Reconhecer o propósito dos principais padrões GoF e as situações nas quais eles podem ser aplicados

Factory Method

- **Problema**

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

Os ***frameworks*** definem pontos de extensão (***hot spots***) nos quais devem ser feitas adaptações do código para um sistema específico. Tipicamente, os ***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 este padrão é importante?

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

Comentário

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.

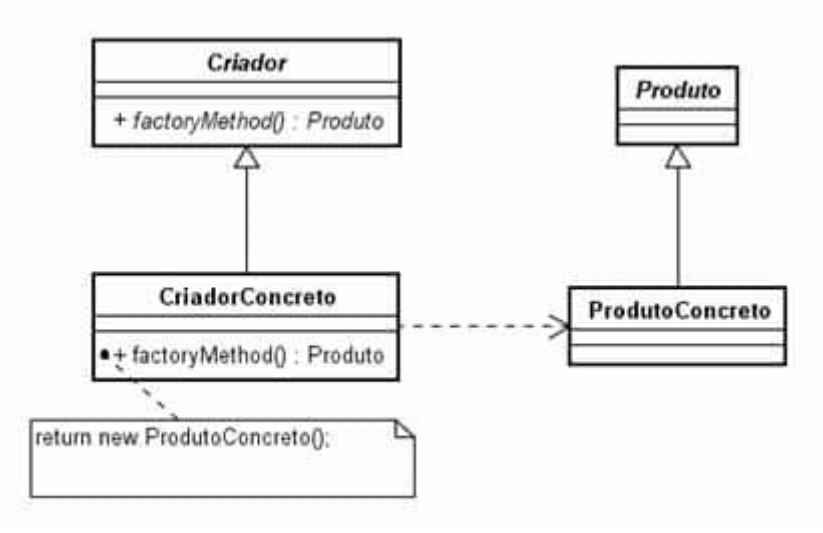
Como a criação de um objeto determina dependência entre a classe onde a instanciação está sendo feita e a classe instanciada, é preciso muito cuidado para não criar dependências que dificultem futuras evoluções do sistema.

- **Solução**

A figura 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 deste 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.



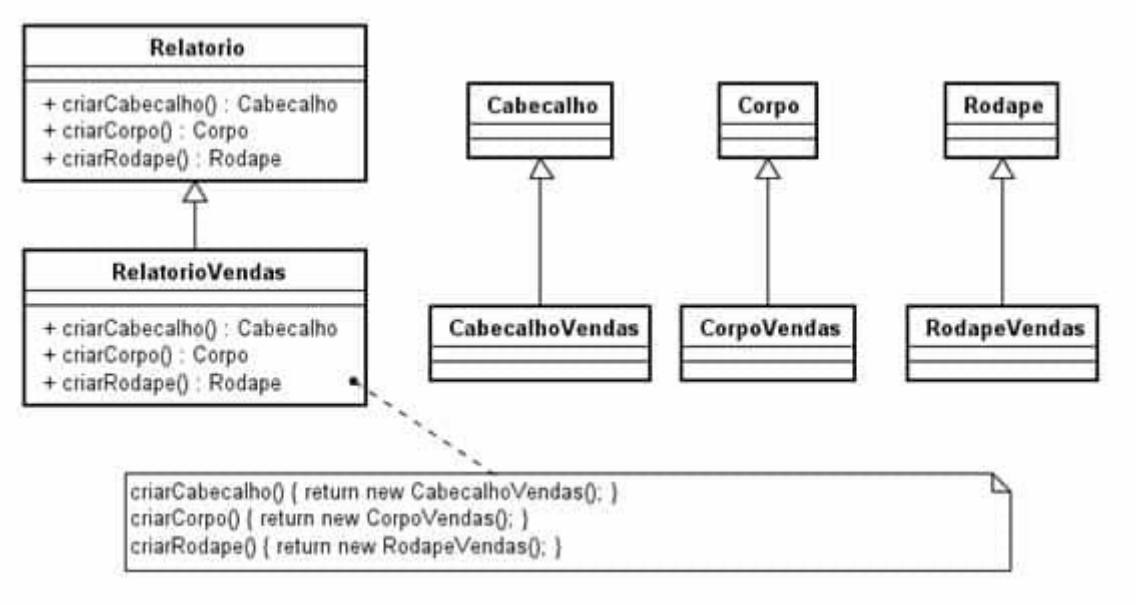
Estrutura do padrão *Factory Method*.
A figura seguinte apresenta um exemplo de aplicação do padrão.

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.



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.

Adapter

- **Problema**

O problema resolvido por este padrão é análogo ao da utilização de tomadas em diferentes partes do mundo.

Exemplo

Quando um turista americano chega em 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?

Clique nas setas para ver o conteúdo.
Imagine que estamos desenvolvendo um software para a operação de vendas de diferentes lojas de departamentos, e que cada loja possa utilizar uma solução de pagamento dentre as diversas disponíveis no mercado.

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.

Portanto, o software de vendas deve ser capaz de ser plugado a diferentes API de pagamento que oferecem 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 software que pudesse trabalhar com novos fornecedores que apareçam no mercado, sem termos que mexer em módulos já existentes.

- **Solução**

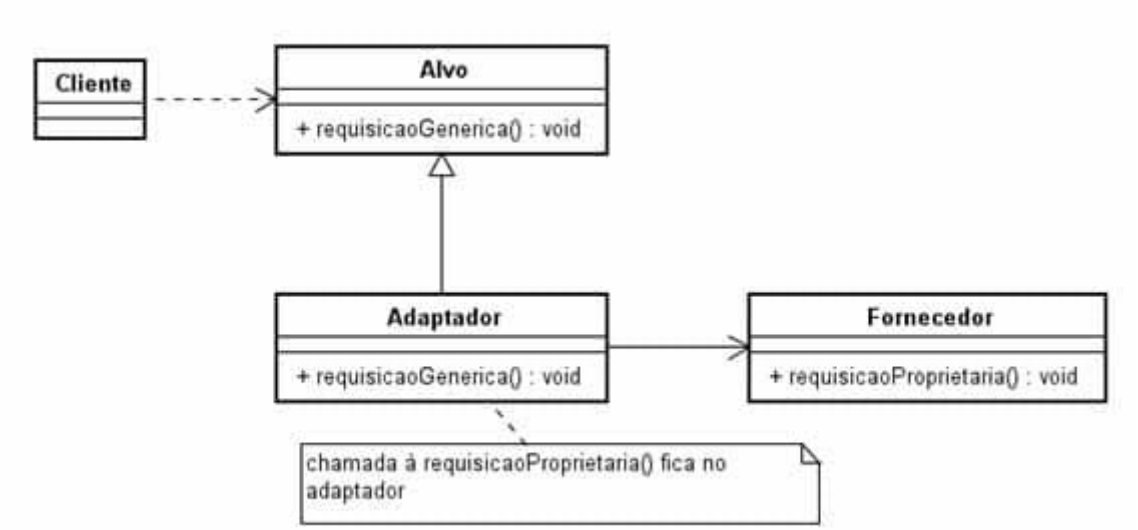
O Adapter é um padrão estrutural, cuja estrutura é apresentada na figura a seguir.

O participante Fornecedor define uma API proprietária e que não pode ser alterada. O participante Alvo representa a generalização dos serviços oferecidos pelos diferentes fornecedores. Os demais módulos do sistema utilizarão apenas este participante, 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

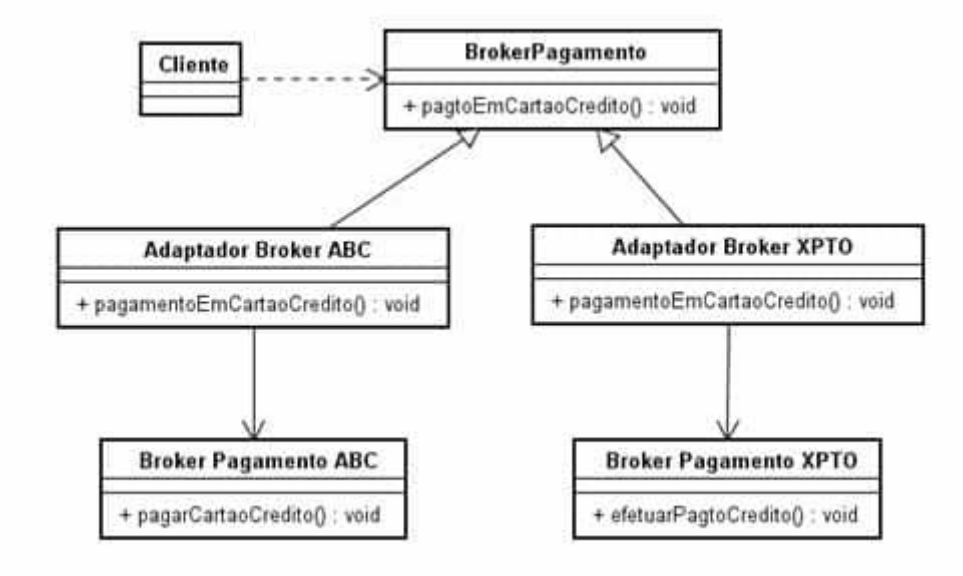
Note que cada fornecedor possuirá um adaptador específico, da mesma forma que existem adaptadores específicos para cada padrão de tomada.



Estrutura do padrão Adapter

A figura a seguir apresenta como este padrão 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. Essas classes são fornecidas pelos fabricantes e não podem ser alteradas.

A aplicação deste padrão consiste em definir uma interface genérica (BrokerPagamento) que será utilizada em todos os módulos do sistema que precisem interagir com o ***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.



Exemplo do padrão Adapter.

- **Consequências**

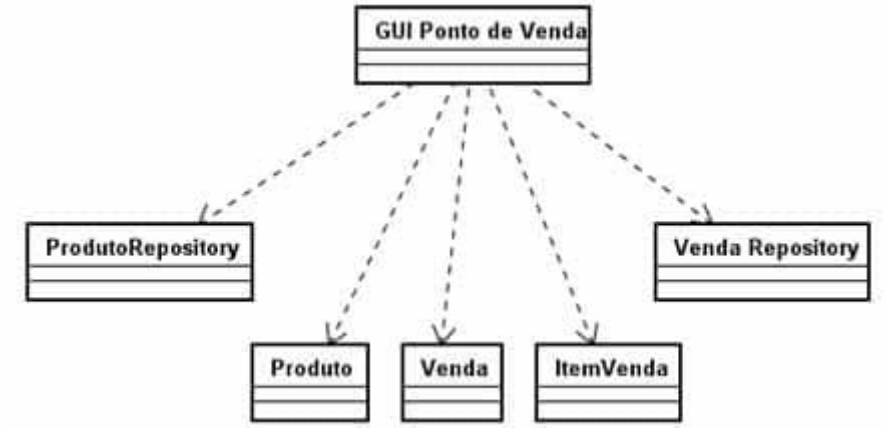
O padrão Adapter permite incorporar módulos previamente desenvolvidos, modificando sua interface original sem que haja necessidade de alterá-los, possibilitando a utilização de diferentes implementações proprietárias por meio de uma única interface, sem criar dependências dos módulos clientes com as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

Facade

- **Problema**

O problema resolvido pelo padrão estrutural ***Facade*** (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.

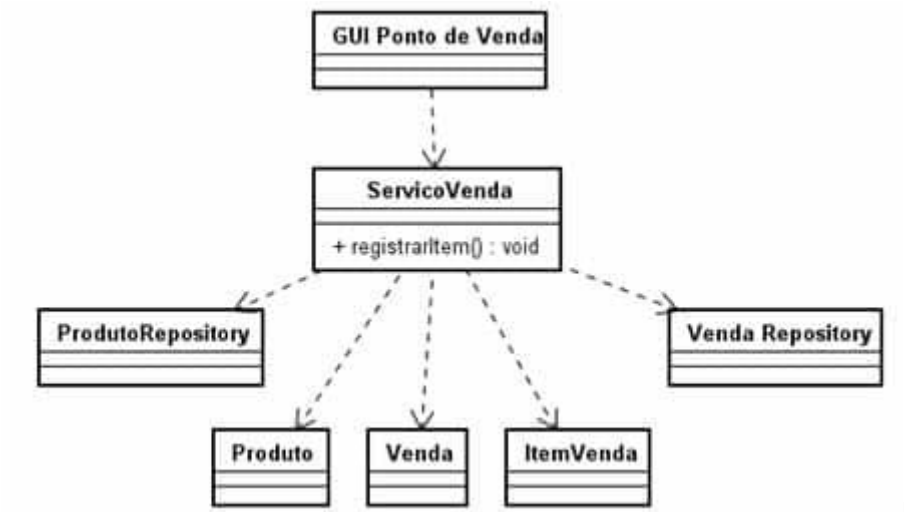
A próxima figura apresenta uma situação típica em que o padrão ***Facade*** pode ser utilizado. A classe GUI Ponto de Venda representa um elemento da camada de interface com o usuário responsável por registrar itens vendidos no caixa da loja. Para isso, quando os dados do item são entrados pelo operador, 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.



Padrão Facade – problema.

- **Solução**

A figura a seguir mostra como essas 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 ela 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.



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, este 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.

Strategy

- **Problema**

O problema resolvido pelo padrão **Strategy** ocorre quando temos diferentes algoritmos para realizar determinada tarefa.

Exemplo

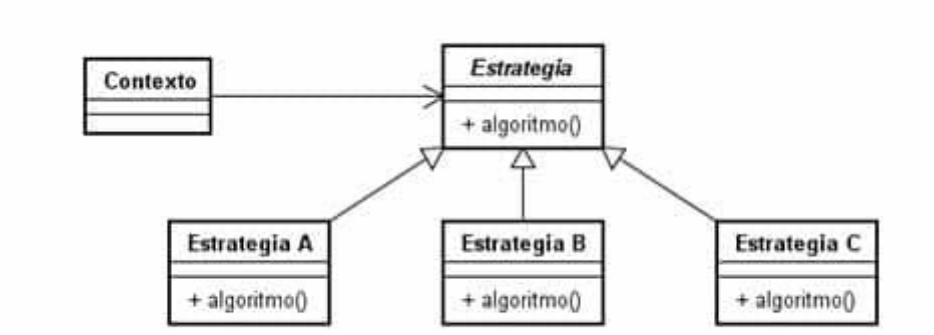
Uma loja de departamentos pode ter diferentes políticas de desconto aplicáveis em função da época do ano (ex.: Natal, Páscoa, Dia das Mães, Dia das Crianças).

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, onde 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 figura 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. Essa interface genérica é utilizada pelo participante Contexto quando este necessitar que o algoritmo seja executado. Estrategia A, B e C são implementações específicas do algoritmo genérico que o participante Contexto pode disparar. A estratégia específica deve ser injetada no módulo Contexto, que pode alimentar a estratégia com seus dados.

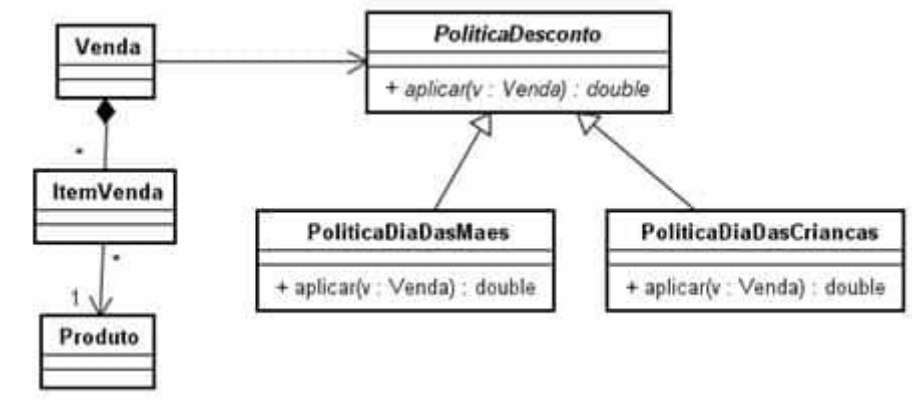


Padrão Strategy – estrutura.

A próxima figura 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.

Para obter o valor a pagar da Venda, basta obter o seu valor total, somando o valor dos 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, bastando 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.



Exemplo do padrão Strategy.

- **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, tais 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, permite a remoção de estruturas condicionais complexas normalmente presentes quando essas variações não são implementadas com este padrão.

Template Method

- **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, com alguns passos em comum e outros específicos:

Máquina de café	Máquina de chá
Esquentar a água	Esquentar a água
Preparar mistura **(via moagem do café)**	Preparar mistura **(via infusão do chá)**

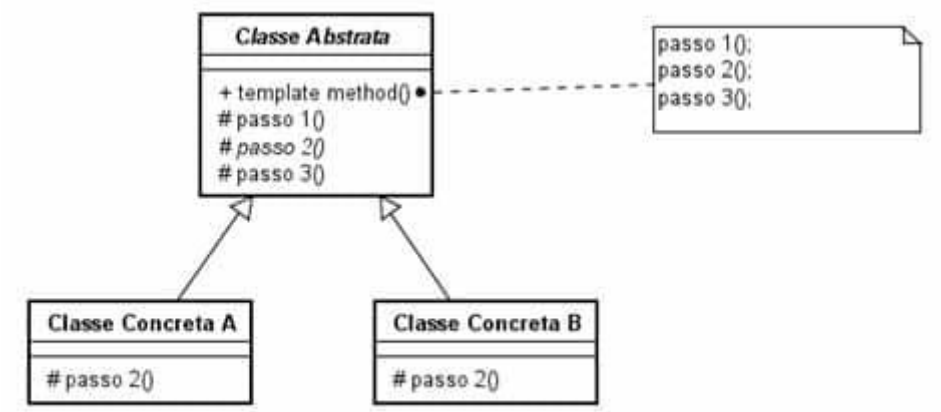
Colocar a mistura no copo	Colocar a mistura no copo
Adicionar açúcar, se selecionado	Adicionar açúcar, se selecionado
Adicionar **leite**, se selecionado	Adicionar **limão**, se selecionado
Liberar a bebida	Liberar a bebida

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**

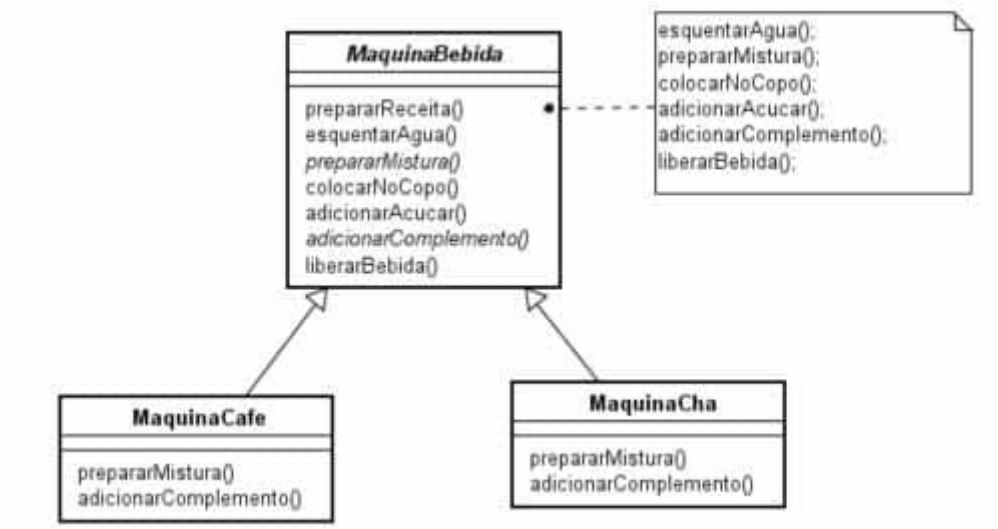
A figura a seguir apresenta a estrutura geral da solução.

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.



Template Method – estrutura.
A figura seguinte apresenta uma solução para o problema da máquina de café e da máquina de chá utilizando este padrão.

O método prepararReceita, definido na superclasse MaquinaBebida, possui seis passos que são definidos como operações nesta 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 dessas duas operações abstratas.



Template Method – exemplo.

- **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 onde são definidos os pontos de variação. 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.