Arquitetura de Software

Padrões de Projetos – Padrões Comportamentais

Nairon Neri Silva

Sumário

Padrões de Projeto Comportamentais

1. *Chain of Responsibility*
2. *Command*
3. *Interpreter*
4. *Iterator*
5. *Mediator*
6. *Memento*
7. *Observer*
8. *State*
9. ***Strategy***
10. ***Template Method***
11. ***Visitor***

Strategy

Policy

Intenção

- Definir uma família de algoritmos, encapsular cada uma deles e torna-los intercambiáveis

- O **Strategy** permite que o algoritmo varie independentemente dos clientes que o utilizam

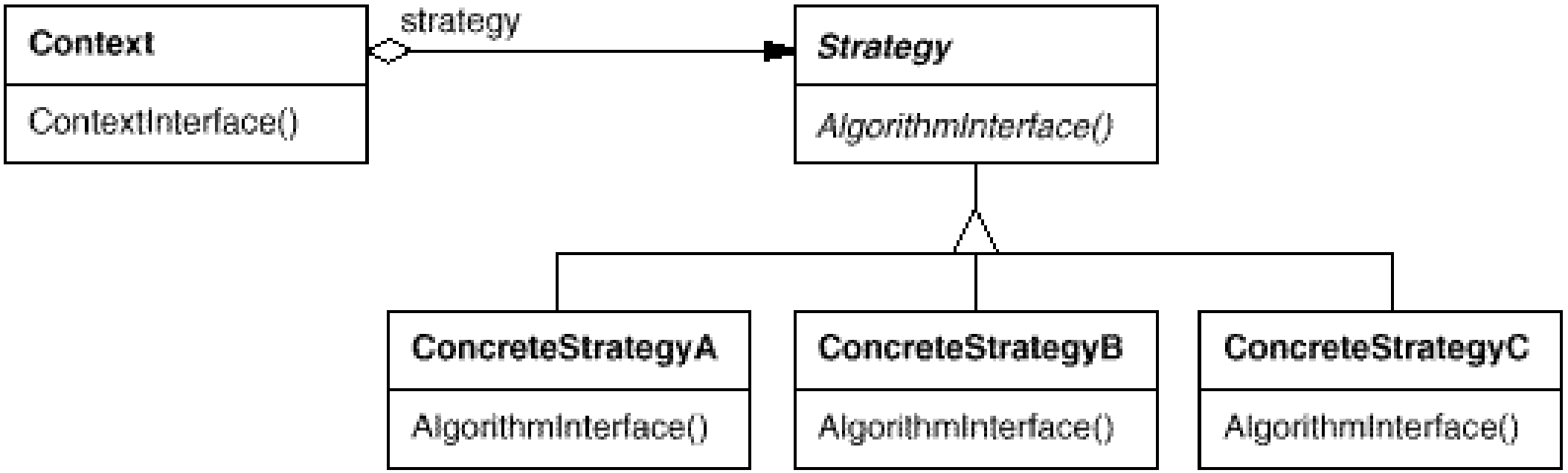
Motivação

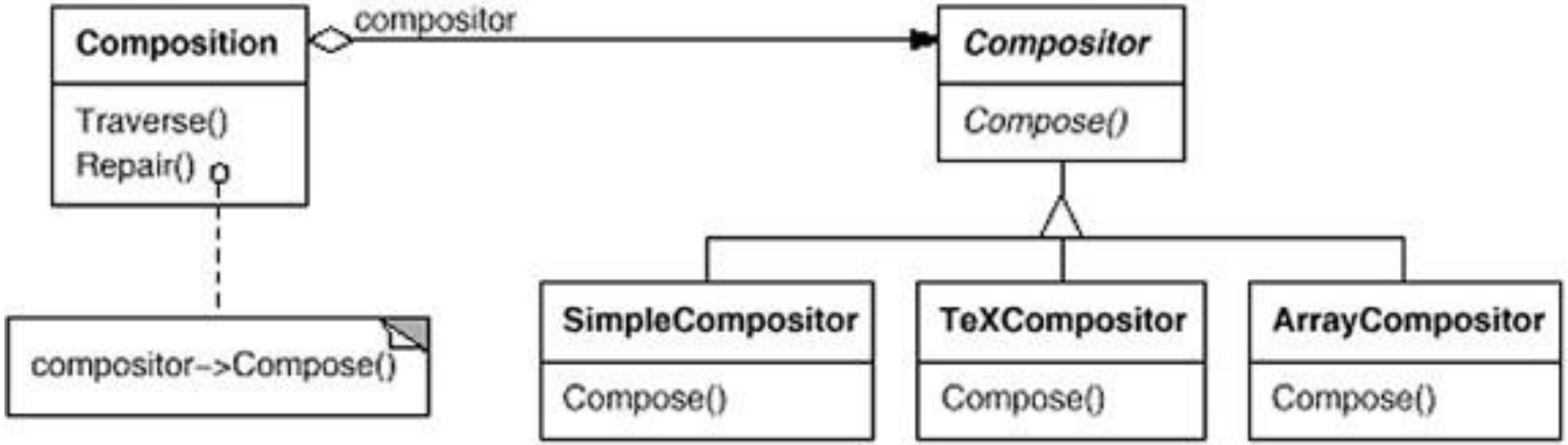
- Ex.: existem muitos algoritmos para quebrar um fluxo (*stream*) de texto em linhas
  - Codificar de maneira fixa e rígida nas classes que o utilizam não é desejável
  - Por que não?
    - Torna as classes “clientes” mais complexas (maiores e difíceis de manter)
    - Diferentes algoritmos serão apropriados em diferentes situações
    - É difícil adicionar novos algoritmos e variar os existentes

- Podemos evitar isso definindo classes que encapsulam diferentes algoritmos – **Strategy**

Aplicabilidade

- Use esse padrão quando:
  - Muitas classes relacionadas diferem apenas no seu comportamento – o Strategy ajuda nessa configuração
  - Você necessita de variantes de um algoritmo
  - Um algoritmo usa dados que o cliente não deveria ter conhecimento – evita exposição de estruturas complexas
  - Uma classe define muitos comportamentos e aplicam muitos condicionais para seleção do adequado


Context
ContextInterface()
strategy
Strategy
AlgorithmInterface()
ConcreteStrategyA
AlgorithmInterface()
ConcreteStrategyB
AlgorithmInterface()
ConcreteStrategyC
AlgorithmInterface()


Composition
Traverse()
Repair()
compositor
Compositor
Compose()
compositor->Compose()
SimpleCompositor
Compose()
TeXCompositor
Compose()
ArrayCompositor
Compose()

Exemplo/Participantes

1. **Strategy (Compositor)**
   - Define uma interface comum para os algoritmos
2. **ConcreteStrategy (SimpleCompositor, TeXCompositor, ArrayCompositor)**
   - Implementa o algoritmo usando a interface **Strategy**
3. **Context (Composition)**
   - Mantém uma referência a um objeto **Strategy**
   - É configurado com o objeto **ConcreteStrategy**
   - Pode definir uma interface que permita ao objeto **Strategy** acessar os seus dados

Consequências

1. Agrupa famílias de algoritmos relacionados
2. Uma alternativa ao uso de subclasses
3. Estratégias eliminam comandos condicionais
4. Possibilidade de escolha de implementações
5. O cliente deve conhecer as diferentes estratégias
6. Custo da comunicação entre **Strategy** e **Context**
7. Aumento no número de objetos

Exemplo Prático

1. Neste exemplo vamos simular uma loja virtual que precisa vender para diferentes regiões (EUA e EUR).
2. Cada região tem as suas particularidades que irão afetar o preço final do produto
3. Através do uso do padrão Strategy vamos definir estratégias diferentes de acordo com cada uma das regiões.

Exemplo baseado no site: https://github.com/MichalFab/Strategy-design-pattern/tree/master/src/main/java/ShoppingStrategy

Saiba mais...

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

Acesse os endereços e veja mais detalhes sobre o padrão Strategy.

Template Method

Intenção

- Definir o esqueleto de um algoritmo em uma operação, postergando alguns passos para as subclasses

- O **Template Method** permite que subclasses redefinam certos passos de um algoritmo sem mudar a estrutura do mesmo

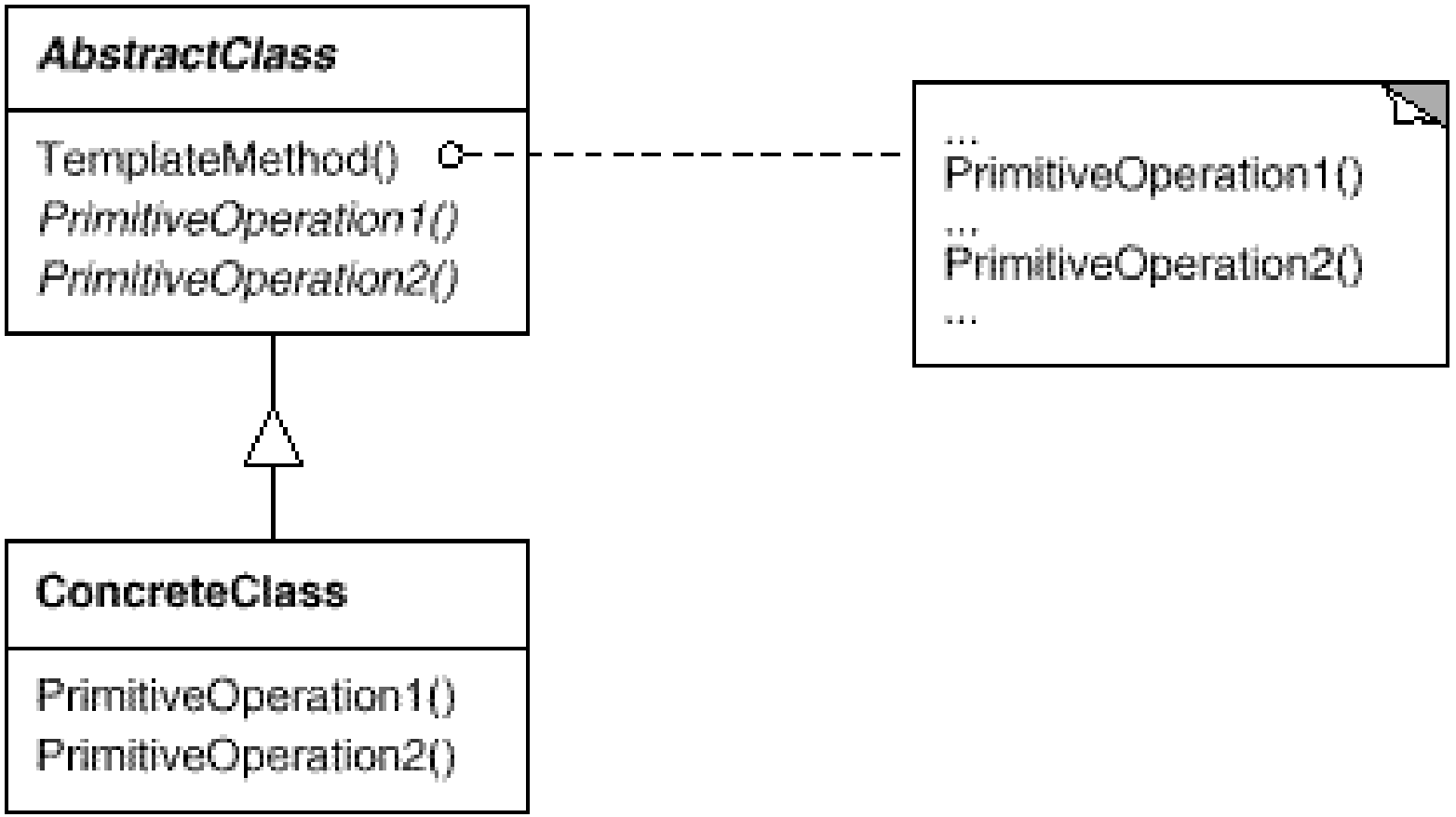
Motivação

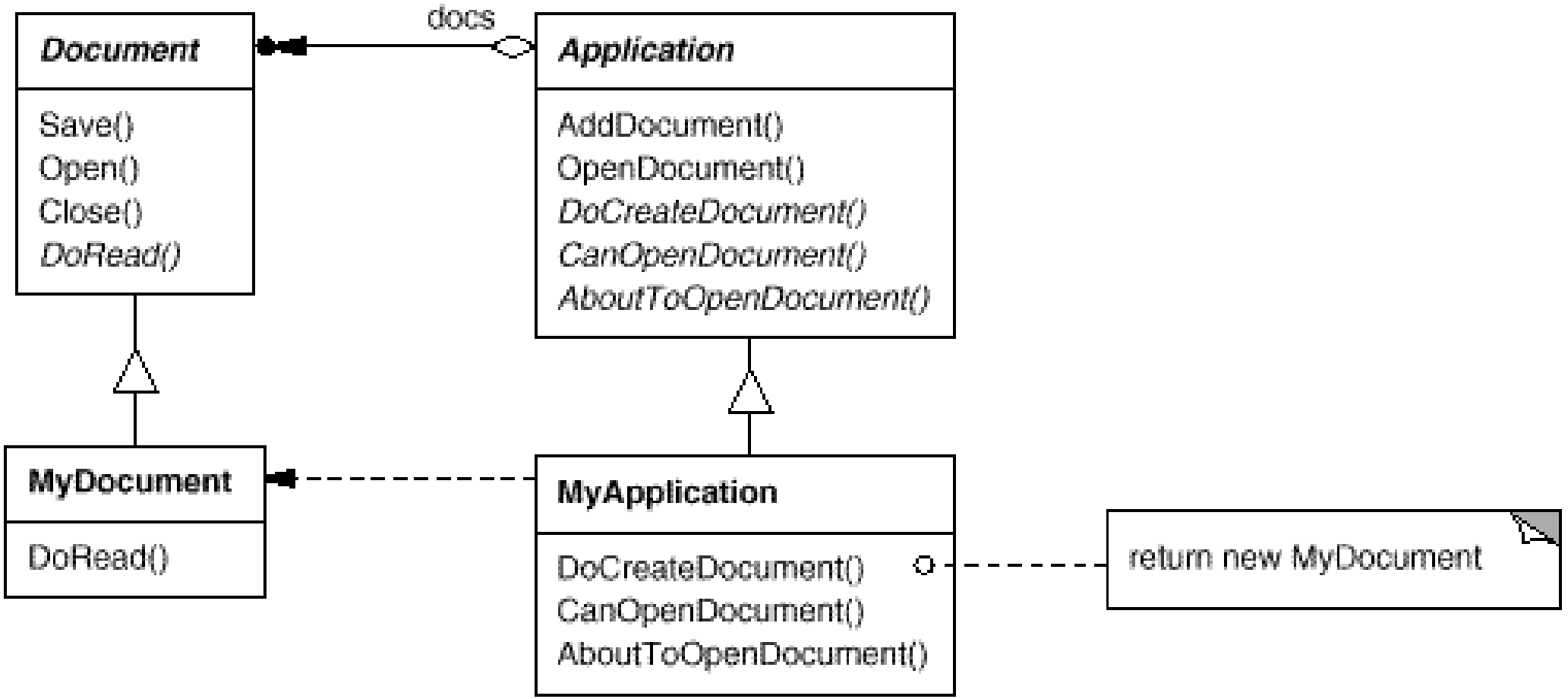
- Ex.: considere um *framework* que fornece as classes **Application** e **Document**
  - A “aplicação” é responsável por abrir “documentos”
  - Os documentos são armazenados em um arquivo – o objeto **Document** armazena as informações após a leitura
  - As aplicações segundo o framework estendem tais classes
  - A classe abstrata **Application** define um algoritmo para abrir e ler um **Document** – **OpenDocument**

- Uma operação como a **OpenDocument** é denominada de um **Template Method**

Aplicabilidade

- Use esse padrão quando:
  - Para implementar partes invariantes de um algoritmo uma vez só em uma superclasse – evitando duplicidade
  - Deve aplicar a estratégia “refatorar para generalizar”
  - Para controlar extensões de subclasses – definindo operação “gancho” em pontos específicos – permitindo extensões somente nesses pontos


AbstractClass
TemplateMethod()
PrimitiveOperation1()
PrimitiveOperation2()
...
PrimitiveOperation1()
...
PrimitiveOperation2()
...
ConcreteClass
PrimitiveOperation1()
PrimitiveOperation2()


docs
Document
Save()
Open()
Close()
DoRead()
Application
AddDocument()
OpenDocument()
DoCreateDocument()
CanOpenDocument()
AboutToOpenDocument()
MyDocument
DoRead()
MyApplication
DoCreateDocument()
CanOpenDocument()
AboutToOpenDocument()
return new MyDocument

Exemplo/Participantes

1. **AbstractClass (Application)**
   - Define operações primitivas – passos de um algoritmo
   - Implementa um método-template que define o esqueleto de um algoritmo – invocando as operações primitivas
2. **ConcreteClass (MyApplication)**
   - Implementa as operações primitivas para executarem os passos específicos do algoritmo da subclasse

Consequências

1. Constituem uma técnica fundamental para a reutilização de código – métodos gancho (*hook*)

2. Definem a uma estrutura de inversão de controle – superclasse chama métodos da subclasse

Exemplo Prático

1. Neste exemplo, o padrão Template Method define um algoritmo de trabalho com uma rede social.
2. As subclasses que correspondem a uma rede social específica, implementam essas etapas de acordo com a API fornecida pela rede social.

Exemplo baseado no site: https://refactoring.guru/pt-br/design-patterns/template-method/java/example

Saiba mais...

- https://www.youtube.com/watch?v=-nSOKE4f2gA
- https://refactoring.guru/pt-br/design-patterns/template-method

Acesse os endereços e veja mais detalhes sobre o padrão Template Method.

Visitor

Intenção

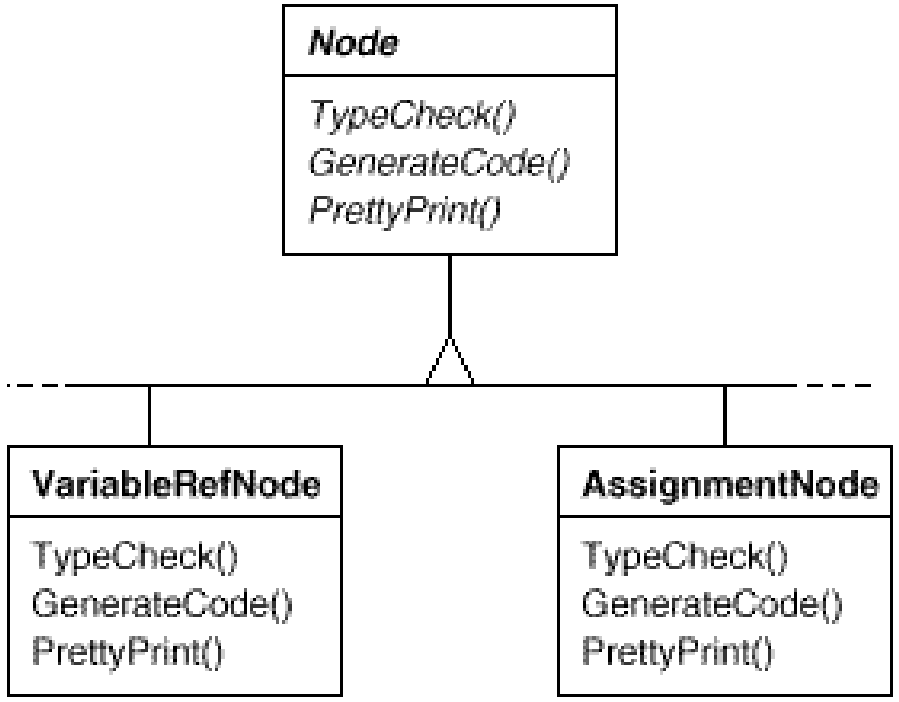
- Representar uma operação a ser executada nos elementos de uma estrutura de objetos

- O **Visitor** permite definir uma nova operação sem mudar as classes dos elementos sobre os quais opera

Motivação

- Ex.: considere um compilador que representa programas como árvores sintáticas abstratas
  - Precisará executar operações na árvore como análise semântica, geração de código, otimização de código, etc.
  - Operações necessitarão tratar os nós de forma diferente – classes para comandos de atribuição, acessos à variáveis, expressões aritméticas, etc.
  - Será necessária a definição dessa hierarquia de classes

Motivação



- Distribuir operações pelas diferentes classes nó nos leva a um sistema difícil de compreender, manter e mudar
- O fato de acrescentar uma operação poderia causar a necessidade de recompilação de toda hierarquia

Motivação

- (1) Seria melhor que cada operação adicional pudesse ser acrescentada separadamente, e

- (2) Que as classes fossem independentes das operações que se aplicam sobre elas

- Podemos ter as duas coisas
  - (i) empacotando as operações relacionadas a cada classe num objeto separado (**Visitor**), e
  - (ii) passando este objeto para os elementos da árvore sintática abstrata à medida que a mesma é percorrida

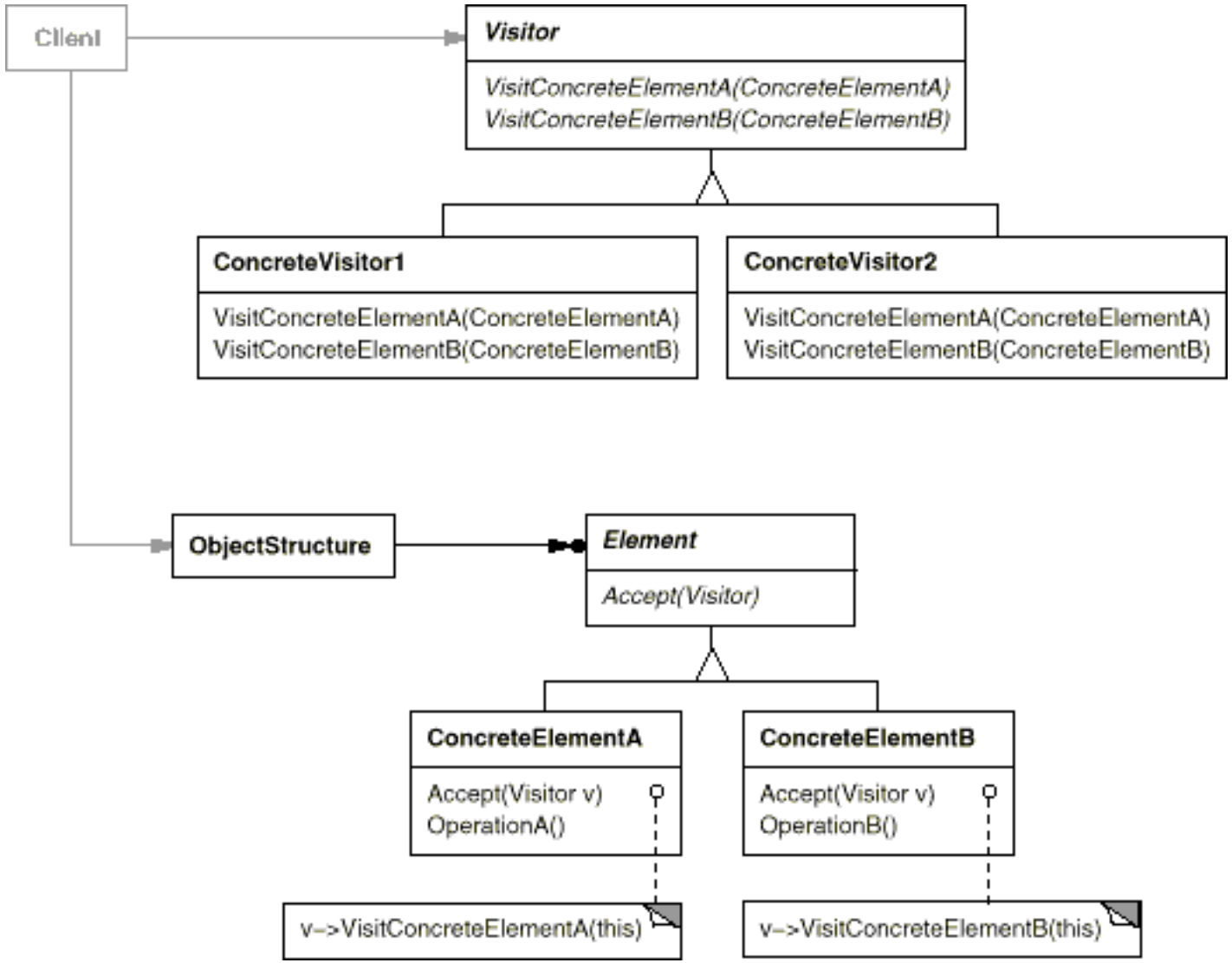
Motivação

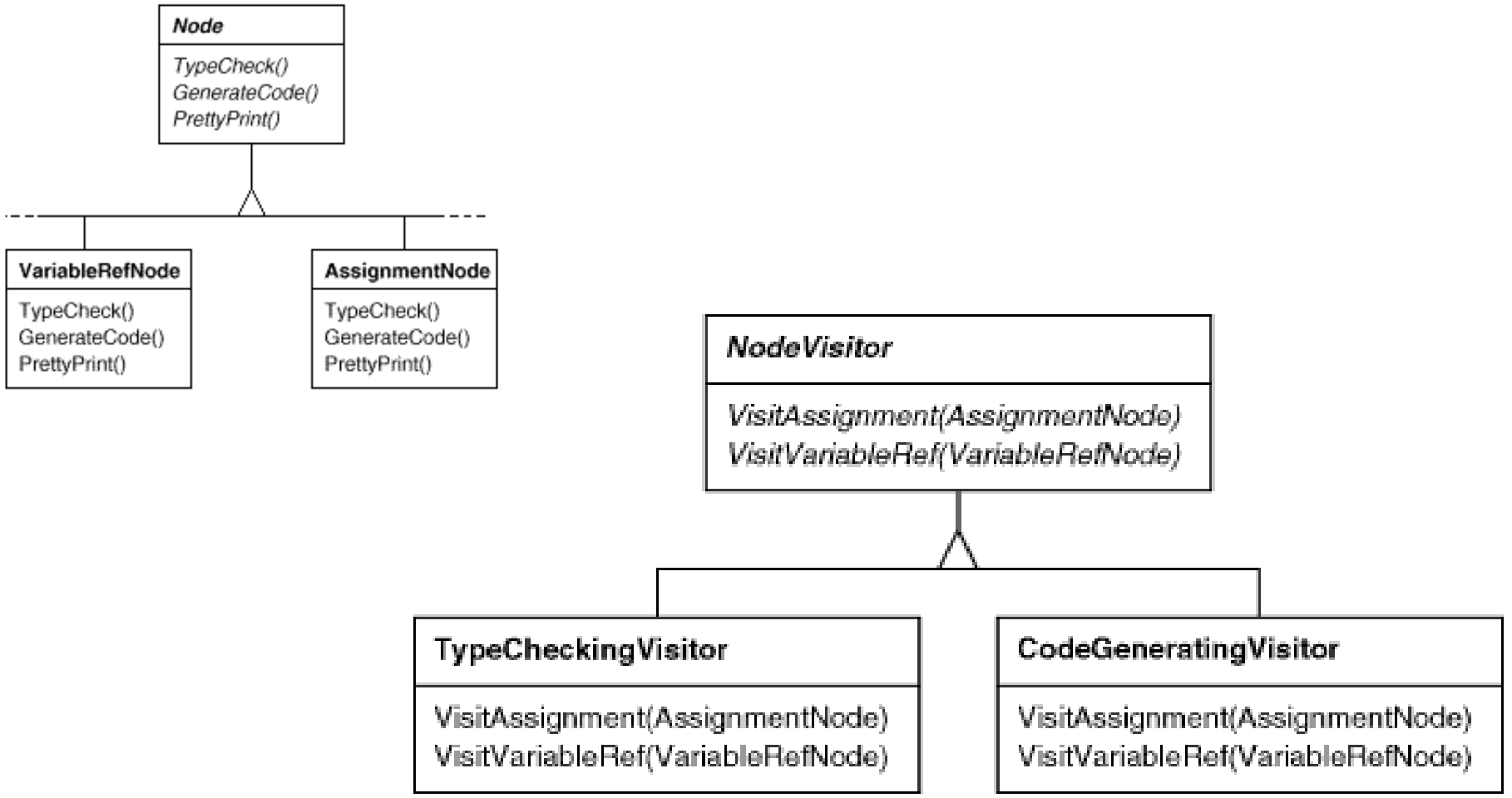
- Quando um elemento “aceita” um visitante, este envia uma solicitação para o visitante que o configura segundo a classe do elemento
  - Ele também inclui o elemento como argumento

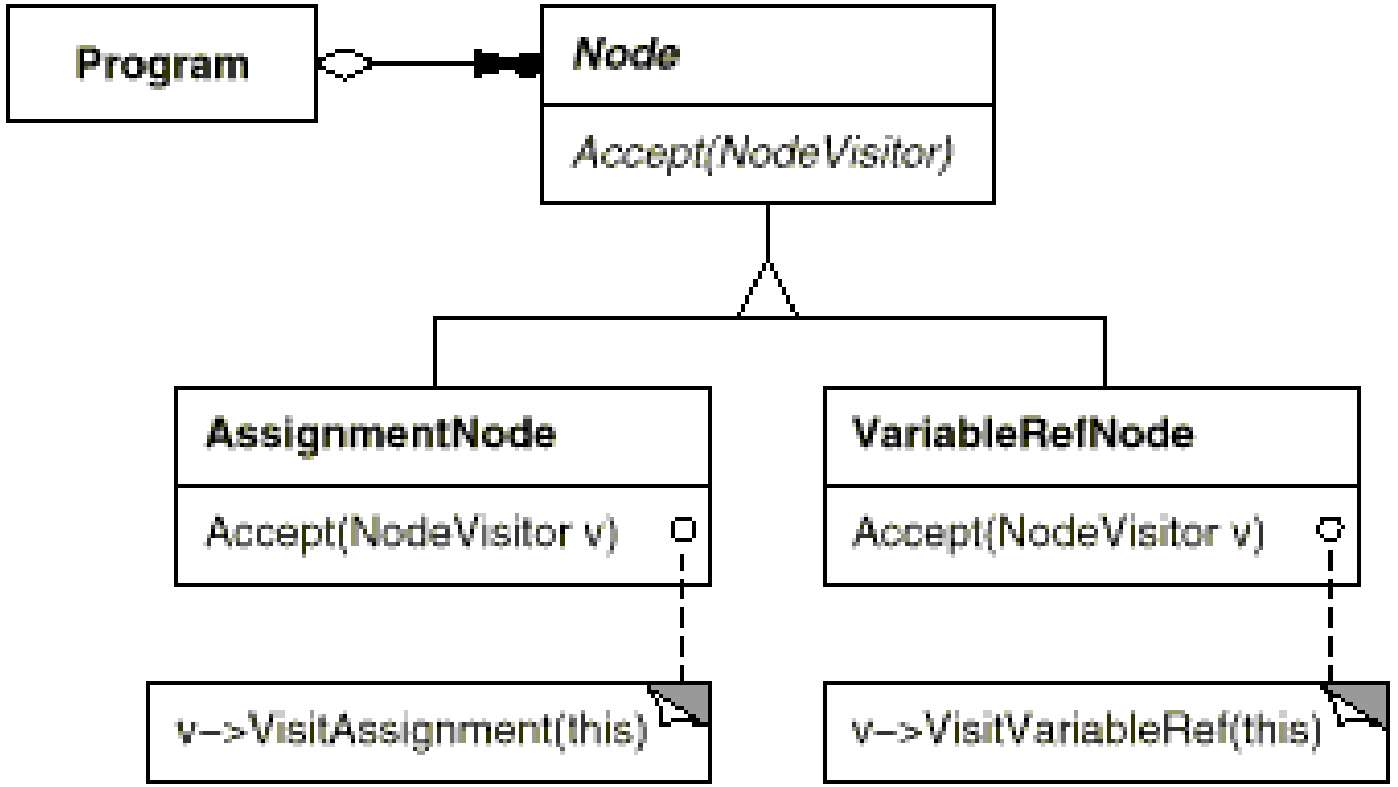
- O visitante executará a operação (que costumava estar no elemento) para cada elemento

Aplicabilidade

- Use esse padrão quando:
  - Uma estrutura de objetos contém muitas classes com interfaces diferentes e você deseja executar operações que dependem das suas classes concretas
  - Muitas operações distintas e não relacionadas necessitam ser executadas sobre objetos de uma estrutura de objetos, evitando a poluição (operações desnecessárias)
  - As classes que definem a estrutura dos objetos raramente mudam; porém, frequentemente você deseja definir novas operações sobre a estrutura


Client
Visitor
VisitConcreteElementA(ConcreteElementA)
VisitConcreteElementB(ConcreteElementB)
ConcreteVisitor1
VisitConcreteElementA(ConcreteElementA)
VisitConcreteElementB(ConcreteElementB)
ConcreteVisitor2
VisitConcreteElementA(ConcreteElementA)
VisitConcreteElementB(ConcreteElementB)
ObjectStructure
Element
Accept(Visitor)
ConcreteElementA
Accept(Visitor v)
OperationA()
ConcreteElementB
Accept(Visitor v)
OperationB()
v->VisitConcreteElementA(this)
v->VisitConcreteElementB(this)


Node
TypeCheck()
GenerateCode()
PrettyPrint()
VariableRefNode
TypeCheck()
GenerateCode()
PrettyPrint()
AssignmentNode
TypeCheck()
GenerateCode()
PrettyPrint()
NodeVisitor
VisitAssignment(AssignmentNode)
VisitVariableRef(VariableRefNode)
TypeCheckingVisitor
VisitAssignment(AssignmentNode)
VisitVariableRef(VariableRefNode)
CodeGeneratingVisitor
VisitAssignment(AssignmentNode)
VisitVariableRef(VariableRefNode)


Program
Node
Accept(NodeVisitor)
AssignmentNode
Accept(NodeVisitor v)
VariableRefNode
Accept(NodeVisitor v)
v->VisitAssignment(this)
v->VisitVariableRef(this)

Exemplo/Participantes

1. **Visitor (NodeVisitor)**
   - Declara a opção **Visit** para as classes **ConcreteElement**
2. **ConcreteVisitor (TypeCheckingVisitor)**
   - Implementa cada operação declarada por **Visitor**
3. **Element (Node)**
   - Define uma operação **Accept** que aceita um visitante
4. **ConcreteElement (AssigmentNode, VariableRefNode)**
   - Implementa a operação **Accept** que aceita um visitante
5. **ObjectStruture (Program)**
   - Pode enumerar os seus elementos
   - Fornece uma interface de alto nível para visitar seus elementos

Consequências

1. O **Visitor** torna fácil a adição de novas operações
2. Um visitante reúne operações relacionadas e separa as operações não-relacionadas
3. É difícil acrescentar novas classes **ConcreteElement** – cada novo elemento dá origem a uma nova operação abstrata em **Visitor**
4. Não possibilita a implementação de **Iterator**
5. Os **Visitors** podem acumular estados a medida que visitam cada elemento da estrutura
6. Forçam os elementos visitados a fornecerem operações públicas de acesso ao seu estado

Exemplo Prático

1. Neste exemplo, gostaríamos de exportar um conjunto de formas geométricas para XML. O problema é que não queremos alterar o código de formas, diretamente ou, pelo menos, manter as alterações ao mínimo.
2. No final, o padrão Visitor estabelece uma infraestrutura que nos permite adicionar comportamentos à hierarquia de formas sem alterar o código existente dessas classes.

Exemplo baseado no site: https://refactoring.guru/pt-br/design-patterns/visitor/java/example

Saiba mais...

- https://www.youtube.com/watch?v=5PRG7rT2dcU
- https://refactoring.guru/pt-br/design-patterns/visitor
- https://www.youtube.com/watch?v=KmVZamAB_AA

Acesse os endereços e veja mais detalhes sobre o padrão Visitor.