#R4E
Software Developer

Design Patterns

Strategy

Conteúdo

- Propósito
- Problema
- Solução
- Estrutura
- Aplicabilidade
- Como Implementar
- Prós e Contras
- Relações c/ Outros Padrões

Propósito

- O **Strategy** é um padrão de projeto comportamental (Behavioral Pattern) que permite definir uma família de algoritmos, colocá-los em classes separadas, e fazer os seus objetos intercambiáveis.

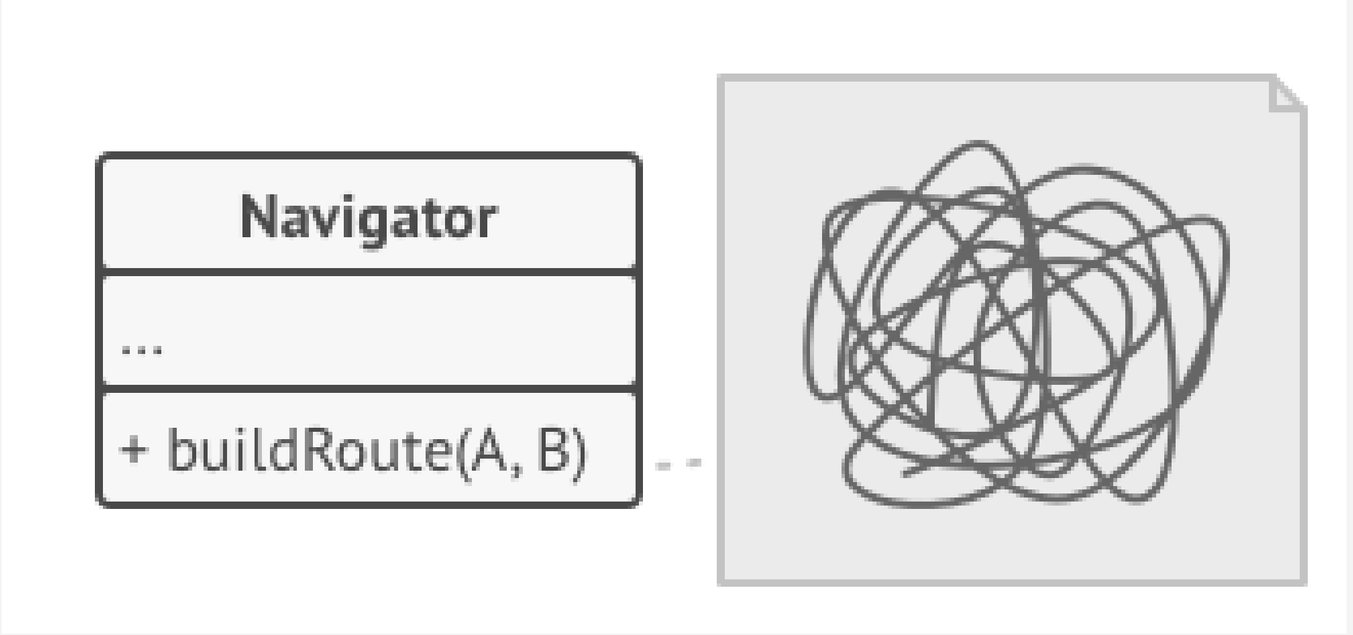
Problema

- Um dia decides criar uma aplicação de navegação para viajantes casuais. A aplicação estava centrada num mapa bonito que ajudava os utilizadores na orientação rapidamente numa cidade.
- Uma das funcionalidades mais pedidas para a aplicação era o cálculo automático de rotas. Um utilizador deveria ser capaz de introduzir uma morada e ver a rota mais rápida no mapa.

Problema

- A primeira versão da aplicação podia apenas construir rotas sobre rodovias, e isso agradou muito aos utilizadores que viajam de carro. Porém aparentemente, nem todos conduzem quando estão de férias. Então na próxima atualização adicionas uma opção de calcular rotas de caminhada. Após isso adicionas outra opção para permitir que as pessoas usem o transporte público.
- Contudo, foi apenas o começo. Mais tarde planeias adicionar um construtor de rotas para ciclistas. E mais tarde, outra opção para construir rotas até todas as atrações turísticas da cidade.

Problema



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

Problema

- 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 de equipa ficou ineficiente. Os membros da equipa, que foram contratados após ao grande lançamento do produto, queixavam-se que gastavam muito tempo a resolver **conflitos de merge**. Implementar novas funcionalidades implicava mudanças na classe gigantesca, gerando conflitos com os códigos criados por outras pessoas.

Solução

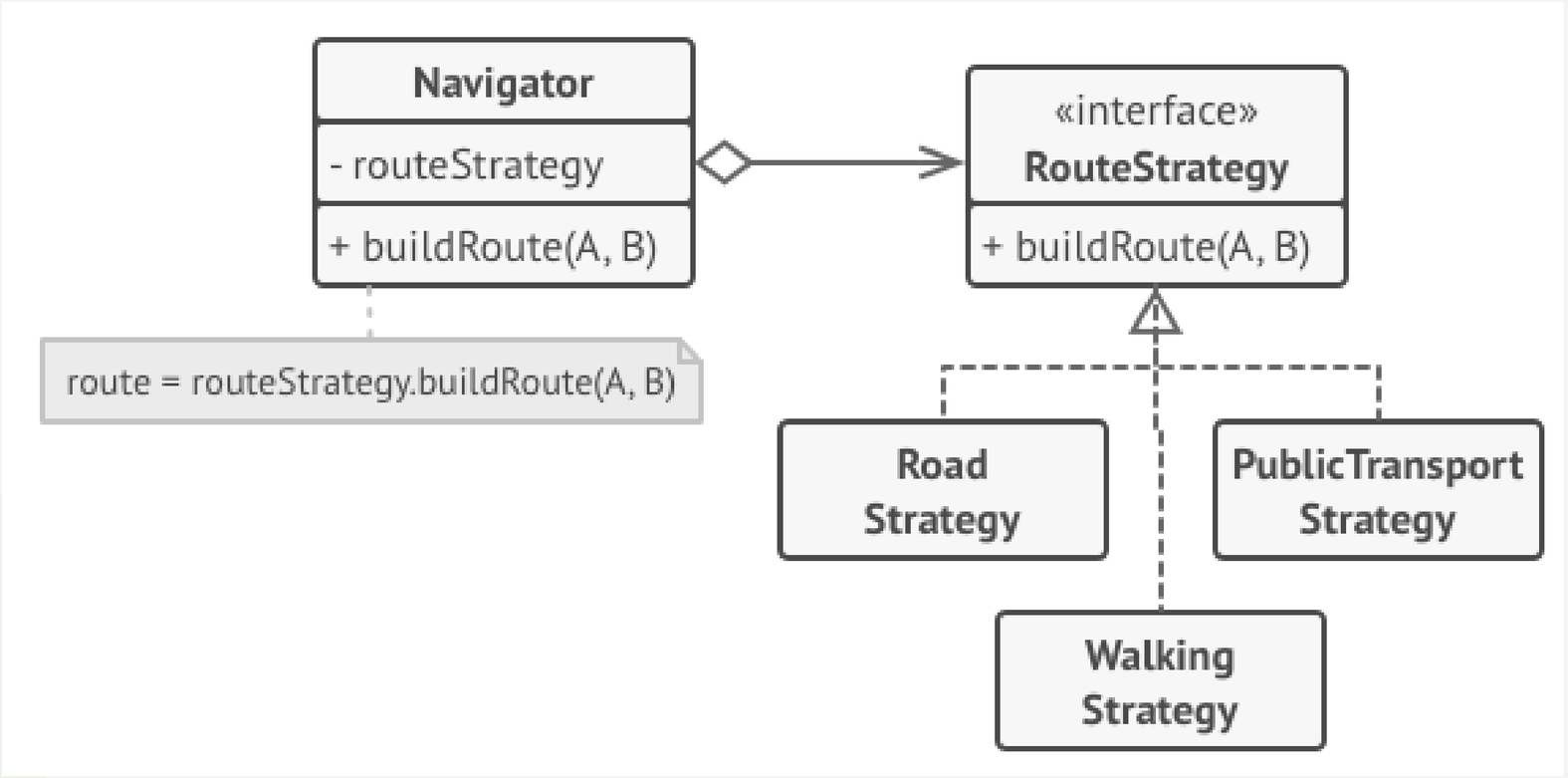
- O padrão **Strategy** sugere que pegue uma classe que faz algo específico de diversas maneiras diferentes e extraia todos esses algoritmos para classes separadas chamadas estratégias.
- A classe original, chamada contexto, deve ter um campo para armazenar uma referência para uma dessas estratégias. O contexto delega o trabalho para um objeto estratégia ao invés de executá-lo por conta própria.

Solução

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

Solução

- Desta forma o contexto torna-se independente das estratégias concretas, então pode adicionar novos algoritmos ou modificar os existentes sem modificar o código do contexto ou outras estratégias.

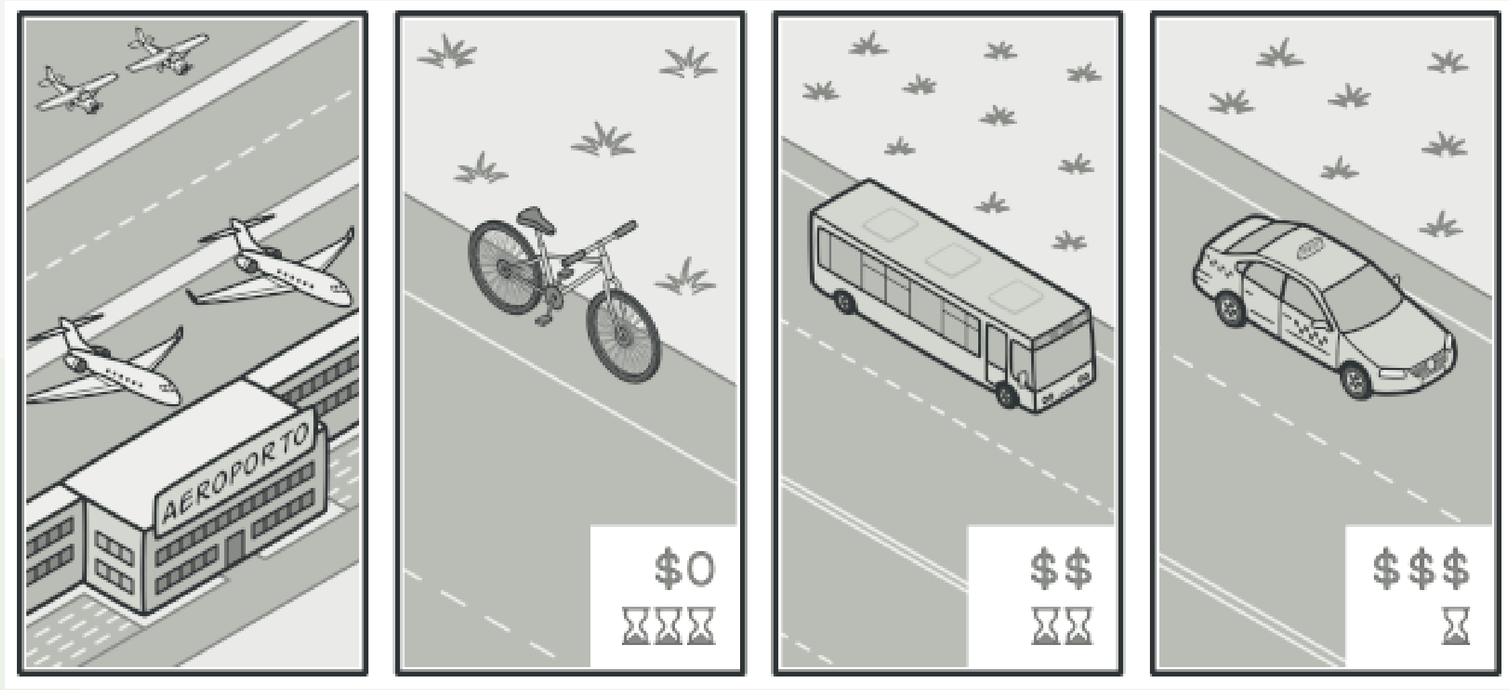


Solução

- Na nossa aplicação de navegação, cada algoritmo de rota pode ser extraído para a 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 rota pode construir uma rota diferente, a classe navegadora principal não se importa qual o algoritmo que está selecionado uma vez que seu trabalho primário é fazer o *render* de um conjunto de pontos num mapa. A classe tem um método para trocar a estratégia ativa de rotas, então os seus clientes, bem como os botões na user interface, podem substituir o comportamento de rotas selecionado por um outro.

Analogia

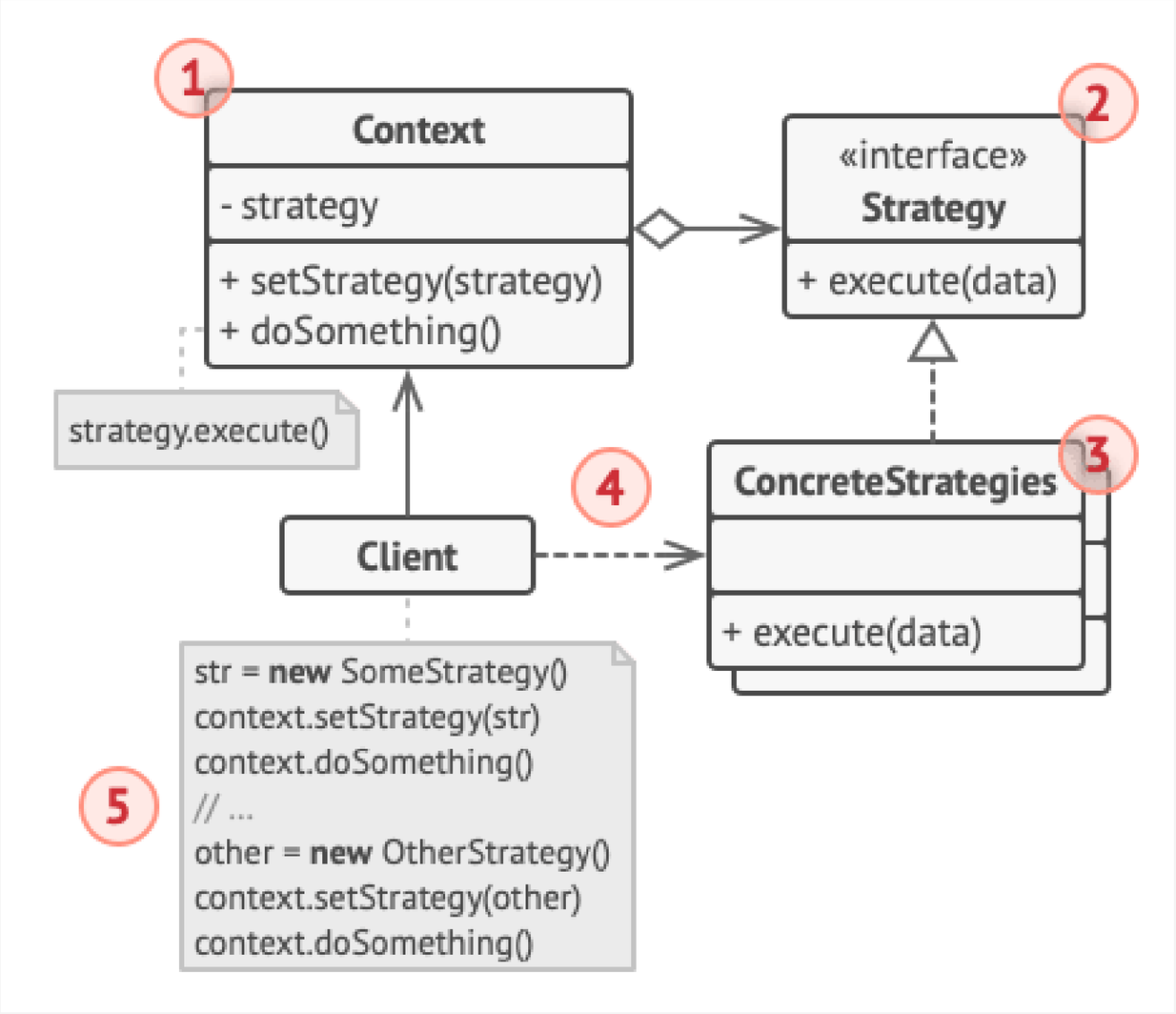
- Imagine que tem que chegar ao aeroporto. Pode apanhar um autocarro, pedir um táxi, ou pegar na sua bicicleta. Estas são as suas estratégias de transporte. Pode escolher uma das estratégias dependendo de fatores como orçamento ou restrições de tempo.



Estrutura

1. O **Contexto** mantém uma referência para uma das estratégias concretas e comunica com esse objeto através da interface da estratégia.

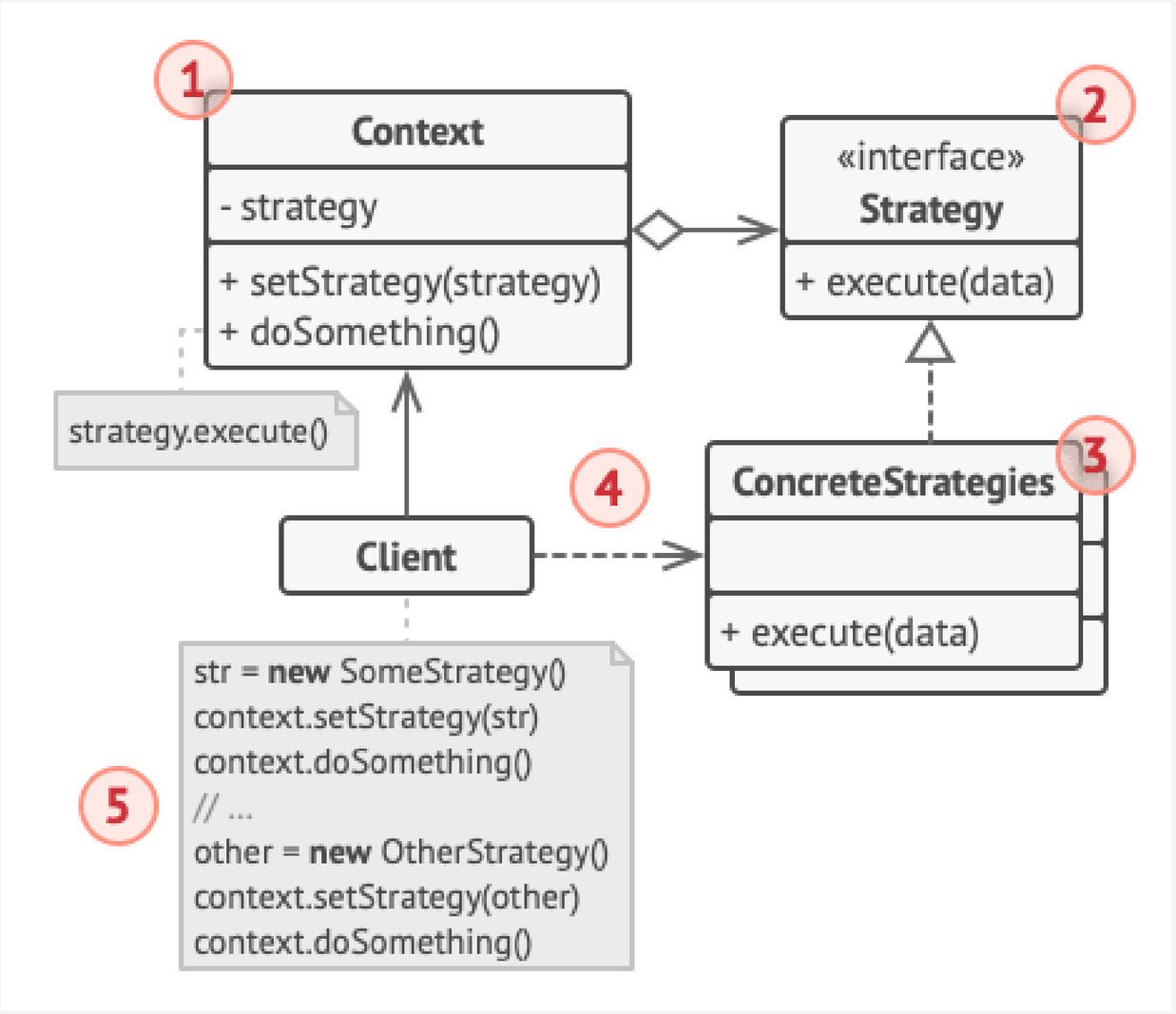
2. A interface **Estratégia** é comum a todas as estratégias concretas. Declara um método que o contexto usa para executar uma estratégia.



Estrutura

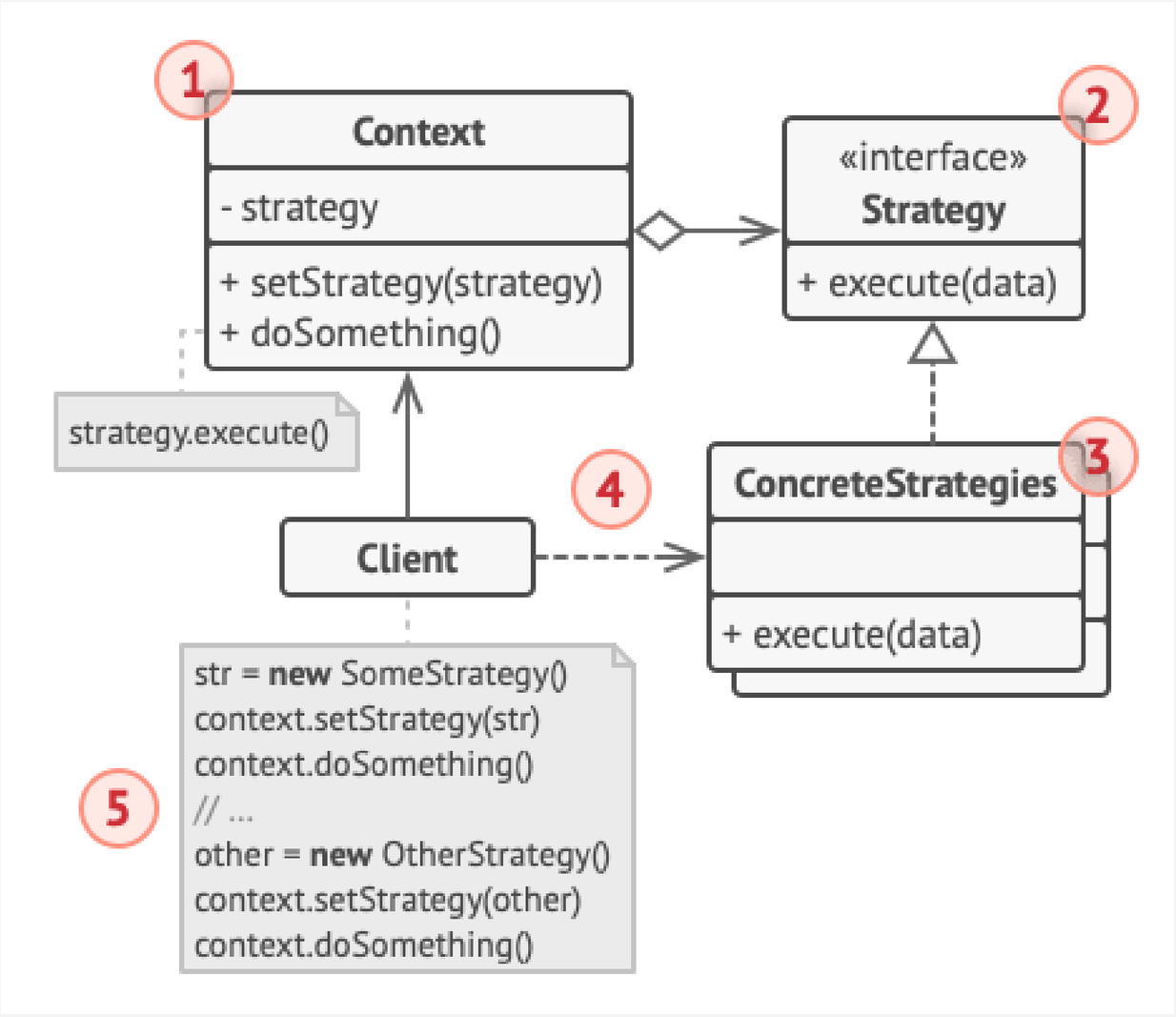
3. **Estratégias Concretas** implementam diferentes variações de um algoritmo que o contexto usa.

4. O contexto chama o método de execução no objeto estratégia ligado cada vez que ele precisa de executar um algoritmo. O contexto não sabe com que tipo de estratégia está a trabalhar ou como o algoritmo é executado.



Estrutura

5. O **Cliente** cria um objeto estratégia específico e passa como argumento para o contexto. O contexto expõe um setter que permite o cliente mudar a estratégia associada com contexto durante a execução.



Aplicabilidade

- Utilize o padrão Strategy quando 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 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.

Aplicabilidade

- **Utilize o Strategy quando tem muitas classes parecidas que somente diferem na forma de executar algum comportamento.**
  - O padrão Strategy permite que extraia o comportamento variante para uma hierarquia de classe separada e combine as classes originais numa só, reduzindo código duplicado.

Aplicabilidade

- 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 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.

Aplicabilidade

- **Utilize o padrão quando a classe tem um operador condicional muito grande que troca entre diferentes variantes do mesmo algoritmo.**
  - O padrão Strategy permite que 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 as suas próprias classes. Elas devem todas implementar a interface estratégia.

Como Implementar

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 aceder aos dados.
5. Os Clientes do contexto devem associá-lo com uma estratégia apropriada que coincide com a maneira que esperam que o contexto atue no seu trabalho primário.

Prós

- Pode trocar algoritmos usados dentro de um objeto durante a execução.
- Pode isolar os detalhes de implementação de um algoritmo do código que o usa.
- Pode substituir a herança por composição.
- Princípio aberto/fechado. Pode introduzir novas estratégias sem mudar o contexto.

Contras

- Se só tem um par de algoritmos e eles raramente mudam, não há motivo 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 anónimas. Então pode usar essas funções exatamente como se estivesse usando objetos estratégia, mas sem complicar o código com classes e interfaces adicionais.

Relações c/ 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, todos resolvem problemas diferentes. Um padrão não é apenas uma receita para estruturar o código de uma maneira específica. Ele também pode comunicar a outros programadores o problema que o padrão resolve.

Relações c/ Outros Padrões

- O Command e o Strategy podem ser parecidos porque pode usar ambos para parametrizar um objeto com alguma ação. Contudo, eles têm propósitos bem diferentes.
  - Pode usar o **Command** para converter qualquer operação num objeto. Os parâmetros da operação transformam-se em campos daquele objeto. A conversão permite que atrase a execução de uma operação, transforme-a numa 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 troque esses algoritmos dentro de uma única classe contexto.

Relações c/ Outros Padrões

- O Decorator permite mudar a pele de um objeto, enquanto o Strategy permite mudar suas entranhas.
- O Template Method é baseado em herança: ele permite que altere partes de um algoritmo ao estender essas partes em subclasses. O Strategy é baseado em composição: 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 troque os comportamentos durante a execução.

Relações c/ Outros Padrões

- 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 com que esses objetos sejam 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.

#R4E
Software Developer

Design Patterns

Strategy