Introdução a padrões de projeto - design patterns

Padrões de projeto de software são fundamentais para desenvolvedores, pois permitem a criação de programas bem estruturados a partir de soluções consagradas para problemas comuns na construção de sistemas. Vamos discutir os padrões GRASP, SOLID e GoF para que possamos entender o conceito de padrões de projeto e reconhecer oportunidades para aplicá-los no desenvolvimento de nossos sistemas.

Prof. Alexandre Luis Correa / Colaboração: Prof. Denis Cople

Objetivos

- Descrever os conceitos gerais de padrões de projeto, seus elementos e suas características.
- Reconhecer o propósito dos padrões GRASP e as situações nas quais eles podem ser aplicados.
- Descrever as características dos princípios SOLID.
- Reconhecer o propósito dos principais padrões GoF e as situações nas quais eles podem ser aplicados.

Introdução

Olá! Antes de começarmos, assista ao vídeo de introdução ao nosso objeto de estudo: padrões de projeto – Design patterns.

Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Padrões de projeto

Melhoram o código e estabelecem um vocabulário comum entre diferentes equipes de desenvolvimento. Ao oferecerem soluções para problemas comuns, eles elevaram o nível de padronização do código e sua organização, facilitando a manutenção e possibilitando o surgimento de diversos frameworks.

Essa característica reduziu o esforço necessário para o desenvolvimento de sistemas e teve impacto direto no projeto de bibliotecas para diversas plataformas. Compreender os padrões aprimora a forma de programar, facilita o uso das ferramentas de desenvolvimento atuais e melhora a comunicação com outros desenvolvedores.

Acompanhe, neste vídeo, a motivação e a origem dos padrões de projeto como soluções gerais para problemas recorrentes durante o desenvolvimento de um software.

> Conteúdo interativo
>
> Acesse a versão digital para assistir ao vídeo.

Imagine que você foi alocado em um novo trabalho e está responsável por evoluir um sistema desenvolvido por outras pessoas ao longo de anos. Você não se sentiria mais confortável se o software tivesse sido projetado com estruturas de solução que você já conhece?

Quando aprendemos a programar em uma linguagem orientada para objetos, por exemplo, somos capazes de definir classes, atributos, operações, interfaces e subclasses com ela. Esse conhecimento nos permite responder perguntas operacionais como: “Como eu escrevo a definição de uma classe nesta linguagem de programação?”.

Entretanto, quando temos que estruturar um sistema, a pergunta se tornar bem mais complexa: “Quais classes, interfaces, relacionamentos, atributos e operações devemos definir para resolver esse problema de forma adequada?”.

Ao desenvolvermos um sistema, deparamo-nos, inúmeras vezes, com essa pergunta. É comum um desenvolvedor iniciante se sentir perdido diante da diversidade de respostas possíveis, ainda mais quando algumas podem facilitar enquanto outras podem dificultar bastante as futuras evoluções do sistema.

Que tipo de conhecimento um desenvolvedor experiente utiliza para estruturar um sistema?

Tabuleiro de xadrez.

Fazendo uma analogia com o jogo de xadrez, um desenvolvedor inexperiente compartilha semelhanças com um jogador iniciante, que sabe colocar as peças no tabuleiro, conhece os movimentos permitidos para cada peça e domina alguns princípios elementares do jogo, como a importância relativa das peças, e o centro do tabuleiro.

Um desenvolvedor inexperiente conhece as construções básicas de programação (ex.: sequência, decisão, iteração, classes, objetos, atributos e operações) e alguns princípios elementares de estruturação de um sistema, como dividi-lo em módulos e evitar construir outros longos e complexos.

O que diferencia os grandes enxadristas dos iniciantes? Vejamos.

1. Os jogadores experiente estudaram ou vivenciaram inúmeras situações de jogo e acumularam experiência sobre as consequências que determinado lance pode gerar no desenrolar da partida.

2. Quanto maior for o arsenal de situações conhecidas por um jogador, maiores serão as chances de ele fazer uma boa jogada em pouco tempo.

3. Os jogadores de xadrez também têm prazo a cumprir. Existem campeonatos que definem o tempo máximo de duas horas para que cada enxadrista realize seus primeiros 40 lances.

Contudo, o jogador iniciante não aprende apenas jogando e vivenciando situações, pois ele também pode estudar inúmeras situações catalogadas e feitas por outros jogadores. Existem catálogos de situações e estratégias para o início, o meio e a finalização de um jogo de xadrez.

As estratégias para o início de jogo são conhecidas como **aberturas**. Cada uma tem um nome e corresponde a um conjunto de jogadas que gera consequências positivas e, eventualmente, negativas para cada lado. Uma dessas aberturas é o nome de uma minissérie original muito popular: ***O gambito da rainha***. A produção, inclusive, possibilita conferir como é possível aprender e se desenvolver a partir do que outros jogadores já fizeram. A protagonista Beth Harmon tinha o hábito de ler materiais sobre partidas e movimentos e, assim, desenvolvia suas habilidades de jogadora. Caso estude bastante as jogadas, quem sabe você consegue ver uma partida no teto, exatamente como a personagem fazia na série.

O gambito da rainha

Uma jovem prodígio do xadrez luta contra vícios enquanto enfrenta os maiores enxadristas do mundo.

Você sabe o que diferencia os desenvolvedores experientes dos iniciantes?

Os experientes acumularam conhecimento sobre as consequências de resolver um problema aplicando determinada estrutura de solução, isto é, dividindo a solução em um conjunto específico de módulos e estabelecendo uma estrutura particular de interação entre eles. A experiência acumulada permite que eles reusem, no presente, soluções que funcionaram bem no passado em problemas similares, produzindo sistemas flexíveis, elegantes e em menor tempo.

Muitos problemas em estruturação de software são recorrentes. Então, que tal registrá-los com as respectivas soluções para que, de forma análoga aos livros de abertura do xadrez, possamos compartilhar esse conhecimento com desenvolvedores que estejam encarando pela primeira vez um problema análogo?

Foi o que pensaram quatro engenheiros de software, lá em 1994:

- Erich Gamma
- Richard Helm
- Ralph Johnson
- John Vlissides

Eles publicaram o livro *Design patterns: elements of reusable object-oriented software*, descrevendo 23 padrões que utilizaram no desenvolvimento de diversos sistemas ao longo de suas carreiras. Os autores ficaram conhecidos como a “gangue dos quatro” (tradução de *gang of four*), e os padrões publicados são

conhecidos como os “**padrões GoF**”. Desde então, diversas publicações surgiram relatando novos padrões, críticas aos existentes, padrões específicos para determinadas linguagens e paradigmas de programação, além de padrões arquiteturais e até mesmo antipadrões – ou seja, construções que tipicamente trazem consequências negativas para a estrutura de um sistema.

Atividade 1

Questão

Assinale a afirmativa verdadeira sobre padrões de projeto.

A

Corresponde a uma biblioteca de códigos pronta para ser utilizada para resolver um problema recorrente em projetos de software.

B

Define uma forma padronizada de documentar a estrutura de projetos de software.

C

Descreve a estrutura fundamental da solução para um problema recorrente em projetos de software, proporcionando reuso de conhecimento em soluções de projeto.

D

Descreve ideias de possíveis soluções para um problema recorrente em projetos de software, que podem (ou não) ter sido utilizadas em outros projetos.

E

Podem ser utilizados somente com linguagens orientadas a objetos.

A alternativa C está correta.

- A alternativa A é falsa, pois um padrão não é uma biblioteca de código.
- A alternativa B é falsa, pois um padrão de projeto não está relacionado à forma de documentação de um software.
- A alternativa C é verdadeira, pois padrões de projeto correspondem a reuso de conhecimento de projetistas que já passaram pelo mesmo problema em outros projetos e catalogaram a estrutura de uma solução bem-sucedida.
- A alternativa D é falsa, pois um padrão precisa ter sido utilizado com sucesso em pelo menos três contextos diferentes de aplicação.
- A alternativa E é falsa, pois existem padrões de projeto para diferentes paradigmas de programação.

Conhecendo o padrão de projeto

Vamos iniciar definindo o que é o padrão de projeto. Trata-se de uma solução nomeada para determinado problema, com uma estrutura básica e observações acerca de sua utilização. Embora estruturado em termos de UML, o padrão de projeto não se restringe ao paradigma orientado a objetos, podendo ser aplicado em praticamente todas as plataformas. O uso de padrões permite adotar uma base de conhecimento amplamente testada e aceita no mercado de desenvolvimento, porém, existem diversas famílias de padrões e, muitas vezes, o que um padrão indica, outro desqualifica, o que exige do desenvolvedor a capacidade de analisar cada cenário, escolhendo o padrão mais adequado em cada situação.

Neste vídeo, você compreenderá o que é o padrão de projeto e as respectivas vantagens e desvantagens em sua aplicação.

Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

O que é um padrão de projeto?

Também conhecido como *design pattern*, ele descreve um problema recorrente e a estrutura fundamental da sua solução definida por módulos e comunicações entre eles.

> Em uma estrutura de solução orientada a objetos, a solução é descrita por classes, interfaces e mecanismos de colaboração entre objetos.

O conceito de padrões de projeto é mais abrangente, uma vez que existem padrões voltados para outros paradigmas de programação, como **programação funcional** e **reativa**.

Um padrão de projeto apresenta quatro elementos fundamentais:

Nome

Possibilita a comunicação entre os desenvolvedores em um nível mais abstrato. Em uma discussão de projeto, em vez de descrevermos todos os elementos da solução para um problema, bem como a forma de comunicação entre eles e as consequências do seu uso, basta mencionarmos o nome do padrão apropriado.

Um desenvolvedor experiente pode recomendar algo como: “use o Observer para resolver esse problema!”, e isso será suficiente, pois conhecendo esse padrão, você já saberá como estruturar a solução. Portanto, os nomes dos padrões foram um vocabulário compartilhado pelos desenvolvedores para agilizar a discussão de soluções de projeto.

Problema

Podem ser específicos, por exemplo: “como podemos representar diferentes algoritmos com o mesmo propósito na forma de classes?” ou “como podemos instanciar uma classe específica, entre várias similares, sem criar dependência rígida da implementação escolhida?”. Um padrão deve descrever as situações nas quais a sua utilização é apropriada, explicando o problema e o contexto.

Esse elemento também pode descrever e discutir soluções inadequadas ou pouco flexíveis para o contexto apresentado. Se você estiver pensando em uma solução similar às descritas, o seu problema passará a ser: “ok, estou vendo que a estrutura em que estava pensando aplicar não é adequada. Como devo estruturar a solução?”.

Solução

É uma descrição abstrata que pode ser replicada em diferentes situações e sistemas. O desenvolvedor deve traduzir a descrição abstrata em uma implementação específica no problema particular que estiver resolvendo. Note que a solução é uma ideia que precisa ser construída e implementada pelo desenvolvedor. Um catálogo de padrões de projeto é bem diferente de uma biblioteca de código, pois um padrão de projeto não tem o propósito de promover reutilização de código, mas, sim, de conhecimento.

O padrão deve descrever os elementos recomendados para a implementação, bem como as suas responsabilidades, os seus relacionamentos e as suas colaborações.

Consequências

Um padrão precisa discutir os custos e benefícios da solução proposta. Uma solução pode dar maior flexibilidade ao projeto, ao mesmo tempo em que pode trazer maior complexidade ou até mesmo resultar em problemas de performance. Antes de utilizarmos um padrão, devemos sempre avaliar os custos e benefícios em relação ao problema particular que estamos querendo resolver.

Vantagens e desvantagens do uso de padrões de projeto

Por que você deve conhecer padrões de projeto? Podemos enumerar alguns motivos:

1

Produtividade

Possibilitam maior produtividade ao desenvolvimento, uma vez que não desperdiçamos tempo pensando, fazendo e refinando soluções até encontrarmos a mais adequada. Padrões nos fornecem um atalho para a boa solução.

2

Reutilização

Reforçam práticas de reutilização de código com estruturas que acomodam mudanças por meio do uso de mecanismos, como delegação, composição e outras técnicas que não sejam baseadas em estruturas de herança.

3

Troca

Fornecem uma linguagem comum para os desenvolvedores, agilizando a troca de ideias e experiências.

4

Facilitação

Facilitam o desenvolvimento, pois permitem a reutilização de soluções bem-sucedidas em problemas similares. Um padrão não pode ser simplesmente uma ideia, pois só pode ser considerado como tal se passar pela regra dos três, isto é, ele deve ter sido utilizado em, pelo menos, três diferentes situações ou aplicações.

E quais são as desvantagens ou os pontos de atenção na utilização de padrões de projeto?

- Desde que o conceito foi apresentado, na década de 1990, uma grande quantidade de padrões surgiu. Portanto, a curva de aprendizado pode ser grande.
- Decidir se um padrão pode ser empregado em um problema específico nem sempre é uma tarefa fácil, exigindo alguma experiência.
- É necessário, às vezes, adaptar ou mudar alguma característica da solução proposta para um padrão para tornar sua utilização adequada a um problema especifico, o que também requer experiência.
- É comum um iniciante achar que os padrões devem estar por toda a implementação e fazer uso inadequado deles.
- Não existe consenso sobre a qualidade de todos os padrões. Por exemplo, singleton é um padrão GoF que gera grande controvérsia, sendo considerado um antipadrão por muitos.

Atividade 2

Questão

Assinale a afirmativa falsa sobre a descrição dos padrões.

A

Devem ter um nome, pois formam um vocabulário que agiliza o debate sobre uma solução entre os desenvolvedores de software.

B

Precisam descrever o problema ou a situação na qual a solução proposta é aplicável.

C

Devem descrever a estrutura de solução proposta, identificando os participantes, as suas responsabilidades e os mecanismos de colaboração entre eles.

D

Como todo padrão corresponde a uma solução bem-sucedida para o problema, ele deve apresentar as vantagens obtidas com o seu uso.

E

Um padrão não deve descrever as desvantagens do seu uso, pois isso caracterizaria que a solução não é adequada para o problema.

A alternativa E está correta.

Um padrão de projeto pode trazer consequências positivas, mas, eventualmente, negativas para o projeto. Portanto, antes de usar um padrão, o desenvolvedor deve verificar se os problemas são relevantes no contexto específico da aplicação e se podem ser contornados de alguma forma, avaliando, os custos e os benefícios da sua utilização.

Principais padrões em linhas gerais

Embora existam diversas famílias de padrões de projeto, o conjunto de padrões GoF assume um papel extremamente importante na área de desenvolvimento. Conhecê-los é essencial para criar sistemas consistentes e adequados ao uso de diversos frameworks. Com uma divisão entre padrões de criação, estruturais e comportamentais, eles conseguem absorver a complexidade de diversas tarefas de desenvolvimento, como a combinação e o reuso das etapas de construção de objetos complexos por meio de builder, uso de composite para a definição de estruturas hierárquicas ou implementação de comportamento assíncrono via observer.

Neste vídeo você compreenderá, de forma resumida, as três grandes categorias dos padrões de projeto GoF: criação, estrutura e comportamento.

Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Os 23 padrões de projeto GoF são classificados em três grandes categorias. Confira a seguir!

Criação

Têm como objetivo tornar a implementação independente da forma com que os objetos são criados, compostos ou representados.

Essa categoria possui **cinco** padrões GoF: Abstract Factory, Builder, Factory Method, Prototype e Singleton.

Estrutura

Compreende formas de combinar classes e objetos para formar estruturas maiores.

Essa categoria possui **sete** padrões GoF: Adapter, Bridge, Composite, Decorator, Facade, Flyweight e Proxy.

Comportamento

Tratam de algoritmos, da distribuição de responsabilidades pelos objetos e da comunicação entre eles.

Essa categoria possui **onze** padrões GoF: Chain of Responsibility, Command, Interpreter, Iterator, Mediator, Memento, Observer, State, Strategy, Template Method e Visitor.

Existem inúmeros padrões publicados para diferentes problemas: arquitetura e projeto detalhado de software, padrões específicos para uma linguagem de programação, testes, gerência de projeto e de configuração, entre outros. Você não precisa estudar todos os padrões disponíveis, mas é importante saber que eles existem, onde estão descritos e os problemas que eles resolvem. Assim, no dia em que tiver que resolver um problema similar, você poderá aproveitar uma solução utilizada com sucesso por outras pessoas.

Saiba mais

Experimente fazer uma busca na internet com termos gerais, tais como pattern books, patterns catalog, e mais específicos, como java design patterns, cloud architecture patterns e node js patterns.

Atividade 3

Questão

Conhecer os padrões GoF é essencial para qualquer desenvolvedor, pois eles são ampla e geralmente utilizados no mercado de desenvolvimento e particularmente na construção de frameworks. Esses padrões são divididos em três grandes grupos, relacionados a criação de componentes, definições estruturais e modelagem de comportamentos. Entre as opções a seguir, qual delas apresenta apenas padrões comportamentais?

A

Builder, factory method e bridge

B

Command, decorator e facade

C

Adapter, bridge e composite

D

Factory method, decorator e interpreter

E

Command, iterator e visitor

A alternativa E está correta.

Os padrões criacionais estão relacionados à geração de objetos, como builder e factory method. Os padrões estruturais definem estruturas complexas, como adapter, bridge, composite, decorator e facade. Quanto aos padrões comportamentais, eles organizam a modelagem de processos, como interpreter, command, iterator e visitor. Logo, entre as opções, o único grupo que é composto apenas por padrões comportamentais é command, iterator e visitor.

Especialista

GRASP é o acrônimo para o termo em inglês *general responsibility assignment software patterns*. Ele foi definido por Craig Larman no livro *Applying UML and patterns*, que define padrões gerais para a atribuição de responsabilidades em software. Veja a diferenciação entre os padrões GoF e GRASP:

Padrões GoF	Padrões GRASP
Abordam problemas específicos encontrados no projeto de software.	São princípios gerais de design para software orientado a objetos.

Neste conteúdo, você vai aprender seis padrões GRASP:

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

Confira, neste vídeo, o padrão GRASP especialista, que aborda o princípio geral de atribuição de responsabilidades aos objetos de um sistema.

Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Problema

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

Solução

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

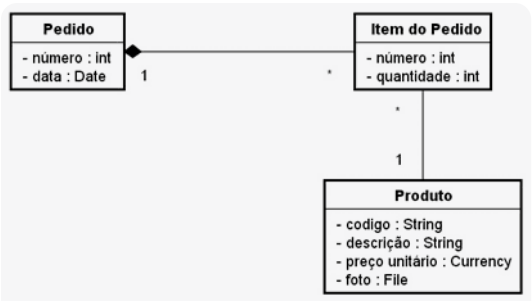
Comentário

Atribuir a responsabilidade ao especialista é uma especulação que utilizamos em 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 a atividade (um especialista), ou seja, um eletricista.

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

A imagem 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 as informações desejadas?



Padrão especialista – classes de domínio.

O valor total do pedido pode ser definido como a soma total 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 o pedido? Ele próprio, não é mesmo? Então, vamos definir a operação obterValorTotal na classe pedido.

Vamos responder, primeiro, a algumas perguntas:

- Onde ficaria o cálculo do preço de um item do pedido?
- Quais informações são necessárias para esse cálculo?
- Quem conhece as informações sobre a quantidade e o preço do produto?

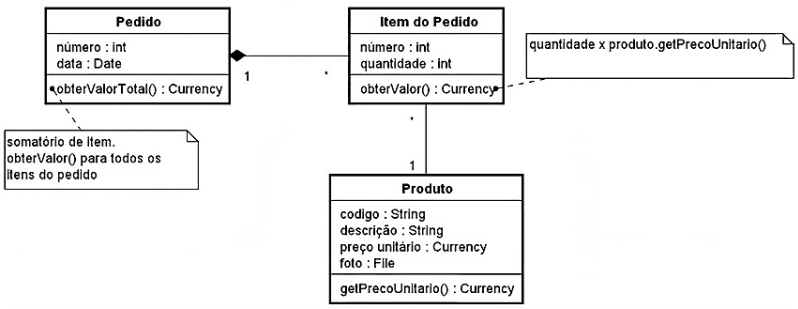
Como a classe **item do pedido** conhece a quantidade e o produto associado, vamos definir nela a operação **obterValor**.

Para o preço do produto, basta o objeto referente ao item do pedido pedir essa informação para o produto, por meio da operação de acesso **getPrecoUnitario**.

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

Conteúdo interativo

Acesse a versão digital para ver mais detalhes da imagem abaixo.

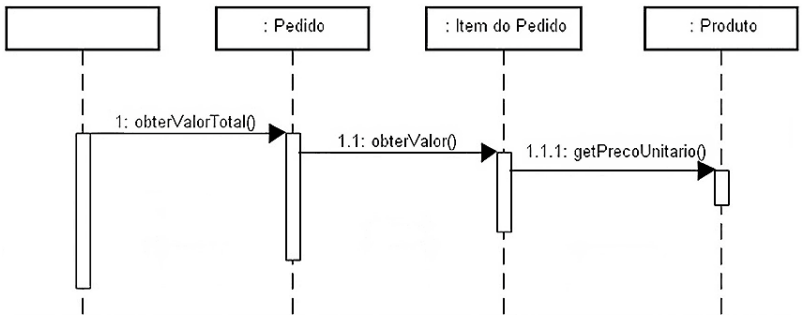


Padrão especialista – alocação de responsabilidades.

Na próxima imagem, o diagrama de sequência ilustra a colaboração definida para implementar a obtenção do valor total de um pedido. Veja!

Conteúdo interativo

Acesse a versão digital para ver mais detalhes da imagem abaixo.



Padrão especialista – colaboração (opção 1).

Consequências

Quando o padrão especialista não é seguido, é comum encontrarmos a solução deficiente (antipadrão) conhecida como *God Class*. Ela consiste em definir apenas operações de acesso (chamadas de *getters* e *setters*) nas classes de domínio. É usual também concentrarmos a lógica de determinada funcionalidade do sistema em uma única classe, geralmente definida na forma de uma classe de controle ou 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 o uso 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: no negócio e na tecnologia de armazenamento, o que é absolutamente inadequado.

Atividade 1

Questão

Quando trabalhamos com o padrão especialista, buscamos implementar as funções do sistema nas classes capazes de obter as informações necessárias, ou seja, que conhecem o domínio do problema. Segundo o padrão especialista, considerando um sistema acadêmico e uma classe prova, qual das classes a seguir deve fornecer acesso às provas realizadas?

A

Professor

B

Turma

C

Aluno

D

Período

E

Curso

A alternativa C está correta.

De forma análoga a um sistema de pedidos: um pedido reconhece seus itens. e os itens reconhecem os produtos relacionados. Assim, no sistema acadêmico, esse tipo de hierarquia também pode ser adotado. A classe curso fornece os objetos de período, que devem dar acesso aos objetos de turma, os quais estarão relacionados a um conjunto de objetos aluno e um objeto professor. Como o aluno realiza e visualiza as provas, ele será considerado o especialista, e qualquer outro objeto deverá saber quem é o aluno antes de acessar suas provas.

Criador e baixo acoplamento

Os padrões GRASP se concentram na definição das responsabilidades de cada componente de software.

É fundamental identificar o objeto mais apropriado para realizar determinadas tarefas. Segundo o padrão criador, você, pessoa desenvolvedora, deve definir qual objeto será responsável por instanciar outros. Isso se dá no caso de uma classe desenho com um método para criar objetos da classe polígono, o que gera uma figura completa com os polígonos criados ao final.

Outro padrão GRASP é o baixo acoplamento. Nele, o desenvolvedor visa assegurar o mínimo de acoplamento entre os objetos e os componentes, facilitando tanto a manutenção como a evolução do sistema.

Acompanhe, neste vídeo, como o padrão GRASP criador trata da instanciação de objetos de um sistema e como o padrão GRASP baixo acoplamento aborda o grau de dependência de um módulo em relação a outros módulos.

Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Padrão GRASP 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.

A pergunta que esse 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.

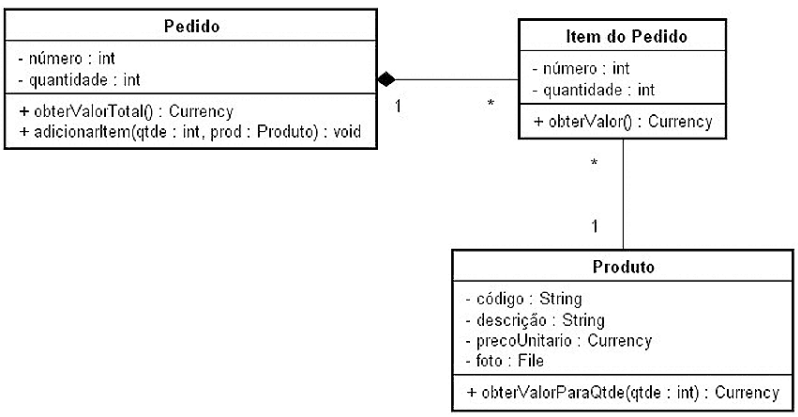
No exemplo do site de vendas de produtos pela internet, considerando o modelo de classes da imagem **especialista – classes de domínio**, qual classe deveria ser responsável pela criação das instâncias de item do pedido?

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 de suas partes (criação e destruição).

A imagem 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.

Conteúdo interativo

Acesse a versão digital para ver mais detalhes da imagem abaixo.



Padrão creator – classes de domínio.

Consequências

Veja, a seguir, a indicação e a contraindicação do padrão Criador.

Indicação

É especialmente indicado para criar instâncias que formam parte de um agregado por promover uma solução de menor acoplamento.

Contraindicação

Não é apropriado em algumas situações especiais, como a criação condicional de uma instância dentro de uma família de classes similares.

Padrão GRASP baixo acoplamento

Problema

Vamos começar definindo o que é acoplamento. Ele corresponde ao grau de dependência de um módulo em relação a outros 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 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. Veja:

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

2. Se a classe A depende da interface I, A depende de uma abstração e pode funcionar com diferentes implementações de I sem depender de uma específica.

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) ou por englobar diversas questões de negócio (ex.: diferentes regras de negócio e diversos fornecedores de uma mesma solução de pagamento, entre outros).

A pergunta que esse padrão tenta responder é:

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

Solução

Distribuir as responsabilidades de modo que gere baixo acoplamento entre os módulos.

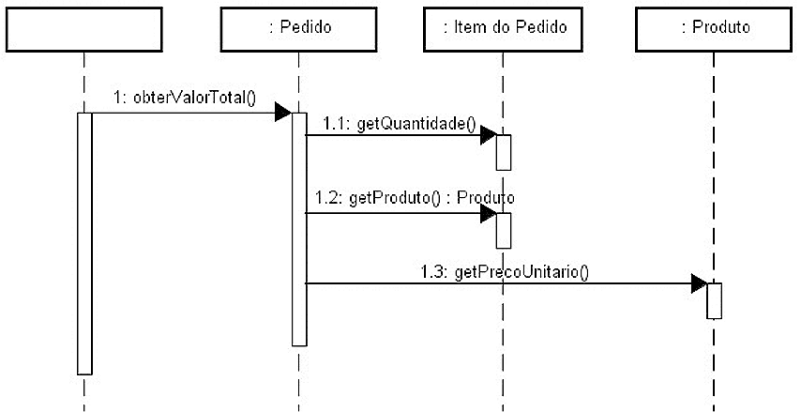
Para você entender o conceito de baixo acoplamento, vamos apresentar um contraexemplo, isto é, uma situação em que um acoplamento maior do que o desejado foi criado.

A imagem a seguir 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, o produto de cada item e 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. Confira!

Conteúdo interativo

Acesse a versão digital para ver mais detalhes da imagem abaixo.



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 acoplamento menor ou 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 e integração entre sistemas) é uma política geral de acoplamento, que deve ser seguida, pois é recomendada por diversas proposições de arquitetura.

Atividade 2

Questão

Assinale a afirmativa que apresenta a recomendação expressa pelo padrão GRASP baixo acoplamento.

A

Alocar as responsabilidades nos módulos que contenham as informações para realizá-las.

B

Alocar as responsabilidades nos módulos, a fim de tornar as dependências entre os módulos gerenciáveis, evitando criar dependências inadequadas.

C

Alocar as responsabilidades nos módulos de forma que minimize a presença de estruturas condicionais complexas nos módulos.

D

Alocar as responsabilidades nos módulos de modo que cada módulo reúna elementos que cumpram um único propósito.

E

Alocar as responsabilidades nos módulos para evitar a instanciação indiscriminada e sem critérios bem definidos de objetos no sistema.

A alternativa B está correta.

- A alternativa A corresponde à intenção do padrão Especialista.
- A alternativa B corresponde à intenção do padrão Baixo acoplamento, que define relações de dependência entre classes, distribuindo responsabilidades de modo que diminua o acoplamento entre módulos do sistema, minimizando o impacto de mudanças e facilitando o reuso.
- A alternativa C corresponde à intenção do padrão Polimorfismo.
- A alternativa D corresponde à intenção do padrão Alta coesão.
- A alternativa E corresponde à intenção do padrão Criador.

Alta coesão

Vamos iniciar entendendo o que é coesão. Trata-se da propriedade de determinado componente de software que define o nível de coerência e eficácia no agrupamento e na execução de tarefas. O padrão de alta coesão do GRASP tem foco nessa propriedade, pois um objeto coeso apresenta maior independência de outros, além de apresentar uma semântica mais clara ao agrupar funcionalidades semelhantes.

Como pessoa desenvolvedora, é importante seguir o padrão Alta coesão, criando, por exemplo, pacotes destinados a áreas específicas e com nomes representativos, nos quais as classes associadas estão restritas às funcionalidades exigidas pelos pacotes.

Neste vídeo, você compreenderá que o padrão GRASP alta coesão se refere à forma de avaliar se as responsabilidades de um módulo estão, de fato, relacionadas em um sistema e se têm o mesmo propósito. Confira!

Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Problema

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

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

A pergunta que esse padrão tenta responder é:

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

Solução

Consiste em definir módulos de alta coesão. Mas como se mede a coesão?

Ela está ligada ao critério utilizado para reunir um conjunto de elementos em um mesmo módulo. Veja:

1

Coesão de um método de uma classe

Um método reúne um conjunto de instruções.

2 Coesão de uma classe
Uma classe agrupa um conjunto de atributos e operações.

3 Coesão de um pacote
Um pacote junta um conjunto de classes e interfaces.

4 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 pela saída de um programa. Nesse pacote, encontramos classes com responsabilidades bem específicas, como:

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

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

Atividade 3

Questão

Assinale a afirmativa que expressa a intenção do padrão GRASP alta coesão.

A

Alocar as responsabilidades nos módulos que contenham as informações para realizá-las.

B

Alocar as responsabilidades nos módulos a fim de tornar as dependências entre eles gerenciáveis, evitando criar dependências inadequadas.

C

Alocar as responsabilidades nos módulos de forma que minimize a presença de estruturas condicionais complexas nos módulos.

D

Alocar as responsabilidades nos módulos de forma que cada módulo reúna elementos que cumpram um único propósito.

E

Alocar as responsabilidades nos módulos a fim de evitar a instanciação indiscriminada e sem critérios bem definidos de objetos no sistema.

A alternativa D está correta.

- A alternativa A corresponde à intenção do padrão Especialista.
- A alternativa B corresponde à intenção do padrão Baixo acoplamento.
- A alternativa C corresponde à intenção do padrão Polimorfismo.

- A alternativa D corresponde à intenção do padrão Alta coesão, que atende à necessidade de manter os elementos dos módulos do sistema agrupados, seja uma classe, um pacote ou um subsistema, a fim de tornar a sua complexidade gerenciável.
- A alternativa E corresponde à intenção do padrão Criador.

Conhecendo o padrão controlador

Em muitas situações, é essencial ter um líder coordenando a equipe, como o técnico em um time esportivo ou o maestro de uma orquestra. Da mesma forma, em um sistema, o padrão controlador visa definir uma classe que inicia operações específicas quando ocorrem determinados eventos.

Um exemplo prático disso é a arquitetura MVC, que é amplamente utilizada em sistemas de cadastro. O controlador recebe as solicitações da interface do usuário e executa as operações necessárias na camada de persistência. A arquitetura MVC é implementada em vários frameworks, como o Spring, que faz uso da anotação @Controller.

Veja, neste vídeo, que o padrão GRASP controlador é responsável por atribuir a responsabilidade de receber um evento do sistema e coordenar a produção de sua resposta a uma classe.

Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Problema

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

Exemplo

Quando o cliente solicita o fechamento de um pedido na loja on-line, isso precisa ser capturado e processado pelo sistema.

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

Solução

Deve-se 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 corresponde ao sistema ou a um subsistema específico, e essa solução é conhecida como **controlador fachada**. Ela é utilizada em sistemas com poucos eventos.

Opção 2

Uma classe corresponde ao caso de uso em que o evento ocorre. Normalmente, o nome dessa classe é formado pelo nome do caso de uso com o sufixo processador, controlador, sessao ou algo similar.

A classe controlador deve reunir o tratamento de todos os eventos que o sistema recebe no contexto do caso de uso. Essa solução evita que as responsabilidades de tratamento de eventos de diferentes funcionalidades se concentrem em um único controlador fachada, evitando a criação de um módulo com baixa coesão.

Note que a classe estudada não cumpre as responsabilidades de interface com o usuário. Em um sistema de *home banking*, por exemplo, o usuário informa todos os dados de uma transferência em um componente de interface com o usuário e, ao clicar no botão transferir, esse componente delega a requisição para o controlador realizar o processamento lógico da transferência. Assim, o mesmo controlador pode atender às solicitações realizadas por diferentes interfaces com o usuário (web, dispositivo móvel e totem 24 horas, por exemplo).

Consequências

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

Maestro e orquestra.

Em uma orquestra, o maestro comanda o momento em que cada músico deve entrar em ação, mas ele mesmo não toca nenhum instrumento. Da mesma forma, o módulo Controlador é o grande orquestrador de um conjunto de objetos, em que cada um tem sua responsabilidade específica na produção da resposta ao evento.

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

Atividade 4

Questão

O padrão controlador é essencial para diversos frameworks que trabalham na arquitetura MVC, como Spring, Structs e JSF. Considerando a definição do GRASP de controlador, qual será o papel do controlador nesses frameworks?

A

Implementar a interface de usuário.

B

Definir o processamento da resposta para requisições específicas.

C

Efetuar a persistência no banco de dados.

D

Estabelecer um método organizado para a criação de objetos.

E

Fornecer elementos para o comportamento assíncrono.

A alternativa B está correta.

Um controlador deve atuar como gestor de processos, recebendo requisições e orquestrando os demais objetos na realização das tarefas necessárias. Ele não é utilizado para definir a interface de usuário, mas trata das requisições recebidas a partir dela, e a persistência deve ser efetuada pelas entidades do domínio, segundo o padrão especialista. O padrão criador define quem será responsável pela geração de objetos, e, para o comportamento assíncrono, é fundamental que haja baixo acoplamento.

Padrão polimorfismo e suas características

Um elemento que oferece grande poder ao paradigma orientado a objetos é o polimorfismo, pois permite que um método herdado seja modificado, adaptando a funcionalidade original às necessidades do descendente. O padrão polimorfismo se baseia nesse princípio e visa substituir a tradicional execução condicional da programação estruturada por métodos polimórficos definidos em diversas classes descendentes.

Você, como pessoa desenvolvedora, deverá ser capaz de identificar processos genéricos que possam ser especializados para contextos distintos e implementá-los por meio de famílias de classes. Nesse contexto, o polimorfismo permite que os métodos sejam modificados nas classes descendentes, proporcionando utilização transparente pelo sistema.

Confira, neste vídeo, que o padrão GRASP polimorfismo estabelece como escrever uma solução genérica para alternativas baseadas no tipo de um elemento, criando módulos plugáveis.

Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Problema

Como evitar construções condicionais complexas no código?

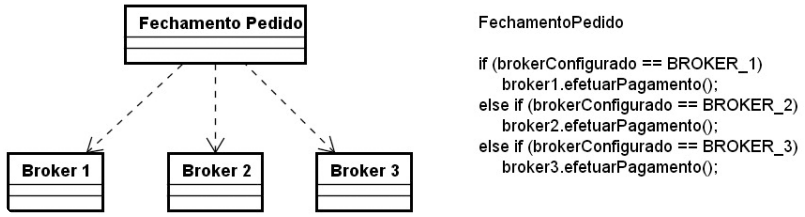
Suponha que você esteja implementando a parte de pagamentos de uma loja virtual via cartão. Para realizar um pagamento interagindo diretamente com a administradora de cartão, temos que passar por um processo longo e complexo de homologação com ela. Imagine realizar esse processo com cada administradora!

Para simplificar o problema, existem diferentes **brokers** (software intermediário que permite que programadores façam chamadas a programas de um computador para outro) de pagamento, que já estão homologados com as diversas administradoras e fornecem uma **API** (do inglês *application programming interface* ou interface de programação de aplicação) para integração com a nossa aplicação. Cada broker tem uma política de preços e volume de transações. Eventualmente, podem surgir novos com políticas mais atrativas no mercado.

Imagine que fornecemos uma solução de software de loja virtual para diferentes estabelecimentos, 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* (imagem a seguir).

Pense como ficaria esse código se houvesse 20 brokers diferentes. Veja:



Solução sem polimorfismo.

O problema que o padrão polimorfismo resolve é o seguinte:

Como escrever uma solução genérica para alternativas que se baseiam 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

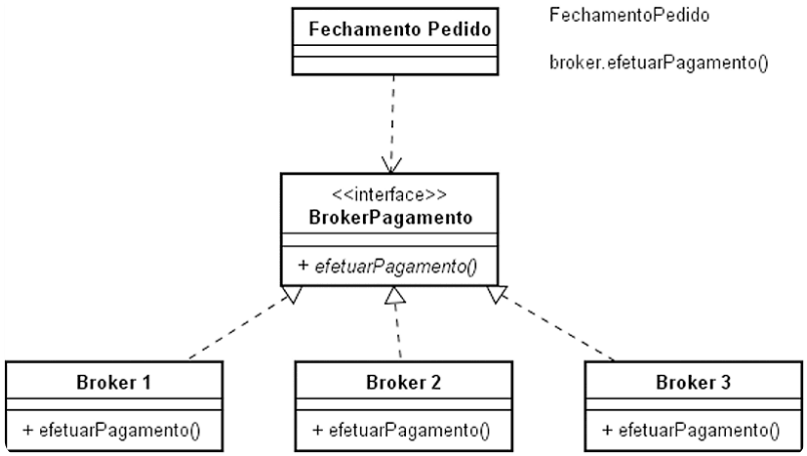
Percebeu como a solução baseada em estruturas condicionais do tipo *if-then-else* ou *switch-case*, além de serem mais complexas, criam um acoplamento do módulo chamador com cada implementação específica? Note como a classe Fechamento pedido depende diretamente de todas as implementações de broker de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

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

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

Conteúdo interativo

Acesse a versão digital para ver mais detalhes da imagem abaixo.



Solução com polimorfismo.

Consequências

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

Dica

O princípio geral do polimorfismo é utilizado para definir diversos padrões GoF, como: Adapter, Command, Composite, Proxy, State e Strategy.

Lembre-se: 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.

Atividade 5

Questão

O padrão polimorfismo utiliza a característica da orientação a objetos de mesmo nome, permitindo que uma funcionalidade herdada possa ser modificada na classe descendente para adaptação a um novo contexto de utilização. Qual dos problemas a seguir tiraria proveito do polimorfismo?

A

A implementação de um carrinho de compras com pagamento via Pix ou cartão.

B

A definição de uma classe que ofereça apenas uma instância global para o sistema.

C

O estabelecimento de comunicação remota com um web service REST.

D

A pesquisa do conjunto de automóveis em um estacionamento.

E

A definição de uma classe que receba todas as requisições HTTP e direcione para os objetos responsáveis pelo tratamento de cada uma.

A alternativa A está correta.

Ao criar um carrinho de compras, o pagamento via Pix ou cartão pode ser definido de maneira polimórfica, pois o próprio nome indica que são múltiplas as formas de executar a mesma ação (que, no caso, é o pagamento). A definição de uma instância global é o padrão singleton, do GoF, e não tem relação com polimorfismo. Para a comunicação remota é essencial que haja baixo acoplamento.

O estacionamento detém as informações acerca dos automóveis presentes, logo, é o especialista da informação. Da mesma forma, usar um objeto para controlar todas as requisições HTTP se enquadra no padrão controlador.

Conhecendo o princípio da responsabilidade única (SRP)

Segundo o SRP, cada módulo deve estar direcionado para um ator específico, e apenas um garante, por exemplo, que, em um sistema de vendas, as funções voltadas para o vendedor e o gestor não estejam presentes no mesmo módulo. A vantagem de adotar o SRP é a possibilidade de dividir o sistema em componentes menores voltados para atores específicos, o que viabiliza a distribuição deles em diferentes locais e plataformas, como o ponto de venda na loja física, para o vendedor, e a busca de produtos na web, para o cliente.

Assista, neste vídeo, ao SOLID, acrônimo para os cinco princípios de projeto orientado a objetos, e à aplicação do princípio da responsabilidade única (SRP).

Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Princípios SOLID

O termo SOLID é frequentemente visto como um requisitado em perfis de vagas para desenvolvimento de software.

Trata-se de um acrônimo para cinco princípios de projeto orientado a objetos. Definidos por Robert C. Martin, eles devem ser seguidos para nos ajudar a produzir software com uma estrutura sólida, isto é, uma estrutura que permita sua manutenção com menor esforço.

Cada letra do acrônimo corresponde a um dos princípios: Confira a seguir!

S

Single-responsibility principle (SRP).

O

Open-closed principle (OCP).

L

Liskov substitution principle (LSP).

I

Interface segregation principle (ISP).

D

Dependency inversion principle (DIP).

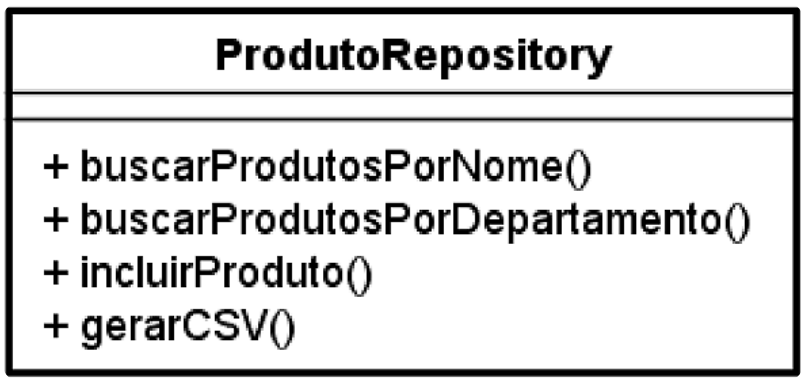
SRP

Sigla do termo em inglês *Single responsibility principle*, estabelece que o módulo deve ser responsável por um – e apenas isso – ator, representando o papel desempenhado por alguém envolvido com o software. Nesse contexto, um módulo corresponde a um arquivo-fonte.

Exemplo

Em Java, um módulo corresponde a uma classe.

Suponha que uma loja virtual tenha requisitos de busca de produtos por nome e departamento, além de cadastro de um novo produto e exportação de dados dos produtos em formato CSV. Se pensarmos conforme o padrão Especialista, por exemplo, todas essas funcionalidades trabalham com os dados dos produtos e poderiam ser alocadas no módulo ProdutoRepository, conforme ilustrado na imagem a seguir.



Violação do princípio SRP.

A alocação de responsabilidades viola o princípio SRP, pois define um módulo que atende a diferentes atores.

Vamos analisar os atores envolvidos com cada responsabilidade do módulo:

1. Pesquisar produtos por nome e departamento são demandas do cliente da loja.

2. Incluir produto é uma demanda da área de vendas.

3. GerarCSV é uma demanda da área de integração de sistemas.

O módulo ProdutoRepository apresenta diferentes razões para mudar: a área de integração pode demandar novo formato para a exportação dos dados dos produtos, enquanto os clientes podem requerer novas formas de pesquisar os produtos.

Segundo o princípio SRP, devemos separar essas responsabilidades em três módulos, o que nos daria a flexibilidade de separar a busca de produtos e a inclusão deles em serviços fisicamente independentes, o que possibilitaria atender diversas demandas de escalabilidade de forma mais racional.

Exemplo

A busca de produtos pode ter que atender a milhares de clientes simultâneos. No caso da inclusão de itens, esse número não deve passar de algumas dezenas de usuários.

Atividade 1

Questão

O princípio da responsabilidade única (SRP) estipula que cada classe deve ter apenas uma responsabilidade atrelada a um ator específico, o que traz vantagens tanto para a manutenção como para o particionamento do sistema. Em termos da orientação a objetos, o que se busca ao utilizar o padrão?

A

Explorar o polimorfismo.

B

Controlar a geração de instâncias.

C

Diminuir o acoplamento.

D

Melhorar o gerenciamento de composições.

E

Aumentar a coesão.

A alternativa E está correta.

Ao deixar uma classe circunscrita a uma reponsabilidade específica e relacionada a apenas um dos atores do sistema, aumentamos o nível de coesão, o qual estabelece que uma classe não deve assumir responsabilidades que não são suas. O polimorfismo é a alteração de um método herdado, correspondendo a um padrão GRASP sem relação com a restrição do SRP. Mesmo com múltiplas responsabilidades, uma classe pode ter baixo acoplamento quando essas responsabilidades não envolvem elementos externos. O controle de instâncias pode ser feito pelos padrões criacionais do GoF, e o gerenciamento de composições, por padrões estruturais, como o composite, mas nenhum deles restringe as responsabilidades da classe.

O princípio aberto fechado (OCP) e suas características

OCP é uma sigla que representa, em inglês *open closed principle*. Ela estabelece que um módulo deve estar aberto para extensões, mas fechado para modificações. Ou seja, seu comportamento deve ser extensível sem que seja necessário alterá-lo para acomodar as novas extensões.

Neste vídeo, exploramos um dos princípios fundamentais da programação orientada a objetos. Confira como o OCP permite que o software seja aberto para extensão, mas fechado para modificação, garantindo que novos comportamentos possam ser adicionados sem alterar o código existente. Acompanhe exemplos práticos e veja como aplicar esse princípio para criar sistemas mais robustos, flexíveis e fáceis de manter.

Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Quando uma mudança em um módulo causa verdadeira cascata de modificações em módulos dependentes, isso é sinal de que a estrutura do software não acomoda mudanças adequadamente. Se for aplicado de forma adequada, esse princípio permitirá que futuras alterações desse tipo sejam feitas pela adição de novos módulos, e não pela modificação dos já existentes.

Você deve estar pensando como é possível fazer um módulo estender seu comportamento sem que seja necessário mudar uma linha do seu código. A chave para esse enigma se chama **abstração**.

Em uma linguagem orientada a objetos, é possível criar uma interface abstrata que apresente diferentes implementações concretas. Um novo comportamento, em conformidade com essa abstração, pode ser adicionado simplesmente quando um módulo é criado.

Violação do princípio OCP (clonagem)

O exemplo a seguir apresenta um código que não segue o princípio *open closed*. A classe **CalculadoraGeometrica** contém operações para calcular a área de triângulos e quadrados.

```java
public class Triangulo {
	private double base;
	private double altura;

	public double getBase() {
		return base;
	}

	public void setBase(double base) {
		this.base = base;
	}

	public double getAltura() {
		return altura;
	}

	public void setAltura(double altura) {
		this.altura = altura;
	}
}

public class Quadrado {
	double lado;

	public double getLado() {
		return lado;
	}

	public void setLado(double lado) {
		this.lado = lado;
	}
}

public class CalculadoraGeometrica {
	public double obterArea(Quadrado quadrado) {
		return quadrado.getLado() * quadrado.getLado();
	}

	public double obterArea(Triangulo triangulo) {
		return triangulo.getBase() * triangulo.getAltura() / 2;
	}
}
```

Imagine que o programa tenha que calcular a área de um círculo, por exemplo.

O módulo CalculadoraGeometrica poderá calcular a área de um círculo sem que o seu código seja alterado?

Não, pois teremos que adicionar uma nova operação, obterArea, à CalculadoraGeometrica. Essa é uma forma comum de violação, em que uma classe concentra diversas operações da mesma natureza, que operam sobre tipos diferentes.

O exemplo a seguir ilustra outra forma comum de violação do princípio OCP. Acompanhe!

```java
public class FiguraGeometrica {
}

public class Triangulo extends FiguraGeometrica {
    private double base;
    private double altura;

    public double getBase() {
        return base;
    }

    public void setBase(double base) {
        this.base = base;
    }

    public double getAltura() {
        return altura;
    }

    public void setAltura(double altura) {
        this.altura = altura;
    }
}

public class Quadrado extends FiguraGeometrica {
    double lado;

    public double getLado() {
        return lado;
    }

    public void setLado(double lado) {
        this.lado = lado;
    }
}

public class CalculadoraGeometrica {
    public double obterArea(FiguraGeometrica figura) {
        if (figura instanceof Quadrado) {
            return obterAreaQuadrado((Quadrado) figura);
        } else if (figura instanceof Triangulo) {
            return obterAreaTriangulo((Triangulo) figura);
        } else {
            return 0.0;
        }
    }

    private double obterAreaQuadrado(Quadrado quadrado) {
        return  quadrado.getLado() * quadrado.getLado();
    }

    private double obterAreaTriangulo(Triangulo triangulo) {
        return  triangulo.getBase() * triangulo.getAltura() / 2;
    }
}
```

No exemplo, a classe CalculadoraGeometrica possui uma operação obterArea, que recebe um tipo genérico FiguraGeometrica, mas continua concentrando a lógica de cálculo para todos os tipos de figuras. Para incluir um círculo, precisaríamos adicionar a operação obterAreaCirculo e modificar a implementação da operação

obterArea, inserindo nova condição. Imagine como ficaria o código dessa operação se ela tivesse que calcular a área de 50 tipos diferentes de figuras geométricas!

O que devemos fazer para que o módulo CalculadoraGeometrica possa estar aberto para trabalhar com novos tipos de figuras sem precisarmos alterar o seu código?

A resposta está no emprego da abstração e, nesse caso, dos princípios do **especialista** e do **polimorfismo**. Veja:

Figura geométrica

Definimos que toda figura geométrica é capaz de calcular sua área, sendo que a fórmula de cálculo de cada figura específica deve ser implementada nas realizações concretas de FiguraGeometrica.

Nova figura

Dessa forma, para adicionarmos uma nova figura geométrica, basta adicionarmos novo módulo com uma implementação específica para essa operação.

O próximo exemplo ilustra a nova estrutura do projeto.

Reestruturação adequada ao princípio OCP

A classe FiguraGeometrica define uma operação abstrata obterArea.

Aplicando o princípio do especialista, cada subclasse fica responsável pelo cálculo da sua área. Já com o princípio do polimorfismo, o módulo CalculadoraGeometrica passa a depender apenas da abstração FiguraGeometrica, que define que todo elemento desse tipo é capaz de calcular a sua área. Dessa forma, a calculadora geométrica pode funcionar com novos tipos de figuras sem ser necessário alterar o seu código, uma vez que ela deixou de depender da forma específica da figura. Veja:

```
python
```

```
public abstract class FiguraGeometrica {
    public abstract double obterArea();
}

public class Triangulo extends FiguraGeometrica {
    private double base;
    private double altura;

    public double obterArea() {
        return  this.getBase() * this.getAltura() / 2;
    }

    public double getBase() {
        return base;
    }
    public void setBase(double base) {
        this.base = base;
    }
    public double getAltura() {
        return altura;
    }
    public void setAltura(double altura) {
        this.altura = altura;
    }
}

public class Quadrado extends FiguraGeometrica {
    double lado;

    public double obterArea() {
        return  this.getLado() * this.getLado();
    }
    public double getLado() {
        return lado;
    }
    public void setLado(double lado) {
        this.lado = lado;
    }
}

public class CalculadoraGeometrica {
    public double obterArea(FiguraGeometrica figura) {
        return figura.obterArea();
```

Atividade 2

Questão

Assinale a afirmativa que expressa a intenção do princípio *open closed*.

A

Módulos clientes de um tipo genérico devem ser capazes de utilizar qualquer especialização do princípio OCP sem precisar conhecer ou se adaptar a qualquer especialização específica.

B

Idealmente, a incorporação de novas funcionalidades deve ser realizada pela adição de novos módulos, e não pela alteração dos existentes.

C

Módulos clientes não devem ser forçados a depender de outros que trazem elementos desnecessários ou irrelevantes para as suas necessidades de uso.

D

Um módulo de alto nível não deve depender de uma implementação concreta de nível inferior. Ambos devem depender de abstrações.

E

Se um módulo possuir elementos que possam ser modificados por diferentes razões para atender às necessidades de vários clientes, ele deve ser decomposto em um ou mais módulos, de forma que cada um atenda a apenas um cliente.

A alternativa B está correta.

- A alternativa A corresponde ao princípio de substituição de Liskov.
- A alternativa B trata do princípio *open closed* (OCP), que estabelece que o comportamento de um módulo seja extensível, admitindo novas funcionalidades, porém, sem estar aberto para modificações, o que poderia provocar alterações em cascata.
- A alternativa C se relaciona ao princípio de segregação de interfaces.
- A alternativa D corresponde ao princípio de inversão de dependências.
- A alternativa E se refere ao princípio da responsabilidade única.

Princípio da Substituição de Liskov (LSP)

Foi definido em 1988 por Barbara Liskov, a primeira doutora em ciências da computador dos Estados Unidos. Ele estabelece que um tipo deve poder ser substituído por qualquer um de seus subtipos sem alterar o funcionamento correto do sistema.

Veja, neste vídeo, um dos pilares da programação orientada a objetos. Entenda como garantir que subclasses possam substituir suas classes base sem comprometer a funcionalidade do sistema. Aprenda com exemplos práticos e confira por que o LSP é crucial para criar código flexível e robusto.

Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Princípio LSP (quadrado)

O próximo exemplo apresenta um caso clássico de violação do princípio de Liskov. Na geometria, um quadrado pode ser visto como um caso particular de um retângulo. Se levarmos essa propriedade para a

implementação em software, poderemos definir uma classe **quadrado** como uma extensão da classe retangulo.

Note que, como a largura do quadrado é igual ao comprimento, a implementação dos métodos de acesso (setLargura e setComprimento) do quadrado deve se preocupar em preservar essa propriedade. Acompanhe:

```
java

public class Retangulo {
    private double largura;
    private double comprimento;

    public double area() {
        return largura * comprimento;
    }

    public double getLargura() {
        return largura;
    }
    public void setLargura(double largura) {
        this.largura = largura;
    }
    public double getComprimento() {
        return comprimento;
    }
    public void setComprimento(double comprimento) {
        this.comprimento = comprimento;
    }
}

public class Quadrado extends Retangulo {
    public void setLargura(double largura) {
        super.setLargura(largura);
        super.setComprimento(largura);
    }
    public void setComprimento(double largura) {
        super.setLargura(largura);
        super.setComprimento(largura);
    }
}
```

O próximo exemplo mostra como a solução vista viola o princípio da substituição de Liskov. O método verificarArea da classe ClienteRetangulo recebe um objeto r do tipo retangulo. Vamos lá!

Violação do princípio LSP

De acordo com esse princípio, o código desse método deveria funcionar com qualquer objeto de um tipo derivado de retangulo. Entretanto, no caso de um objeto do tipo quadrado, a execução da chamada setComprimento fará com que os dois atributos do quadrado passem a ter o valor 10. Em seguida, a execução da denominada setLargura fará com que os dois atributos do quadrado passem a ter o valor 8, e o valor retornado pelo método área definido na classe retangulo será 64, e não 80, violando o comportamento esperado para um retângulo, como pode ser visto a seguir.

```java
public class ClienteRetangulo {
    public void verificarArea(Retangulo r) throws Exception {
        r.setComprimento(10);
        r.setLargura(8);
        if (r.area() != 80) {
            throw new Exception("area incorreta");
        }
    }
}
```

Embora um quadrado possa ser classificado como um caso particular de um retângulo, o princípio de Liskov estabelece que um subtipo não pode estabelecer restrições adicionais que violem o comportamento genérico do supertipo. Nesse caso, o fato de o quadrado exigir que comprimento e largura sejam iguais o torna mais restrito que o retângulo.

Portanto, do ponto de vista da orientação a objetos, definir a classe quadrado como um subtipo de retângulo é uma solução inapropriada para o problema por violar o princípio de Liskov.

Atividade 3

Questão

Um dos princípios da orientação a objetos é o de que qualquer descendente pode ser utilizado no lugar da classe original, algo que, na prática, corresponde ao LSP. No entanto, a modelagem incorreta das classes pode levar à violação do LSP. Qual dos cenários a seguir representa a isso?

A

Uma classe tributo apresenta um método getValorTributo, e um descendente altera a fórmula de cálculo por meio de polimorfismo.

B

A classe original define métodos abstratos, que são substituídos nos descendentes por versões concretas, obedecendo ao padrão template method.

C

Uma classe Https, derivada de Http, é utilizada para fazer a conexão com uma URL sem TLS, ou seja, um endereço inseguro.

D

Ao programar a resposta para os eventos de uma interface gráfica, no Java, é necessário implementar os métodos da interface Listener associada ao evento.

E

Determinada classe é utilizada em um processo, assim como seus descendentes, mas os descendentes apresentam novos atributos, que não fazem parte do processo.

A alternativa C está correta.

Ao utilizar o descendente Https ocorrerá a tentativa de realizar o handshake do TLS, gerando um erro. Portanto, Https não pode ser utilizada no lugar de Http em termos da conexão, o que representa uma violação do LSP. Quanto às demais opções, se a classe usa polimorfismo ou acrescenta novos atributos, com os processos definidos ao utilizar a classe original, as novas características não interferem na funcionalidade, sendo mantido o LSP. No caso do Java, qualquer implementação do Listener responderá corretamente ao evento, garantindo o LSP – e o padrão template method se baseia justamente no LSP, pois delega para os descendentes a complementação de um processo com a implementação das lacunas que foram definidas por meio de métodos abstratos.

Fundamentos do princípio da segregação de interfaces (ISP)

Estabelece que clientes de uma classe não devem ser forçados a depender de operações que não utilizem.

Neste vídeo, exploramos um dos princípios SOLID de design de software. Veja como o ISP ajuda a criar sistemas mais flexíveis e fáceis de manter ao dividir interfaces grandes em outras menores e mais específicas. Aprenda a aplicar esse princípio para melhorar a modularidade e reduzir a dependência entre componentes, tornando seu código mais robusto e fácil de evoluir.

Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Violação do princípio ISP

O código do exemplo a seguir apresenta uma interface que viola o princípio ISP por reunir operações que dificilmente serão utilizadas em conjunto. As operações login e registrar estão ligadas ao registro e à autorização de acesso de um usuário, enquanto a operação logErro registra mensagens de erro ocorridas durante algum processamento. A operação enviarEmail permite enviar um correio eletrônico para o usuário logado.

```
java

public interface IUsuario {
    boolean login(String login, String senha);
    boolean registrar(String login, String senha, String email);
    void logErro(String msgErro);
    boolean enviarEmail(String assunto, String conteudo);
}
```

Segregação das interfaces

Segundo o princípio ISP, devemos manter a coesão funcional das interfaces, evitando colocar operações de diferentes naturezas em uma única interface. Aplicando ao exemplo, poderíamos segregar a interface original em outras três. Confira o exemplo a seguir.

IAutorizacao

Com as operações de login e registro de usuários.

IMensagem

Com as operações de registro de erro, aviso e informações.

IEmail

Para o envio de e-mail.

Veja o exemplo a seguir:

```
java

public interface IAutorizacao {
    boolean login(String login, String senha);
    boolean registrar(String login, String senha, String email);
}

public interface IEmail {
    boolean enviarEmail(String assunto, String conteudo);
}

public interface IMensagem {
    void logErro(String msgErro);
    void logAviso(String msgAviso);
    void logInfo(String msgInfo);
}
```

Atividade 4

Questão

De acordo com o princípio da segregação de interfaces (ISP), uma classe não deve ser forçada a implementar métodos que não são essenciais para suas funcionalidades específicas. No entanto, esse problema é comum no desenvolvimento de software. Qual das opções a seguir apresenta um componente de software que segue o princípio ISP?

A

Uma interface para conexão com banco de dados estabelece os métodos abrirConexao e fecharConexao, que devem ser implementados pelo programador.

B

Ao usar a interface MouseListener para responder ao clique do mouse, o programador precisou implementar também os demais métodos previstos.

C

Um programador cria um jogo e modela uma classe inimigo, que contém os métodos ataqueMachado e ataqueEspada, derivando a classe espadachim de inimigo.

D

Uma interface gráfica foi criada para o cadastramento de pessoa física, com os campos nome, endereço e CPF, e o programador usou o princípio de herança para estender essa interface, criando uma tela específica para o cadastro de pessoas jurídicas, adicionando o campo CNPJ.

E

Um framework fornece uma interface para conexão com web services, em que os métodos getXML e getJSON são previstos, correspondendo aos formatos de dados recebidos.

A alternativa A está correta.

Sempre que trabalhamos com bancos de dados, o processo inicial é a abertura da conexão, e o processo final é o fechamento dela. Logo, a interface de conexão não apresenta métodos desnecessários, respeitando o ISP. Quanto à demais opções, MouseListener é um bom exemplo de violação do ISP, pois, para responder ao clique, devem ser implementados métodos que não serão necessários. Um espadachim usa apenas uma espada, e o machado é um objeto desnecessário, que não será utilizado ao atacar. Ao derivar a interface gráfica, os campos CPF e CNPJ estarão presentes, mas a empresa utiliza apenas CNPJ. Web services do tipo REST respondem com JSON, enquanto os do tipo SOAP retornam XML, ou seja, sempre há um método que não será necessário, a depender do tipo de web service.

Conhecendo o princípio da inversão de dependências

Também denominado de DIP (do inglês *dependency inversion principle*), estabelece que as entidades concretas devem depender de abstrações, e não de outras entidades concretas. Isso é especialmente importante quando estabelecemos dependências entre entidades de camadas diferentes do sistema.

Neste vídeo, explicamos um dos pilares do design de software orientado a objetos. Veja como o DIP promove a criação de sistemas flexíveis e robustos ao inverter a dependência tradicional entre módulos de alto e baixo nível.

Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Apresentação

Segundo o princípio DIP, um módulo de alto nível não deve depender de um de baixo nível. Entenda:

Módulos de alto nível

São aqueles ligados estritamente ao domínio da aplicação (ex.: loja, produto, ServiçoVenda etc.).

Módulos de baixo nível

São ligados a alguma tecnologia específica (ex.: acesso a banco de dados, interface com o usuário etc.). Por isso são mais voláteis.

Violação do princípio da inversão de dependências

O próximo exemplo apresenta um caso de dependência de um módulo de alto nível em relação a outro de baixo nível. A classe ServicoConsultaProduto depende diretamente da implementação de um repositório de produtos que acessa um banco de dados relacional Oracle.

Essa dependência é estabelecida na operação obterPrecoProduto da classe ServicoConsultaProduto pelo comando *new ProdutoRepositoryOracle()*. Acompanhe no exemplo a seguir.

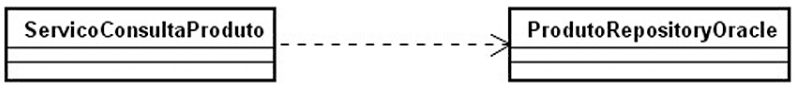
```
java

 public class ProdutoRepositoryOracle {
    public Produto obterProduto(String codigo) {
        Produto p = new Produto();
        // implementacao SQL de recuperação do produto
        return p;
    }

    public void  salvarProduto(Produto produto) {
        // implementacao SQL de insert ou update do produto
    }
}

public class ServicoConsultaProduto {
    public double obterPrecoProduto(String codigo) {
        ProdutoRepositoryOracle repositório
           = new ProdutoRepositoryOracle();
        Produto produto = repositorio.obterProduto(codigo);
        return produto.getPreco();
    }
}
```

A imagem a seguir apresenta, em um diagrama UML, a relação de dependência existente entre a classe concreta ServicoConsultaProduto e a ProdutoRepositoryOracle.



Violação do DIP (Diagrama UML).

Por que essa dependência é inadequada?

Como aspectos tecnológicos são voláteis, devemos isolar a lógica de negócio, permitindo que a implementação concreta desses aspectos possa variar sem afetar as classes que implementam a lógica de negócio.

O exemplo a seguir apresenta a solução resultante da aplicação do princípio estudado. A interface abstrata (ProdutoRepository) define um contrato abstrato entre cliente (ServicoConsultaProduto) e fornecedor (ProdutoRepositoryOracle). Em vez de o cliente depender de um fornecedor específico, ambos passam a depender da interface abstrata.

Assim, a classe ServicoConsultaProduto pode trabalhar com qualquer implementação concreta da interface ProdutoRepository, sem que seu código precise ser alterado. Veja o exemplo:

```java
public interface ProdutoRepository {
    Produto obterProduto(String codigo);
    void  salvarProduto(Produto produto);
}
public class ProdutoRepositoryOracle implements ProdutoRepository {
    public Produto obterProduto(String codigo) {
        Produto p = new Produto();
        // implementacao SQL de recuperação do produto
        return p;
    }

    public void  salvarProduto(Produto produto) {
        // implementacao SQL de insert ou update do produto
    }
}

public class ServicoConsultaProduto {
    ProdutoRepository repositorio;

    public ServicoConsultaProduto(ProdutoRepository repositorio) {
           this.repositorio = repositorio;
    }
    public double obterPrecoProduto(String codigo) {
        Produto produto = repositorio.obterProduto(codigo);
        return produto.getPreco();
    }
}
```

A imagem a seguir apresenta o diagrama UML correspondente à nova estrutura da solução após a aplicação do princípio. A classe ServicoConsultaProduto depende apenas da interface ProdutoRepository, enquanto a classe ProdutoRepositoryOracle é uma das possíveis implementações concretas dessa interface.

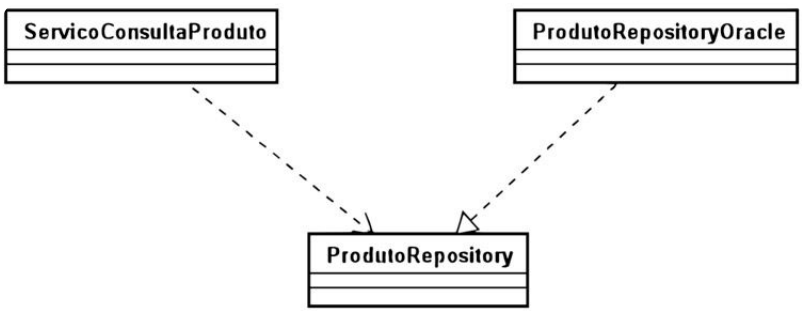


Diagrama UML da solução de acordo com o DIP.

Atividade 5

Questão

Assinale a afirmativa que expressa a intenção do princípio de inversão de dependências.

A

Módulos clientes de um tipo genérico devem ser capazes de utilizar qualquer especialização dele, sem precisar conhecer ou se adaptar a qualquer especialização específica.

B

Idealmente, a incorporação de novas funcionalidades deve ser realizada pela adição de novos módulos, e não pela alteração de módulos existentes.

C

Módulos clientes não devem ser forçados a depender de módulos que trazem elementos desnecessários ou irrelevantes para as suas necessidades de uso.

D

Um módulo de alto nível não deve depender de uma implementação concreta de nível inferior. Ambos devem depender de abstrações.

E

Se um módulo possuir elementos que possam ser modificados por diferentes razões para atender às necessidades de vários clientes, ele deve ser decomposto em um ou mais módulos, de forma que cada um atenda a apenas um cliente.

A alternativa D está correta.

- A alternativa A corresponde ao princípio de substituição de Liskov.
- A alternativa B aborda o princípio Open Closed.
- A alternativa C se refere ao princípio de segregação de interfaces.
- A alternativa D corresponde ao princípio da inversão de dependências (DIP), pelo qual um módulo pode ter dependência em relação a módulos genéricos de nível mais alto (abstrações), mas não em relação a módulos específicos de nível mais baixo.
- A alternativa E se relaciona ao princípio da responsabilidade única.

Padrão factory method

Define uma estrutura simples para que diferentes tipos de objetos possam ser criados, dependendo da classe que os instancia.

Suponha que você esteja criando um sistema na área de big data para a recuperação dos dados contidos em arquivos de diferentes formatos, que exigem leitores específicos para cada tipo de arquivo. Nessa situação, o factory method pode retornar um objeto do tipo leitor, no qual os descendentes podem definir como retorno objetos de classes descendentes, como LeitorJSON, LeitorXML e LeitorPDF.

Acompanhe, neste vídeo, o padrão de design factory method, uma poderosa técnica de programação orientada a objetos, e como ele permite criar objetos de maneira flexível e extensível, delegando a responsabilidade da instância para subclasses especializadas.

Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Problema

O padrão estudado define uma interface genérica de criação de um objeto, deixando a decisão da classe específica a ser instanciada para as implementações concretas da interface. O padrão factory method é muito utilizado no desenvolvimento de frameworks.

> Frameworks definem pontos de extensão (hot spots) nos quais adaptações do código para um sistema específico devem ser feitas.

Tipicamente, frameworks definem classes e interfaces abstratas, que devem ser implementadas nas aplicações que os utilizem. Um exemplo de framework muito usado em Java é o Spring.

Por que esse padrão é importante?

Em aplicações desenvolvidas em Java, por exemplo, podemos criar um objeto de uma classe simplesmente utilizando o operador *new*.

Embora seja uma solução aceitável para pequenos programas, à medida que o sistema cresce, a quantidade de códigos para criar objetos também cresce, espalhando-se por todo o sistema.

Como a criação de um objeto determina a dependência entre a classe em que a instanciação é feita e a classe instanciada, é preciso muito cuidado para não criar dependências que dificultem futuras evoluções do sistema.

Solução

A imagem a seguir apresenta a estrutura de solução proposta pelo padrão factory method.

Os participantes são:

- **Produto**: define o tipo abstrato dos objetos criados pelo padrão. Todos os objetos concretamente instanciados serão derivados desse tipo.

- **ProdutoConcreto**: corresponde a qualquer classe concreta que implementa a definição abstrata do tipo produto. Representa as classes específicas em uma aplicação desenvolvida com um framework.

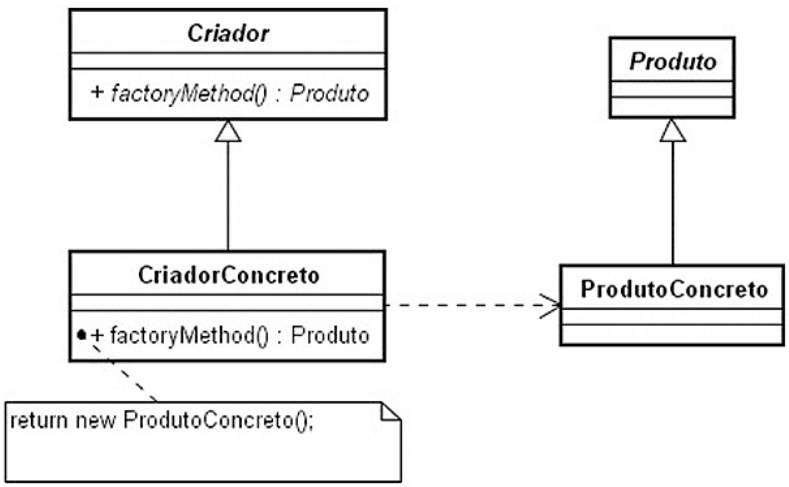
- **Criador**: declara um factory method que define uma interface abstrata, que retorna um objeto genérico do tipo produto.

- **CriadorConcreto**: corresponde a qualquer classe concreta que implemente o factory method definido abstratamente no criador.

A imagem seguinte apresenta um exemplo de aplicação do padrão.

Conteúdo interativo

Acesse a versão digital para ver mais detalhes da imagem abaixo.



Estrutura do padrão factory method.

Na parte **superior**, estão as classes que pertencem a um framework genérico de geração de relatório.

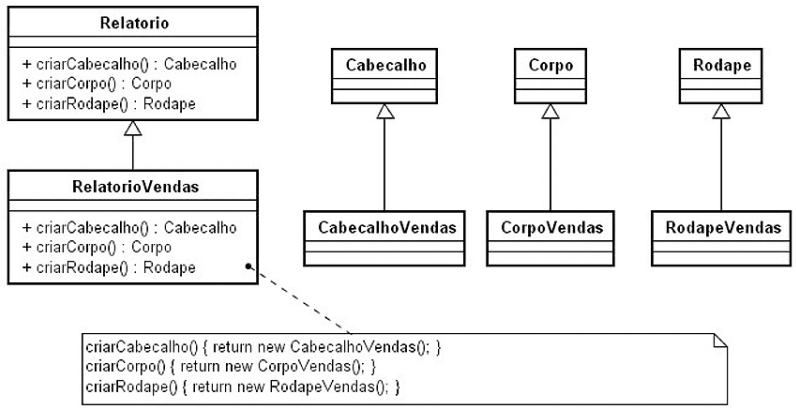
Na parte **inferior**, estão as classes concretas de uma aplicação que utiliza o framework para gerar um relatório de vendas.

As operações criarCabecalho, criarCorpo e criarRodape, definidas na classe genérica, são implementadas concretamente na classe RelatorioVendas. Cada método dessa subclasse cria um objeto específico que implementa os tipos abstratos (Cabecalho, corpo e rodape) definidos no framework.

A classe relatorio desempenha o papel de criador, a classe RelatorioVendas corresponde ao participante CriadorConcreto, enquanto as classes cabecalho, corpo e rodape desempenham o papel de produto. Finalmente, as classes CabecalhoVendas, CorpoVendas e RodapeVendas correspondem ao participante ProdutoConcreto. Veja a imagem a seguir.

Conteúdo interativo

Acesse a versão digital para ver mais detalhes da imagem abaixo.



Exemplo do padrão factory method.

Consequências

O padrão factory method permite criar estruturas arquiteturais genéricas (frameworks), provendo pontos de extensão por meio de operações que instanciam os objetos específicos da aplicação. Dessa forma, todo código genérico pode ser escrito no framework e reutilizado em diferentes aplicações, evitando soluções baseadas em clonagem ou em estruturas condicionais complexas.

Atividade 1

Questão

Os padrões de criação organizam a forma de instanciar objetos e têm o objetivo de tornar a implementação independente da forma com que esses objetos são criados, compostos ou representados. Como se caracteriza o padrão de criação factory method?

A

Define uma interface para criação de objetos na superclasse, mas permite que o retorno do tipo de objeto correto ocorra nas subclasses.

B

Organiza o relacionamento hierárquico entre objetos.

C

Permite copiar objetos existentes, sem que o código fique dependente de suas classes.

D

Utiliza construtores de partes menores, organizadas de alguma forma, para construir objetos mais complexos.

E

Garante que determinada classe forneça apenas uma instância para o sistema.

A alternativa A está correta.

A criação do tipo de objeto correto nas subclasses é definida no factory method. Por exemplo, podemos ter uma classe de gerenciamento logístico, com a definição do método abstrato CriarTransporte, em que a subclasse LogisticaRodoviaria retornaria um carro, enquanto a subclasse LogisticaMarinha retornaria um navio. Relacionamentos hierárquicos são definidos por meio do composite. A cópia de objetos é feita com o Prototype. O padrão builder é utilizado para construir objetos complexos a partir de construtores para as partes menores. Com o uso de singleton, apenas uma instância de determinada classe ficará disponível para o sistema.

Padrão adapter e suas características

Trata-se de um padrão bastante simples, mas que representa uma solução eficiente para a adaptação de sistemas ao uso de componentes não previstos inicialmente. Ele apenas converte a chamada de uma função ou método de classe para o formato requerido pelo sistema, podendo utilizar herança e interface ou instância do objeto adaptado.

Por exemplo, você pode ter um sistema de plotagem que trabalha com ângulos em graus, mas adquire uma biblioteca para cálculo trigonométrico que trabalha com radianos. Portanto, o padrão adapter deve ter métodos com os parâmetros em graus, garantindo que haja conversão para radianos antes de invocar a biblioteca correspondente.

Assista, neste vídeo, ao padrão de design adapter, que ajuda a integrar diferentes sistemas de software, e a como ele permite que classes com interfaces incompatíveis trabalhem juntas, facilitando a reutilização de código e a flexibilidade do sistema.

Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Problema

A questão solucionada por esse padrão é análoga à da utilização de tomadas em diferentes partes do mundo.

Celular sendo carregado com o uso de um adaptador.

Quando um turista americano chega a um hotel no Brasil, ele provavelmente não conseguirá colocar o seu carregador de celular na tomada que está na parede do quarto. Como ele não pode mudar a tomada que está na parede, pois ela é propriedade do hotel, a solução é utilizar um adaptador que converta os pinos do carregador americano em uma saída de três pinos compatível com as tomadas brasileiras.

Como esse problema ocorre em um software?

Imagine que estamos desenvolvendo um software para a operação de vendas de diferentes lojas de departamentos, a fim de que que cada uma possa utilizar uma solução de pagamento entre as diversas disponíveis no mercado.

1. O problema é que cada fornecedor permite a realização de um pagamento em cartão de crédito por meio de uma API específica. fornecida por ele.

1. O software de vendas deve ser capaz de ser plugado em diferentes API de pagamento que ofereçam basicamente o mesmo serviço: intermediar as diversas formas de pagamento existentes no mercado.

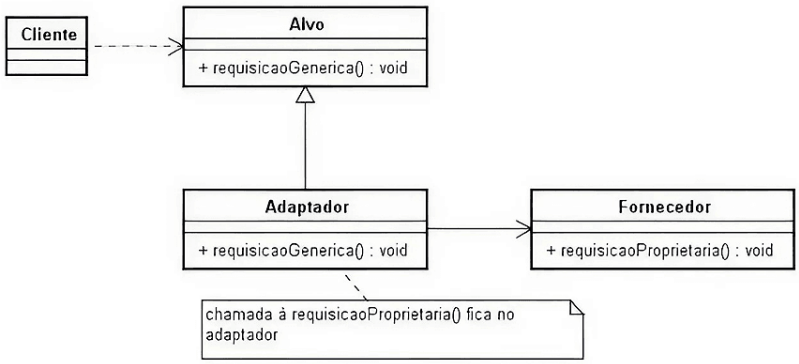
Gostaríamos de poder fazer um software de vendas aderente ao princípio *open closed*, isto é, um que trabalhasse com os novos fornecedores que aparecem no mercado, mas sem termos que mexer em módulos já existentes.

Solução

O adapter é um padrão estrutural. Confira a apresentação de sua estrutura na imagem a seguir.

Conteúdo interativo

Acesse a versão digital para ver mais detalhes da imagem abaixo.



Estrutura do padrão adapter.

O participante fornecedor define uma API proprietária, que não pode ser alterada. O participante alvo representa a generalização dos serviços oferecidos pelos diferentes fornecedores. Os demais módulos do sistema utilizarão apenas o participante alvo, ficando isolados do fornecedor concreto da implementação. O participante adaptador transforma uma requisição genérica feita pelo cliente em uma chamada à API específica do fornecedor com o qual ele está conectado.

Atenção

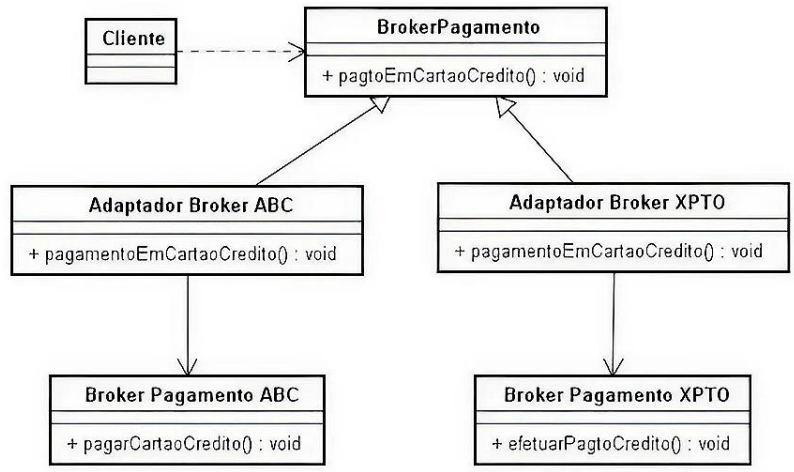
Note que cada fornecedor possuirá um adaptador específico, da mesma forma que existem adaptadores específicos para cada padrão de tomada.

A imagem a seguir apresenta como o padrão adapter poderia ser aplicado no problema do software para a loja de departamentos.

Suponha que existam dois brokers de pagamento (ABC e XPTO), sendo as suas respectivas API representadas pelas classes BrokerPagamentoABC e BrokerPagamentoXPTO. Note que as operações dessas classes, embora similares, têm nomes diferentes e poderiam também ter parâmetros de chamada e retorno diferentes. Além disso, essas classes são fornecidas pelos fabricantes e não podem ser alteradas. Veja:

Conteúdo interativo

Acesse a versão digital para ver mais detalhes da imagem abaixo.



Exemplo do padrão adapter.

A aplicação do padrão adapter consiste em definir uma interface genérica (BrokerPagamento) que será utilizada em todos os módulos do sistema que precisem interagir com brokers de pagamento (representados genericamente pela classe cliente do diagrama). Para cada API específica, criamos um adaptador que implementa a interface genérica BrokerPagamento, traduzindo a chamada da operação genérica pagtoEmCartaoCredito para o protocolo específico da API. Dessa forma, a operação do AdaptadorBrokerABC chamará a operação pagarCartaoCredito do BrokerPagamentoABC, enquanto a operação do AdaptadorBrokerXPTO chamará a operação efetuarPagtoCredito de BrokerPagamentoXPTO.

Consequências

O padrão adapter permite incorporar módulos previamente desenvolvidos, modificando sua interface original sem que haja necessidade de alterá-los, possibilitando a utilização de diferentes implementações proprietárias por meio de uma única interface, sem criar dependências dos módulos clientes com as implementações proprietárias.

Atividade 2

Questão

O padrão adapter pertence ao grupo estrutural do GoF e tem como objetivo adaptar a interface de um novo componente de software às interfaces requeridas pelo sistema. Em qual das opções a seguir o padrão adapter seria indicado?

A

Um sistema de controle de clínica precisou acrescentar algumas rotas no controlador para se comunicar com o web service do SUS.

B

Um sistema de gerenciamento de pessoal precisou reorganizar os níveis hierárquicos, devido a uma mudança estrutural recente na organização.

C

Um sistema ERP teve que modificar o processo de cálculo atual, a fim de incorporar o modelo tributário de imposto único.

D

Um sistema de vendas aceitava apenas pagamento com cartões Mastercard e Visa, que se baseiam na mesma interface, mas foi obrigado a acrescentar a biblioteca do Pix na nova versão.

E

Um aplicativo móvel foi alterado para utilizar menu oculto, com base em arraste, no lugar das abas presentes na primeira versão.

A alternativa D está correta.

Dificilmente a biblioteca do Pix utilizaria a mesma interface adotada atualmente para os cartões no sistema e, para não modificar o restante do código, um adapter pode ser definido para intermediação das chamadas à biblioteca disponibilizada.

O acréscimo de rotas em um controlador não utiliza um adapter, pois envolve os mesmos processos atuais com outros endereços. A refatoração em termos de dados hierárquicos não utilizaria o adpater, mas tiraria proveito do composite. A modificação do cálculo não envolve um adapter e, na melhor das hipóteses, utilizaria polimorfismo. Por último, a modificação do aplicativo móvel seria estrutural, com a substituição de um componente de navegação, e não uma adaptação de chamadas.

Padrão facade

Em muitas situações, determinadas chamadas complexas precisam ser efetuadas em diversos pontos do sistema, o que deixa o código extenso, principalmente nas interfaces gráficas. Isso dificulta a manutenção e diminui o nível de reuso.

O padrão facade oferece uma solução para esse problema, concentrando as chamadas em componentes e subsistemas em classes que irão funcionar como fachadas, eliminando grande parte do código nas demais classes. Você pode, por exemplo, concentrar as chamadas para diversos serviços REST em uma fachada, o que permite que suas interfaces gráficas não tenham códigos voltados para a comunicação HTTP.

Acompanhe, neste vídeo, o padrão GoF facade, um dos padrões de design fundamentais para simplificar a interação com sistemas complexos, e como ele cria uma interface unificada para um conjunto de interfaces em um subsistema, facilitando o uso e promovendo a redução da complexidade no desenvolvimento de software.

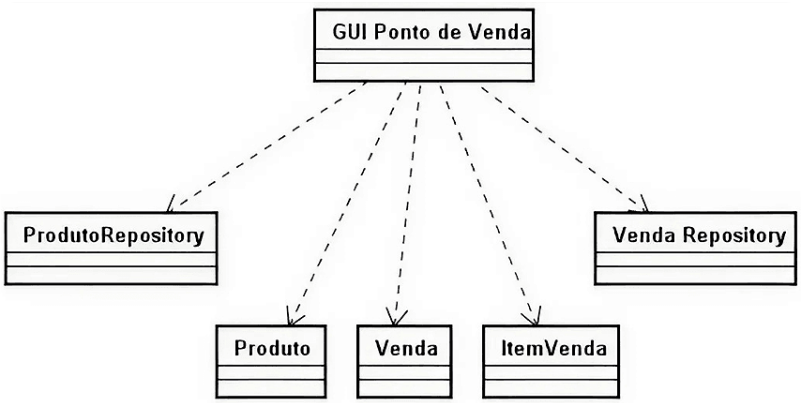
Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Problema

A questão resolvida pelo padrão estrutural facade (ou fachada) é reduzir a complexidade de implementação de uma operação do sistema, fornecendo uma interface simples de uso, ao mesmo tempo em que evita que a implementação precise lidar com diferentes tipos de objetos e chamadas de operações específicas.

A próxima imagem apresenta uma situação típica em que o padrão facade pode ser utilizado. A classe GUI ponto de venda representa um elemento da camada de interface com o usuário responsável por registrar itens vendidos no caixa da loja. Para isso, quando os dados do item são entrados pelo operador, a implementação busca o produto chamando uma operação da classe ProdutoRepository, cria um ItemVenda, adicionando-o a um objeto venda, e salva a nova configuração da venda chamando uma operação da classe VendaRepository. Essa solução é inadequada, pois cria inúmeras dependências e gera maior complexidade em uma classe de interface com o usuário. Observe:



Padrão facade – problema.

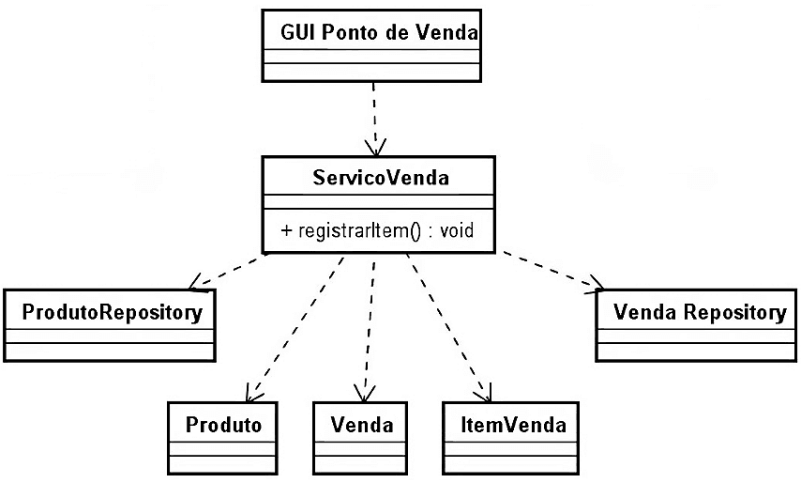
Solução

A imagem a seguir mostra como as dependências entre um elemento de interface com o usuário e os elementos de negócio e armazenamento podem ser simplificadas pela introdução da classe ServicoVenda. Ela passa a servir de fachada para a execução de uma operação complexa do sistema, oferecendo uma interface única e simplificada para a classe GUI ponto de venda.

Em vez de chamar várias operações de diferentes objetos, basta chamar a operação registrarItem, definida na classe fachada. Dessa forma, os elementos da camada de interface com o usuário ficam isolados da complexidade de implementação e da estrutura interna dos objetos pertencentes à lógica de negócio envolvida na operação. Veja:

Conteúdo interativo

Acesse a versão digital para ver mais detalhes da imagem abaixo.



Padrão facade – solução.

Consequências

O padrão facade é bastante utilizado na estruturação dos serviços lógicos que um sistema oferece, isolando a camada de interface com o usuário da estrutura da lógica de negócio do sistema. Portanto, esse padrão nada mais é do que a aplicação do princípio da abstração, em que isolamos um cliente de detalhes irrelevantes de implementação, promovendo uma estrutura com menor acoplamento entre as camadas.

Atividade 3

Questão

Assinale a afirmativa que expressa a intenção do padrão de projeto facade.

A

Definir uma interface genérica de criação de um objeto, deixando a decisão da classe específica a ser instanciada para as implementações concretas dessa interface.

B

Fornecer uma interface simples para uma operação complexa do sistema, evitando que o módulo cliente dessa operação tenha que lidar com diferentes tipos de objetos e chamadas de operações.

C

Encapsular uma família de algoritmos em classes, isolando os módulos clientes das implementações concretas desses algoritmos por meio de uma interface genérica.

D

Permitir a utilização de diferentes interfaces proprietárias de um mesmo serviço lógico, a partir da utilização de uma interface abstrata e de módulos que fazem a conversão da interface abstrata para as interfaces proprietárias.

E

Garantir que exista uma (e apenas uma) instância de uma classe, fornecendo um ponto de acesso global a ela.

A alternativa B está correta.

- A alternativa A corresponde ao padrão factory method.
- A alternativa B se refere ao padrão facade, uma vez que esse padrão visa propiciar uma interface de alto nível simples para as operações oferecidas pelo módulo representado pela fachada.
- A alternativa C se relaciona ao padrão strategy.
- A alternativa D corresponde ao padrão adapter.
- A alternativa E diz respeito ao padrão singleton.

Conhecendo o padrão strategy

Se você já se deparou com cenários nos quais várias ações distintas precisam ser executadas dependendo de uma condição específica, é provável que reconheça a complexidade que pode surgir em um código repleto de estruturas condicionais, como *if* e *else*.

O padrão strategy oferece uma alternativa mais organizada para esse tipo de situação, sendo cada estratégia de ação definida em uma classe diferente. Uma classe de contexto verifica as variáveis de ambiente para selecionar a estratégia correta. Um exemplo simples: em um sistema de controle de armazéns, diferentes estratégias para alocação de espaço podem ser definidas, de acordo com a natureza e as dimensões do produto.

Confira, neste vídeo, um dos padrões de design mais poderosos e versáteis do catálogo Gang of four e como o padrão strategy permite que você defina uma família de algoritmos, encapsule cada um deles e os torne intercambiáveis, facilitando a manutenção e a expansão do seu código.

Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Problema

O padrão strategy soluciona o problema que ocorre quando temos diferentes algoritmos para realizar determinada tarefa.

Exemplo

Uma loja de departamentos pode ter diferentes políticas de desconto aplicáveis em função da época do ano (ex.: Natal, Páscoa, Dia das mães e Dia das crianças).

Como organizar esses algoritmos de forma que possamos respeitar o princípio open closed, fazendo com que o sistema de vendas possa aplicar novas políticas de desconto sem que seja necessário modificar o que já está implementado?

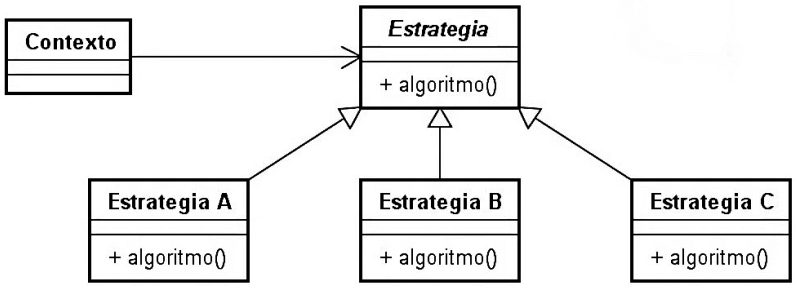
Solução

O padrão strategy define uma família de algoritmos, em que cada um é encapsulado em sua própria classe. Os diferentes algoritmos compõem uma família que pode ser utilizada de forma intercambiável, e novos algoritmos podem ser adicionados à família sem afetar o código existente.

A imagem a seguir apresenta a estrutura do padrão. O tipo estrategia define uma interface comum a todos os algoritmos, podendo ser implementado como uma classe abstrata ou como uma interface. A interface genérica é utilizada pelo participante contexto, quando esse necessitar que o algoritmo seja executado. Estrategia A, B e C são implementações específicas do algoritmo genérico que o participante xontexto pode disparar. A estratégia específica deve ser injetada no módulo contexto, que pode alimentar a estratégia com seus dados. Veja:

Conteúdo interativo

Acesse a versão digital para ver mais detalhes da imagem abaixo.

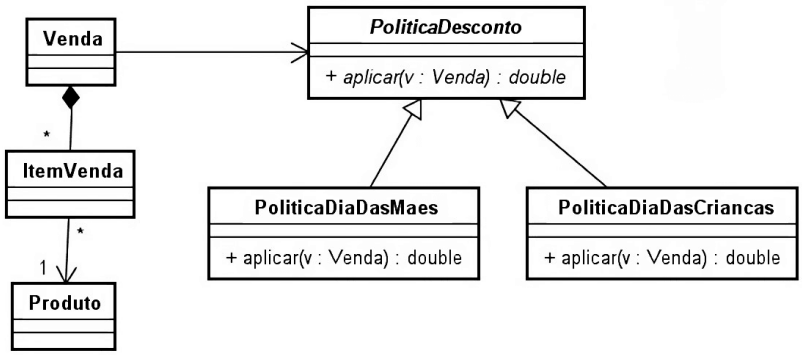


Padrão strategy – estrutura.

A próxima imagem apresenta um exemplo de aplicação do padrão strategy. A classe venda representa uma venda da loja de departamentos. Uma política de desconto pode ser associada a uma venda no momento de sua instanciação e utilizada para cálculo do valor a ser pago pelo cliente. Acompanhe:

Conteúdo interativo

Acesse a versão digital para ver mais detalhes da imagem abaixo.



Exemplo do padrão strategy.

Para obter o valor a pagar da venda, basta obter o seu valor total, somando o valor de seus itens, e subtrair o valor retornado pela operação aplicar da instância de PoliticaDesconto, associada ao objeto venda.

Note que novas políticas podem ser agregadas à solução. Basta adicionar uma nova implementação da interface PoliticaDesconto. Nenhuma alteração na classe venda é necessária, pois ela faz uso apenas da interface genérica, não dependendo de nenhuma política de descontos específica.

Consequências

O padrão strategy permite separar algoritmos de diferentes naturezas dos elementos necessários para sua execução. No exemplo da loja de departamentos, podemos aplicar à venda diferentes tipos de algoritmo, como: política de desconto, cálculo de frete e prazo de entrega, entre outros. Cada tipo de algoritmo conta com especializações que podem ser implementadas em uma família própria de algoritmos.

O padrão strategy oferece uma solução elegante e extensível para o encapsulamento de algoritmos, que podem ter diferentes implementações, isolando, de suas implementações concretas, os módulos clientes desses algoritmos. Além disso, ele permite a remoção de estruturas condicionais complexas normalmente presentes quando as variações não são implementadas com esse padrão.

Atividade 4

Questão

Assinale a afirmativa que expressa a intenção do padrão de projeto strategy.

A

Definir uma interface genérica de criação de um objeto, deixando a decisão da classe específica a ser instanciada para as implementações concretas dessa interface.

B

Fornecer uma interface simples para uma operação complexa do sistema, evitando que o módulo cliente dessa operação tenha que lidar com diferentes tipos de objetos e chamadas de operações.

C

Encapsular uma família de algoritmos em classes, isolando os módulos clientes das implementações concretas desses algoritmos por meio de uma interface genérica.

D

Permitir a utilização de diferentes interfaces proprietárias de um mesmo serviço lógico, a partir da utilização de uma interface abstrata e de módulos que fazem a conversão da interface abstrata para as interfaces proprietárias.

E

Garantir que exista uma (e apenas uma) instância de uma classe, fornecendo um ponto de acesso global a ela.

A alternativa C está correta.

- A alternativa A corresponde ao padrão factory method.
- A alternativa B se refere ao padrão facade.
- A alternativa C se relaciona ao padrão strategy, pois ele visa encapsular algoritmos de uma mesma família em uma estrutura de classes que realizam uma interface genérica, fazendo com que os clientes desses algoritmos utilizem apenas essa interface.
- A alternativa D corresponde ao padrão adapter.
- A alternativa E diz respeito ao padrão singleton.

Template method

Não são raras as situações em que grande parte de um processo bem estabelecido possa ser reutilizada em outros contextos, mas com pequenas modificações. O padrão template method parte desse princípio, definindo o esqueleto principal de uma operação genérica, com chamada para métodos abstratos em determinados trechos, delegando para os filhos a implementação desses métodos.

Por exemplo, você pode criar um método de persistência genérico, com a abertura da conexão, inicialização da transação, execução do comando SQL, finalização da transação e desconexão, mas de forma que o comando a ser executado seja definido nos descendentes.

Neste vídeo, exploramos uma técnica poderosa que define a estrutura de um algoritmo em uma classe-mãe, permitindo que subclasses específicas personalizem certos passos do processo sem alterar a estrutura geral. Veja como esse padrão facilita a reutilização de código e promove a flexibilidade em projetos de software orientado a objetos, enquanto exemplificamos sua aplicação prática em cenários do mundo real.

> **Conteúdo interativo**
>
> Acesse a versão digital para assistir ao vídeo.

Problema

O padrão template method é aplicável quando temos diferentes implementações de uma operação, em que alguns passos são idênticos e outros são específicos de cada implementação.

Para ilustrar o problema, suponha a implementação de duas máquinas de bebidas: uma de café e outra de chá. O procedimento de preparação é bem similar, pois há alguns passos em comum, mas outros específicos:

Máquina de café

- Esquentar a água.
- Preparar mistura (via moagem do café).
- Colocar a mistura no copo.
- Adicionar açúcar, se selecionado.
- Adicionar leite, se selecionado.
- Liberar a bebida.

Máquina de chá

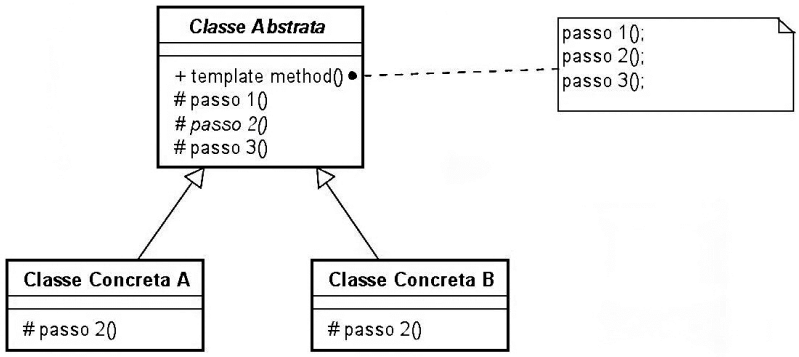
- Esquentar a água.
- Preparar mistura (via infusão do chá).
- Colocar a mistura no copo.
- Adicionar açúcar, se selecionado.
- Adicionar limão, se selecionado.
- Liberar a bebida.

O problema consiste em implementar diferentes formas concretas de um algoritmo padrão contendo pontos de variação, sem clonar o algoritmo em cada forma concreta nem produzir uma única implementação contendo diversas estruturas condicionais.

Solução

A imagem a seguir apresenta a estrutura geral da solução.

> **Conteúdo interativo**
>
> Acesse a versão digital para ver mais detalhes da imagem abaixo.



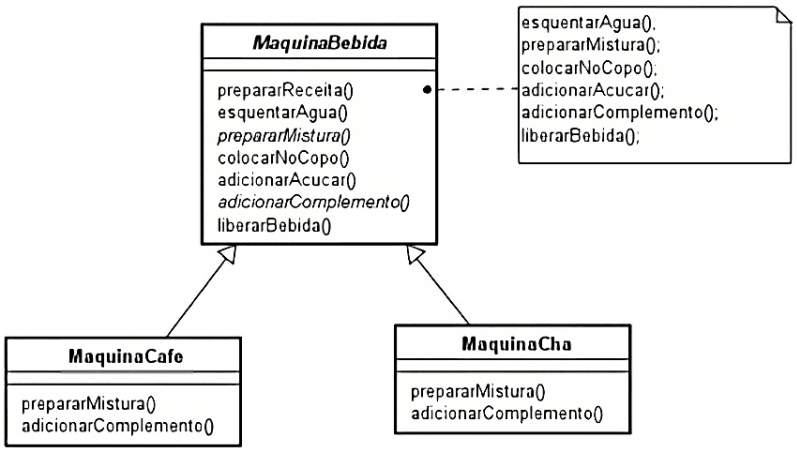
Template method – estrutura.

O procedimento genérico é definido em um método na classe abstrata. Cada passo do procedimento é definido como uma operação na classe abstrata. Alguns passos são comuns e, portanto, podem ser implementados na própria classe abstrata. Os passos específicos são definidos como operações abstratas na classe abstrata, sendo as implementações específicas implementadas nas subclasses, como é o caso do passo 2.

A imagem seguinte apresenta uma solução para o problema das máquinas de café e de chá, utilizando o padrão template method.

Conteúdo interativo

Acesse a versão digital para ver mais detalhes da imagem abaixo.



Template method – exemplo.

O método prepararReceita, definido na superclasse MaquinaBebida, possui seis passos estabelecidos como operações na mesma classe. Note que existem dois passos definidos como operações abstratas, pois são executados de forma específica nas duas máquinas: prepararMistura e adicionarComplemento. Cada máquina derivada da classe abstrata MaquinaBebida implementa apenas os passos específicos, isto é, apenas a implementação das duas operações abstratas.

Consequências

O padrão template method evita a duplicação de algoritmos que apresentem a mesma estrutura, com alguns pontos de variação entre eles. O algoritmo é implementado apenas uma vez em uma classe abstrata, em que os pontos de variação são definidos. As subclasses específicas implementam esses pontos de variação. Dessa forma, qualquer alteração no algoritmo comum pode ser feita em apenas um módulo.

Esse padrão é muito utilizado na implementação de frameworks, permitindo a construção de estruturas de controle invertidas, também conhecidas como princípio de Hollywood ou Não nos chame, nós o chamaremos, referindo-se a estruturas em que a superclasse é quem chama os métodos definidos nas subclasses, e não o contrário.

Atividade 5

Questão

O padrão template method é muito utilizado em diversos frameworks, em processos nos quais a funcionalidade geral é definida, permitindo que os programadores personalizem os pontos de extensão do processo para atingir os objetivos específicos do sistema. Em relação ao padrão template method, é correto afirmar que:

A

garante que apenas uma instância do objeto seja criada.

B

serve como base para a inversão de controle.

C

duplica algoritmos com estruturas semelhantes.

D

deve ser classificado como um padrão estrutural.

E

permite centralizar o tratamento das requisições do usuário.

A alternativa B está correta.

Com o template method é possível implementar um macroprocesso que tem pontos de extensão especializados pelo programador, e quando um framework utiliza esse padrão, o programador o deixa executar o processo, apenas complementando os passos específicos da aplicação, o que é chamado de inversão de controle. Com relação às demais opções, o template method evita a duplicação do código ao centralizar a parte comum, por isso se trata de um padrão comportamental, e quem pode centralizar as requisições é o controlador do GRASP.

Considerações finais

O que você aprendeu neste conteúdo?

- O que é um padrão de projeto.
- Principais padrões de projeto.
- Padrões GRASP.
- Princípios SOLID.
- Principais padrões GoF.

Podcast

Confira agora um resumo sobre os padrões de projeto.

Conteúdo interativo

Acesse a versão digital para ouvir o áudio.

Explore +

Indicamos a leitura dos livros:

- **Patterns of enterprise application architecture**, de Martin Fowler. O livro apresenta padrões para módulos de interface com o usuário, persistência de dados, lógica de negócio e integração de sistemas.

- **Java EE 8 design patterns and best practices**, de Rhuan Rocha e João Purificação. O livro aborda padrões para interface com o usuário, lógica do negócio, integração de sistemas, orientação a aspectos, programação reativa e microsserviços.

Além de livros, é possível encontrar muitos artigos técnicos por meio de buscas na internet, usando os acrônimos dos tipos de padrões e princípios abordados no conteúdo (GoF, GRASP, SOLID) ou os nomes específicos dos padrões e princípios que você pretenda utilizar, aproveitando soluções adotadas com sucesso por outras pessoas.

Referências

BUSCHMANN, F. *et al.* **Pattern-oriented software architecture:** a system of patterns. Nova Jersey: John Wiley & Sons, 2000.

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

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

MARTIN, R.C. **Clean architecture:** a craftsman´s guide to software structure and design. Nova Jersey: Prentice Hall, 2017.