Arquitetura de Software

Padrões de Projetos – Padrões Estruturais

Nairon Neri Silva

Sumário

Padrões de Projeto Estruturais

1. *Adapter*
2. ***Bridge***
3. ***Composite***
4. *Decorator*
5. *Façade*
6. *Flyweight*
7. *Proxy*

Bridge

Também conhecido como ***Handle/Body***

Intenção

- Desacoplar uma abstração da sua implementação, de modo que as duas possam variar independentemente

Motivação

- Quando uma abstração pode ter uma entre várias implementações possíveis, a maneira de acomodá-las é usando herança

- Uma classe abstrata define a interface para uma abstração, e as subclasses concretas a implementam de formas diferentes

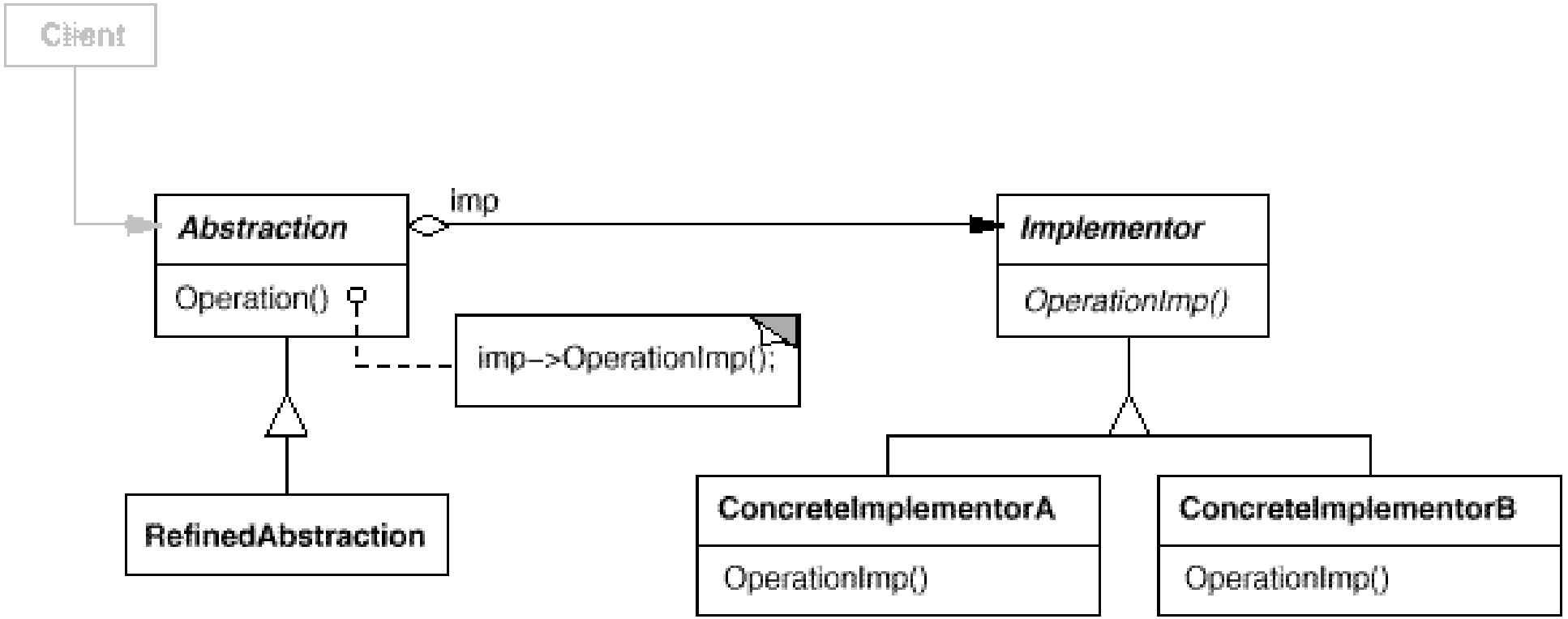
- Porém, a herança liga uma implementação a uma abstração permanentemente e dificulta modificar, aumentar e reutilizar abstrações e implementações independentemente

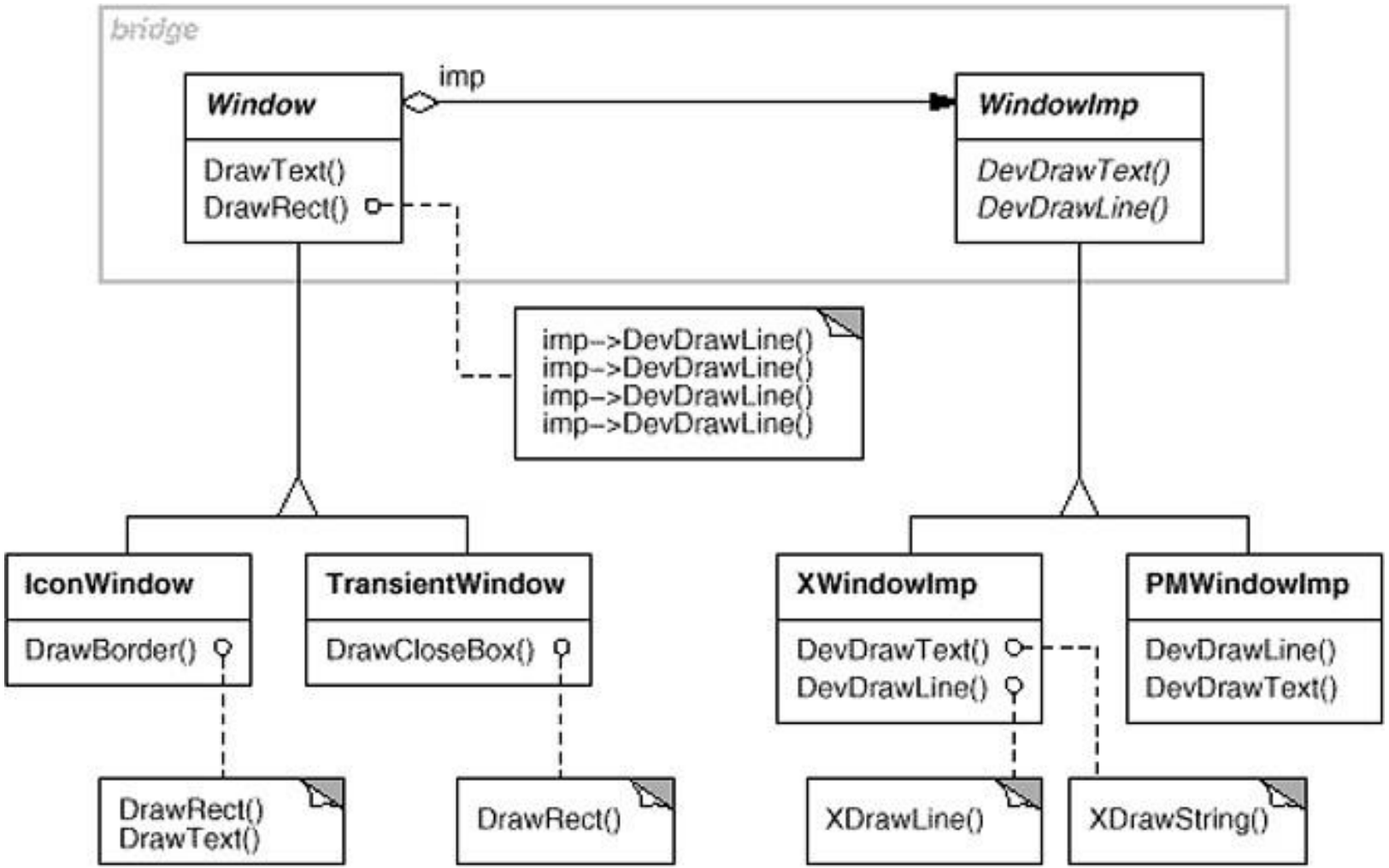
Motivação

- Considere a implementação de uma ”Janela” em um toolkit para construir interfaces de usuário
  - Essa abstração deveria nos possibilitar escrever aplicações para diferentes ambientes operacionais
  - Usando herança, podemos definir a abstração ***Window***
  - E uma subclasse para cada ambiente operacional
  - Teríamos alguns problemas:
    - É inconveniente estender a abstração para cobrir todos as variações da abstração e ambientes operacionais possíveis
    - Torna o código dependente da plataforma – torna difícil portar o código do cliente para outras plataformas

Aplicabilidade

1. Desejamos evitar um vínculo permanente entre uma abstração e sua implementação
2. Tanto abstrações como suas implementações tiverem de ser extensíveis por meio de subclasses
3. Mudanças na implementação de uma abstração não deveriam ter impacto sobre os clientes
4. Tivermos uma proliferação de classes
5. Desejamos compartilhar uma implementação entre múltiplos objetos e ocultar isso do cliente


Client
Abstraction
Operation()
imp
Implementor
OperationImp()
imp->OperationImp();
RefinedAbstraction
ConcreteImplementorA
OperationImp()
ConcreteImplementorB
OperationImp()


bridge
Window
DrawText()
DrawRect()
imp
WindowImp
DevDrawText()
DevDrawLine()
imp->DevDrawLine()
imp->DevDrawLine()
imp->DevDrawLine()
imp->DevDrawLine()
IconWindow
DrawBorder()
TransientWindow
DrawCloseBox()
XWindowImp
DevDrawText()
DevDrawLine()
PMWindowImp
DevDrawLine()
DevDrawText()
DrawRect()
DrawText()
DrawRect()
XDrawLine()
XDrawString()

Exemplo/Participantes

1. ***Abstraction (Window)***
   - Define a interface da abstração
   - Mantém uma referência para um objeto ***Implementor***
2. ***RefinedAbstraction (IconWindow)***
   - Estende a interface definida por ***Abstraction***
3. ***Implementor (WindowImp)***
   - Define a interface para as classes de implementação
4. ***ConcreteImplementor (XWindowImp, PMWindowImp)***
   - Implementa a interface de ***Implementor***

Consequências

1. Desacopla a interface da implementação
2. Melhora a extensibilidade da abstração e implementação (de forma independente)
3. Oculta detalhes de implementação dos clientes
4. Tem as mesmas vantagens do Adapter
5. Aumenta a complexidade quando aplicado de forma incorreta

Exemplo Prático

1. Em uma aplicação será possível desenhar diversos tipos de formas (quadrado, triângulo, círculo, linha....) e cada forma pode ter uma determinada cor.
2. Inicialmente, seriam criadas as classes no padrão: quadradoVermelho, quadradoAzul, quadradoVerde, trianguloVermelho, trianguloAmarelo, trianguloVerde...
3. Porém, desta forma dificultará tanto o acréscimo de novas formas quanto o acréscimo de cores...
4. Como resolver este problema?

Saiba mais...

- https://www.youtube.com/watch?v=-gsuMWLxAko
- https://refactoring.guru/pt-br/design-patterns/bridge
- https://www.baeldung.com/java-bridge-pattern

Acesse os endereços e veja mais detalhes sobre o padrão Bridge

Composite

Intenção

- Compor objetos em estruturas de árvore para representar hierarquias partes-todo

- Permite aos clientes tratarem de maneira uniforme objetos individuais e composições de objetos

Motivação

- Aplicações gráficas, tais como editores de desenhos permitem aos usuários construir diagramas complexos a partir de componentes simples
  - O usuário pode agrupar componentes para formar componentes maiores, os quais por sua vez, podem ser agrupados para formar componentes ainda maiores

- Uma implementação simples poderia definir classes representando os elementos básicos e classes que funcionariam como recipientes (*containers*)

Motivação

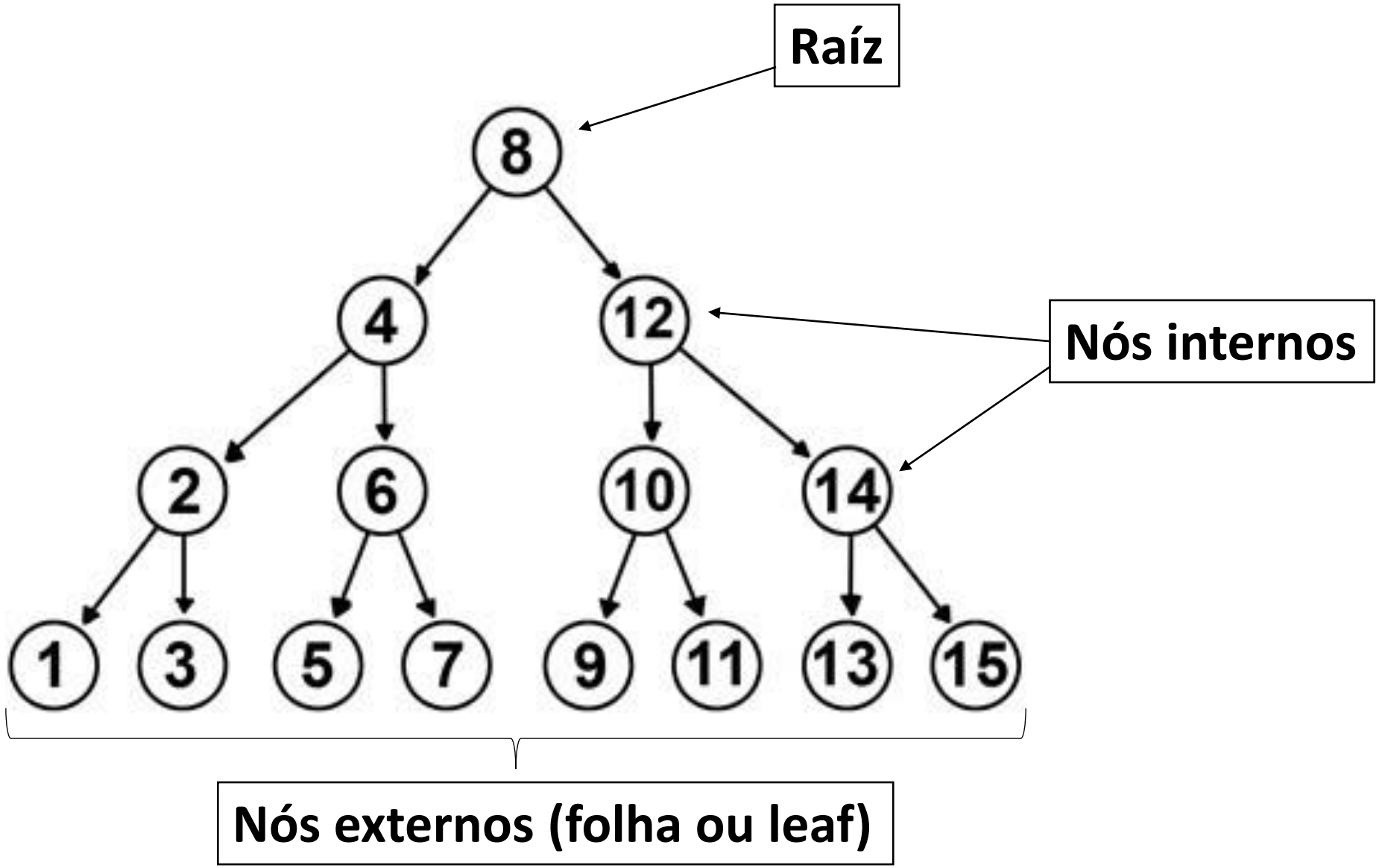
- Problema dessa abordagem:
  - O código que usa essas classes deve tratar objetos primitivos e objetos ”recipientes” de forma diferente

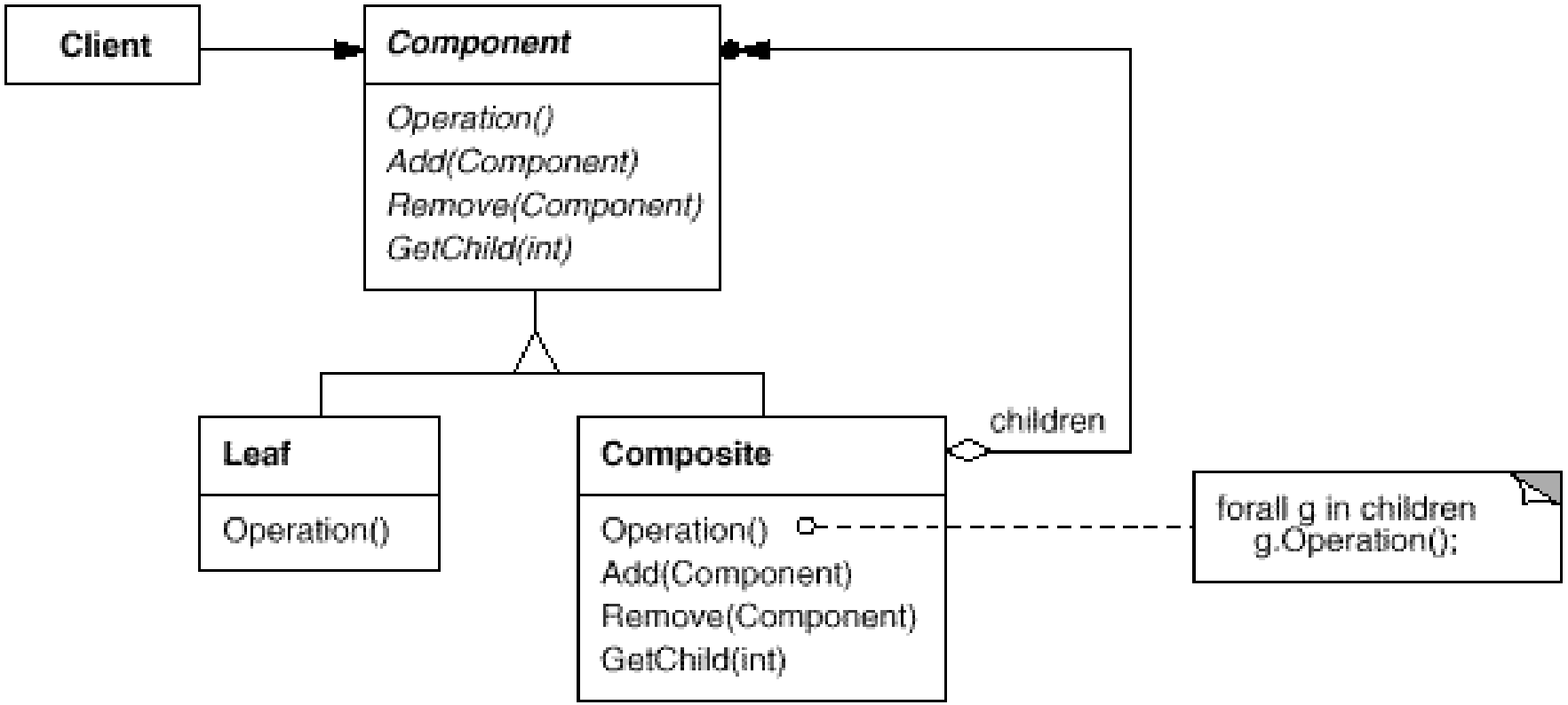
- O padrão ***Composite*** descreve como usar a composição recursiva de maneira que os clientes não tenham que fazer essa distinção

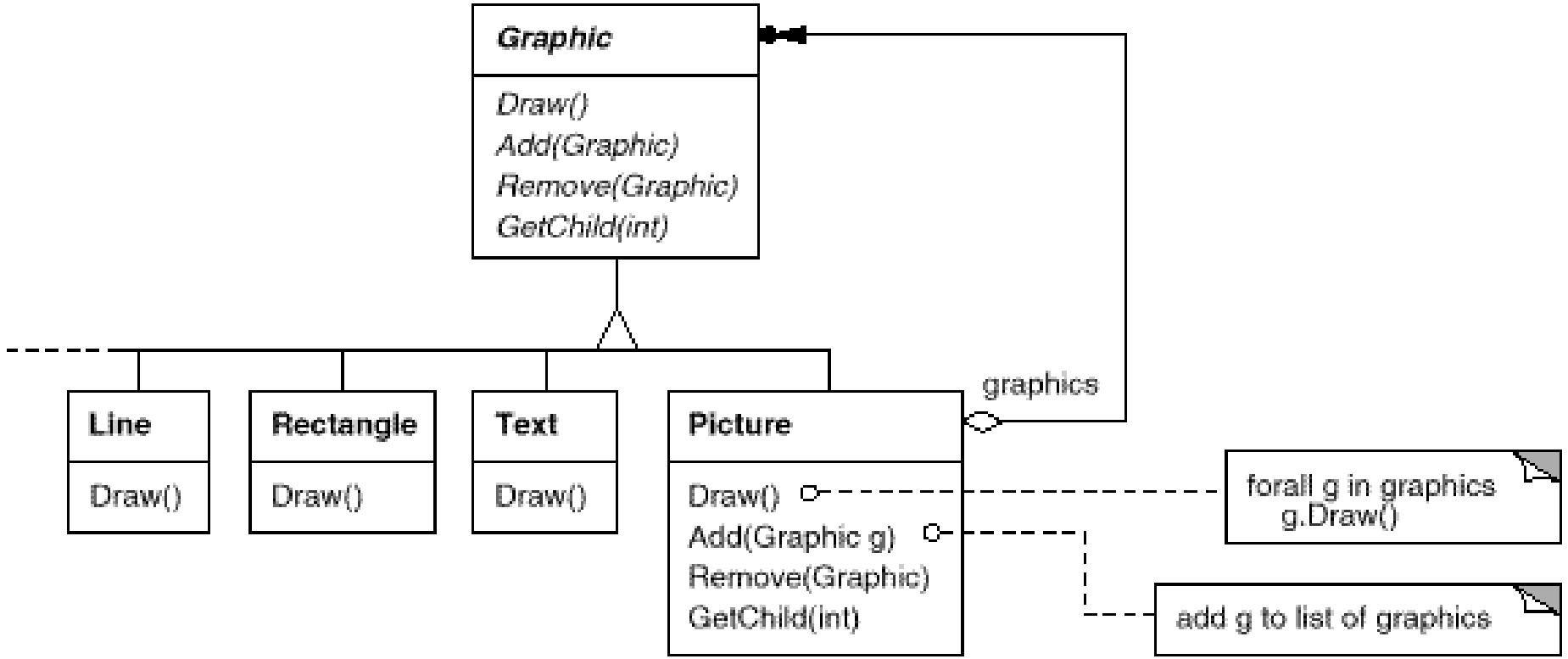
Aplicabilidade

1. Quando queremos representar hierarquia de objetos

2. Quando queremos que os clientes sejam capazes de ignorar a diferença entre composições de objetos e objetos individuais – uniformemente


Raíz
8
4
12
Nós internos
2
6
10
14
1
3
5
7
9
11
13
15
Nós externos (folha ou leaf)


Client
Component
Operation()
Add(Component)
Remove(Component)
GetChild(int)
Leaf
Operation()
Composite
Operation()
Add(Component)
Remove(Component)
GetChild(int)
children
forall g in children
g.Operation();


Graphic
Draw()
Add(Graphic)
Remove(Graphic)
GetChild(int)
graphics
Line
Draw()
Rectangle
Draw()
Text
Draw()
Picture
Draw()
Add(Graphic g)
Remove(Graphic)
GetChild(int)
forall g in graphics
g.Draw()
add g to list of graphics

Exemplo/Participantes

1. ***Component (Graphic)***
   - Declara a interface para os objetos na composição
   - Implementa o comportamento comum
   - Declara uma interface para acessar os seus ”filhos”
   - (opcional) declara uma interface para acessar o seu ”pai”
2. ***Leaf (Rectangle, Line, Text, etc.)***
   - Representa os objetos-folha na composição
   - Define o comportamento para objetos básicos na composição
3. ***Composite (Picture)***
   - Define o comportamento para objetos que possuem ”filhos”
4. ***Client***
   - Manipula objetos na composição através de ***Component***

Consequências

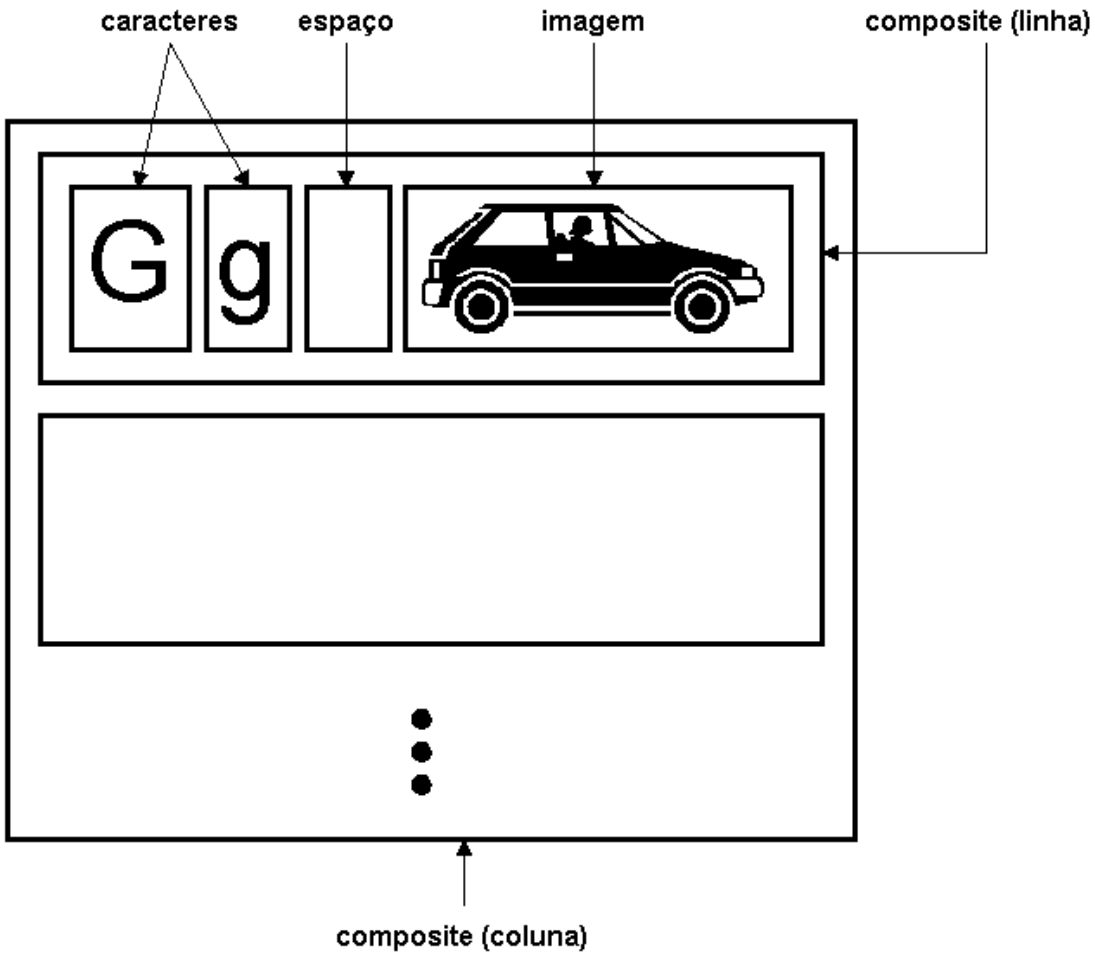
1. Define hierarquias de classe que consistem de objetos primitivos e objetos compostos

2. Torna o cliente mais simples

3. Torna mais fácil de acrescentar novas espécies de componentes

4. Pode tornar o projeto excessivamente genérico
   - E quando quisermos um composto que tenham somente certos componentes?

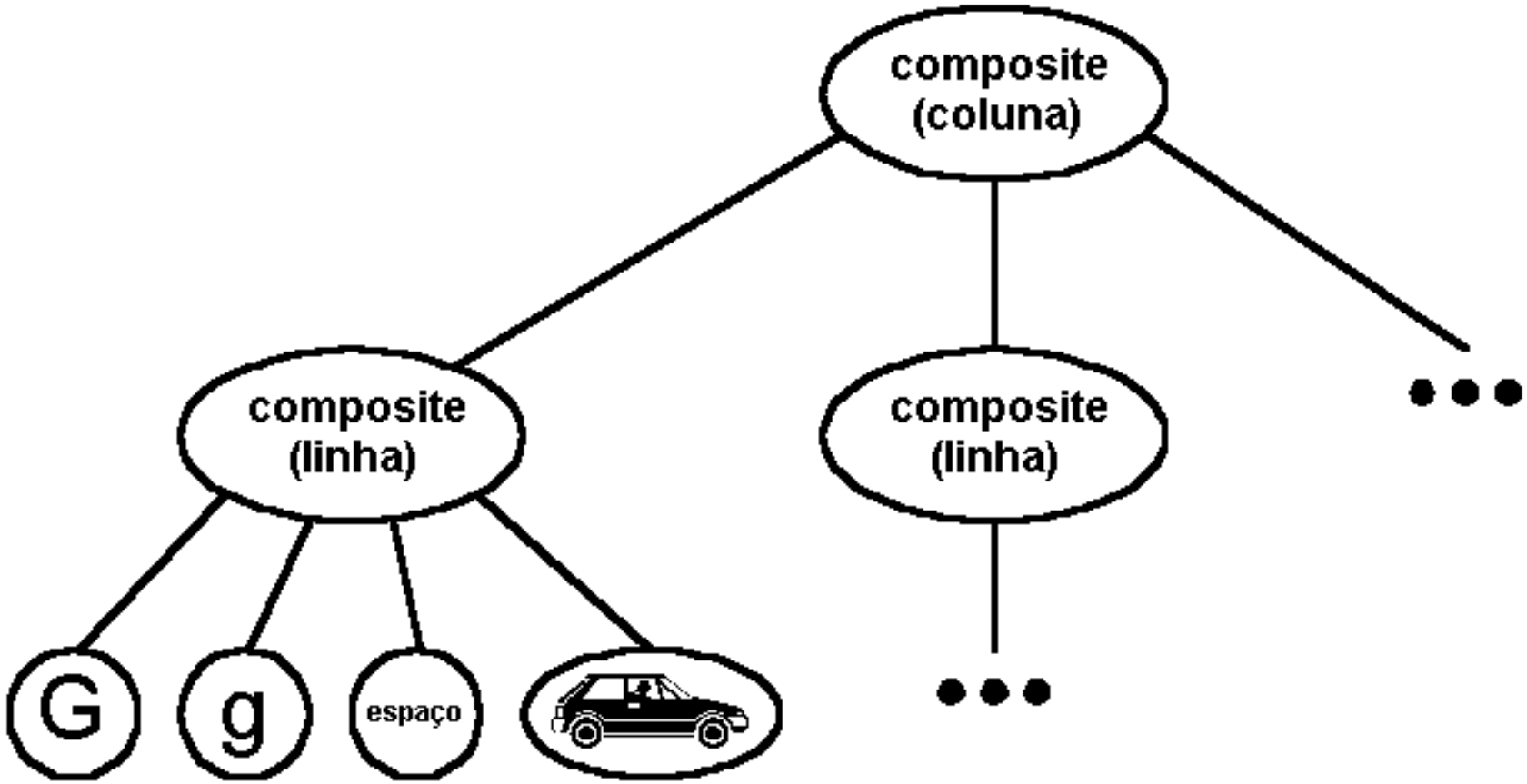
Exemplo


caracteres
espaço
imagem
composite (linha)
G
g
composite (coluna)

Exemplo

- Podemos representar essa estrutura física usando um objeto para cada elemento. Isso inclui elementos visíveis e elementos estruturais (linhas, colunas)
- A estrutura de objetos seria como abaixo
  - Na prática, talvez um objeto não fosse usado para cada caractere por razões de eficiência

Exemplo


composite
(coluna)
composite
(linha)
composite
(linha)
G
g
espaço

Exemplo

- Podemos agora tratar texto e gráficos uniformemente
- Podemos ainda tratar elementos simples e compostos uniformemente
- Teremos que ter uma classe para cada tipo de objeto e essas classes terão que ter a mesma interface (para ter uniformidade de tratamento)
  - Uma forma de ter interfaces compatíveis é de usar herança
- Definimos uma interface "ElementoDeDocumentoIF" e uma classe abstrata "ElementoDeDocumento" para todos os elementos que aparecem na estrutura de objetos
- Suas subclasses definem elementos gráficos primitivos (caracteres e gráficos) e elementos estruturais (linhas, colunas, frames, páginas, documentos, ...)

Exemplo Prático

- Precisamos criar um editor de formas gráficas complexas, compostas por formas mais simples e tratar as duas de maneira uniforme.

Fontes de Consulta

- https://www.youtube.com/watch?v=I0RqHDFQjVY
- https://refactoring.guru/pt-br/design-patterns/composite
- https://www.baeldung.com/java-composite-pattern

Acesse os endereços e veja mais detalhes sobre o padrão Composite