Padrões Comportamentais

- Representam e descrevem padrões de comunicação entre objetos a fim de realizar um comportamento específico
- Seu objetivo é que foco do projeto seja a interconexão entre os objetos a fim de obter este comportamento
- Padrões definidos por Gamma et. al. (GoF)
  - Chain of Responsibility
  - Command
  - Flyweigth
  - Interpreter
  - Iterator
  - Mediator
  - Memento
  - Observer
  - State
  - Strategy
  - Template
  - Visitor

Padrões Comportamentais

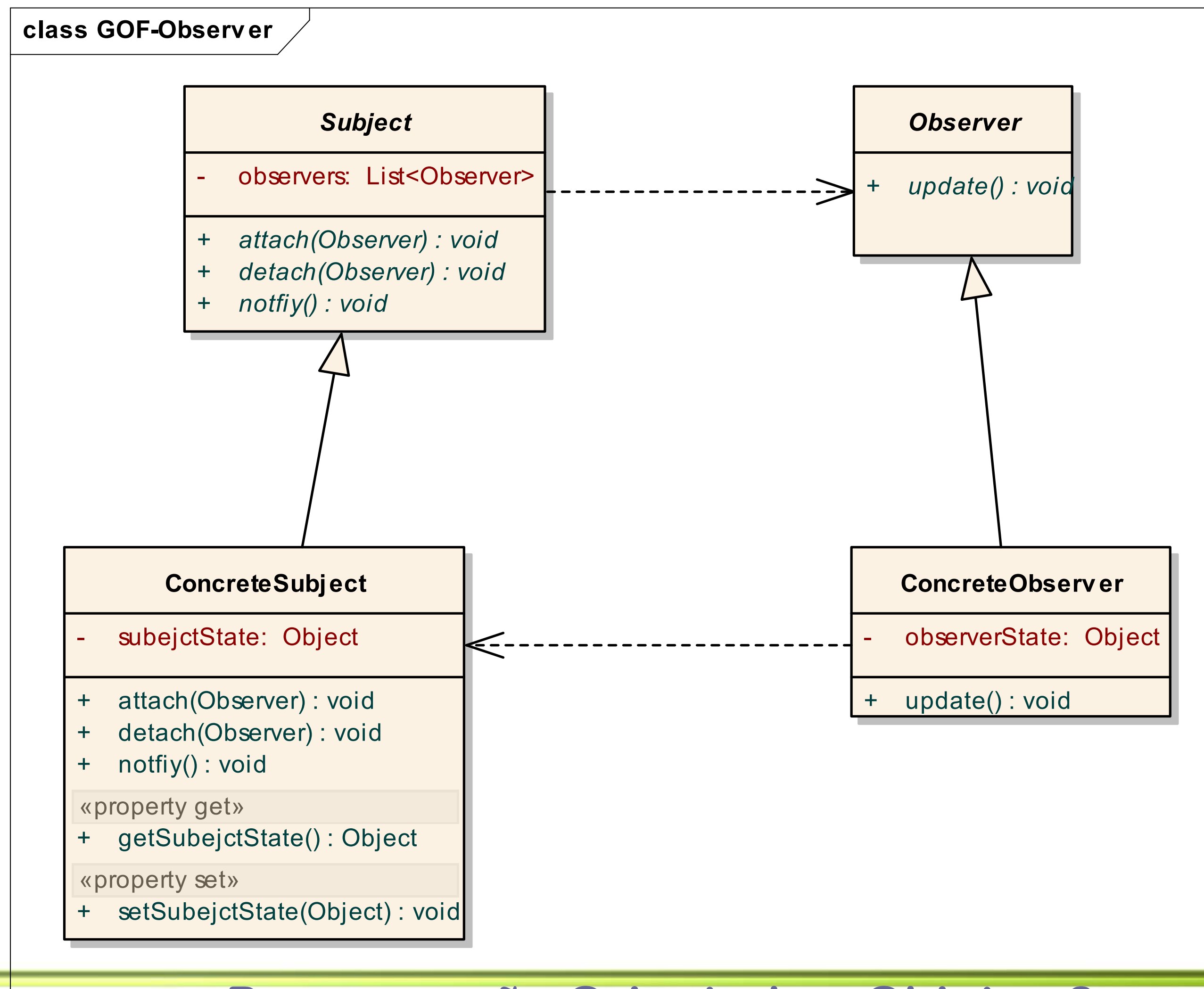
- Observer
- Command
- Strategy
- Template
- Iterator
- Visitor
- Mediator

Observer

- Objetivo
  - Define uma dependência entre objeto e vários outros(um-para-muitos) sendo que em uma alteração neste objeto, todos os outros são notificados
- Motivação
  - Em muitas situações um objeto pode ter vários outros dependentes.
  - Neste caso sempre que este objeto (sujeito) for alterado é interessante que outros sejam notificados
  - Isto porém pode levar a um forte acoplamento entre os mesmos
- Uso
  - Quando uma abstração (Classe) possuir dois ou mais aspectos estes podem ser encapsulados em diferentes classes
  - Em situações onde a alteração de um objeto pode afetar um grupo de outros objetos não conhecidos previamente
  - Permitir que um objeto seja capaz de notificar outros sem que estejam acoplados

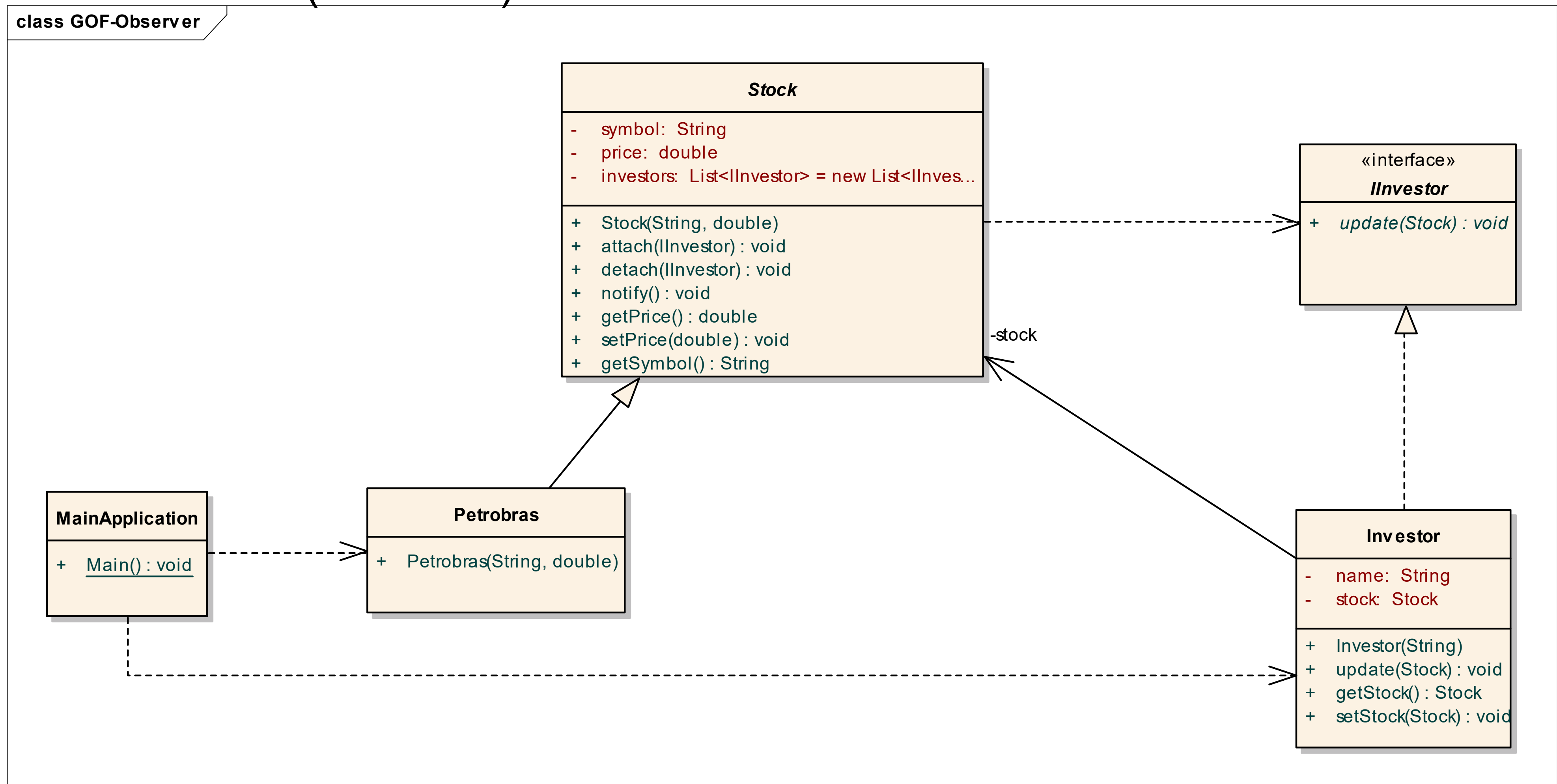
Observer
Estrutura

- O sujeito (subject) é observado por uma lista de observadores (Observer)
- Um método notify() invoca o método update() nos observadores



Observer
Exemplo

- A medida que as ações de uma empresa (Stock) forem alteradas os investirores (Inverstor) serão notificados



Observer
Exemplo - Código

```
/// MainApp startup class for Real-World
public class MainApplication   {
    public static void Main(){
      // Create Petrobras stock and attach investors
      Petrobras petro = new Petrobras("Petrobras", 120.00);
      petro.attach(new Investor("Sorros"));
      petro.attach(new Investor("Berkshire"));

      // Fluctuating prices will notify investors
      petro.setPrice(120.10);
      petro.setPrice(121.00);
      petro.setPrice(120.50);
      petro.setPrice(120.75);
    }
  }
```

Observer
Exemplo - Código

```
/// The 'Subject' abstract class
 public abstract class Stock {
    private String symbol;     private double price;
    private List<IInvestor> investors = new List<IInvestor>();
    public Stock(String symbol, double price){
      this.symbol = symbol;
      this.price = price;
    }
     public void attach(IInvestor investor){    investors.add(investor);   }
     public void detach(IInvestor investor){     investors.remove(investor);   }
     public void notify()     {
      for (IInvestor investor : investors){
         investor.update(this);      }
        System.out.println("");
     }
```

Observer
Exemplo - Código

```
public double getPrice(){  return price; }
 public void setPrice(double nprice){
    if (price != nprice){
        price = nprice;
        notify();
    }
  }
// Gets the symbol
  public String getSymbol(){
   return symbol;
  }
```

Observer
Exemplo - Código

```
// The 'ConcreteSubject' class
public class Petrobras extends Stock
 {
   // Constructor
   public Petrobras(double price){
       super(“PETR”, price)
   }
 }
```

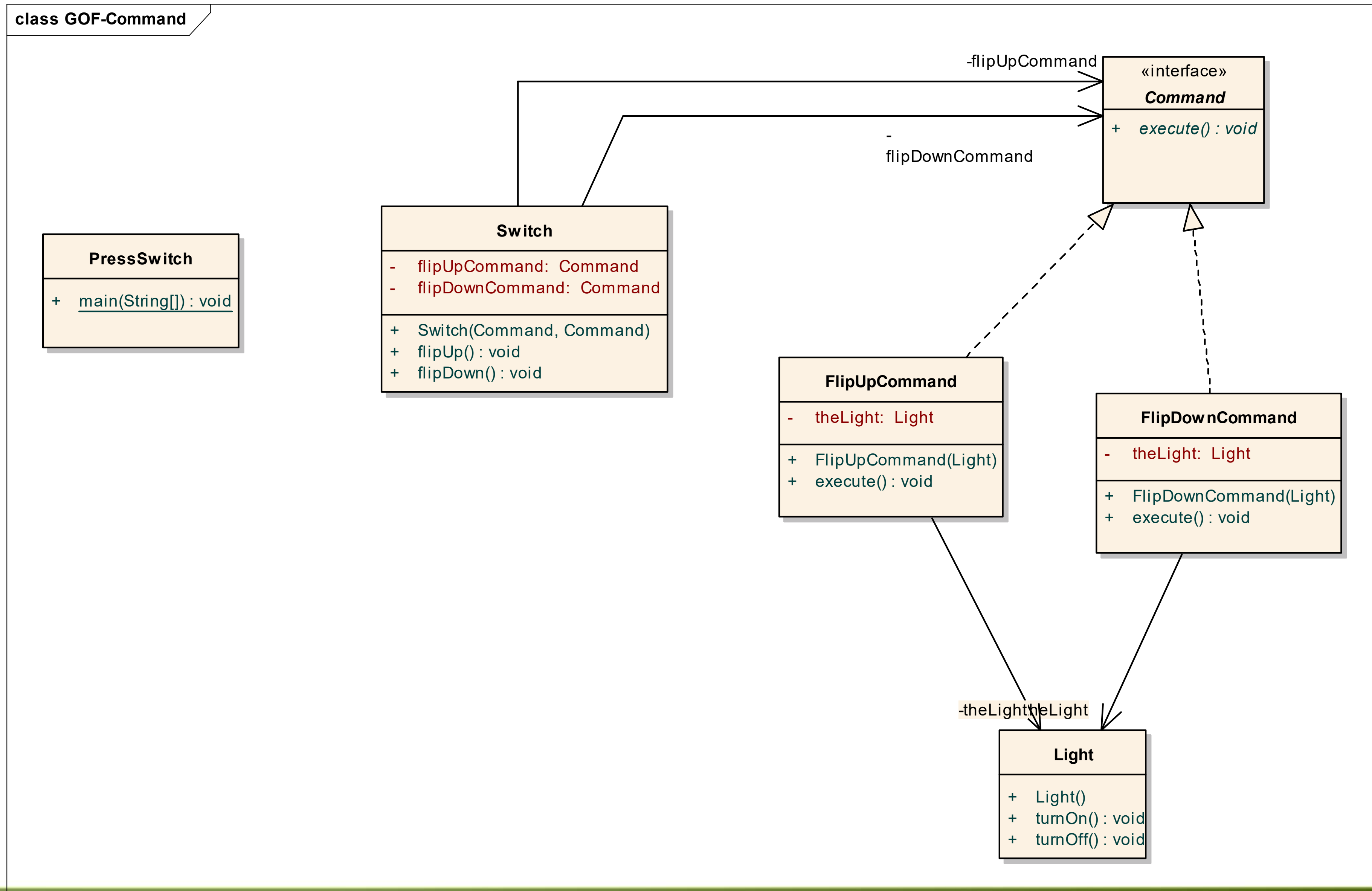
Observer
Exemplo - Código

```
// The 'Observer' interface
interface Iinvestor   {
    void update(Stock stock);
 }

// The 'ConcreteObserver' class
public class Investor implements IInvestor {
    private String name;
    private Stock stock;
   public Investor(String name){    this.name = name;   }
    public void update(Stock stock){
        System.out.println("Notificando " + this.name + "que " + stock.getSymbol() + " alterou para
      " + stock.getPrice();
    }
    public Stock getStock() {   return stock; }
    public void setStock(Stock value){      stock = value;     }
  }
```

Command

- Objetivo
- Motivação
- Uso
- Estrutura
- Exemplo Código

Command
Exemplo - Estrutura



Command
Código

```
/* The Invoker class */
public class Switch {
     private Command flipUpCommand;
    private Command flipDownCommand;
     public Switch(Command flipUpCmd, Command flipDownCmd) {
         this.flipUpCommand = flipUpCmd;
         this.flipDownCommand = flipDownCmd;
    }
     public void flipUp() {
         flipUpCommand.execute();
    }

    public void flipDown() {
         flipDownCommand.execute();
    }
}
```

Command
Código

```
/* The Receiver class */
public class Light {
    public Light() { }
    public void turnOn() {
       System.out.println("The light is on");
    }
    public void turnOff() {
       System.out.println("The light is off");
    }
}
```

Command
Código

```
/* The Command interface */
public interface Command {
    void execute();
}

/* The Command for turning the light on in North America, or turning the light off in most other
      places */
public class FlipUpCommand implements Command {
   private Light theLight;
   public FlipUpCommand(Light light) {
        this.theLight = light;
   }
   public void execute(){
      theLight.turnOn();
   }
}
```

Command
Código

```
/* The Command for turning the light off in North America, or turning the light on in most other
     places */
public class FlipDownCommand implements Command {

   private Light theLight;

   public FlipDownCommand(Light light) {
        this.theLight = light;
   }

   public void execute() {
      theLight.turnOff();
   }
}
```

Command
Código

```
public class PressSwitch {   // The test class or client
      public static void main(String[] args) {
       Light lamp = new Light();
       Command switchUp = new FlipUpCommand(lamp);
       Command switchDown = new FlipDownCommand(lamp);
      Switch s = new Switch(switchUp, switchDown);
       try {
           if (args[0].equalsIgnoreCase("ON")) {
                s.flipUp();
           } else if (args[0].equalsIgnoreCase("OFF")) {
               s.flipDown();
           } else {
               System.out.println("Argument \"ON\" or \"OFF\" is required.");
           }
       } catch (Exception e) {
           System.out.println("Arguments required.");
       }
   }
}
```

Command
Código

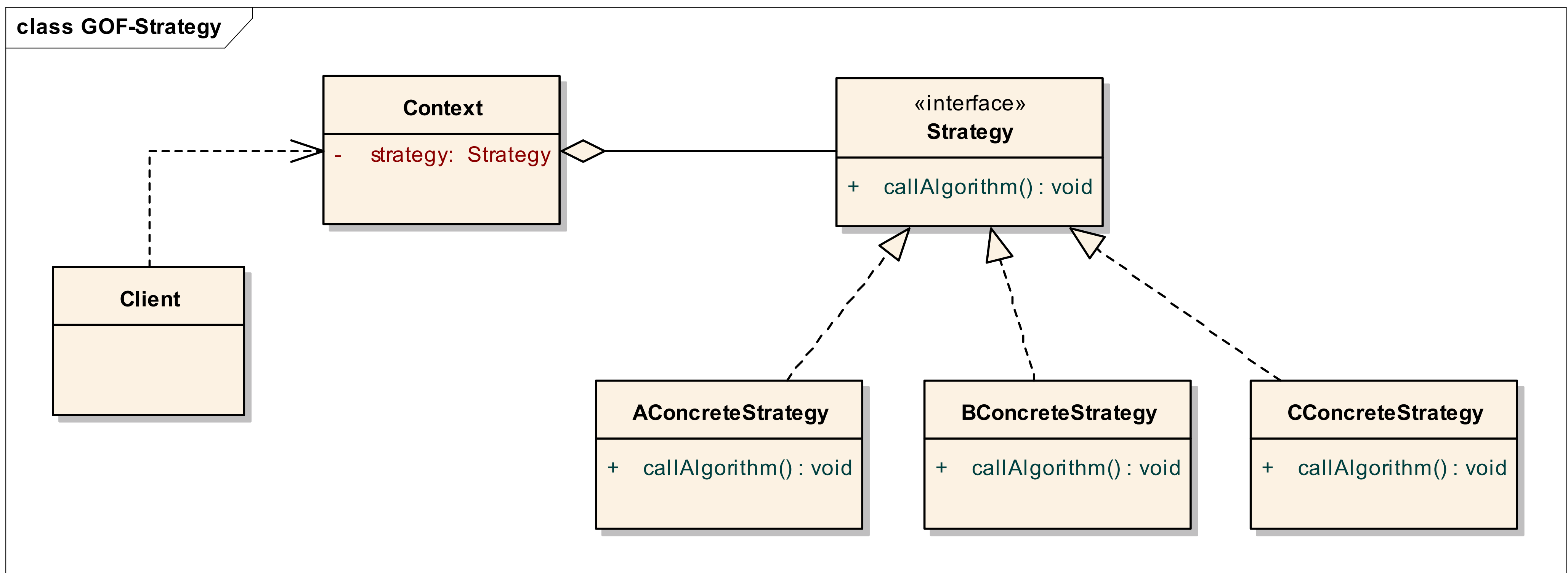
```
/* The Invoker class */
public class Switch {

    private Command flipUpCommand;
    private Command flipDownCommand;

    public Switch(Command flipUpCmd, Command flipDownCmd) {
        this.flipUpCommand = flipUpCmd;
        this.flipDownCommand = flipDownCmd;
    }

    public void flipUp() {
        flipUpCommand.execute();
    }

    public void flipDown() {
        flipDownCommand.execute();
    }
}
```

Strategy

- Objetivo
  - Definir uma família de algoritmos que podem ser intercambiáveis sendo possível alterar os algoritmos sem alterar o cliente que os utiliza
- Motivação
  - Considere a existência de vários algoritmos para uma mesma tarefa, por exemplo, ordenar uma lista de objetos
  - Uma aplicação que deve suportar diferentes algoritmos pode tornar-se complexa e no geral não permite a adição de novos que possam ser desenvolvidos
- Uso
  - Configurar o uso considerando que existem classes relacionadas com diferentes comportamentos
  - Existência de diferentes algoritmos ou métodos que podem ser implementados e que devem ser selecionados em tempo de execução
  - Impedir que o cliente tenha que manter dados específicos para diferentes algoritmos
  - Remover o uso de estruturas de seleção, como um switch, em uma operação. Neste caso cada possibilidade de código a ser executado se transforma em uma diferente estratégia

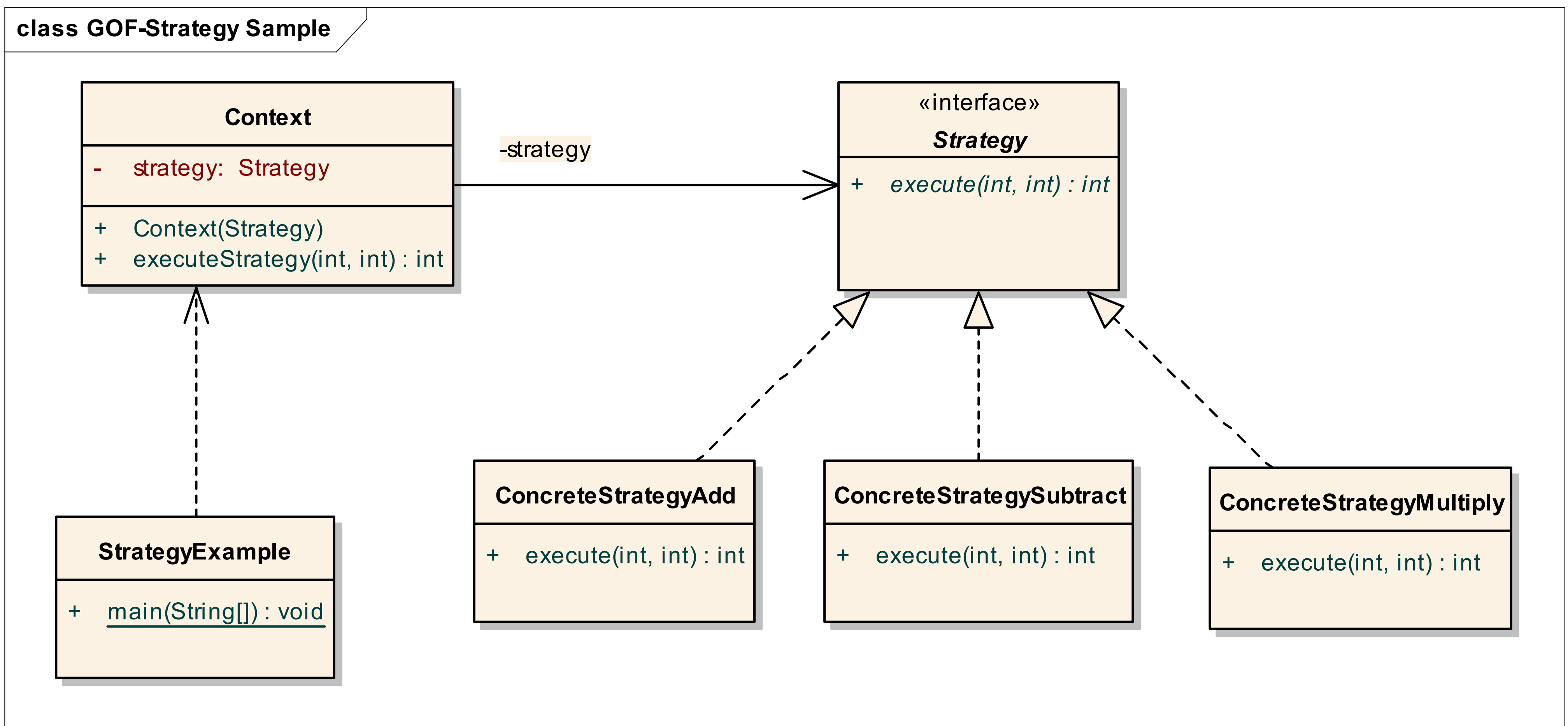
Strategy
Estrutura

- Um mesma mesma estratégia (Strategy) pode ser implementada de diversas formas (AConcreteStategy, BConcreteStategy, CConcreteStategy)
- Em um contexto (Context) contém uma estratégia genérica
- O contexto é utlizado por um cliente (Client). Uma alteração no contexto pode indicar o uso de uma nova estratégia



Strategy Exemplo

- Neste caso a estratégia permite que o cliente (StrategyExample) utilize diferentes operações aritméticas conforme o contexto (Context) utilizado



Strategy
Exemplo - Código

```
interface Strategy {
   int execute(int a, int b);
}
class ConcreteStrategyAdd implements Strategy {
    public int execute(int a, int b) {
        System.out.println("Called ConcreteStrategyAdd's execute()");
        return a + b; }
}
class ConcreteStrategySubtract implements Strategy {
    public int execute(int a, int b) {
        System.out.println("Called ConcreteStrategySubtract's execute()");
        return a - b; }
}
 class ConcreteStrategyMultiply implements Strategy {
     public int execute(int a, int b) {
        System.out.println("Called ConcreteStrategyMultiply's execute()");
        return a * b; }
}
```

Strategy Exemplo - Código

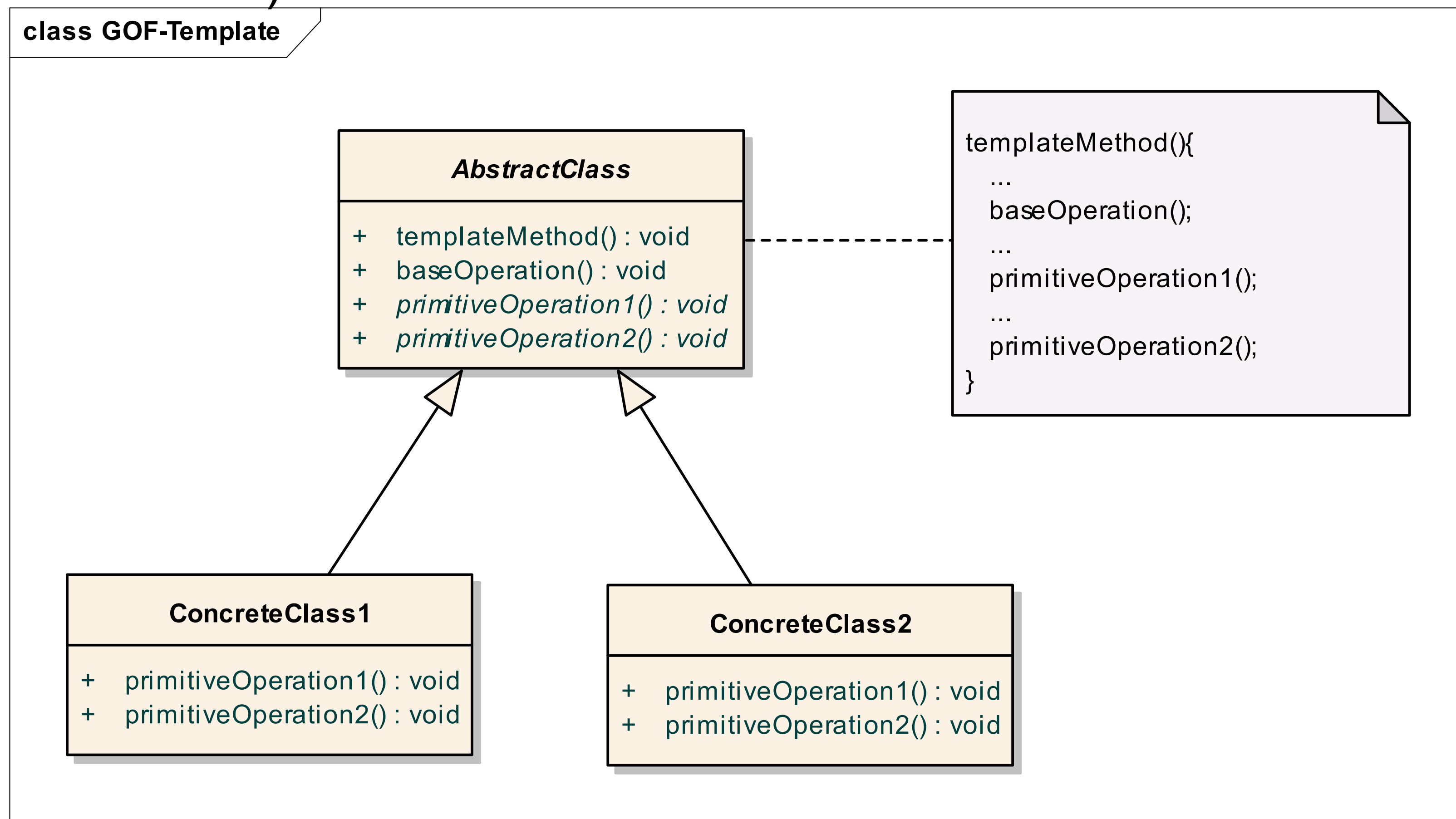
```
class Context {
    private Strategy strategy;
    // Constructor
    public Context(Strategy strategy) {
       this.strategy = strategy;  }
    public int executeStrategy(int a, int b) {
       return strategy.execute(a, b);   }
}
class StrategyExample {
    public static void main(String[] args) {
        Context context;
        context = new Context(new ConcreteStrategyAdd());
        int resultA = context.executeStrategy(3,4);
         context = new Context(new ConcreteStrategySubtract());
        int resultB = context.executeStrategy(3,4);
         context = new Context(new ConcreteStrategyMultiply());
        int resultC = context.executeStrategy(3,4);
    }
}
```

Template

- Objetivo
  - Define um modelo (template) de um algoritmo em deixando alguns passos para suas subclasses
- Motivação
- Uso
  - Implementar as partes invariantes de um algoritmo na superclasse, deixando para a subclasse o comportamento que pode variar
  - Evitar a duplicação de código, localizado o comportamento comum na superclasse
- Estrutura
- Exemplo Código

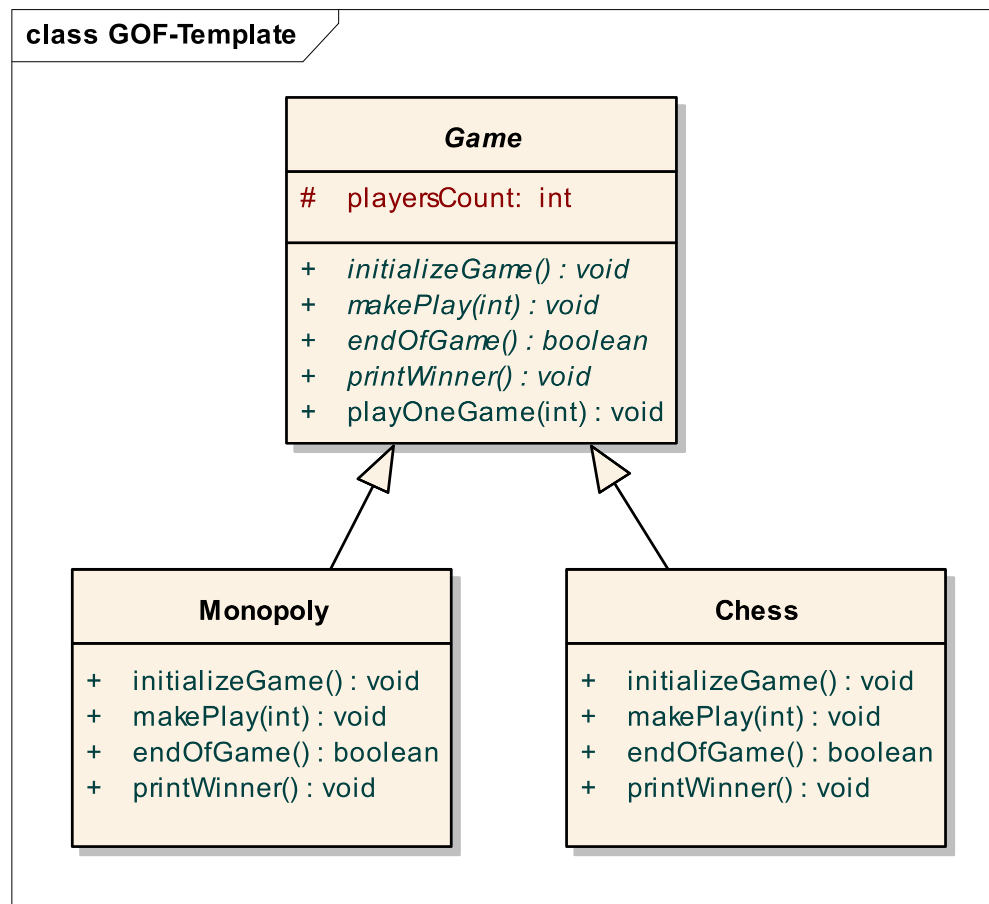
Template
Estrutura

- A classe abstrata (AbstractClass) possui um conjunto de operações e um método (templateMethod) que é responsável por invocar as operaçoes que são especializadas nas subclasses (ConcreteClass1 e ConcreteClass2)



Template
Exemplo

- Um jogo contém operações básicas que serão especializadas
- O template method aqui é playOneGame()



Template
Exemplo – Código

```
public abstract class Game {
    protected int playersCount;
   public abstract void initializeGame();
   public abstract void makePlay(int player);
   public abstract boolean endOfGame();
   public abstract void printWinner();
    /* A template method : */
   public final void playOneGame(int playersCount) {
       this.playersCount = playersCount;
       initializeGame();
       int j = 0;
       while (!endOfGame()) {
           makePlay(j);
           j = (j + 1) % playersCount;
       }
       printWinner();
   }
}
```

Template
Exemplo – Código

```
//Now we can extend this to implement other games
public class Monopoly extends Game {
    /* Implementation of necessary concrete methods */
    public void initializeGame() {
        // Initialize players
        // Initialize money
    }
    public void makePlay(int player) {
        // Process one turn of player
    }
    public boolean endOfGame() {
        // Return true if game is over
        // according to Monopoly rules
    }
    public void printWinner() {
        // Display who won
    }
}
```

Template
Exemplo – Código

```
public class Chess extends Game {
    /* Implementation of necessary concrete methods */
     public void initializeGame() {
        // Initialize players
        // Put the pieces on the board
    }
    public void makePlay(int player) {
        // Process a turn for the player
    }
    public boolean endOfGame() {
        // Return true if in Checkmate or
        // Stalemate has been reached
    }
    public void printWinner() {
        // Display the winning player
    }
}
```

Template
Exemplo – Código

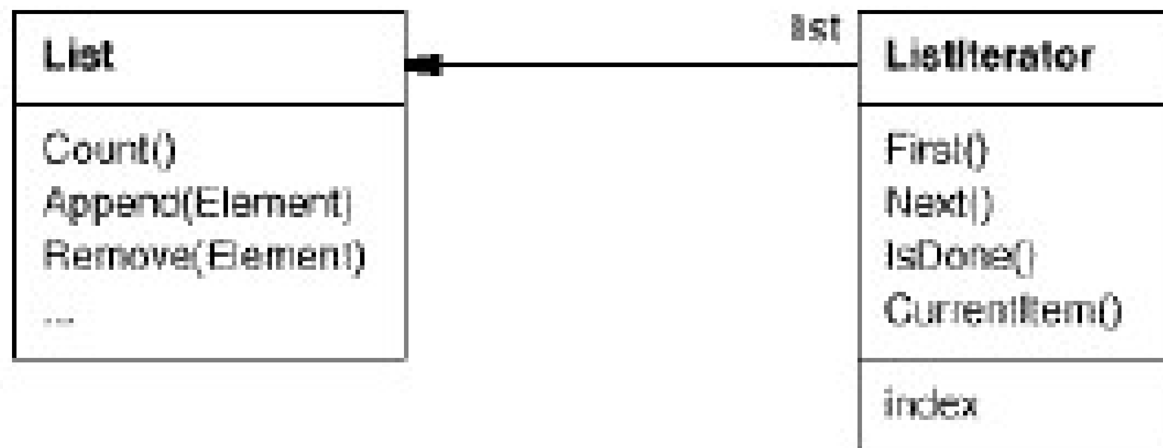
```
public class Monopoly extends Game {
    /* Implementation of necessary concrete methods */
    public void initializeGame() {
        // Initialize players
        // Initialize money
    }
    public void makePlay(int player) {
        // Process one turn of player
    }
    public boolean endOfGame() {
        // Return true if game is over
        // according to Monopoly rules
    }
    public void printWinner() {
        // Display who won
    }
}
```

Iterator

- Objetivo
  - Fornece uma maneira de acessar os componentes de um objeto sem expor a sua representação interna
- Motivação
- Uso
- Estrutura
- Exemplo Código

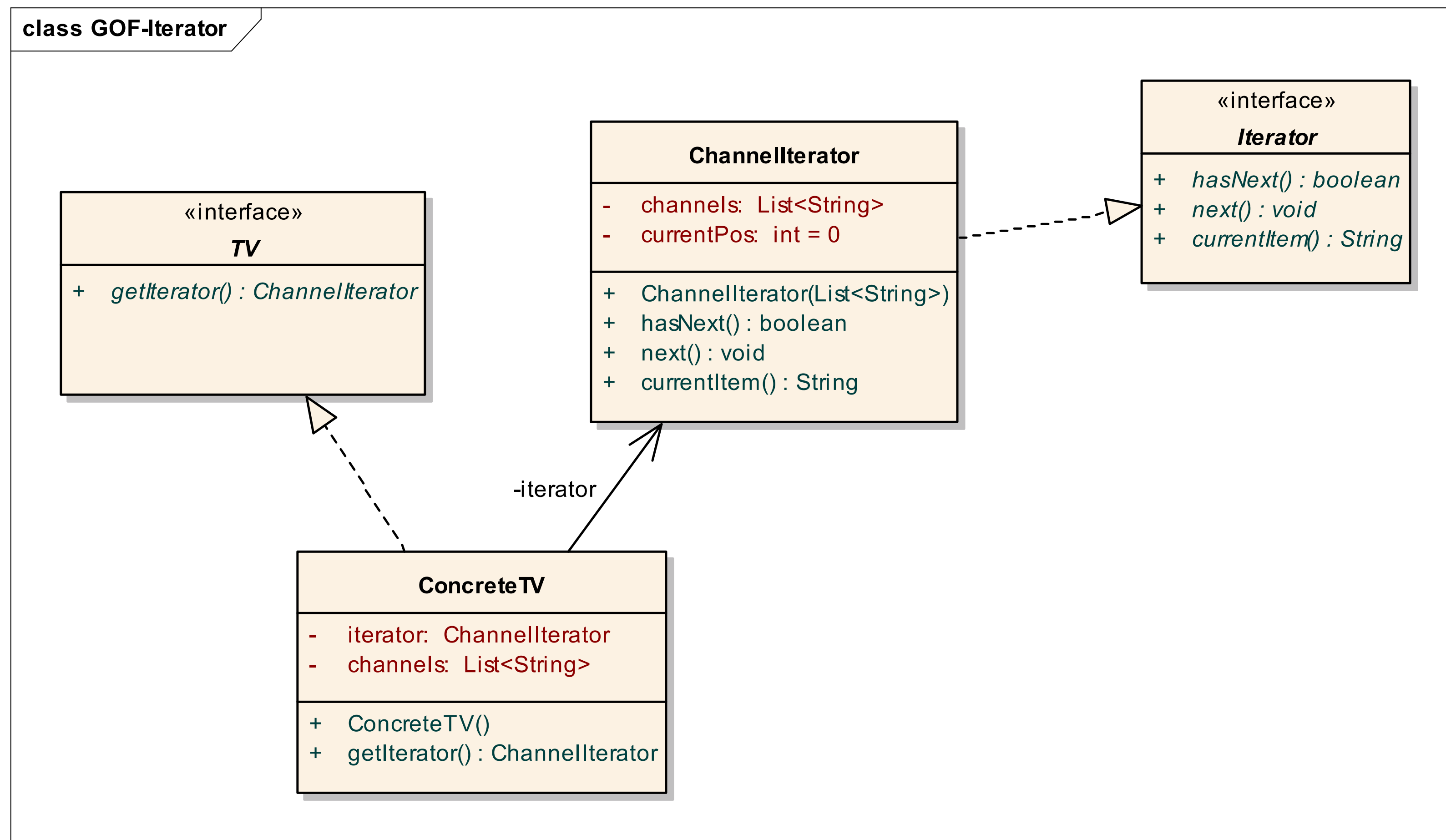
Iterator
Estrutura

- O iterator (ListIterator) permite o acesso ao objeto da lista em que seja necessário conhecer o seu interior



Iterator Exemplo

- Em uma televisão (ConcreteTV) existe um Iterator (ChannelIterator) sobre os canais desta televisão
- Idepentende de como o acesso aos canais é feito é possível obtê-los através do Iterator



Iterator
Código

```
//Iterator interface
public interface Iterator {
    public boolean hasNext();
    public void next();
    public String currentItem();

}

//Aggregate interface
public interface TV {
    public ChannelIterator getIterator();
    //other TV methods
}
```

Iterator
Código

```
//Concrete Aggregator
public class ConcreteTV implements TV{
    private ChannelIterator iterator;
    private List<String> channels;

    public ConcreteTV()        {
        iterator = new ConcreteChannelIterator(channels);
    }

    public ChannelIterator getIterator()    {
        return iterator;
    }

}
```

Iterator
Código

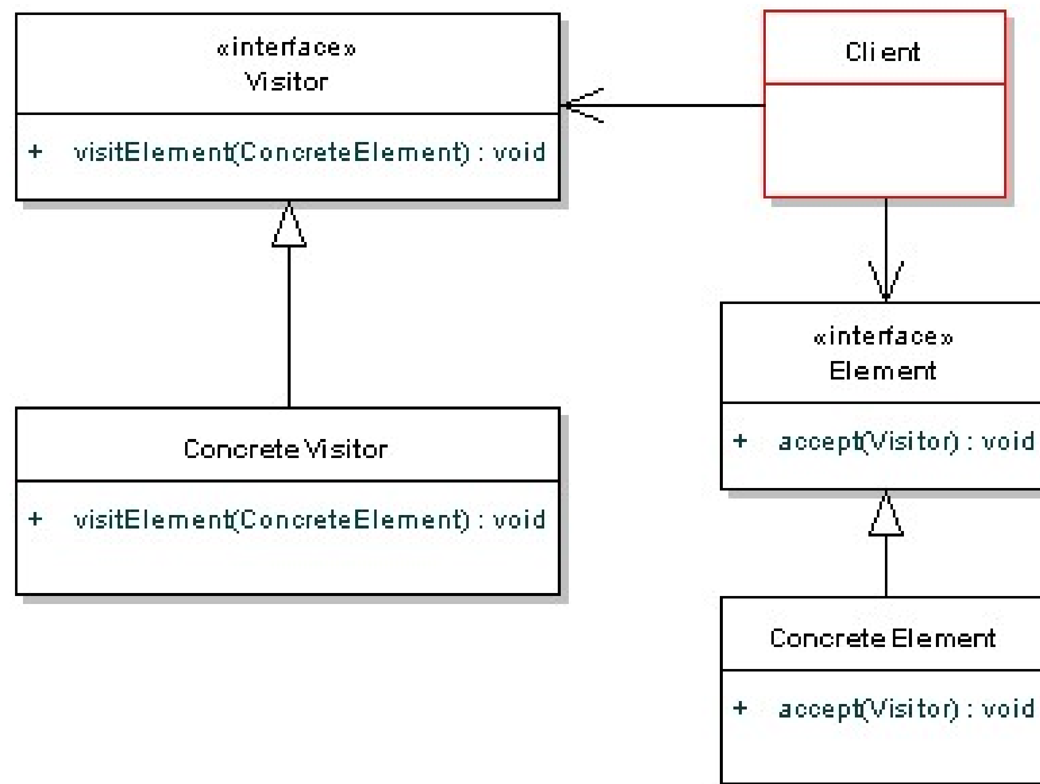
```
//Concrete Iterator
public class ChannelIterator implements Iterator {
      private List<String> channels;
      private int currentPos = 0;
      public ChannelIterator(List<String> channels){
            this.channels = channels;
      }
      public boolean hasNext(){
            if(currentPos + 1 < channels.size()){
              return true;
            }
            return false;
      }
      public void next() { currentPos++; }

      public String currentItem() { return channels.get(currentPos);
      }
}
```

Visitor

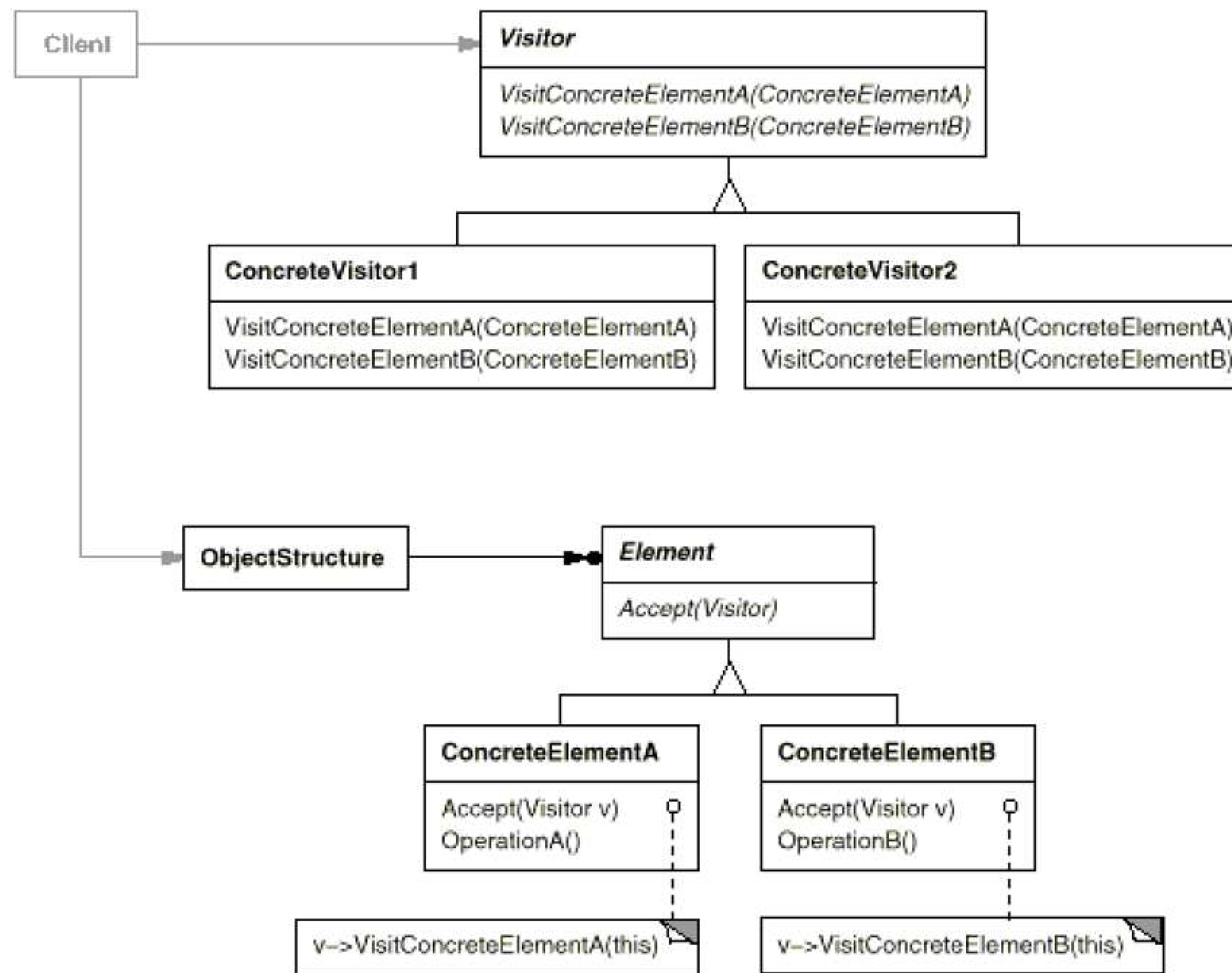
- Objetivo
  - Permite que uma ou mais operações seja aplicada a um conjunto ou estrutura de objetos, desacoplando as operações desta estrutura
  - A operação “visita” o conjunto de objetos e é executada sobre o mesmo
- Motivação
  - Para calcular o total de uma compra, diferentes operações devem ser executadas sobre o produto. Em alguns caso é necessário pesar o produto para obter seu preço, em outros é necessário apenas ler seu código de barras, em outro finalmente é aplicado um desconto por compras em quantidade
  - Ao passar no caixa, o atendente será o Visitor, que irá obter um produto e então calculará o seu preço.
- Uso
  - Desacoplar a lógica de uma operação dos objetos que são submetidos a esta o operação
  - Permitir que operações diferentes e não relacionadas seja aplicadas em uma estrutura de objetos

Visitor
Estrutura

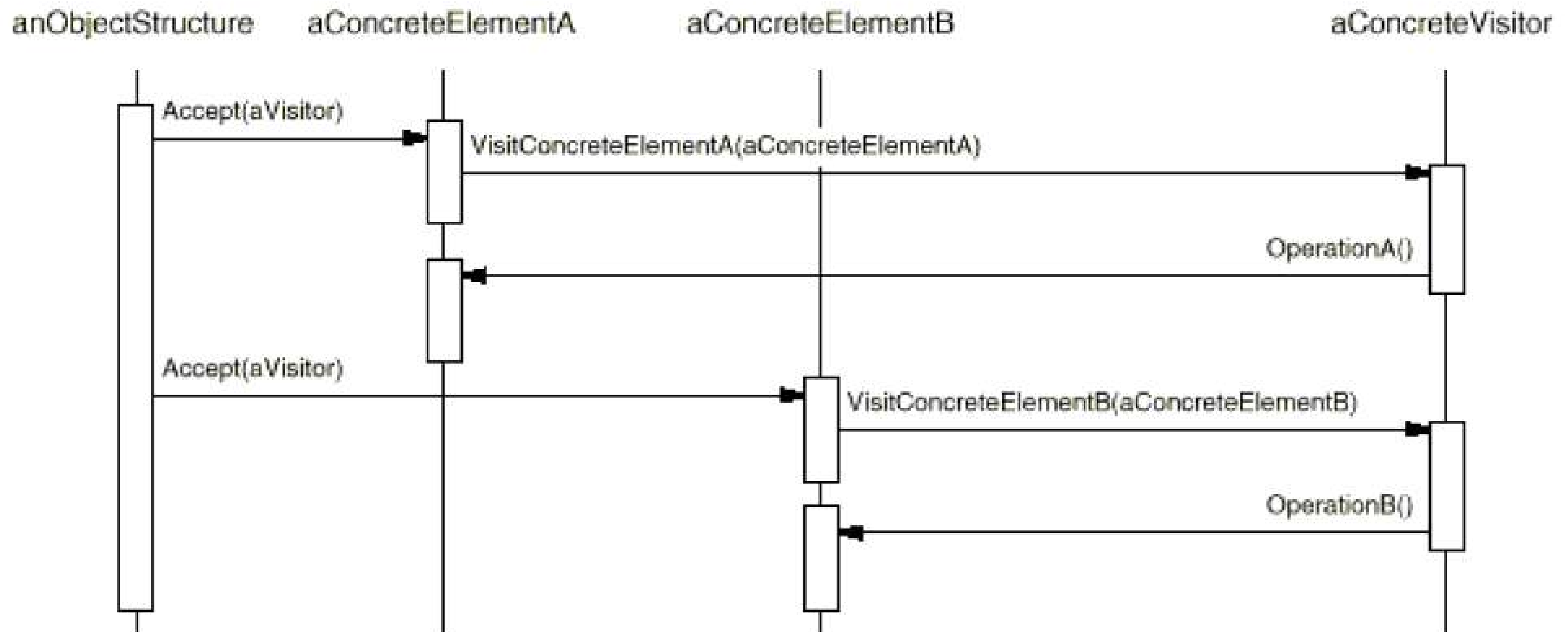
- A interface Visitor define uma operação sobre um elemento (ConcreteElement). Um elemento por sua vez possui uma operação que permite aceitar um visitante (accept), associando-se ao vistante



Visitor
Estrutura

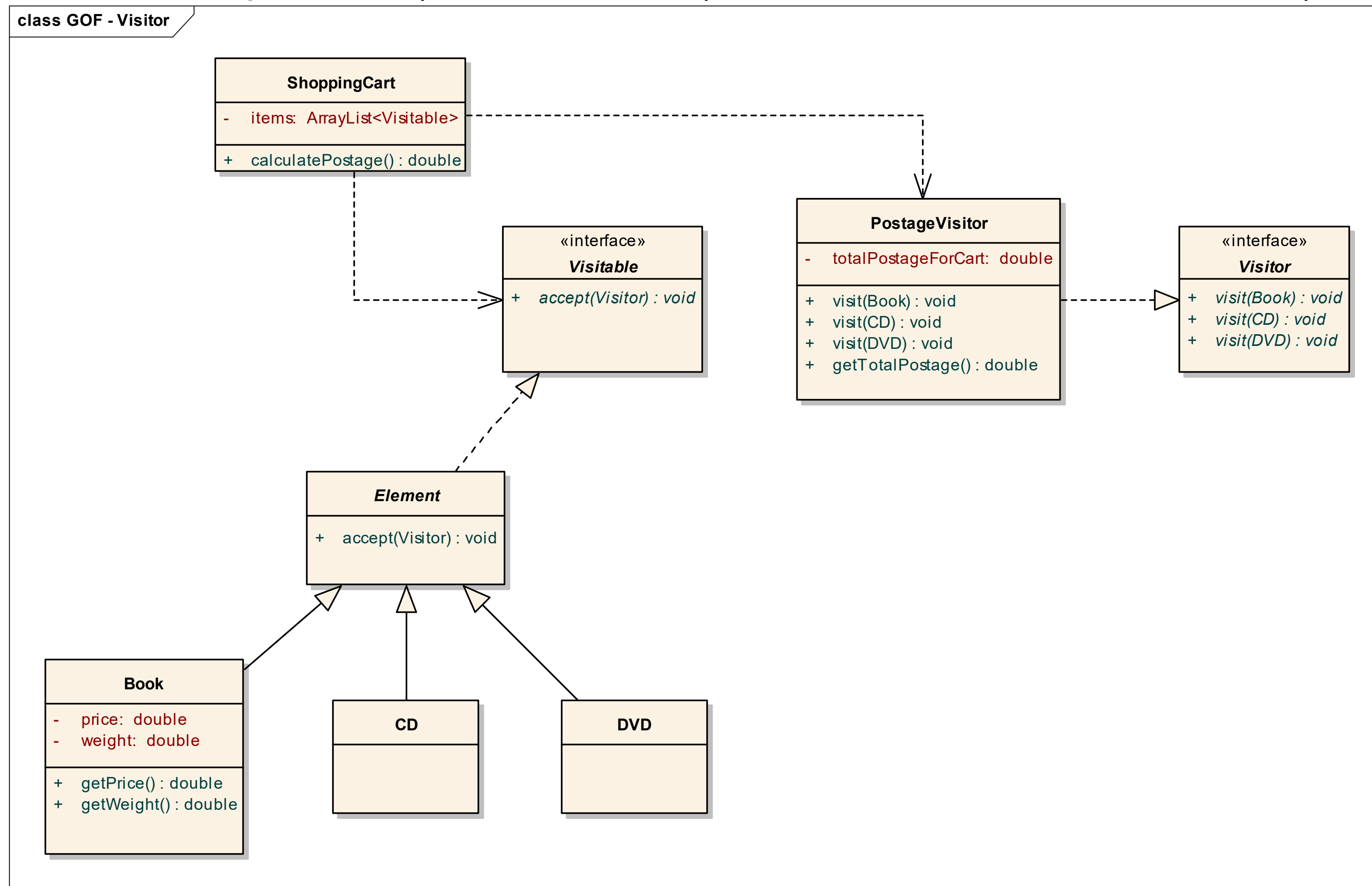


Visitor
Estrutura



Visitor
Exemplo

- Utilizando um visitor (PostageVisitor) será possível implementar o cálculo do frente para diferentes produtos (Book, CD, DVD) e então calcular o valor do frete (calculatePostage)



Visitor
Exemplo - Código

```
//Element interface
public interface Visitable {
    public void accept(Visitor visitor);
}
//abstract element
public abstract class Element implements Visitable {
     //accept the visitor
      public void accept(Visitor visitor) {
         visitor.visit(this);
      }
}
//concrete element
public class CD extends Element{
 }
//concrete element
public class DVD extends Element{
 }
```

Visitor
Exemplo - Código

```
//concrete element
public class Book extends Element {
    private double price;
    private double weight;

    public double getPrice()
    {
        return price;
    }

    public double getWeight()
    {
        return weight;
    }
}
```

Visitor Exemplo - Código

```
public interface Visitor {
   public void visit(Book book);
   //visit other concrete items
   public void visit(CD cd);
   public void visit(DVD dvd);
}
```

Visitor
Exemplo - Código

```
public class PostageVisitor implements Visitor  {
   private double totalPostageForCart;
   //collect data about the book
    public void visit(Book book)      {
      //assume we have a calculation here related to weight and price
      //free postage for a book over 10
      if(book.getPrice() < 10.0)       {
           totalPostageForCart += book.getWeight() * 2;
      }
    }
    //add other visitors here
    public void visit(CD cd){...}
    public void visit(DVD dvd){...}
    //return the internal state
    public double getTotalPostage()     {
      return totalPostageForCart;
    }
}
```

Visitor
Exemplo - Código

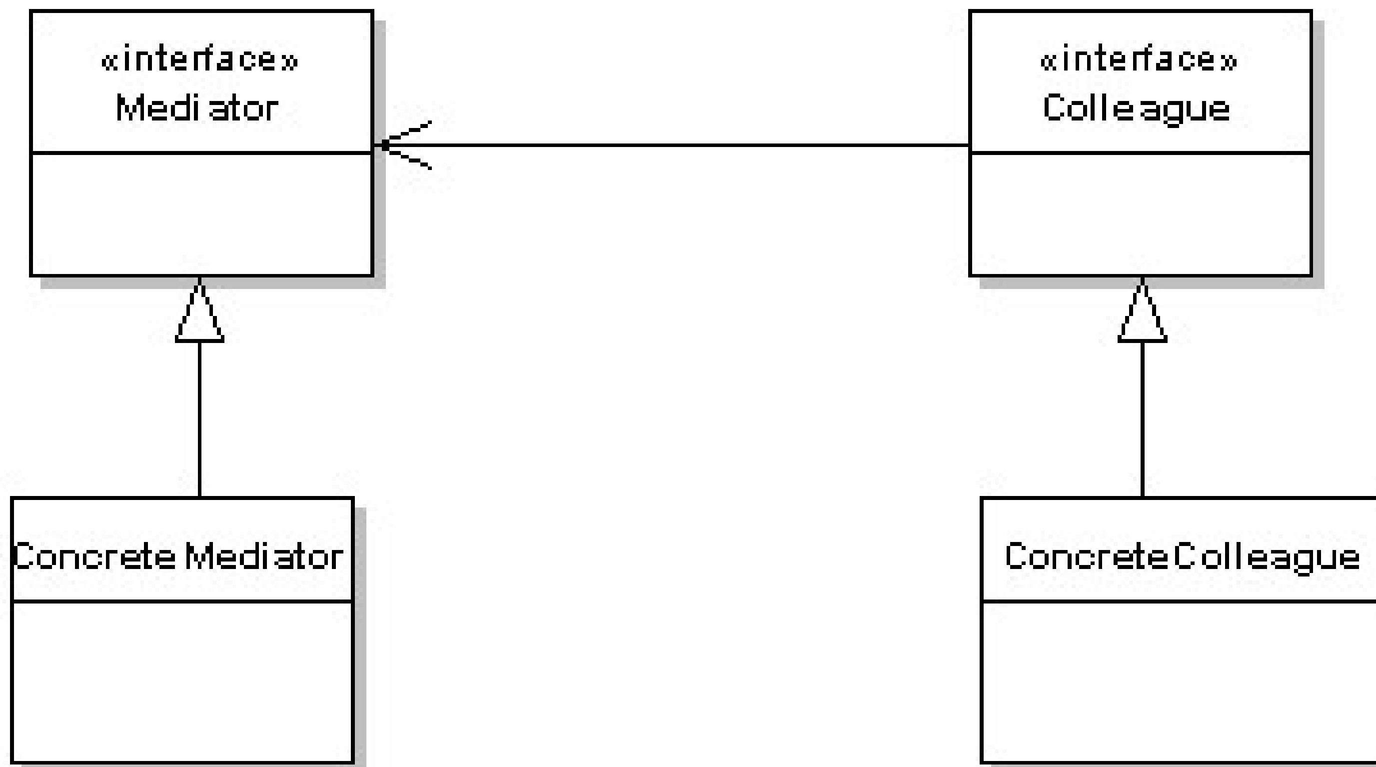
```
public class ShoppingCart {
   //normal shopping cart stuff
   private ArrayList<Visitable> items;
   public double calculatePostage()   {
      //create a visitor
      PostageVisitor visitor = new PostageVisitor();
      //iterate through all items
      for(Visitable item: items)   {
         item.accept(visitor);
      }
     double postage = visitor.getTotalPostage();
     return postage;

  }
}
```

Mediator

- Objetivo
  - Permite criar um baixo acoplamento entre um conjunto de objetos que se comunica e interage
  - O mediator é o objeto que realiza a comunicação com cada objeto permitindo que esta comunicação entre estes objetos ocorra de forma independende uma das outras
- Motivação
  - Uma torre de controle em um aeroporto é o mediator entre os aviões e o aeroporto
  - Ao invés dos aviões comunicarem entre si cada um comunica diretamente com a torre
- Uso
  - Quando existe uma comunicação complexa, porém bem definida, entre objetos
  - Em situações onde existem vários relacionamentos entres objetos produzindo uma diagrama de classes que indica alto acoplamento

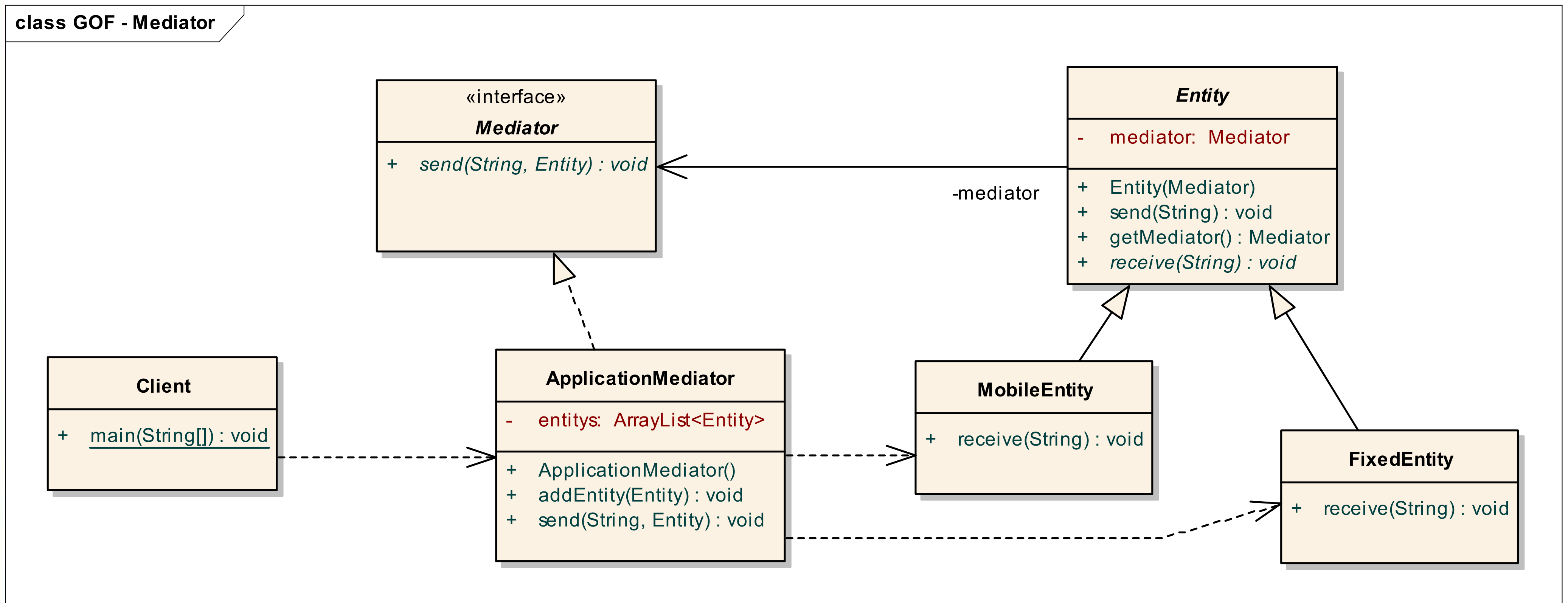
Mediator
Estrutura

- O mediator define uma interface para a comunicação entre diferentes partes (Colleage). Um mediator é então criado (ConcreteMediator) a fim de implementar o comportamento para a comunicação



Mediator Exemplo

- A comunicação entre as entidades (FixedEntity e MobileEntity) é realizada através do mediator (ApplicationMediator) associado a cada entidade



Mediator
Código

```
//Mediator interface
public interface Mediator {
    public void send(String message, Entity entity);
}
public abstract class Entity {//Entity Abstract Base Class
    private Mediator mediator;
    public Entity(Mediator m){
        mediator = m;
    }
    public void send(String message){   //send a message via the mediator
        mediator.send(message, this);
    }
    public Mediator getMediator(){  //get access to the mediator
        return mediator;
    }
    public abstract void receive(String message);

}
```

Mediator
Código

```
public class ApplicationMediator implements Mediator {
	private ArrayList<Entity> entitys;
	public ApplicationMediator(){
			entitys = new ArrayList<Entity>();
	}
	public void addEntity(Entity entity) {
			entitys.add(entity);
	}
	public void send(String message, Entity originator) {
		//let all other screens know that this screen has changed
		for(Entity entity: entitys){
				//don't tell ourselves
				if(entity != originator){
						entity.receive(message);
				}
		}
	}
}
```

Mediator
Código

```
public class FixedEntity extends Entity {
    public void receive(String message){
        System.out.println("FixedEntity Received: " + message);
    }
}

public class MobileEntity extends Entity {
    public void receive(String message) {
        System.out.println("MobileEntity Received: " + message);
    }
}
```

Mediator
Código

```
//Client shows the use
public class Client {
  public static void main(String[] args)   {
      ApplicationMediator mediator = new ApplicationMediator();

     FixedEntity desktop = new FixedEntity(mediator)
     FixedEntity mobile = new MobileEntity(mediator)

      mediator.addEntity(desktop);
      mediator.addEntity(mobile);

      desktop.send("Hello World");
      mobile.send("Hello");

  }
}
```