INSTITUTO FEDERAL
Paraná

PLANO DE AULA

Professor: João Vitor Campõe Galescky
Público Alvo: Programadores
Previsão de Tempo: 10 minutos
Tema: *Design Patters - Behaviour Patterns*
Título da Aula: *Strategy*

https://refactoring.guru/design-patterns/strategy
https://www.geeksforgeeks.org/system-design/strategy-pattern-set-1/
https://www.geeksforgeeks.org/system-design/behavioral-design-patterns/
Diagrama PlantUML - traduzido de GURU

Strategy

Strategy é um padrão de design comportamental que permite definir a família dos algoritmos, pondo cada um em classes separadas, tornando seus objetos intercambiáveis [GURU]. Permite que os clientes troquem algoritmos dinamicamente sem alterar a estrutura do código, útil quando deseja alterar dinamicamente o comportamento de uma classe sem modificar seu código [GEEK].

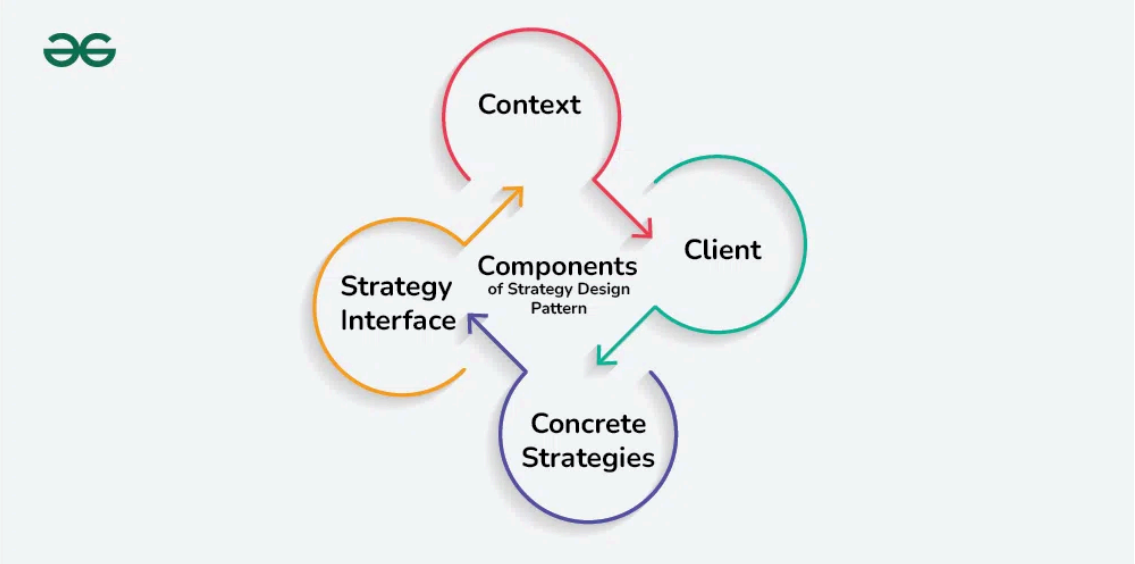
Padrão Comportamental

Categoria de padrões de design que se concentra nas interações e na comunicação entre objetos. Eles ajudam a definir como os objetos colaboram e distribuem responsabilidades entre si, facilitando o gerenciamento de fluxos de controle complexos e a comunicação em um sistema [GEEK].

Características

- Define uma família de algoritmos: O padrão permite encapsular vários algoritmos ou comportamentos em classes separadas, conhecidas como estratégias.
- Encapsula comportamentos: cada estratégia encapsula um comportamento ou algoritmo específico, fornecendo uma maneira limpa e modular de gerenciar diferentes variações ou implementações.
- Permite a troca dinâmica de comportamento: o padrão permite que os clientes alternem entre diferentes estratégias em tempo de execução, permitindo mudanças de comportamento flexíveis e dinâmicas.
- Promove a colaboração de objetos: o padrão incentiva a colaboração entre um objeto de contexto e objetos de estratégia, onde o contexto delega a execução de um comportamento a um objeto de estratégia.

Componentes



Fonte: GeekForGeeks

Problema

Um dia você decide criar um aplicativo de navegação para viajantes casuais. O app estava centralizado em volta de um mapa estilizado que ajuda os usuários a rapidamente se orientarem em qualquer cidade.

Uma das funcionalidades mais requisitadas para o app era o planejamento automático de rotas. O usuário deveria ser capaz de digitar um endereço e ver a rota mais rápida para aquele destino exibido no mapa.

A primeira versão do app somente conseguia construir rotas pelas ruas. Pessoas que viajavam de carro ficaram maravilhadas. Entretanto, nem todo mundo gosta de dirigir nas férias. Então, na próxima atualização, você adicionou uma opção para construir rotas para pedestres. Em seguida, adicionou outra opção para permitir o uso de transporte público nas rotas.

Porém, aquilo era apenas o começo. Mais tarde, planejou adicionar o planejamento de rotas para ciclistas. Mais a frente, outra opção para construir rotas passando por todas as atrações turísticas da cidade.

Enquanto, por uma perspectiva comercial, o app era um sucesso, a parte técnica causava muitas dores de cabeça. Cada vez que adiciona-se um novo algoritmo de roteamento, a classe principal do navegador duplica de tamanho. Em certo ponto, ficou-se difícil de controlar.

Qualquer mudança em um dos algoritmos, um simples ajuste de bug ou de pontuação nas ruas, afetava toda a classe, aumentando a chance de criar erros em código que antes funcionava.

Além disso, o trabalho em equipe se tornou ineficiente. Seus colegas, contratados após o lançamento de sucesso, reclamavam que passavam muito tempo resolvendo conflitos de merge. Implementar uma nova funcionalidade exigia que todos mudassem a mesma classe enorme, entrando em conflito com o código dos outros.

Solução

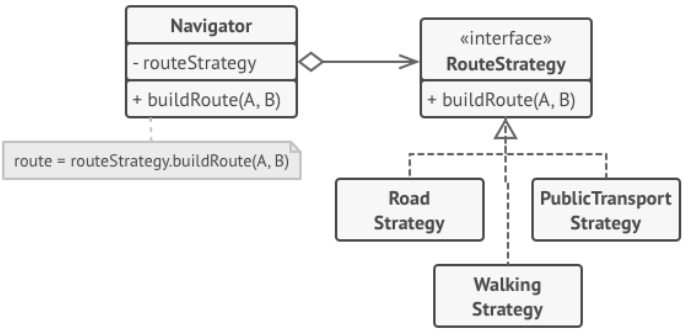
O padrão *Strategy* sugere que pegue uma classe que faz algo específico de muitas formas diferentes e extraia todos esses algoritmos para classes separadas chamadas strategies.

A classe original, chamada *context*, deve ter um campo para armazenar uma referência a uma das strategies. O *context* delega o trabalho para o objeto *strategy* vinculado, em vez de executá-lo por conta própria.

O *context* não é responsável por selecionar o algoritmo apropriado para o trabalho. Em vez disso, o cliente passa a *strategy* desejada para o *context*. Na verdade, o *context* não sabe muito sobre as *strategies*. Trabalha com todas as *strategies* através da mesma interface genérica, que expõe apenas um método para disparar o algoritmo encapsulado na strategy selecionada.

Assim, o *context* se torna independente das *strategies* concretas, permitindo adicionar novos algoritmos ou modificar os existentes sem alterar o código do *context* ou das outras *strategies*.

Estratégia de Planejamento de Rotas



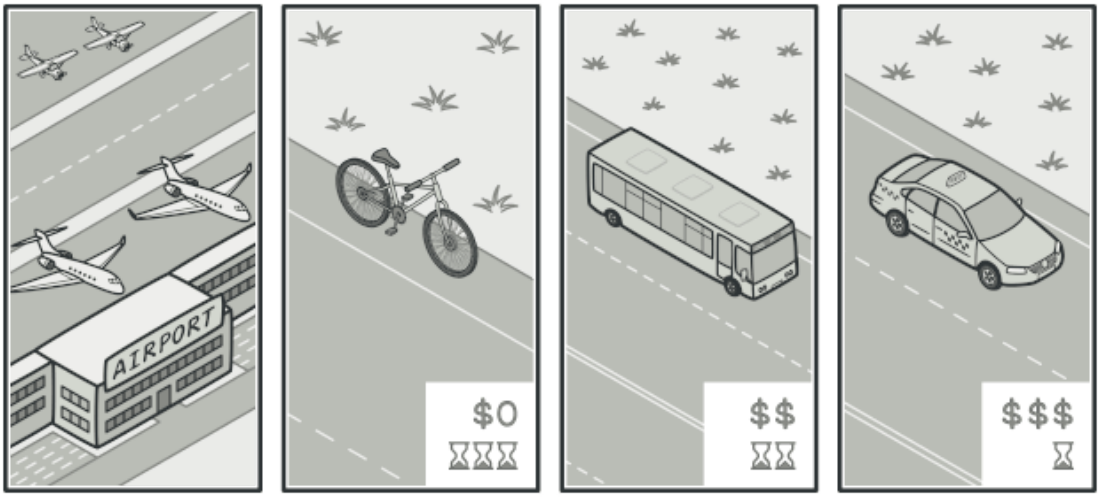
No nosso app de navegação, cada algoritmo de roteamento pode ser extraído para sua própria classe com um único método buildRoute. Esse método recebe uma origem e um destino, e retorna uma coleção de pontos de verificação da rota.

Mesmo que, dado o mesmo argumento, cada classe possa construir uma rota diferente, a classe principal do navegador não se importa qual algoritmo está selecionado, já que seu trabalho principal é renderizar um conjunto de pontos no mapa. Essa classe tem um método para alternar a

estratégia de roteamento ativa, para que os clientes, como botões na interface, possam substituir o comportamento de roteamento atualmente selecionado por outro.

Analogia

Imagine que você precisa ir para o aeroporto. Você pode pegar um ônibus, chamar um táxi ou ir de bicicleta. Estas são suas estratégias de transporte. Você escolhe uma estratégia dependendo de fatores como orçamento ou tempo.



Aplicação

Use o padrão Strategy quando você quiser usar diferentes variantes de um algoritmo dentro de um objeto e poder alternar de um para outro em tempo de execução.

O padrão Strategy permite alterar indiretamente o comportamento do objeto em tempo de execução associando-o a diferentes sub-objetos que podem executar sub tarefas específicas de diferentes formas.

Use o Strategy quando você tem várias classes similares que diferem apenas na forma como executam algum comportamento.

O padrão Strategy permite extrair o comportamento variável para uma hierarquia separada de classes, combinando as classes originais em uma só, reduzindo assim código duplicado.

Use o padrão para isolar a lógica de negócio de uma classe dos detalhes de implementação de algoritmos que podem não ser tão importantes no contexto daquela lógica.

O padrão Strategy isola o código, dados internos e dependências de vários algoritmos do restante do código. Vários clientes recebem uma interface simples para executar os algoritmos e alterná-los em tempo de execução.

INSTITUTO FEDERAL
Paraná

Use o padrão quando sua classe tem um enorme bloco condicional que escolhe entre diferentes variantes do mesmo algoritmo.

O padrão Strategy elimina esse condicional ao extrair todos os algoritmos para classes separadas, todas implementando a mesma interface. O objeto original delega a execução para um desses objetos, ao invés de implementar todas as variantes do algoritmo.

Como implementar

1. Na classe *context*, identifique um algoritmo que costuma mudar com frequência. Pode ser um grande condicional que seleciona e executa uma variante do mesmo algoritmo em tempo de execução.
2. Declare uma interface *strategy* comum a todas as variantes do algoritmo.
3. Extraia, uma a uma, todas as variantes do algoritmo para suas próprias classes. Elas devem implementar a interface *strategy*.
4. Na classe *context*, adicione um campo para armazenar a referência ao objeto *strategy*. Forneça um método para substituir essa referência. O *context* deve trabalhar com o objeto *strategy* apenas via interface *strategy*. O contexto pode definir uma interface para permitir que strategy acesse seus dados.
5. Os clientes do contexto devem associá-lo a uma *strategy* adequada que corresponda à forma como esperam que ele execute seu trabalho principal.

Prós e Contras

Vantagens:

- Você pode trocar os algoritmos usados dentro de um objeto em tempo de execução.
- Pode isolar os detalhes de implementação de um algoritmo do código que o usa.
- Pode substituir herança por composição.
- Princípio Aberto/Fechado: você pode introduzir novas estratégias sem alterar o contexto.

Desvantagens:

- Se você tem só alguns algoritmos e eles raramente mudam, não há razão para complicar o programa com novas classes e interfaces do padrão.
- Os clientes precisam conhecer as diferenças entre as estratégias para escolher a correta.
- Muitas linguagens modernas suportam funções anônimas, o que permite implementar variantes de um algoritmo como funções, usando-as como estratégias, sem precisar adicionar muitas classes e interfaces.