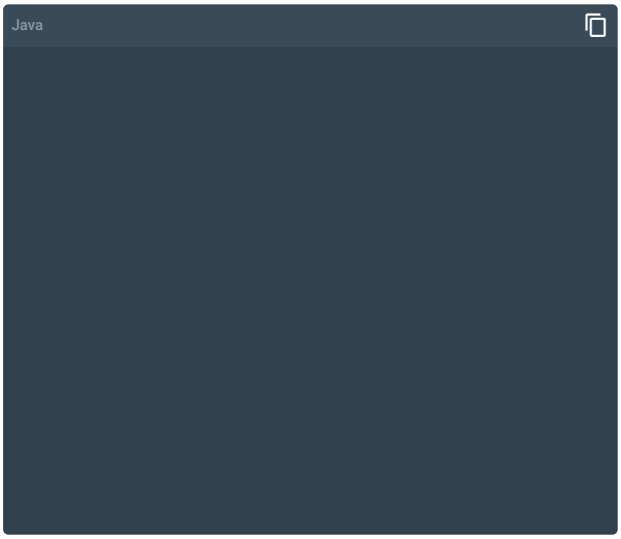
Classe ProdutoRepository e class ModuloCliente.

Observe que o módulo cliente não apenas está exposto a todas as implementações do módulo de leitura de dados do produto, como também controla esse módulo, indicando a implementação desejada. Uma solução menos acoplada teria o módulo cliente dependendo de uma abstração, ou seja, de um serviço capaz de obter um produto, sem que ele precise saber que existem diferentes implementações para esse serviço.

Acoplamento de estrutura

Ele acontece quando um módulo chamador passa uma estrutura de dados para um módulo chamado, o qual, por sua vez, utiliza apenas um pequeno subconjunto de dados dessa estrutura. O código à frente ilustra um exemplo de acoplamento de estrutura.

A classe Pedido contém todas as informações de um pedido. Já a classe CalculadoraFrete possui uma operação que calcula o frete de um pedido recebido como parâmetro. Entretanto, o algoritmo de cálculo do frete utiliza apenas o endereço de destino, o que configura um exemplo de utilização de uma pequena parcela (endereço de destino) de uma estrutura de dados (Pedido).



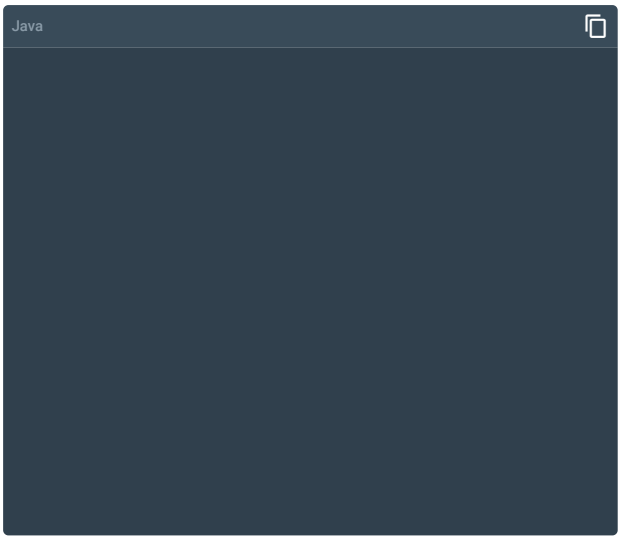
Classe Pedido, classe CalculadoraFrete e classe Exemplo.

Acoplamento de dados

O acoplamento de dados ocorre quando um módulo se comunica com outro, passando apenas os dados dos quais o módulo chamado precisa para cumprir a sua responsabilidade. Nesse tipo de acoplamento, os módulos são independentes e se comunicam por meio de dados.

O código adiante mostra um exemplo de acoplamento de dados no qual o módulo CalculadoraFrete recebe o endereço de destino, que é a informação necessária para calcular o frete.

Java



Classe Pedido, classe CalculadoraFrete e classe Exemplo.

Portanto, os níveis de acoplamento, do mais alto para o mais baixo, são:

Conteúdo – Global – Externo – Controle – Estrutura – Dados

Ao dividir as responsabilidades pelos módulos, devem ser balanceadas **três heurísticas** ligadas às relações de dependência entre eles:

Heurística 1

Minimizar o número de dependências entre os módulos sem comprometer a coesão, o que significa dizer que dependências desnecessárias não devem ser criadas.

Heurística 2

Utilizar, sempre que possível, os níveis mais baixos de acoplamento, preferencialmente o acoplamento de dados.

Heurística 3

Fazer com que as implementações dependam das abstrações, especialmente nas relações de dependência entre os módulos que implementam a lógica de negócio e aqueles relacionados à tecnologia, como, por exemplo, interface gráfica com o usuário, armazenamento de dados e integração com componentes externos.

Consequências do Acoplamento Baixo

O acoplamento é um princípio fundamental da estruturação de software. Ele, por isso, precisa ser considerado em qualquer decisão de projeto de software.

> Você sabe como os acoplamentos são criados concretamente em um programa?

Em linguagens como Java, por exemplo, um acoplamento direto entre o módulo X e o módulo Y é criado quando:

- Y é utilizado como o tipo de um atributo definido em X.
- Um método definido em X invoca operações definidas em Y (operações com escopo de classe).
- Um método definido em X utiliza uma instância de Y, seja ela criada localmente no método, recebida como parâmetro do método ou obtida com o retorno de uma operação invocada dentro desse método.
- Uma instância de Y é criada dentro de X via inicialização ou dentro de algum método específico de X.
- X é descendente (herda direta ou indiretamente) de Y.
- Y é uma interface, e X a implementa.

Além disso, os acoplamentos indiretos podem ser definidos pelo compartilhamento de recursos, como variáveis globais, arquivos e bancos de dados, por exemplo. Sempre que você utilizar uma dessas construções, um acoplamento será definido.

Sendo assim, avalie com cuidado se esse acoplamento é realmente necessário ou se há alternativas que levariam a um menor acoplamento. Observe ainda se você pode criar alguma abstração para não gerar uma dependência de uma implementação específica.

Atenção

Em geral, manter as classes de domínio isoladas e não dependentes de tecnologia – de armazenamento, de interface com usuário ou de integração entre sistemas, entre outras opções – constitui uma política geral de

acoplamento que deve ser seguida e que está presente em proposições, como, por exemplo, a arquitetura hexagonal ou a limpa.

O padrão Polimorfismo

Suponha que você esteja implementando os requisitos de pagamento em cartão de uma loja on-line. Para implementar um pagamento em cartão que interaja diretamente com uma administradora de cartão, é preciso passar por um longo e complexo processo de homologação perante a administradora. Imagine então realizar esse processo com cada uma!

Comentário

Atualmente existem diferentes brokers de pagamento homologados perante as diversas administradoras que fornecem uma API para a integração com sistemas de terceiros. No entanto, cada broker tem uma política de preços; eventualmente, podem surgir novos brokers com políticas mais atrativas no mercado.

Agora imagine que você seja o responsável por fornecer uma solução de sistema para diferentes lojistas que podem escolher os brokers de pagamento em função das suas exigências de segurança, preço e volume de transações, entre outros fatores de influência. Isso significa que o nosso software tem de ser capaz de funcionar com diferentes brokers, cada um com a sua API proprietária.

Quem não conhece polimorfismo resolveria esse problema com uma solução baseada em if-then-else ou switch-case, sendo cada alternativa de broker mapeada em um case no switch ou em um else no if-then-else, como ilustra o trecho de código a seguir. Pense como ficaria tal código se houvesse vinte brokers diferentes!

Java

Classe FechamentoPedido.

Esse problema possui uma complicação adicional, já que cada broker fornece uma API proprietária com interfaces diferentes. Os sufixos Broker1, Broker2 e Broker3 nos nomes das operações representam o fato de que as operações de cada API não possuem o mesmo nome e que também podem diferir nos argumentos passados na chamada de cada operação. Em resumo, o padrão Polimorfismo procura resolver os seguintes problemas:

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

Como evitar construções condicionais complexas contendo um código condicional baseado no tipo do objeto ou em alguma propriedade específica?

Comentário

No nosso exemplo, as alternativas baseadas no tipo de um elemento correspondem aos diferentes tipos de brokers com as suas APIs proprietárias.

Solução do Polimorfismo

Você percebeu como a solução baseada em estruturas condicionais do tipo if-then-else ou switch-case, além de ser mais complexa, cria um acoplamento do módulo chamador com cada implementação específica? Note como a classe FechamentoPedido depende diretamente de todas as implementações de broker de pagamento. Você se lembra de que um princípio geral de bons projetos orientados a objetos é haver implementações dependentes de abstrações, especialmente em casos nos quais essas abstrações possam ter várias implementações?

O polimorfismo é uma alternativa à complexidade imposta pelas estruturas condicionais.

> **A solução via polimorfismo consiste em criar uma interface genérica para a qual podem existir diversas implementações específicas.**

A estrutura condicional é substituída por uma única chamada, a qual, aliás, é feita utilizando essa interface genérica. O chamador, portanto, não precisa conhecer a classe que está do outro lado da interface provendo a implementação. A capacidade de um elemento (a interface genérica) poder assumir diferentes formas concretas (broker1, broker2 ou broker3, no exemplo anterior) é conhecida como **polimorfismo**.

O código adiante ilustra a estrutura de solução para o exemplo do fechamento do pedido com a utilização do padrão Polimorfismo.

Java

Fechamento do pedido com a utilização do padrão Polimorfismo.

Vamos analisar o código:

Criação da interface BrokerPagamento

Em vez de a classe FechamentoPedido utilizar diretamente todos os brokers, é criada uma interface BrokerPagamento para separar o módulo cliente dos módulos que implementam essa interface.

Criação de adaptadores

Como cada broker possui uma interface específica, é criado um conjunto de adaptadores que implementa a interface genérica BrokerPagamento e faz a conversão entre a interface genérica e a específica de cada broker. Desse modo, um adaptador de interface é criado para cada broker de pagamento.

Implementação da operação efetuarPagamento

Verifique como a classe Broker_1_Adapter implementa a operação efetuarPagamento chamando uma operação específica da API do broker 1. De forma análoga, a classe Broker_2_Adapter opera com a API do broker 2. Observe ainda como o código condicional anteriormente presente na implementação da classe FecharPedido foi substituído por uma simples chamada à operação efetuarPagamento de um objeto que implementa a interface BrokerPagamento.

Dica

Como esse objeto pode assumir múltiplas formas em tempo de execução, pois ele pode ser a instância de qualquer um dos adaptadores para os brokers específicos, dizemos que essa chamada é polimórfica.

Você percebeu que o polimorfismo permite adicionar novas soluções de broker sem que haja a necessidade de alterar o módulo chamador (FechamentoPedido)? Basta, para tal, implementar um novo adaptador e adicioná-lo à configuração do sistema, pois o restante do sistema já está preparado para trabalhar com esse novo broker.

Consequências do Polimorfismo

O 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 se lidar com variantes de implementação de uma mesma operação conceitual. O conceito dele está presente na definição de diversos padrões GoF. Listaremos alguns deles a seguir:

- Adapter;
- Command;
- Composite;

- Proxy;
- State;
- Strategy;

Entretanto, é preciso ter cuidado para não construir estruturas genéricas para situações nas quais não haja uma possibilidade de variação. Uma solução genérica se mostra mais flexível, mas é preciso estar atento a fim de não investir um esforço na produção de soluções genéricas para os problemas que sejam específicos por natureza, ou seja, que não apresentem variantes de implementação.

Padrão GRASP Acoplamento Baixo

No vídeo a seguir, abordaremos os diferentes tipos de acoplamento em projetos de software.

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.

Falta pouco para atingir seus objetivos.

Vamos praticar alguns conceitos?

Questão 1

Marque a alternativa correta sobre o acoplamento entre módulos.

A. Buscamos produzir módulos com acoplamento alto, isto é, que cumprem um propósito bem definido.

B. Buscamos produzir módulos com acoplamento baixo, ou seja, que cumprem um propósito bem definido.

C. Buscamos produzir módulos em que a lógica de negócio fica centralizada em um número reduzido de classes a fim de minimizar as relações de dependência entre elas.

D. Buscamos produzir módulos com acoplamento baixo, minimizando o grau de dependência entre eles.

E. Buscamos produzir módulos com acoplamento alto, minimizando as dependências de módulos em relação a abstrações.

Parabéns! A alternativa D está correta.

%0A%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%3Cp%20class%3D'c-paragraph'%3EAs%20alternativas%20A%20e%20B%20est%C3%A3o%20incorretas%2C%20pois%20o%20cumprimento%20de%20um%20prop%

Questão 2

Assinale a alternativa correta sobre o padrão Polimorfismo.

A. O polimorfismo pode ser implementado por um mecanismo de composição em que um objeto A recebe uma mensagem e delega a sua implementação para um objeto B que faz parte de A.

B. O polimorfismo recomenda a utilização do comando switch-case em vez de construções if-then-else com o objetivo de tornar o código mais legível.

C. O polimorfismo permite que um objeto referenciado em uma chamada de operação possa assumir diferentes formas em momentos distintos de execução dessa chamada.

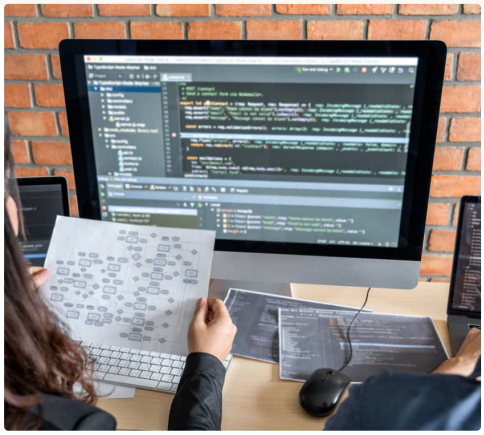
D. O polimorfismo permite que um objeto pertença a diversas classes simultaneamente.

E. O polimorfismo é um mecanismo utilizado para reduzir a abstração dos módulos, permitindo que eles possam ser

entendidos e modificados mais facilmente.

Parabéns! A alternativa C está correta.

%0A%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%3Cp%20class%3D'c-paragraph'%3ENa%20alternativa%20A%2C%20o%20polimorfismo%20%C3%A9%20implementado%20por%20heran%C3%A7a%20entre%20clas
then-
else)%3B%20na%20D%2C%20ele%20n%C3%A3o%20est%C3%A1%20associado%20ao%20fato%20de%20um%20objeto%20poder%20ser%20ir



4 - Padrões Invenção Pura, Indireção e Variações Protegidas

Ao final deste módulo, você será capaz de reconhecer o propósito e as situações de aplicação dos padrões Invenção Pura, Indireção e Variações Protegidas.

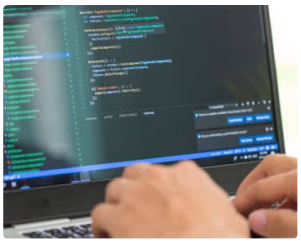
O padrão Invenção Pura

Você está desenvolvendo um sistema de vendas para uma loja on-line e identificou algumas classes que correspondem aos conceitos do negócio, tais como Produto, Cliente e Pedido. O padrão Especialista recomenda que uma responsabilidade seja atribuída ao módulo que possua o conhecimento necessário para realizá-la. Vimos que o cálculo do valor total do Pedido envolve o conhecimento de todos os itens e os seus respectivos valores. Ele, assim, deveria ser responsabilidade da classe Pedido.

> **Qual classe deveria ser a responsável por salvar os dados do Pedido em um banco de dados relacional?**

Seguindo o padrão Especialista, deveríamos alocar essa responsabilidade na classe detentora das informações, que é a própria classe Pedido. Contudo, essa solução comprometeria o padrão Coesão Alta, já que alocaríamos em um mesmo módulo responsabilidades de diferentes naturezas, ou seja, a lógica de negócio e de armazenamento em banco de dados relacional.

Solução da Invenção



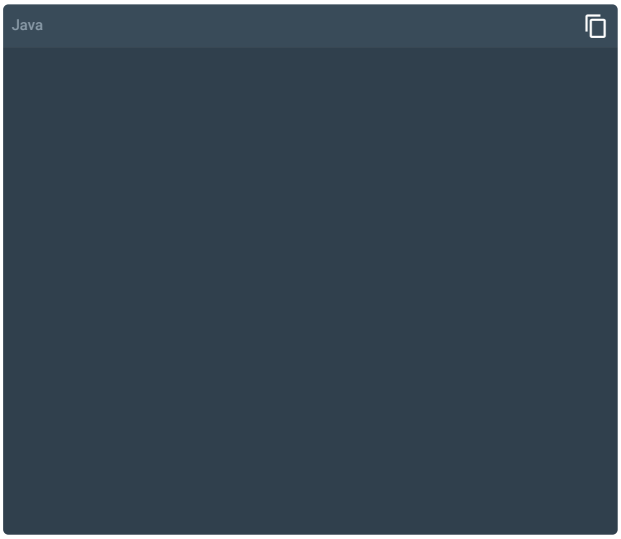
O padrão Invenção Pura consiste na criação de classes artificiais.

A solução proposta pelo padrão **Invenção Pura** diz respeito à criação de classes artificiais, isto é, classes que não representam um conceito do domínio do problema, para gerar soluções com coesão alta e acoplamento baixo. O problema do armazenamento do Pedido em um banco de dados pode ser implementado em uma classe PedidoRepository, a qual, por sua vez, fica responsável pelas operações básicas com pedidos em um meio de armazenamento como, por exemplo,

armazenar um novo pedido, atualizar os dados de um já existente, remover um pedido e recuperar um ou mais pedidos.

O código a seguir ilustra a implementação da classe Pedido com os atributos e as responsabilidades ligados ao domínio do negócio e à classe PedidoRepository, que é uma “Invenção Pura”, definida com a responsabilidade de armazenar e recuperar pedidos de um banco de dados relacional. Com isso, é gerada uma solução com módulos de coesão alta, já que as funcionalidades de diferentes propósitos estão segregadas em classes distintas.

Java



Classe Pedido.

Consequências da Invenção Pura

Esse padrão permite a criação de soluções com módulos de coesão alta e acoplamento baixo, sendo particularmente útil nas situações em que a heurística padrão para a aplicação do padrão Especialista na Informação gera módulos de coesão baixa.

A criação de classes correspondentes a conceitos e elementos concretos encontrados no domínio do negócio é um princípio natural de projetos orientados a objetos. Essa estratégia é chamada de **decomposição por representação**. Já a criação de classes por Invenção Pura resulta da estratégia chamada **decomposição por comportamento**. Vamos entender a diferença:

Decomposição por representação	Decomposição por comportamento
Essa estratégia visa produzir módulos que representem os elementos do domínio sem uma transformação abruta do universo real para a sua representação em software.	Essa estratégia é aplicada nas situações em que se desejar agrupar comportamentos ou algoritmos sem necessariamente vinculá-los diretamente a classes que representam os conceitos do domínio do negócio.

Exemplo

A loja on-line pode ter diferentes políticas de desconto para pedidos dependendo da época do ano, do total do pedido, do endereço de entrega ou da frequência de compras do cliente. Em vez de colocar todos esses algoritmos na classe Pedido, que é a detentora primária das informações do pedido, separam-se essas políticas em uma família de classes correspondente aos diversos algoritmos de cálculo de desconto aplicando o padrão Invenção Pura. Como consequência,

pode-se evoluir os algoritmos de desconto de forma independente da classe Pedido, gerando módulos com coesão mais alta e uma solução mais flexível.

É importante equilibrar as duas estratégias (decomposição por representação e por comportamento). Quando a estratégia de decomposição por representação está sendo subutilizada, surgem sinais, como, por exemplo, classes do domínio do problema contendo apenas atributos e operações de acesso (get/set) ou algumas classes concentrando a lógica de negócio em muitos métodos com dezenas ou centenas de linhas de código.

Padrões GoF, como o Adapter, o Strategy e o Command, são alguns exemplos de aplicação do padrão Invenção Pura, pois eles são baseados na decomposição de responsabilidades por comportamento.

O padrão Indireção

Indireção é uma das técnicas mais simples e mais utilizadas em projetos de software. Ela pode ser utilizada em diversas situações.

Veja a seguir alguns problemas que podem ser solucionados pelo padrão Indireção:

Comunicação entre objetos remotos

Um objeto A precisa chamar uma operação de um objeto remoto B. Como fazer com que A não precise lidar com questões, como sockets, serialização da mensagem e interpretação dos dados de retorno, entre outros aspectos inerentes à comunicação em rede?

Utilização de API de terceiros

Um objeto A deve chamar uma operação de um serviço B cujo acesso é fornecido por um fornecedor terceiro. Por isso, não é possível modificar o código da API de acesso ao serviço B. Imagine que eventualmente você tenha de trabalhar com outros fornecedores, cada qual com a sua API proprietária. Como isolar o objeto A das diferenças entre as APIs dos diversos fornecedores?

Abstração de tecnologia

Você precisa conectar diversos componentes clientes a uma camada de serviços que pode ser implementada em diferentes tecnologias, como, por exemplo, EJB, Web Services e REST API. Como fazer para que os clientes não precisem ser alterados em função de uma mudança na tecnologia de implementação da camada de serviços?

Redução de acoplamento

Um objeto A que captura uma interação do usuário com o sistema tem de interagir com diversos objetos de negócio para cumprir sua função. Como reduzir o acoplamento de A com os diversos objetos da camada de negócio, tornando a implementação desse objeto menos complexa?

Solução da Indireção

A solução proposta pelo padrão Indireção consiste em substituir a conexão direta entre dois ou mais objetos por uma estrutura de comunicação mediada por um objeto intermediário. Assim, se um objeto A enviar uma mensagem

diretamente para B, ele passará a mandá-la para um objeto intermediário X, o qual, por sua vez, fica responsável pela comunicação com B.

Observemos como o padrão Indireção pode ser utilizado nas situações descritas anteriormente:

Comunicação entre objetos remotos

Para isolar um objeto A cliente das complexidades de interação com um objeto remoto B, pode-se utilizar o padrão Gof Proxy, que consiste em definir um par de objetos (proxy e stub) entre os objetos A e B. O objeto proxy roda no mesmo processo de A oferecendo as mesmas operações de B, permitindo que A possa chamar operações de B como se eles estivessem rodando no mesmo processo. Já o objeto stub roda no mesmo processo de B, recebendo as requisições do proxy e repassando-as para o objeto B. Dessa forma, os objetos proxy e stub cuidam dos aspectos da comunicação remota, reduzindo a complexidade da implementação dos módulos A e B.

Utilização de API de terceiros

Quando um objeto A depende de um serviço que pode ter diversas implementações fornecidas por diferentes terceiros, deve-se segregar A desses fornecedores por intermédio da inserção de objetos adaptadores. Esses objetos traduzem uma interface comum conhecida por A nas chamadas proprietárias fornecidas por cada API. Essa é a solução proposta pelo padrão GoF Adapter.

Abstração de tecnologia

Padrões J2EE, como Service Locator e Business Delegate, são exemplos de solução para o problema da abstração de tecnologia. Eles encapsulam a forma com que um serviço é localizado e a tecnologia utilizada na sua implementação (exemplos: EJB, web service e REST API), fazendo com que todos os objetos clientes desse serviço se comuniquem com um objeto intermediário (Business Delegate) de forma independente de tecnologia. Esse objeto delega a chamada para o objeto real utilizando a tecnologia específica de comunicação. Assim, quando essa tecnologia mudar, basta alterar o Business Delegate, pois os clientes estão isolados do impacto dessa mudança.

Redução de acoplamento

Dois padrões GoF (Facade e Mediator) têm o propósito de reduzir o acoplamento entre objetos. O padrão Facade oferece uma interface de alto nível para clientes de um subsistema, reduzindo o número de objetos com os quais eles precisam interagir. Já o padrão Mediator reduz uma rede de dependências entre objetos do tipo N x N para uma topologia 1 x N, substituindo as comunicações diretas entre os objetos por aquelas mediadas por um objeto intermediário responsável por receber as notificações dos objetos e encaminhar o processamento para os objetos correspondentes.

Você conseguiu perceber como diversos padrões de projeto GoF são aplicações específicas do padrão Indireção?

Consequências da Indireção

Indireção é um princípio fundamental muito utilizado em padrões de projeto.

> **Trata-se de introduzir uma camada entre o cliente e o fornecedor a fim de desacoplá-los, promovendo soluções mais flexíveis, reusáveis e mais fáceis de estender e evoluir.**

O padrão Indireção permite:

- Isolar o cliente das tecnologias específicas utilizadas pelo fornecedor e das variações nos fornecedores de serviços consumidos pelo cliente.
- Reduzir o acoplamento entre os módulos.

Além de ser amplamente utilizado em padrões de projeto, esse padrão também é utilizado em áreas, como, por exemplo, balanceamento de carga e PAAS (Platform as a Service).

Atenção

A adição de um ou mais níveis de indireção pode causar algum impacto na performance do sistema. Embora, para a maioria dos sistemas de informação, o impacto não seja relevante, nos sistemas cujo tempo de resposta seja muito crítico, como jogos e sistemas de controle em tempo real, por exemplo, o uso excessivo de indireções poderá penalizar a performance.

O padrão Variações Protegidas

Você sabe qual é um dos maiores pesadelos de quem precisa evoluir um sistema tendo de atender a novas demandas em um prazo cada vez mais reduzido? É quando uma alteração aparentemente simples para o cliente demanda um enorme esforço de implementação, pois diversos módulos precisarão ser modificados para atender a esse novo requisito. Essa é uma situação claramente indesejável, ocorrendo em sistemas que possuem uma estrutura frágil. **Mas o que causa tal fragilidade?**

Resposta

Essa fragilidade é resultante das relações de dependência e da forma como organizamos as decisões nos módulos que criamos. Sempre que se projeta um sistema, é importante avaliar os aspectos que podem variar e as consequências resultantes dessas variações.

Pense em um sistema da operação de bancos. Trata-se daquele que permite alguém ter uma conta corrente e fazer depósitos, saques, transferências, por exemplo. Essas operações existem há décadas.

> **Agora reflita: nos últimos 20 anos, o que mudou nesses sistemas?**

Os sistemas bancários estão em constante evolução.

Um aspecto que claramente vem mudando com frequência nesse período está ligado às tecnologias utilizadas para o cliente interagir com o sistema: caixa eletrônico, internet banking e aplicativo no celular são só alguns exemplos.

Outro aspecto de mudança diz respeito às regras de execução dessas operações, como ocorre nos limites para saques ou nas retiradas limitadas pelo local onde você faça tal operação. Existem, enfim, muitas regras relacionadas à segurança das operações.

Nesse pequeno exemplo, você já percebe que há pelo menos dois aspectos sujeitos a variações:

Aspectos de interação com o usuário

A forma com que ele solicita uma transferência de valores, por exemplo, em cada interface onde ela esteja disponível.

Regras de negócio

As políticas e regras que precisam ser observadas em cada operação desempenhada.

Sabendo que esses elementos estão sujeitos a variações, nossa missão, como projetistas, é estruturar os módulos a fim de que essas variações possam ser acomodadas com um impacto localizado em um pequeno número de módulos, gerando, com isso, uma estrutura flexível e adaptável às mudanças cuja ocorrência julgamos possível.

Vamos ilustrar essa questão com um exemplo. Você está implementando um módulo do sistema de vendas on-line responsável por coordenar o fechamento do pedido realizado pelo cliente. Uma das tarefas desse módulo é solicitar o armazenamento do pedido àquele responsável pela persistência dos pedidos. Uma possível implementação para essa tarefa será ilustrada no código a seguir.

Java

Classe ServicoFechamentoPedido e classe PedidoRepository.

Vimos no código que a classe ServicoFechamentoPedido recebe um repositório (PedidoRepository) para o qual ele solicita a execução da operação inserirPedido. A classe PedidoRepository é uma implementação que faz o armazenamento e a recuperação de pedidos de um banco de dados relacional.

Nessa solução, uma implementação (módulo cliente ServicoFechamentoPedido) está acoplada a outra (módulo fornecedor PedidoRepository), o que compromete a flexibilidade da solução.

Solução das Variações Protegidas

A solução proposta pelo padrão Variações Protegidas é identificar pontos sujeitos à variação e isolar essas variações com a criação de interfaces estáveis no seu entorno.

Suponha que a nossa loja on-line tenha de funcionar com pedidos armazenados em diferentes servidores de bancos de dados, ou seja, nosso sistema tem de se adaptar ao servidor de banco de dados do lojista, que pode ser:

Relacional

NoSQL

Como Oracle, Microsoft SQL Server e MySQL.	Como MongoDB, Redis e Cassandra.

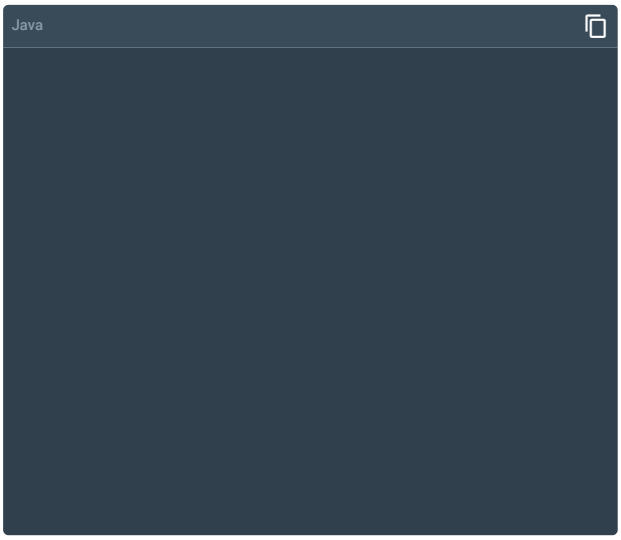
Na solução original, a classe ServicoFechamentoPedido depende diretamente de uma implementação concreta que acessa bancos de dados relacionais.

> **Mas como poderemos projetar a solução a fim de que seja possível variar a implementação de armazenamento de pedidos sem que essa classe precise ser modificada?**

Quando estabelecemos uma relação de dependência, temos de avaliar se o módulo fornecedor pode possuir implementações alternativas. Há, por exemplo, outras formas de armazenar e recuperar pedidos que não seja via banco de dados relacional?

Caso a resposta seja afirmativa, será conveniente isolar o cliente do fornecedor introduzindo uma abstração. O código adiante ilustra uma solução:

Java



Solução das Variações Protegidas.

Na solução apresentada, nós protegemos o módulo cliente de variações na implementação do módulo fornecedor com a criação de uma interface estável, a qual, por sua vez, deve ser implementada em cada possível variante.

A classe ServicoFechamentoPedido passa a utilizar a interface PedidoRepository, que é implementada em cada variante específica.

Atenção

PedidoRDBMSRepository é uma implementação das operações apresentadas em banco de dados relacional, enquanto PedidoMongoRepository corresponde à implementação utilizando o MongoDB.

Consequências das Variações Protegidas

Um dos grandes desafios do trabalho de um arquiteto ou desenvolvedor de software é identificar e proteger os pontos de variação de um software. O padrão Variações Protegidas constitui um dos conceitos mais presentes em mecanismos e padrões de desenvolvimento pela flexibilidade e proteção que ele proporciona em relação a variações em dados, comportamento, componentes e sistemas externos.

Pode-se utilizar esse padrão aplicando conceitos básicos de orientação a objetos, como encapsulamento, interfaces e polimorfismo. Soluções mais avançadas, porém, também podem ser combinadas, como linguagens baseadas em regras, interpretadores de regras e metaprogramação com as APIs de Reflection e Annotation do Java. Vários padrões GoF, como o Adapter, Strategy, Abstract Factory e o Bridge, por exemplo, fazem uso do padrão Variações Protegidas.

Atenção

É importante verificar se o esforço envolvido para projetar e implementar uma solução que proteja partes do sistema de variações terá um retorno adequado. Dessa maneira, deve-se avaliar a probabilidade de as variações em determinados pontos do sistema acontecerem e o impacto dessas mudanças, priorizando as variações mais prováveis e de maior impacto.

Padrão GRASP Indireção

No vídeo a seguir, abordaremos a aplicação do padrão Indireção.

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.

Falta pouco para atingir seus objetivos.

Vamos praticar alguns conceitos?

Questão 1

Aponte a alternativa correta em relação ao padrão Invenção Pura.

A — O padrão Invenção Pura recomenda que o software seja construído por classes que correspondam puramente a conceitos do domínio do negócio.

B — O padrão Invenção Pura corresponde a invenções do projetista para proteger módulos clientes quanto a variações de implementação dos módulos fornecedores.

C — O padrão Invenção Pura consiste em eliminar a dependência direta de um objeto A em relação a um objeto B, introduzindo um objeto intermediário que captura a chamada de A e faz algum processamento adicional, transformação ou adequação antes de repassar a chamada para B.

D — O padrão Invenção Pura diz respeito à criação de classes que não representam um conceito do domínio do problema com o objetivo de gerar soluções com maior coesão e menor acoplamento daquilo que seria obtido pela aplicação do padrão Especialista na Informação.

E — O padrão Invenção Pura significa criar uma interface simplificada de alto nível de abstração para um cliente acessar os serviços oferecidos por um subsistema.

Parabéns! A alternativa D está correta.

%0A%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%3Cp%20class%3D'c-paragraph'%3EA%20alternativa%20A%20est%C3%A1%20incorreta%2C%20pois%20o%20padr%C3%A3o%20sugere%20justamente%20o%20co
se%20inventar%20elementos%20que%20n%C3%A3o%20correspondem%20a%20conceitos%20do%20dom%C3%ADnio%20do%20neg%C3%B3

Questão 2

Assinale a alternativa correta em relação ao padrão de projeto indireção.

A — Padrão que estabelece uma forma de comunicação indireta entre dois módulos com o objetivo de reduzir a coesão desses módulos.

B — Padrão que recomenda introduzir um objeto entre um módulo cliente e um fornecedor de um serviço com o propósito de reduzir o acoplamento entre eles.

C — Padrão que recomenda a centralização da lógica de negócio em classes controladoras, ficando as classes de domínio responsáveis pela definição das informações e das operações get/set a serem utilizadas pelas classes controladoras.

D — Padrão que recomenda a alocação de responsabilidades pelas classes a fim de aumentar o acoplamento entre os módulos do sistema.

E — Padrão que visa reduzir a abstração dos módulos, permitindo que eles possam ser entendidos e modificados mais facilmente.

Parabéns! A alternativa B está correta.

%0A%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%3Cp%20class%3D'c-paragraph'%3EA%20alternativa%20A%20est%C3%A1%20incorreta%2C%20pois%20o%20objetivo%20%C3%A9%20reduzir%20o%20acoplament

Considerações finais

Neste conteúdo, verificamos como os padrões GRASP podem ser utilizados para distribuir as responsabilidades do sistema entre as classes com o propósito de gerar uma solução com módulos mais coesos e menos acoplados. Vimos ainda que o padrão Especialista na Informação estabelece uma heurística simples de atribuição de responsabilidade baseada no conhecimento que cada classe possui.

Em seguida, pontuamos que o padrão Criador recomenda que os agregados sejam responsáveis por criar as suas partes. Observamos também que Coesão Alta e Acoplamento Baixo são padrões fundamentais, pois, enquanto o conceito de coesão está relacionado ao propósito específico de cada módulo, o padrão Acoplamento visa controlar as relações de dependência entre os módulos para que os impactos de futuras modificações sejam minimizados.

Ainda apontamos que o padrão Controlador aborda a atribuição da responsabilidade de coordenar a produção de respostas aos eventos de sistema e que o padrão Polimorfismo é um conceito da orientação a objetos a permitir que as implementações dependam de abstrações, e não de outras implementações concretas. Por fim, destacamos que os padrões Indireção e Variações Protegidas sugerem mecanismos para reduzir o acoplamento entre os objetos, permitindo que os módulos fornecedores de serviços possam variar sem impactar os módulos clientes.

Ouça o podcast. Nele, falaremos da importância do padrão GRASP e das principais dificuldades enfrentadas para a sua aplicação.

Para ouvir o *áudio*, acesse a versão online deste conteúdo.

Explore +

Para saber mais sobre a programação orientada a objetos, acesse o site da DevMedia e leia este artigo: **Utilização dos princípios Solid na aplicação de padrões de projeto.** Consultado na internet em: 18 out. 2021.

O site **Patterns in Practice: Cohesion And Coupling** discute e apresenta exemplos dos conceitos de coesão e acoplamento.

A página **Refactoring guru** apresenta um conteúdo interativo e bastante completo de todos os padrões de projeto GoF com exemplos de código em diversas linguagens de programação. Consultado na internet em: 18 out. 2021.

Referências

GAMMA, E. *et al*. **Design patterns:** elements of reusable object-oriented software. 1. ed. Boston: Addison-Wesley, 1994.

LARMAN, C. **Applying UML and patterns:** an introduction to object-oriented analysis and design and iterative development. 3. ed. Upper Saddle River: Prentice Hall, 2004.

MARTIN, R. C. **Clean architecture:** a craftsman´s guide to software structure and design. 1. ed. Upper Saddle River: Prentice Hall, 2017.