PADRÕES DE PROJETO DE SOFTWARE

ACH 2003 — COMPUTAÇÃO ORIENTADA A OBJETOS

Daniel Cordeiro

15 de junho de 2016

Escola de Artes, Ciências e Humanidades | EACH | USP

PADRÕES COMPORTAMENTAIS

PADRÕES COMPORTAMENTAIS

São os padrões relacionados especificamente ao modo como os objetos se comunicam entre si.

- ~~Chain of responsibility~~
- ~~Command~~
- ~~Interpreter~~
- ~~Iterator~~
- ~~Mediator~~
- ~~Memento~~
- ~~Null Object~~
- ~~Observer~~ ~~(edit)~~
- State
- Strategy
- Template method
- Visitor

STATE

OBJETIVO

- Permitir que um objeto mude seu comportamento quando ocorrerem mudanças no seu estado interno. Parecerá que o objeto mudou de classe
- Uma máquina de estados orientada a objetos

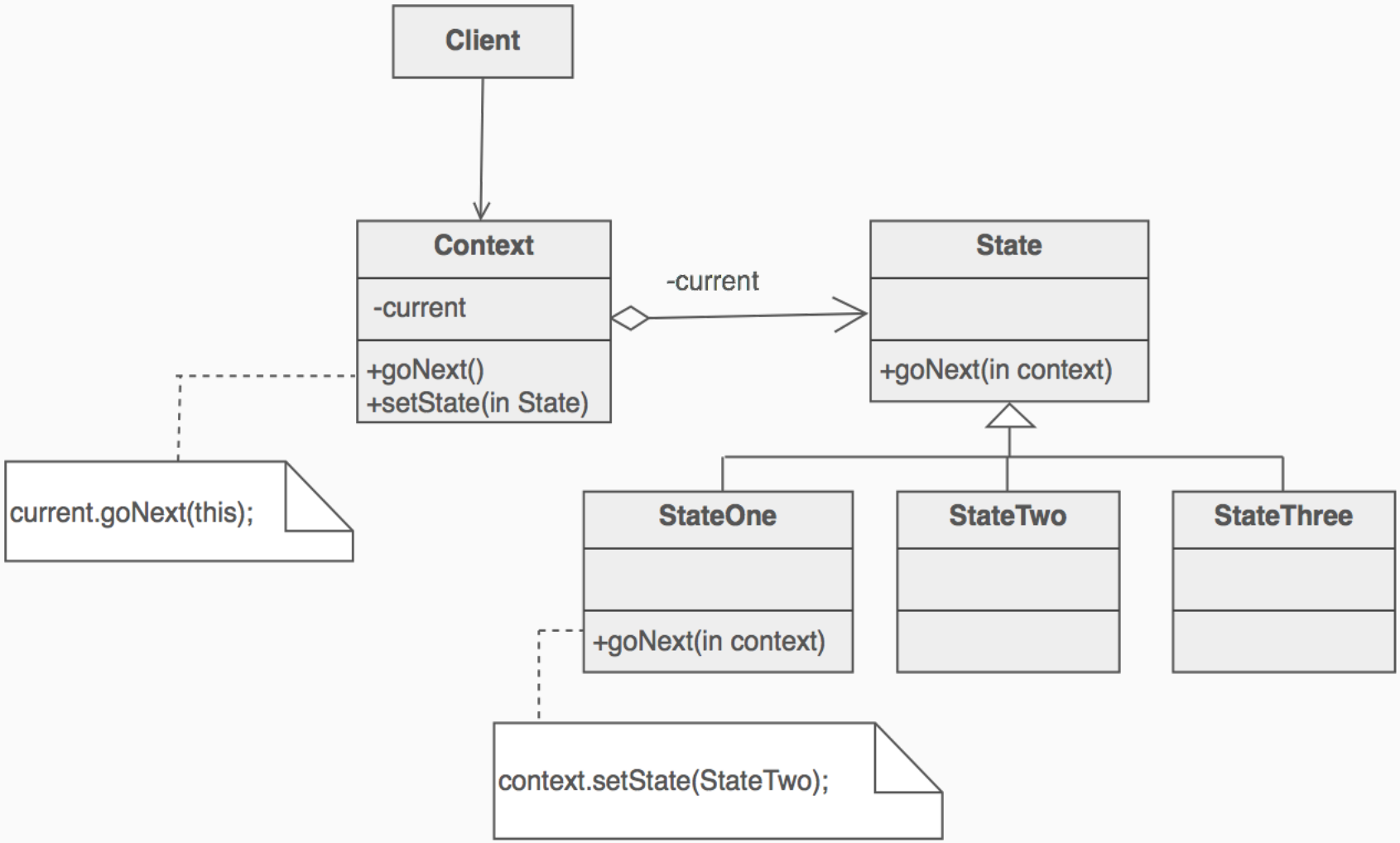
Problema

O comportamento de um objeto é uma função do seu estado. Seu comportamento é determinado em tempo de execução de acordo com o estado atual. Ou sua aplicação é caracterizada por um conjunto grande de condições `case` sobre o estado que controlam o fluxo de execução da aplicação.

DISCUSSÃO

- O padrão State é uma solução para o problema de como fazer o comportamento ser dependente do estado
- Defina uma classe “contexto” que define uma interface única
- Defina uma classe abstrata `State`
- Represente os diferentes “estados” da máquina de estados como uma classe derivada da classe base `State`
- Defina os comportamentos que dependem do estado na classe derivada de `State` apropriada
- Mantenha um ponteiro para o “estado” atual na classe “contexto”
- Para mudar o estado da máquina de estados, mude o ponteiro para o novo “estado”
- O padrão State não especifica **como** as transições são definidas; isso pode ser feito ou pelo contexto, ou pelas classes derivadas

ESTRUTURA


Client
Context
-current
+goNext()
+setState(in State)
-current
State
+goNext(in context)
current.goNext(this);
StateOne
+goNext(in context)
StateTwo
StateThree
context.setState(StateTwo);

LISTA DE VERIFICAÇÃO

1. Identifique ou crie a classe que servirá de máquina de estados da perspectiva do cliente. A classe será a classe *wrapper*
2. Crie uma classe base State que replica os métodos da interface da máquina de estados. Os métodos recebem um parâmetro adicional: uma instância da classe *wrapper*
3. Crie uma classe derivada de State para cada possível estado
4. A classe *wrapper* mantém o objeto State "atual"
5. Todas as requisições do cliente para a classe *wrapper* são delegadas para o objeto State atual e um ponteiro para o `this` do objeto *wrapper* é passado
6. Os métodos de State mudam o objeto estado "atual" do objeto *wrapper* quando apropriado

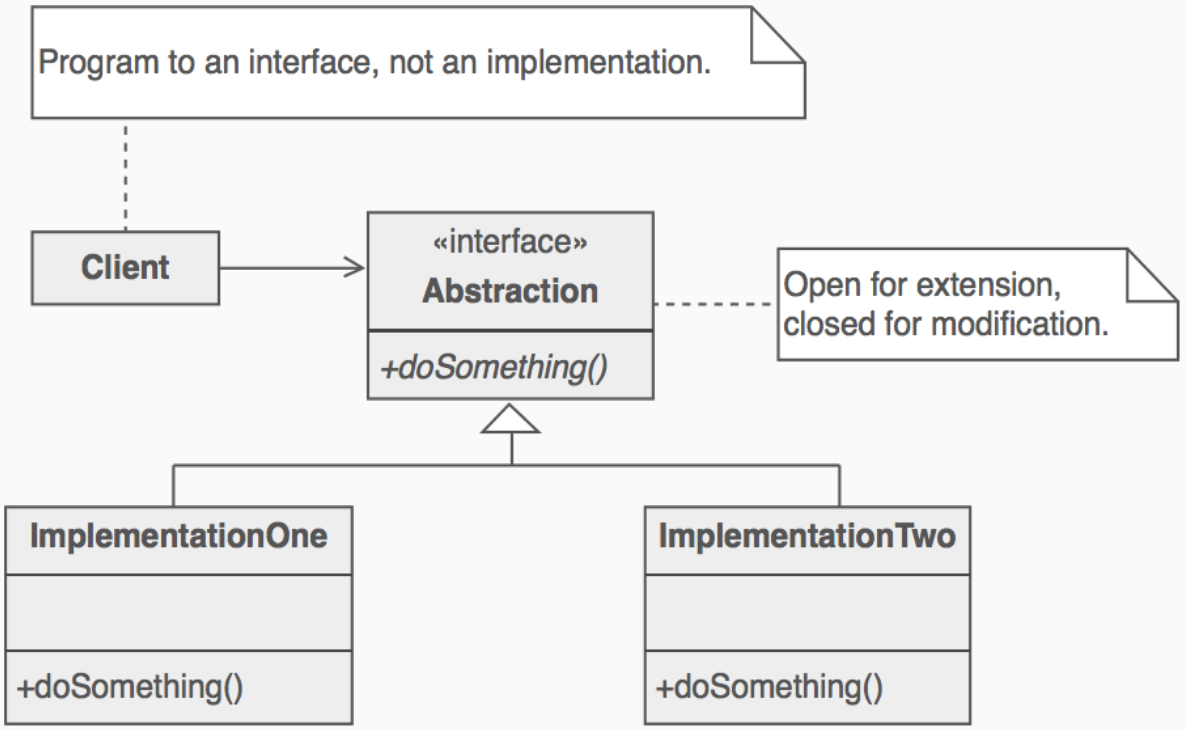
STRATEGY

OBJETIVO

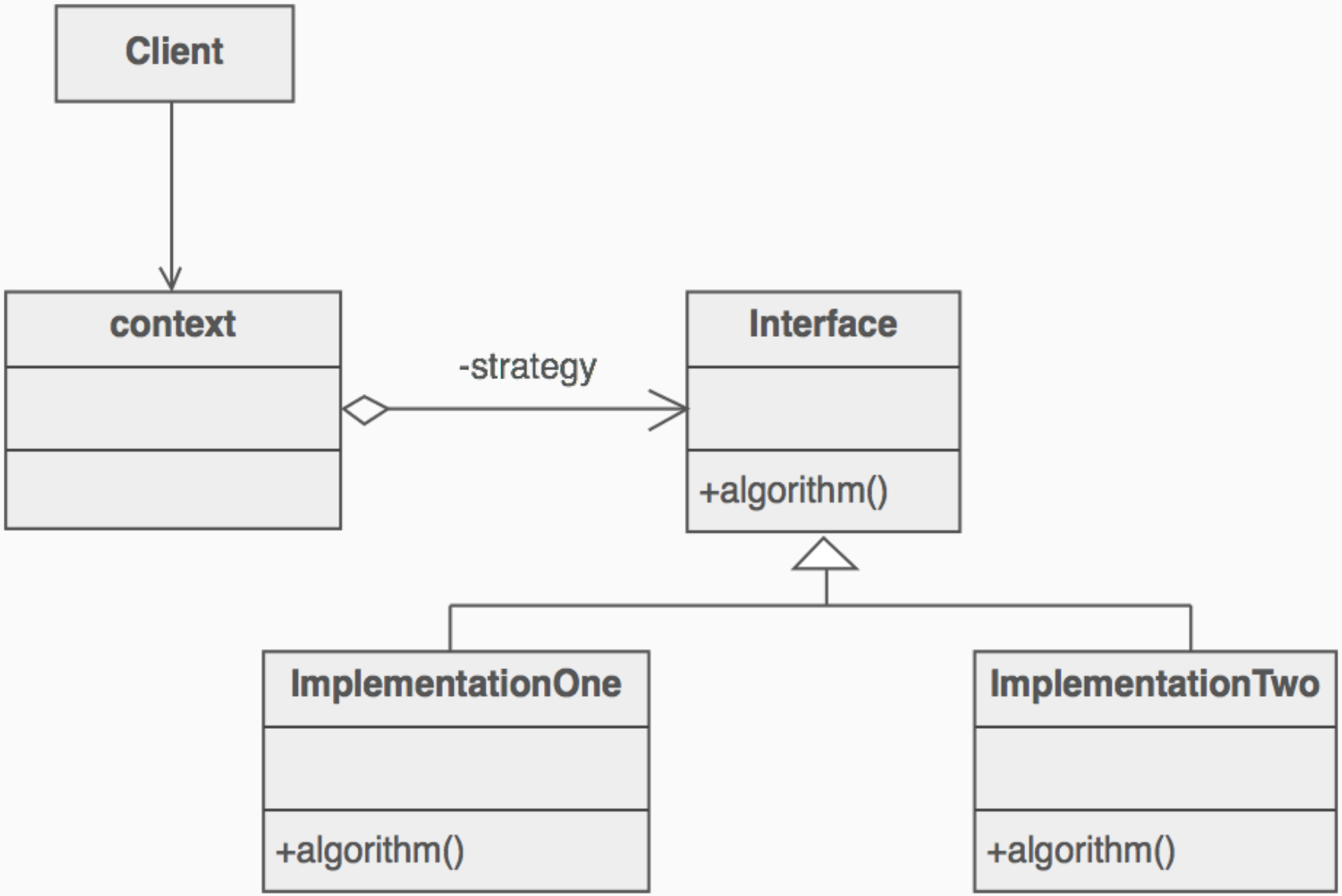
- Definir uma família de algoritmos, encapsulá-los e fazer com que seja fácil trocar um deles por outro. Strategy permite variar o algoritmo independentemente dos clientes que o usam
- Capture a abstração em uma interface e esconda os detalhes de implementação nas classes derivadas

PROBLEMA

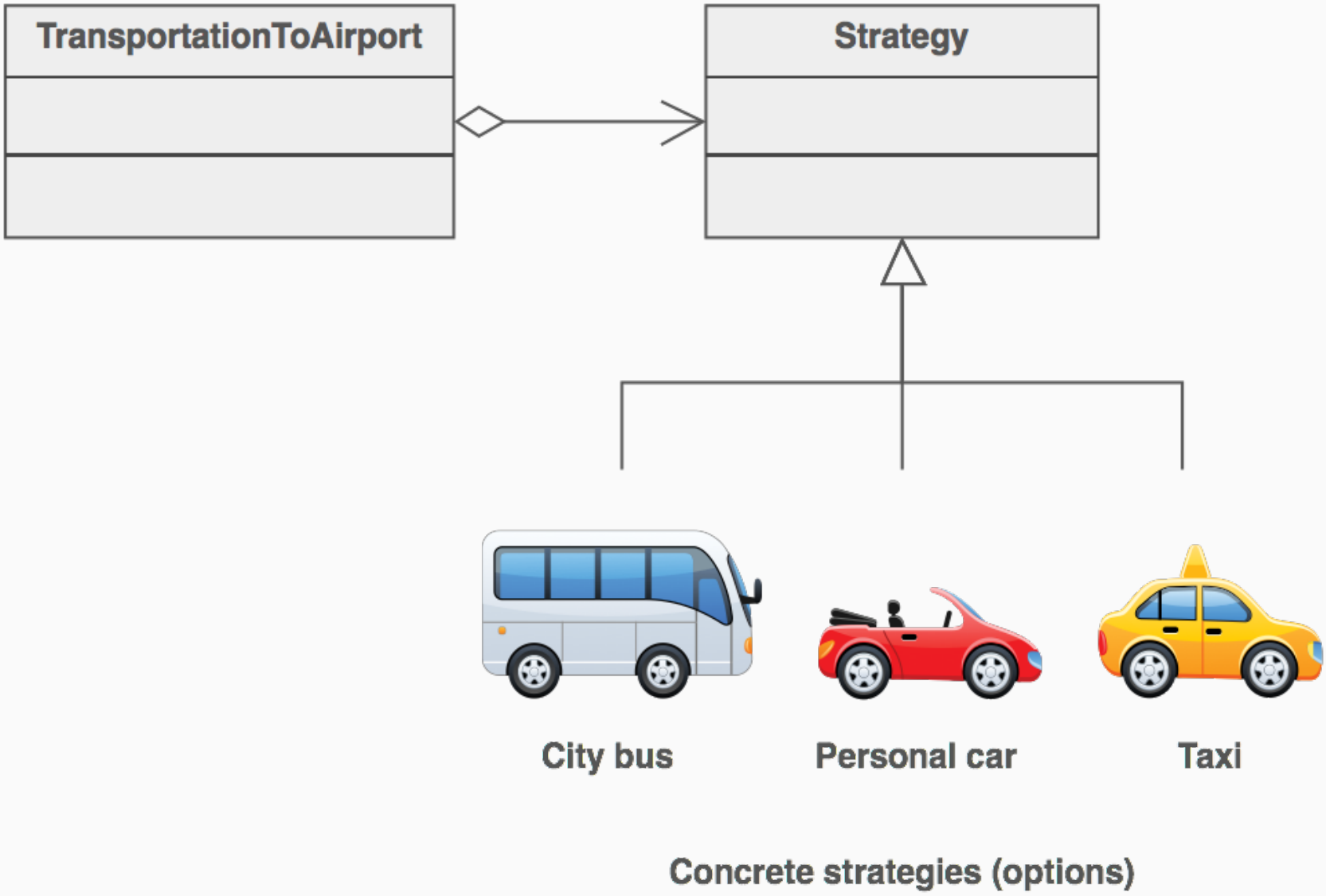
- Em programação orientada a objetos, um princípio de projeto importante é o princípio *open/closed* entidades do software (classes, módulos, funções, etc.) devem estar abertas para extensão, mas fechadas para modificação
- Normalmente alcançado quando programamos usando interfaces e não implementações



ESTRUTURA


Client
context
-strategy
Interface
+algorithm()
ImplementationOne
+algorithm()
ImplementationTwo
+algorithm()

EXEMPLO


TransportationToAirport
Strategy
City bus
Personal car
Taxi
Concrete strategies (options)

LISTA DE VERIFICAÇÃO

1. Identifique um algoritmo (i.e., um comportamento) que o cliente poderia preferir acessar usando um “ponto flexível”
2. Especifique a assinatura da interface para aquele algoritmo
3. Esconda os detalhes das implementações alternativas em classes derivadas
4. Os clientes são acoplados somente à interface

TEMPLATE METHOD

- Definir um esqueleto de um algoritmo, adiando e delegando alguns passos para subclasses cliente. Template Method permite a subclasses redefinir alguns passos de um algoritmo sem mudar a estrutura geral do algoritmo
- Definir “espaços reservados” em um algoritmo e deixar que as classes derivadas os implementem

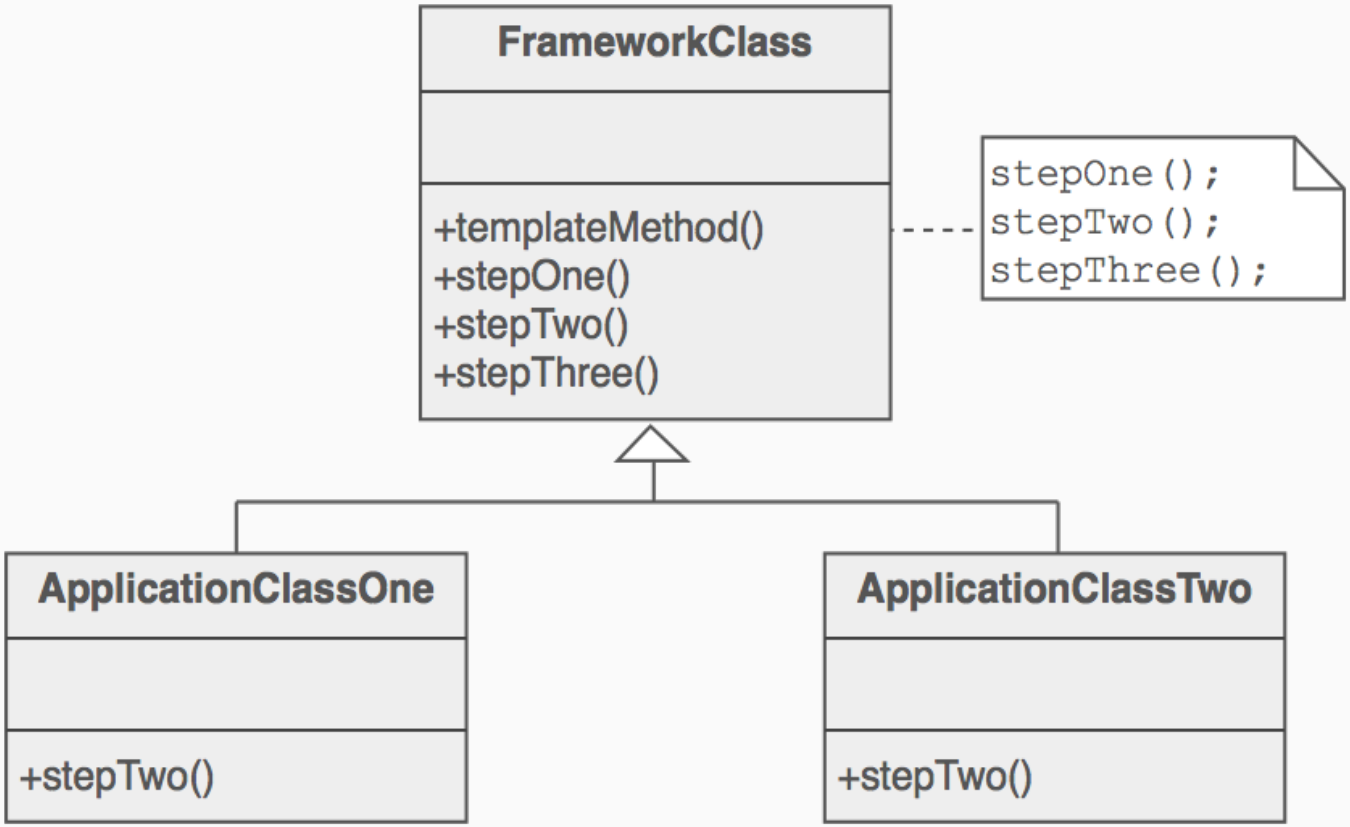
Problema

Dois componentes diferentes possuem muitas coisas em comum, mas não reutilizam nada. Se for preciso fazer uma mudança comum aos dois componentes, esforço duplicado será necessário.

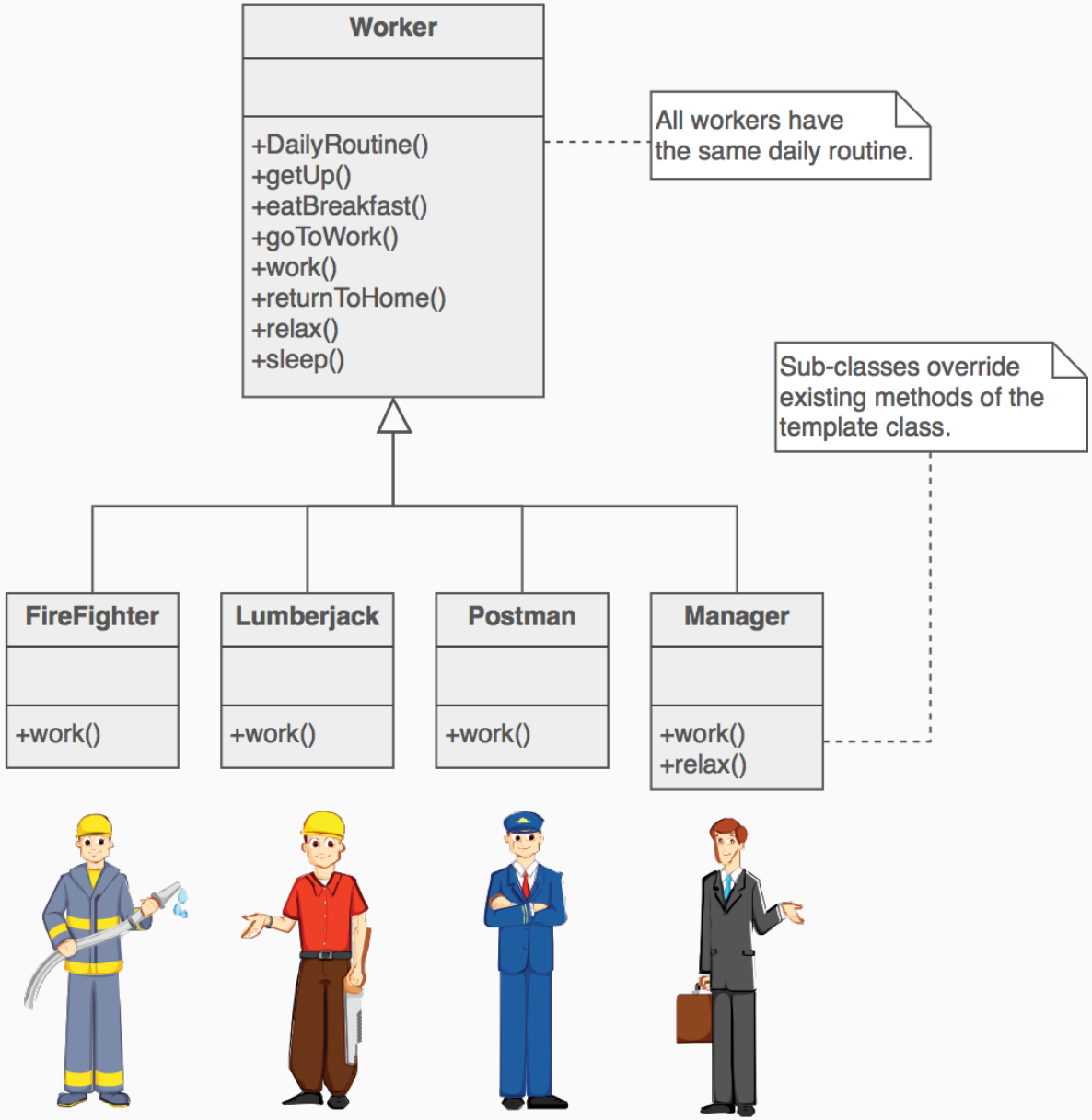
DISCUSSÃO

- O projetista do componente deve decidir quais passos de um algoritmo não variam e quais podem ser personalizados
- Os passos que não variam são implementados na classe base, os outros não são implementados ou são implementados com algum comportamento padrão
- Os passos que variam pode ser vistos como “ganchos” ou “espaços reservados” para modificações e devem ser fornecidos pelas classes derivadas
- O projetista especifica quais passos do algoritmo são obrigatórios e o componente cliente estende ou substitui alguns desses passos

ESTRUTURA


FrameworkClass
+templateMethod()
+stepOne()
+stepTwo()
+stepThree()
stepOne();
stepTwo();
stepThree();
ApplicationClassOne
+stepTwo()
ApplicationClassTwo
+stepTwo()

EXEMPLO


Worker
+DailyRoutine()
+getUp()
+eatBreakfast()
+goToWork()
+work()
+returnToHome()
+relax()
+sleep()
All workers have the same daily routine.
Sub-classes override existing methods of the template class.
FireFighter
+work()
Lumberjack
+work()
Postman
+work()
Manager
+work()
+relax()

IMPLEMENTAÇÃO

```
class SortUp {
  // Shell sort (generalização de bubble sort)
  public:
  void sort(int v[], int n) {
    for (int g = n / 2; g > 0; g /= 2)
      for (int i = g; i < n; i++)
        for (int j = i - g; j >= 0; j -= g)
          if (v[j] > v[j + g])
            doSwap(v[j], v[j + g]);
  }
  private:
  void doSwap(int &a, int &b) {
    int t = a;
    a = b;
    b = t;
  }
};

class SortDown {
  public:
  void sort(int v[], int n) {
    for (int g = n / 2; g > 0; g /= 2)
      for (int i = g; i < n; i++)
        for (int j = i - g; j >= 0; j -= g)
          if (v[j] < v[j + g])
            doSwap(v[j], v[j + g]);
  }
  private:
  void doSwap(int &a, int &b) {
    int t = a;
    a = b;
    b = t;
  }
};}
```

IMPLEMENTAÇÃO

```
class SortUp {
  // Shell sort (generalização de bubble sort)
  public:
  void sort(int v[], int n) {
    for (int g = n / 2; g > 0; g /= 2)
      for (int i = g; i < n; i++)
        for (int j = i - g; j >= 0; j -= g)
          if (v[j] > v[j + g])
            doSwap(v[j], v[j + g]);
  }
  private:
  void doSwap(int &a, int &b) {
    int t = a;
    a = b;
    b = t;
  }
};

class SortDown {
  public:
  void sort(int v[], int n) {
    for (int g = n / 2; g > 0; g /= 2)
      for (int i = g; i < n; i++)
        for (int j = i - g; j >= 0; j -= g)
          if (v[j] < v[j + g])
            doSwap(v[j], v[j + g]);
  }
  private:
  void doSwap(int &a, int &b) {
    int t = a;
    a = b;
    b = t;
  }
};}
```

```
class AbstractSort {
    // Shell sort
  public:
    void sort(int v[], int n) {
        for (int g = n / 2; g > 0; g /= 2)
          for (int i = g; i < n; i++)
            for (int j = i - g; j >= 0; j -= g)
              if (needSwap(v[j], v[j + g]))
                doSwap(v[j], v[j + g]);
    }
  private:
    virtual int needSwap(int, int) = 0;
    void doSwap(int &a, int &b) {
        int t = a;
        a = b;
        b = t;
    }
};

class SortUp: public AbstractSort {
    /* virtual */
    int needSwap(int a, int b) {
        return (a > b);
    }
};
class SortDown: public AbstractSort {
    /* virtual */
    int needSwap(int a, int b)
    {
        return (a < b);
    }
};
```

LISTA DE VERIFICAÇÃO

1. Examine o algoritmo e decida quais passos são padrões e quais são peculiares a cada classe existente
2. Defina uma nova classe abstrata que defina o algoritmo genérico
3. Mova para a nova classe o esqueleto do algoritmo (o “template method”) e a definição dos passos padrões
4. Defina os “espaços reservados” na classe base. Esses métodos podem fornecer uma implementação padrão ou serem declarados como abstratos
5. Chame os métodos abstratos no template method
6. Os detalhes de implementação que sobraram das classes originais serão implementados em cada classe derivada

VISITOR

OBJETIVO

- Representar uma operação que será realizada em diferentes elementos da estrutura de um objeto
- Técnica clássica para recuperar informações sobre informações de tipo que foram perdidas
- Faça a coisa certa baseado no tipo de dois objetos
- “Double dispatch”

Problema

Seu programa tem muitas operações distintas que devem ser aplicadas em objetos “nó” de uma estrutura agregada e heterogênea. Você quer evitar “poluir” as classes dos nós com essas operações e você não quer ter que verificar o tipo de cada nó antes de realizar a operação desejada.

DISCUSSÃO I

- O objetivo principal do Visitor é separar um algoritmo da estrutura do objeto em que ele opera
- O processamento dos elementos é abstraído e removido da classe do elemento
- Funcionalidades adicionais podem ser criadas com a adição de uma nova classe derivada de Visitor
- Visitor implementa o chamado “double dispatch”: a operação executada depende do nome da requisição e do tipo dos dois receptores (o tipo do Visitor) e o tipo do elemento que ele irá visitar

DISCUSSÃO II

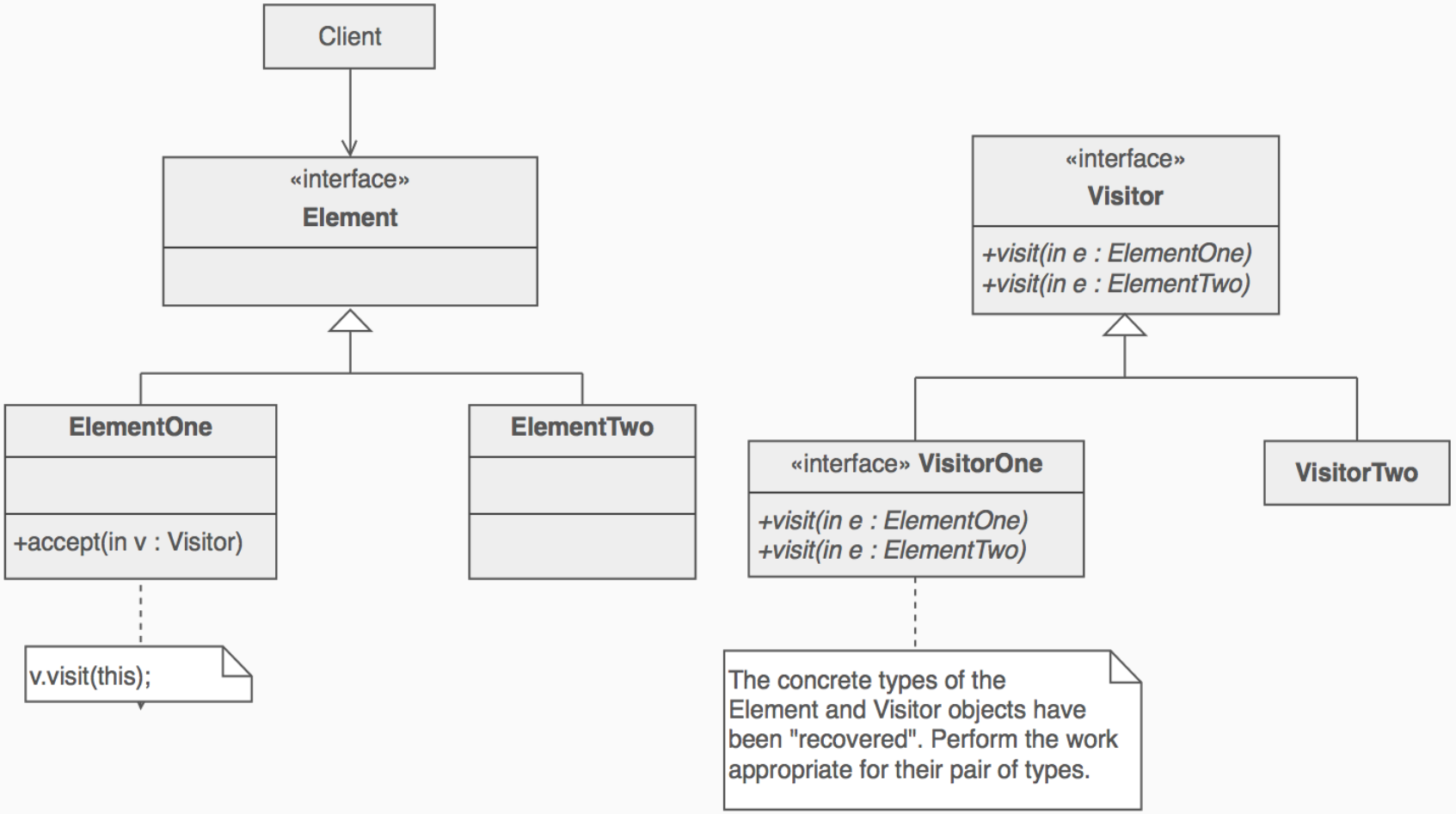
Para implementar o padrão:

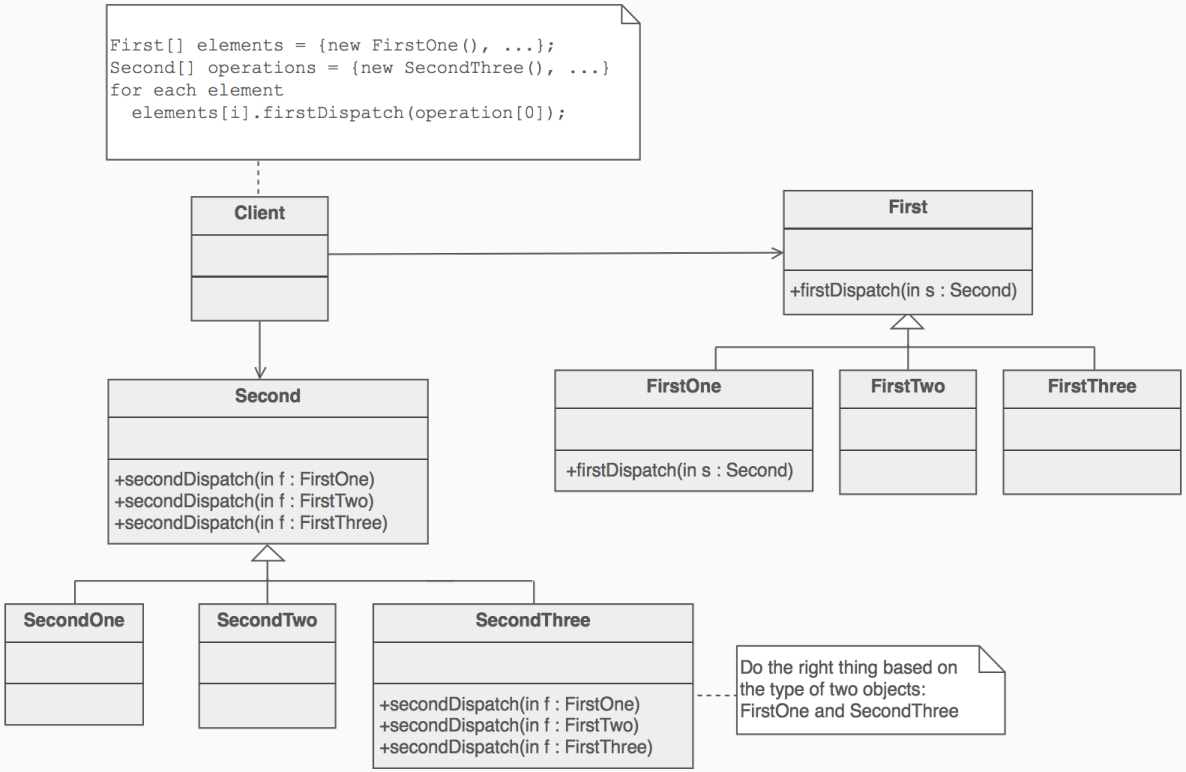
- crie uma hierarquia de classes Visitor que define na classe base um método abstrato `visit` para cada tipo de "nó" que deverá ser processado
- cada método `visit` recebe um único argumento: uma referência à classe derivada do nó original
- cada operação é implementada com uma classe derivada de Visitor, o tipo do argumento definirá qual versão sobrecarregada de `visit` deve ser chamada
- adicione um método abstrato `accept` à classe base dos nós da hierarquia que receberá como parâmetro uma referência à classe base Visitor

discussão III

- cada classe concreta a hierarquia de nós implementará o `accept` como uma chamada de `visit` no objeto argumento e passará `this` como parâmetro
- tudo pronto; quando o cliente precisar ele criará uma instância de objeto Visitor e chamará o método `accept` para cada elemento (nó) da hierarquia
- o `accept` fará que o fluxo de execução encontre a subclasse correta do nó: quando o método `visit` for chamado pelo `accept` ele será despachado para a subclasse correta de Visitor (o despacho do `accept` + despacho do `visit` = “double dispatch”)

ESTRUTURA

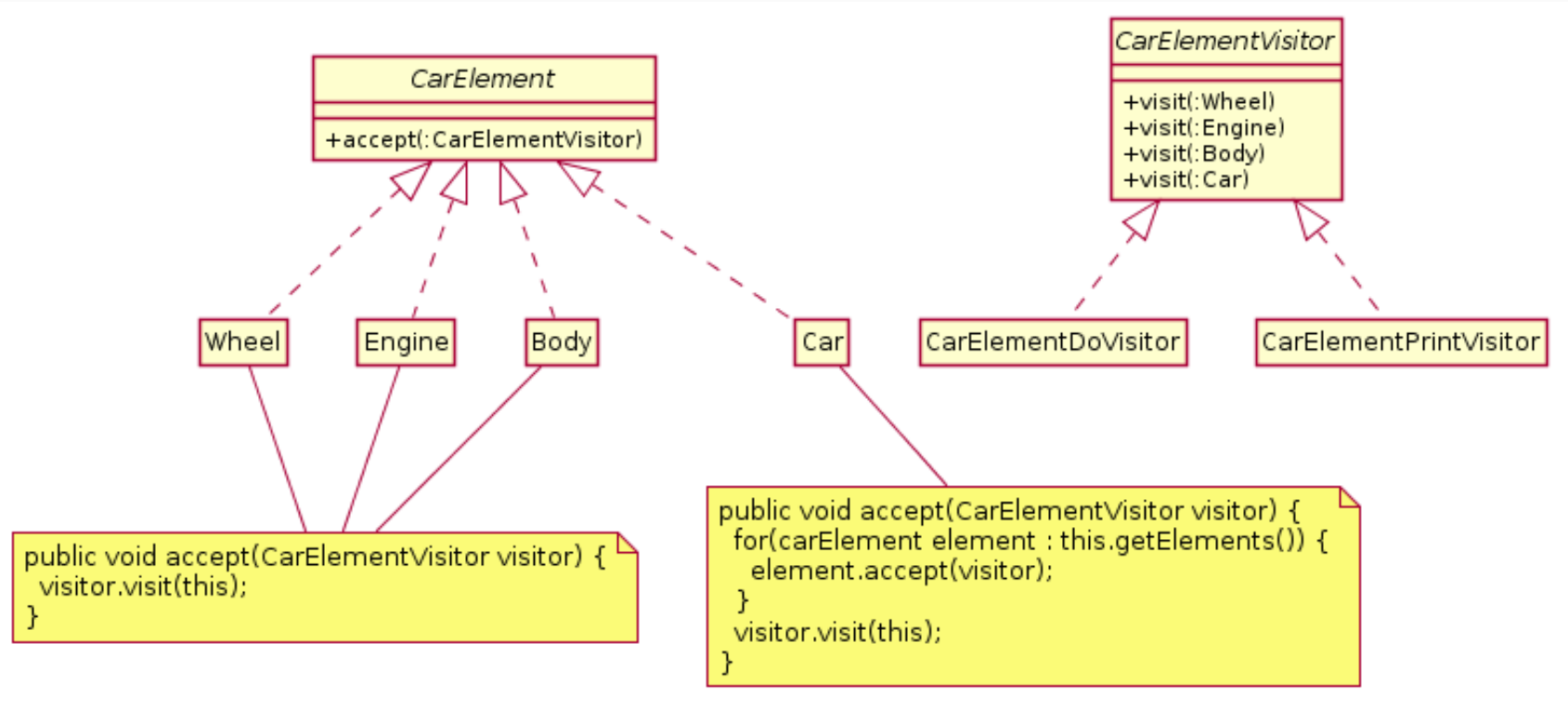

Client
«interface»
Element
ElementOne
+accept(in v : Visitor)
v.visit(this);
ElementTwo
«interface»
Visitor
+visit(in e : ElementOne)
+visit(in e : ElementTwo)
«interface» VisitorOne
+visit(in e : ElementOne)
+visit(in e : ElementTwo)
VisitorTwo
The concrete types of the Element and Visitor objects have been "recovered". Perform the work appropriate for their pair of types.



Exemplo

Quando o método polimórfico `firstDispatch` é chamado com um objeto abstrato `First`, o tipo concreto de daquele objeto é "recuperado". Quando o método polimórfico `secondDispatch` for chamado com um objeto abstrato `Second`, seu tipo concreto também é "recuperado"

EXEMPLO



Adaptado de *Visitor pattern*, `https://en.wikipedia.org/w/index.php?title=Visitor_pattern&oldid=723981660`

IMPLEMENTAÇÃO I

```
interface ICarElementVisitor {
    void visit(Wheel wheel);
    void visit(Engine engine);
    void visit(Body body);
    void visit(Car car);
}

interface ICarElement {
    void accept(ICarElementVisitor visitor);
}

class Wheel implements ICarElement {
    private String name;

    public Wheel(String name) {
        this.name = name;
    }

    public String getName() {
        return this.name;
    }
```

IMPLEMENTAÇÃO II

```
    public void accept(ICarElementVisitor visitor) {
        /*
         * accept(ICarElementVisitor) em Wheel implementa
         * accept(ICarElementVisitor) em ICarElement, de modo que
         * o accept é escolhido em tempo de execução. Este é o
         * "primeiro despacho". Mas a decisão de chamar
         * visit(Wheel) (ao invés de visit(Engine), etc.) pode ser feita
         * em tempo de compilação já que 'this' é sabidamente do
         * do tipo Wheel. Além disso, cada implementação de
         * ICarElementVisitor implementa o visit(Wheel), que é
         * outra decisão que deve ser feita em tempo de execução.
         * Esse é o "segundo despacho".
         */
        visitor.visit(this);
    }
}
```

IMPLEMENTAÇÃO III

```
class Engine implements ICarElement {
    public void accept(ICarElementVisitor visitor) {
        visitor.visit(this);
    }
}

class Body implements ICarElement {
    public void accept(ICarElementVisitor visitor) {
        visitor.visit(this);
    }
}

class Car implements ICarElement {
    ICarElement[] elements;

    public Car() {
        this.elements = new ICarElement[] { new Wheel("front left"),
            new Wheel("front right"), new Wheel("back left") ,
            new Wheel("back right"), new Body(), new Engine() };
    }
```

Implementação IV

```
    public void accept(ICarElementVisitor visitor) {
        for(ICarElement elem : elements) {
            elem.accept(visitor);
        }
        visitor.visit(this);
    }
}

class CarElementPrintVisitor implements ICarElementVisitor {
    public void visit(Wheel wheel) {
        System.out.println("Visiting " + wheel.getName() + " wheel");
    }

    public void visit(Engine engine) {
        System.out.println("Visiting engine");
    }

    public void visit(Body body) {
        System.out.println("Visiting body");
    }

    public void visit(Car car) {
```

IMPLEMENTAÇÃO V

```
        System.out.println("Visiting car");
    }
}

class CarElementDoVisitor implements ICarElementVisitor {
    public void visit(Wheel wheel) {
        System.out.println("Kicking my " + wheel.getName() + " wheel");
    }

    public void visit(Engine engine) {
        System.out.println("Starting my engine");
    }

    public void visit(Body body) {
        System.out.println("Moving my body");
    }

    public void visit(Car car) {
        System.out.println("Starting my car");
    }
}
```

IMPLEMENTAÇÃO VI

```
public class VisitorDemo {
    public static void main(String[] args) {
        ICarElement car = new Car();
        car.accept(new CarElementPrintVisitor());
        car.accept(new CarElementDoVisitor());
    }
}
```

Saída:

```
Visiting front left wheel
Visiting front right wheel
Visiting back left wheel
Visiting back right wheel
Visiting body
Visiting engine
Visiting car
Kicking my front left wheel
Kicking my front right wheel
Kicking my back left wheel
Kicking my back right wheel
Moving my body
Starting my engine
Starting my car
```

LISTA DE VERIFICAÇÃO I

- Confira se a hierarquia de nós se manterá estável e que a interface pública dessas classes é suficiente para o acesso que as classes Visitor precisarão
- Crie uma classe base Visitor com um método `visit(Nó x)` para cada tipo concreto de nó
- Adicione um método `accept(Visitor)` na hierarquia de nós. A implementação em cada classe concreta será sempre a mesma: `accept(Visitor v) { v.visit(this); }`.
- A hierarquia de elementos é acoplada somente à classe base Visitor, mas a hierarquia de Visitor é acoplada a todos os elementos. Se a hierarquia de elementos não for estável, mas a hierarquia de Visitor for, considere trocar os papéis das duas hierarquias

Lista de verificação II

- Crie uma classe derivada de Visitor para cada operação a ser realizada nos elementos. A implementação de `visit` dependerá da interface pública da classe do elemento
- O cliente cria objetos Visitor e os passa para cada objeto nó na chamada de `accept()`

BIBLIOGRAFIA

- The Gang of Four Book, ou GoF: E. Gamma and R. Helm and R. Johnson and J. Vlissides. Design Patterns — Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.
- Alexander Shvets. Design patterns explained simply. `https://sourcemaking.com/design_patterns/`