Padrões de Projeto: Composite

Jonathan Arantes

17 de outubro de 2017

Instituto Federal de Minas Gerais - Campus Formiga

Descrição

Descrição

O padrão bridge é utilizado na engenharia de software onde é necessário "desacoplar uma abstração de sua implementação de forma que estas possam variar de forma independente", introduzido pela gangue dos quatro (Gang of Four). O bridge usa encapsulamento, agregação e pode utilizar herança para separar as responsabilidades em diferentes classes.

Descrição

Quando uma classe varia, as características da programação orientada à objetos podem ser bastante úteis pois mudanças no código podem ser feitas com o mínimo de conhecimento necessário sobre o programa. O padrão bridge é útil quando ambas a classe e seu uso variam. A classe em si pode ser pensada como uma abstração e o que ela é capaz de fazer como implementação. O padrão bridge pode também ser pensado como uma segunda camada de abstração.

Curiosidades

Quando há apenas uma implementação fixa, este padrão é conhecido como idioma Pimpl no mundo da linguagem C++.

O padrão brige é frequentimente confundido com o padrão adapter. Porém, o padrão bridge é frequentemente implementado utilizando o objeto do padrão adapter (exemplo de código abaixo).

Variação: a implementação pode ser ainda mais desacoplada se remover a referência da presença da implementação para o ponto de onde a abstração é utilizada.

Visão Geral

O padrão de projeto bridge é um dos 23 padrões bem conhecidos criados pela gangue dos quatro que descreve como solucionar um problema de design recorrente para flexibilidade de design e reuso de software orientado a objetos, este é, objetos que são fáceis de implementar, mudar, testar e reusar.

Quais problemas o Bridge resolve?

Uma abstração onde sua implementação deve ser definida e extendida independentemente uma da outra.

Uma ligação em tempo de compilação entre a abstração e sua implementação deve ser evitada para que uma implementação seja selecionada em tempo de execução. Quando utilizando subclasses, diferentes subclasses implementam uma classe abstrata de formas diferentes. Mas uma implementação ligada à abstração no tempo de compilação não pode ser mudada no tempo de execução.

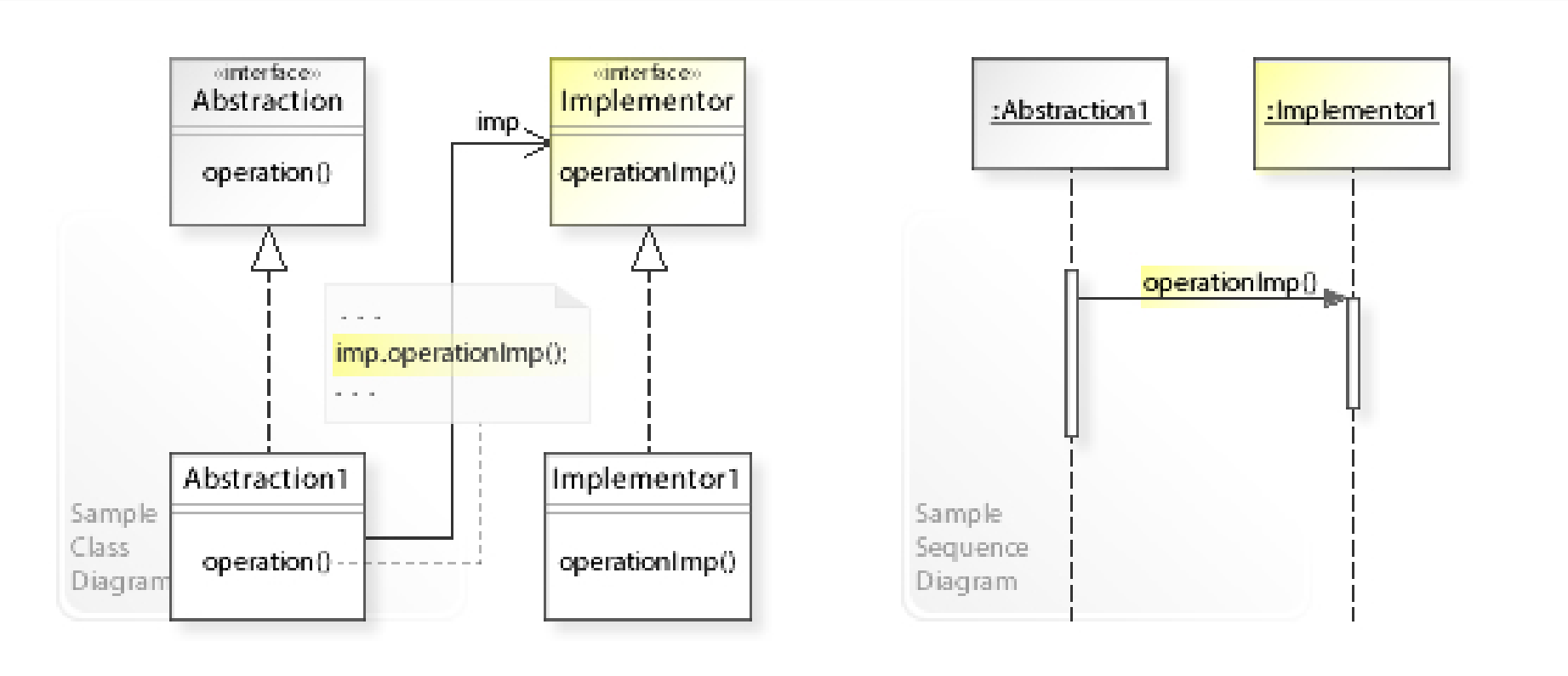
Quais soluções o padrão Bridge descreve?

Separar uma abstração de sua implementação colocando-os em classes com hierarquias diferentes.

Implementar a Abstração nos termos (delegando para) do objeto implementado. Isto habilita a configurar objetos de Abstração e Implementação em tempo de execução.

Diagramas de Sequencia e Classe UML

Veja os diagramas de sequência e UML abaixo:



Diagramas de Sequencia e Classe UML

No diagrama de classe UML acima, uma abstração (Abstraction) não é implementada como normalmente em uma hierarquia de herança. Em vez disso, há uma hierarquia para uma abstração (Abstraction) e uma hierarquia separada para sua implementação (Implementor), o que faz destas duas independentes uma da outra. A interface Abstraction (operation()) é implementada pelos termos de (delegado para) a interface Implementor(imp.operationImp()).

Diagramas de Sequencia e Classe UML

O diagrama de sequência UML mostra as interações em tempo de execução: o objeto Abstraction1 delega a implementação para o objeto Implementor1 (chamando o operationImp() em Implementor1), o que realiza a operação e retorna para Abstraction1.

Exemplo

Exemplo

O código a seguir (feito em Java SE 6) ilustra a implementação de uma 'forma' para desenho.

```java
// Implementador

interface DesenhoAPI {
    public void desenharCirculo(final double x, final double y, final double radius);
}
```

```
6   // Implementador Concreto 1
7
8   class DesenhoAPI1 implements DesenhoAPI {
9       public void desenharCirculo(final double x, final double y, final double radius) {
10          System.out.printf("API1.circulo em %f:%f - radius %f\n", x, y, radius);
11      }
12  }
```

```
13  // Implementador Concreto 2
14
15  class DesenhoAPI2 implements DesenhoAPI {
16      public void desenharCirculo(final double x, final double y, final double radius) {
17          System.out.printf("API2.circulo em %f:%f - radius %f\n", x, y, radius);
18      }
19  }
```

```
20  // Abstracao
21
22  abstract class Forma {
23      protected DesenhoAPI desenhoAPI;
24
25      protected Forma(final DesenhoAPI desenhoAPI) {
26          this.desenhoAPI = desenhoAPI;
27      }
28
29      public abstract void desenhar(); // baixo nivel
30      public abstract void alterarTamanhoPorPorcentagem(final double pct); // alto nivel
31  }
```

```
32  // Abstracao Refinada
33
34  class FormaCirculo extends Forma() {
35      private double x, y, radius;
36
37      public FormaCirculo(final double x, final double y, final double radius, final Desen
38          super(desenhoAPI);
39          this.x = x;
40          this.y = y;
41          this.radius = radius;
42      }
```

```
43  // implementacao de baixo nivel (especifico da implementacao)
44      public void desenhar() {
45          desenhoAPI.desenharCirculo(x, y, radius);
46      }
47
48      // implementacao de alto nivel (especifico da abstracao)
49      public void alterarTamanhoPorPorcentagem(final double pct) {
50          radius *= (1.0 + pct/100.0);
51      }
52  }
```

```
53  // Cliente
54
55  class PadraoBridge {
56      public static void main(final String[] args) {
57          Forma[] formas = new Forma[] {
58              new FormaCirculo(1, 2, 3, new DesenhoAPI1()),
59              new FormaCirculo(5, 7, 11, new DesenhoAPI2())
60          };
61
62          for (Forma forma: formas) {
63              forma.alterarTamanhoPorPorcentagem(2.5);
64              forma.desenhar();
65          }
66      }
67  }
```

Conclusão

Conclusão

Tendo em vista o exemplo anterior, temos um problema facilmente tratado pelo padrão Bridge resolvido.

A interface DesenhoAPI é o topo da hierarquia da parte de implementação do programa, enquanto que a classe abstrata Forma é o topo da parte abstrata, perceba que uma composição de baixo nível do sistema utiliza a interface DesenhoAPI para implementar a função de desenho, enquanto que a composição de alto nível permite alterar o tamanho do desenho ainda na parte abstrata, separando esta da implementação do resto do programa.

Conclusão

Separando a implementação de alto nível podemos evitar a repetição de código, enquanto que separado a implementação de baixo nível nos permite reaproveitar partes de código enquanto que mantém o sistema desacoplado, estas funcionalidades simples do padrão de projeto facilitam o trabalho do programador que o desenvolve, enquanto que ajuda outros desenvolvedores a entenderem partes do projeto sem ter conhecimento de todo o funcionamento do programa.

Referências

- Gamma, E, Helm, R, Johnson, R, Vlissides, J: Design Patterns, page 151. Addison-Wesley, 1995

- Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides (1994). Design Patterns: Elements of Reusable Object-Oriented Software. Addison Wesley. pp. 151ff. ISBN 0-201-63361-2.

- "The Bridge design pattern - Problem, Solution, and Applicability". w3sDesign.com. Retrieved 2017-08-12.

- "The Bridge design pattern - Structure and Collaboration". w3sDesign.com. Retrieved 2017-08-12.

- Bridge Pattern - Wikipedia