refactoring.guru

Strategy

13–17 minutes

Também conhecido como: Estratégia

Propósito

O **Strategy** é um padrão de projeto comportamental que permite que você defina uma família de algoritmos, coloque-os em classes separadas, e faça os objetos deles intercambiáveis.

Problema

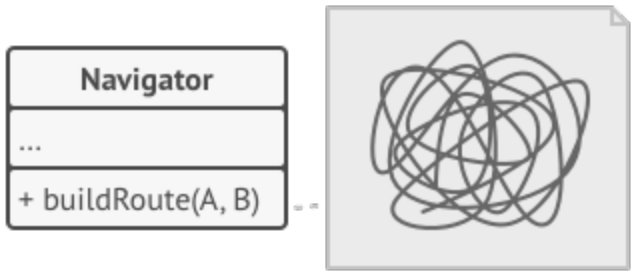
Um dia você decide criar uma aplicação de navegação para

viajantes casuais. A aplicação estava centrada em um mapa bonito que ajudava os usuários a se orientarem rapidamente em uma cidade.

Uma das funcionalidades mais pedidas para a aplicação era o planejamento automático de rotas. Um usuário deveria ser capaz de entrar com um endereço e ver a rota mais rápida no mapa.

A primeira versão da aplicação podia apenas construir rotas sobre rodovias, e isso agradou muito quem viaja de carro. Porém aparentemente, nem todo mundo dirige em suas férias. Então com a próxima atualização você adicionou uma opção de construir rotas de caminhada. Logo após isso você adicionou outra opção para permitir que as pessoas usem o transporte público.

Contudo, isso foi apenas o começo. Mais tarde você planejou adicionar um construtor de rotas para ciclistas. E mais tarde, outra opção para construir rotas até todas as atrações turísticas de uma cidade.



O código do navegador ficou muito inchado.

Embora da perspectiva de negócio a aplicação tenha sido um sucesso, a parte técnica causou a você muitas dores de cabeça. Cada vez que você adicionava um novo algoritmo de roteamento, a classe principal do navegador dobrava de tamanho. Em determinado momento, o monstro se tornou algo muito difícil de se manter.

Qualquer mudança a um dos algoritmos, seja uma simples correção de bug ou um pequeno ajuste no valor das ruas, afetava toda a classe, aumentando a chance de criar um erro no código já existente.

Além disso, o trabalho em equipe se tornou ineficiente. Seus companheiros de equipe, que foram contratados após ao bem sucedido lançamento do produto, se queixavam que gastavam muito tempo resolvendo conflitos de fusão. Implementar novas funcionalidades necessitava mudanças na classe gigantesca, conflitando com os códigos criados por outras pessoas.

Solução

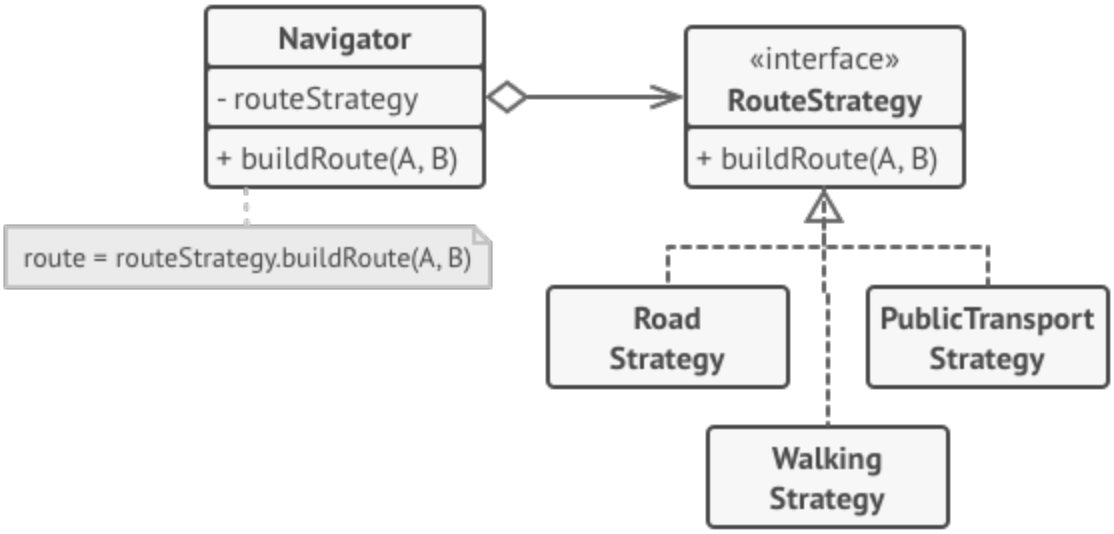
O padrão Strategy sugere que você pegue uma classe que faz algo específico em diversas maneiras diferentes e extraia todos esses algoritmos para classes separadas chamadas *estratégias*.

A classe original, chamada *contexto*, deve ter um campo para armazenar uma referência para um dessas estratégias. O contexto delega o trabalho para um objeto estratégia ao invés de executá-lo por conta própria.

O contexto não é responsável por selecionar um algoritmo apropriado para o trabalho. Ao invés disso, o cliente passa a estratégia desejada para o contexto. Na verdade, o contexto não sabe muito sobre as estratégias. Ele trabalha com todas elas através de uma interface genérica, que somente expõe um único método para acionar o algoritmo encapsulado dentro da estratégia selecionada.

Desta forma o contexto se torna independente das estratégias concretas, então você pode adicionar novos algoritmos ou

modificar os existentes sem modificar o código do contexto ou outras estratégias.

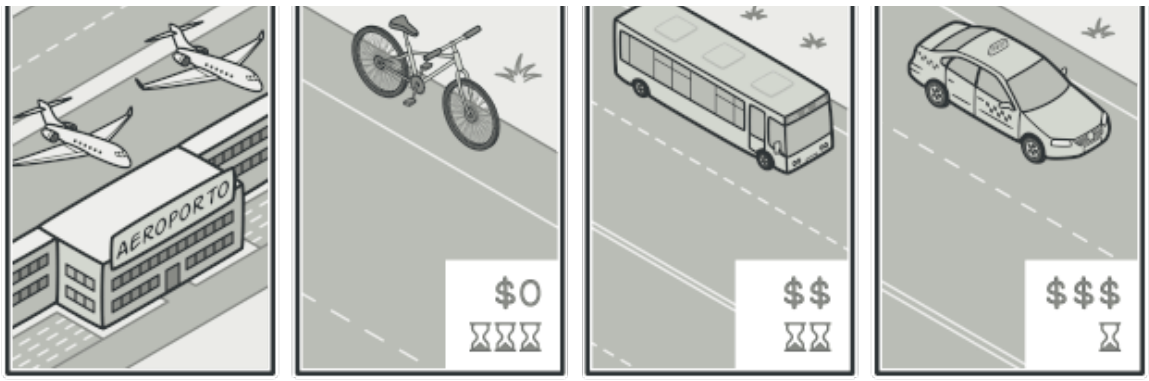


Estratégias de planejamento de rotas.

Em nossa aplicação de navegação, cada algoritmo de roteamento pode ser extraído para sua própria classe com um único método `construirRota`. O método aceita uma origem e um destino e retorna uma coleção de pontos da rota.

Mesmo dando os mesmos argumentos, cada classe de roteamento pode construir uma rota diferente, a classe navegadora principal não se importa qual algoritmo está selecionado uma vez que seu trabalho primário é renderizar um conjunto de pontos num mapa. A classe tem um método para trocar a estratégia ativa de rotas, então seus clientes, bem como os botões na interface de usuário, podem substituir o comportamento de rotas selecionado por um outro.

Analogia com o mundo real



Várias estratégias para se chegar ao aeroporto.

Imagine que você tem que chegar ao aeroporto. Você pode pegar um ônibus, pedir um táxi, ou subir em sua bicicleta. Essas são suas estratégias de transporte. Você pode escolher uma das estratégias dependendo de fatores como orçamento ou restrições de tempo.

Estrutura

1. O **Contexto** mantém uma referência para uma das estratégias concretas e se comunica com esse objeto através da interface da estratégia.
2. A interface **Estratégia** é comum à todas as estratégias concretas. Ela declara um método que o contexto usa para executar uma estratégia.
3. **Estratégias Concretas** implementam diferentes variações de um algoritmo que o contexto usa.
4. O contexto chama o método de execução no objeto estratégia ligado cada vez que ele precisa rodar um algoritmo. O contexto não sabe qual tipo de estratégia ele está trabalhando ou como o algoritmo é executado.
5. O **Cliente** cria um objeto estratégia específico e passa ele para o

contexto. O contexto expõe um setter que permite o cliente mudar a estratégia associada com contexto durante a execução.

Pseudocódigo

Neste exemplo, o contexto usa múltiplas **estratégias** para executar várias operações aritméticas.

```
// A interface estratégia declara operações comuns a todas as
// versões suportadas de algum algoritmo. O contexto usa essa
// interface para chamar o algoritmo definido pelas estratégias
// concretas.
interface Strategy is
    method execute(a, b)

// Estratégias concretas implementam o algoritmo enquanto
seguem
// a interface estratégia base. A interface faz delas
// intercomunicáveis no contexto.
class ConcreteStrategyAdd implements Strategy is
    method execute(a, b) is
        return a + b

class ConcreteStrategySubtract implements Strategy is
    method execute(a, b) is
        return a - b

class ConcreteStrategyMultiply implements Strategy is
    method execute(a, b) is
        return a * b
```

```
// O contexto define a interface de interesse para clientes.
class Context is
    // O contexto mantém uma referência para um dos objetos
    // estratégia. O contexto não sabe a classe concreta de uma
    // estratégia. Ele deve trabalhar com todas as estratégias
    // através da interface estratégia.
    private strategy: Strategy

    // Geralmente o contexto aceita uma estratégia através do
    // construtor, e também fornece um setter para que a
    // estratégia possa ser trocado durante o tempo de execução.
    method setStrategy(Strategy strategy) is
        this.strategy = strategy

    // O contexto delega algum trabalho para o objeto estratégia
    // ao invés de implementar múltiplas versões do algoritmo
    // por conta própria.
    method executeStrategy(int a, int b) is
        return strategy.execute(a, b)


// O código cliente escolhe uma estratégia concreta e passa ela
// para o contexto. O cliente deve estar ciente das diferenças
// entre as estratégia para que faça a escolha certa.
class ExampleApplication is
    method main() is
        Cria um objeto contexto.

        Lê o primeiro número.
        Lê o último número.
```

Lê a ação desejada da entrada do usuário

```
if (action == addition) then
    context.setStrategy(new ConcreteStrategyAdd())

if (action == subtraction) then
    context.setStrategy(new ConcreteStrategySubtract())

if (action == multiplication) then
    context.setStrategy(new ConcreteStrategyMultiply())

result = context.executeStrategy(First number, Second number)
```

Imprimir resultado.

Aplicabilidade

Utilize o padrão Strategy quando você quer usar diferentes variantes de um algoritmo dentro de um objeto e ser capaz de trocar de um algoritmo para outro durante a execução.

O padrão Strategy permite que você altere indiretamente o comportamento de um objeto durante a execução ao associá-lo com diferentes sub-objetos que pode fazer sub-tarefas específicas em diferentes formas.

Utilize o Strategy quando você tem muitas classes parecidas que somente diferem na forma que elas executam algum comportamento.

O padrão Strategy permite que você extraia o comportamento

variante para uma hierarquia de classe separada e combine as classes originais em uma, portando reduzindo código duplicado.

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

O padrão Strategy permite que você isole o código, dados internos, e dependências de vários algoritmos do restante do código. Vários clientes podem obter uma simples interface para executar os algoritmos e trocá-los durante a execução do programa.

Utilize o padrão quando sua classe tem um operador condicional muito grande que troca entre diferentes variantes do mesmo algoritmo.

O padrão Strategy permite que você se livre dessa condicional ao extrair todos os algoritmos para classes separadas, todos eles implementando a mesma interface. O objeto original delega a execução de um desses objetos, ao invés de implementar todas as variantes do algoritmo.

Como implementar

1. Na classe contexto, identifique um algoritmo que é sujeito a frequentes mudanças. Pode ser também uma condicional enorme que seleciona e executa uma variante do mesmo algoritmo durante a execução do programa.
2. Declare a interface da estratégia comum para todas as variantes do algoritmo.
3. Um por um, extraia todos os algoritmos para suas próprias

classes. Elas devem todas implementar a interface estratégia.

4. Na classe contexto, adicione um campo para armazenar uma referência a um objeto estratégia. Forneça um setter para substituir valores daquele campo. O contexto deve trabalhar com o objeto estratégia somente através da interface estratégia. O contexto pode definir uma interface que deixa a estratégia acessar seus dados.
5. Os Clientes do contexto devem associá-lo com uma estratégia apropriada que coincide com a maneira que esperam que o contexto atue em seu trabalho primário.

Prós e contras

- Você pode trocar algoritmos usados dentro de um objeto durante a execução.
- Você pode isolar os detalhes de implementação de um algoritmo do código que usa ele.
- Você pode substituir a herança por composição.
- *Princípio aberto/fechado*. Você pode introduzir novas estratégias sem mudar o contexto.
- Se você só tem um par de algoritmos e eles raramente mudam, não há motivo real para deixar o programa mais complicado com novas classes e interfaces que vêm junto com o padrão.
- Os Clientes devem estar cientes das diferenças entre as estratégias para serem capazes de selecionar a adequada.
- Muitas linguagens de programação modernas tem suporte do tipo funcional que permite que você implemente diferentes versões de

um algoritmo dentro de um conjunto de funções anônimas. Então você poderia usar essas funções exatamente como se estivesse usando objetos estratégia, mas sem inchar seu código com classes e interfaces adicionais.

Relações com outros padrões

- O Bridge, State, Strategy (e de certa forma o Adapter) têm estruturas muito parecidas. De fato, todos esses padrões estão baseados em composição, o que é delegar o trabalho para outros objetos. Contudo, eles todos resolvem problemas diferentes. Um padrão não é apenas uma receita para estruturar seu código de uma maneira específica. Ele também pode comunicar a outros desenvolvedores o problema que o padrão resolve.
- O Command e o Strategy podem ser parecidos porque você pode usar ambos para parametrizar um objeto com alguma ação. Contudo, eles têm propósitos bem diferentes.
- Você pode usar o *Command* para converter qualquer operação em um objeto. Os parâmetros da operação se transformam em campos daquele objeto. A conversão permite que você atrase a execução de uma operação, transforme-a em uma fila, armazene o histórico de comandos, envie comandos para serviços remotos, etc.
- Por outro lado, o *Strategy* geralmente descreve diferentes maneiras de fazer a mesma coisa, permitindo que você troque esses algoritmos dentro de uma única classe contexto.
- O Decorator permite que você mude a pele de um objeto, enquanto o Strategy permite que você mude suas entranhas.
- O Template Method é baseado em herança: ele permite que você

altere partes de um algoritmo ao estender essas partes em subclasses. O Strategy é baseado em composição: você pode alterar partes do comportamento de um objeto ao suprir ele como diferentes estratégias que correspondem a aquele comportamento. O *Template Method* funciona a nível de classe, então é estático. O *Strategy* trabalha a nível de objeto, permitindo que você troque os comportamentos durante a execução.

- O State pode ser considerado como uma extensão do Strategy. Ambos padrões são baseados em composição: eles mudam o comportamento do contexto ao delegar algum trabalho para objetos auxiliares. O *Strategy* faz esses objetos serem completamente independentes e alheios entre si. Contudo, o *State* não restringe dependências entre estados concretos, permitindo que eles alterem o estado do contexto à vontade.