Padrões Estruturais

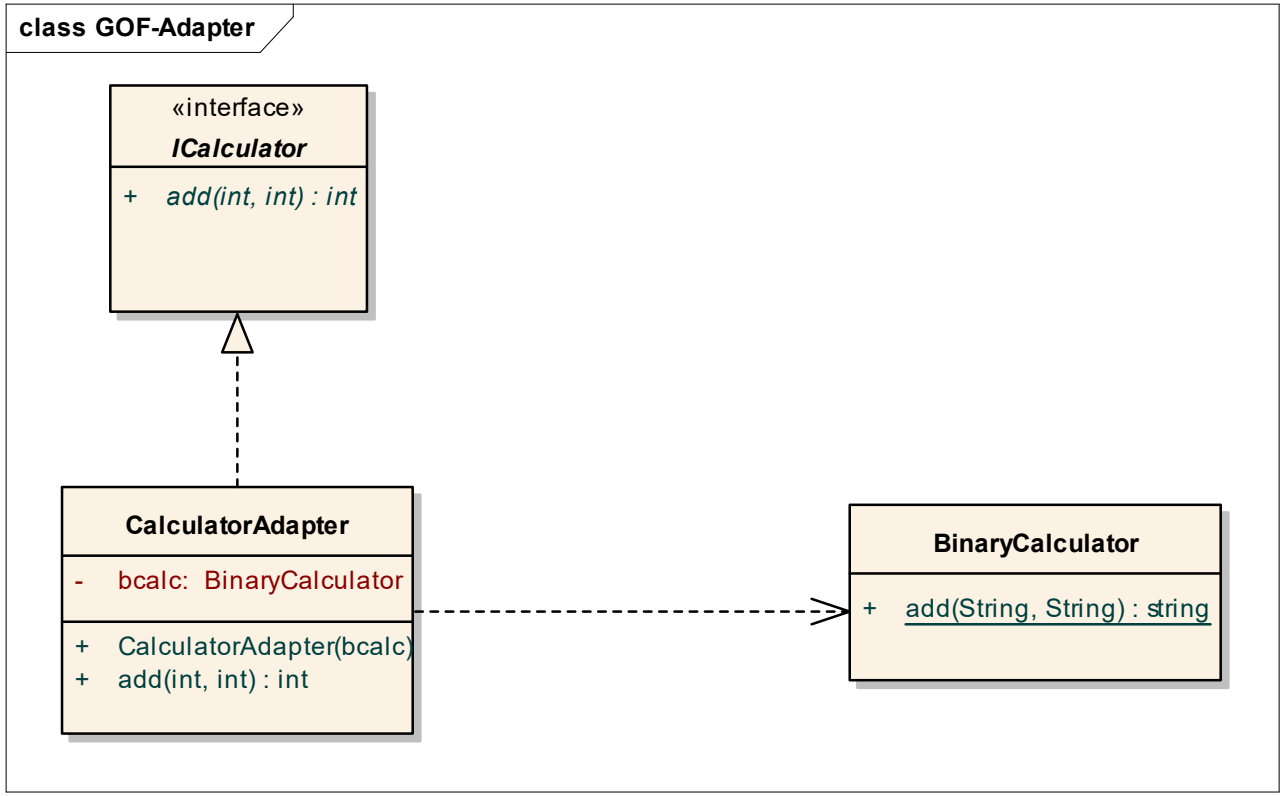
- Adapter
- Bridge
- Composite
- Decorator
- Facade
- Proxy

Adapter

- Objetivo
  - Converter uma interface de uma classe para uma interface compatível com a esperada
  - Permite que classes possam cooperar o que não seria possível pela incompatibilidade entre as interfaces
  - Traduz as chamadas de sua interface para chamadas da interface da classe adaptada
  - Também conhecida como “Wrapper”
- Uso
  - Este padrão de projeto é útil em situações onde uma classe já existente possui serviços que serão utilizados, porém não na Interface necessária
  - Por exemplo, uma classe que espera valores boleanos

Adapter
Exemplo

- Um novo método para adicionar inteiros será utilizado em uma implementação
- O código disponível porém apenas permite a adição de números binários (BinaryCalulator)
- A classe **CalculatorAdapter** permitirá o uso da implementação disponível (**BinaryCalculator**), porém adatapada o

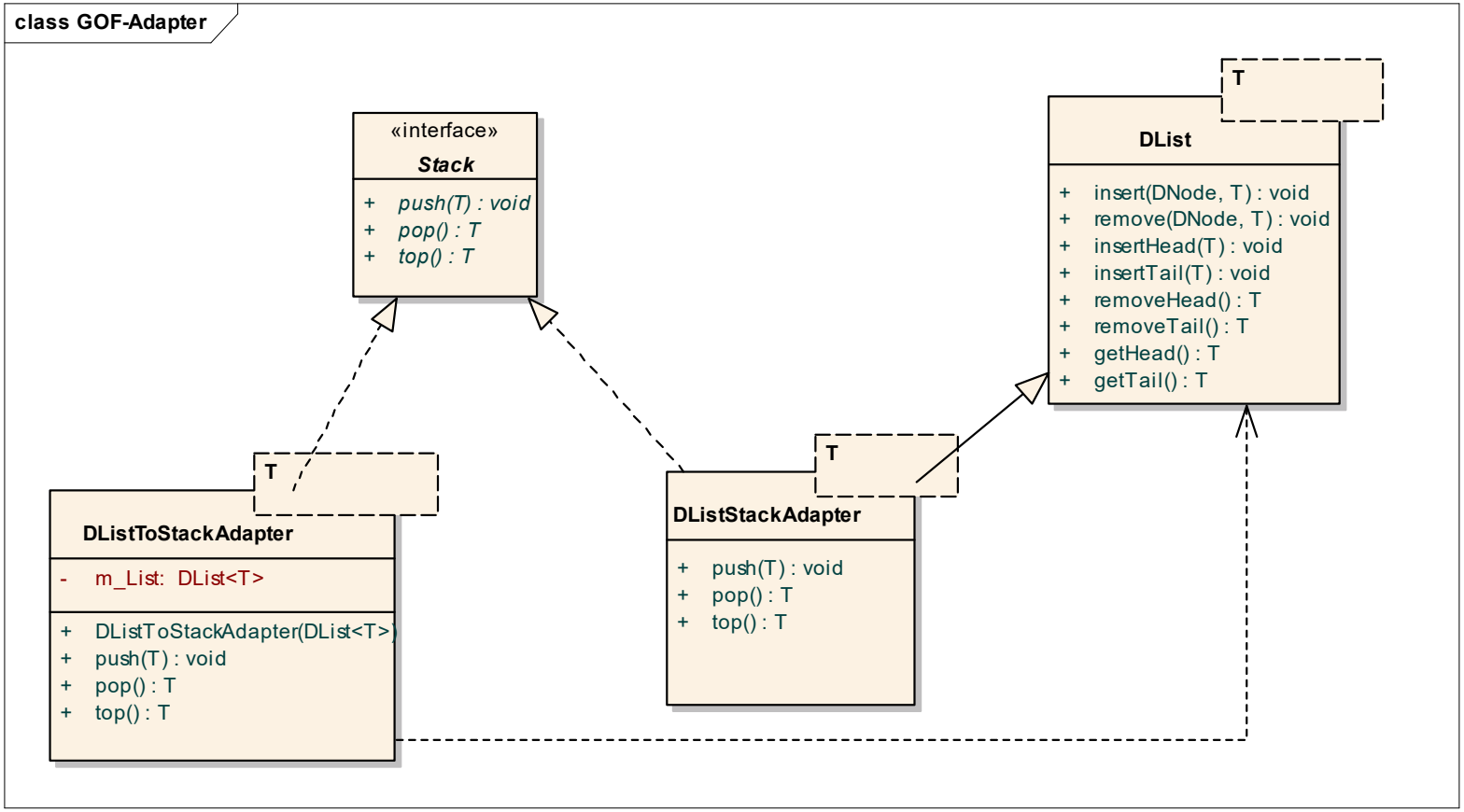


Adapter
Exemplo

```
public interface ICalculator{
    public int add(int ia , int ib);
}
public class BinaryCalculator {
    public static string add(String sa,String sb){   //…
    }
}
public class CalculatorAdapter implements ICalculator {
    private BinaryCalculator bcalc;
    public CalculatorAdapter(bcalc c){
        bcalc = c;
    }
   public int add(int ia, int ib){
        String result;
        result = bcalc.add(Integer.toBinaryString(ia),   Integer.toBinaryString(ib),
        //converts binary string to a decimal representation return is value
        return Integer.valueOf(result,10).intValue();
    }
}
```

Adapter
Outro Exemplo

- Dois exemplos de classes adptadoras. Uma baseada em uso (**DListToStackAdapter**) e outra em herança múltipla (**DListStackAdapter**)

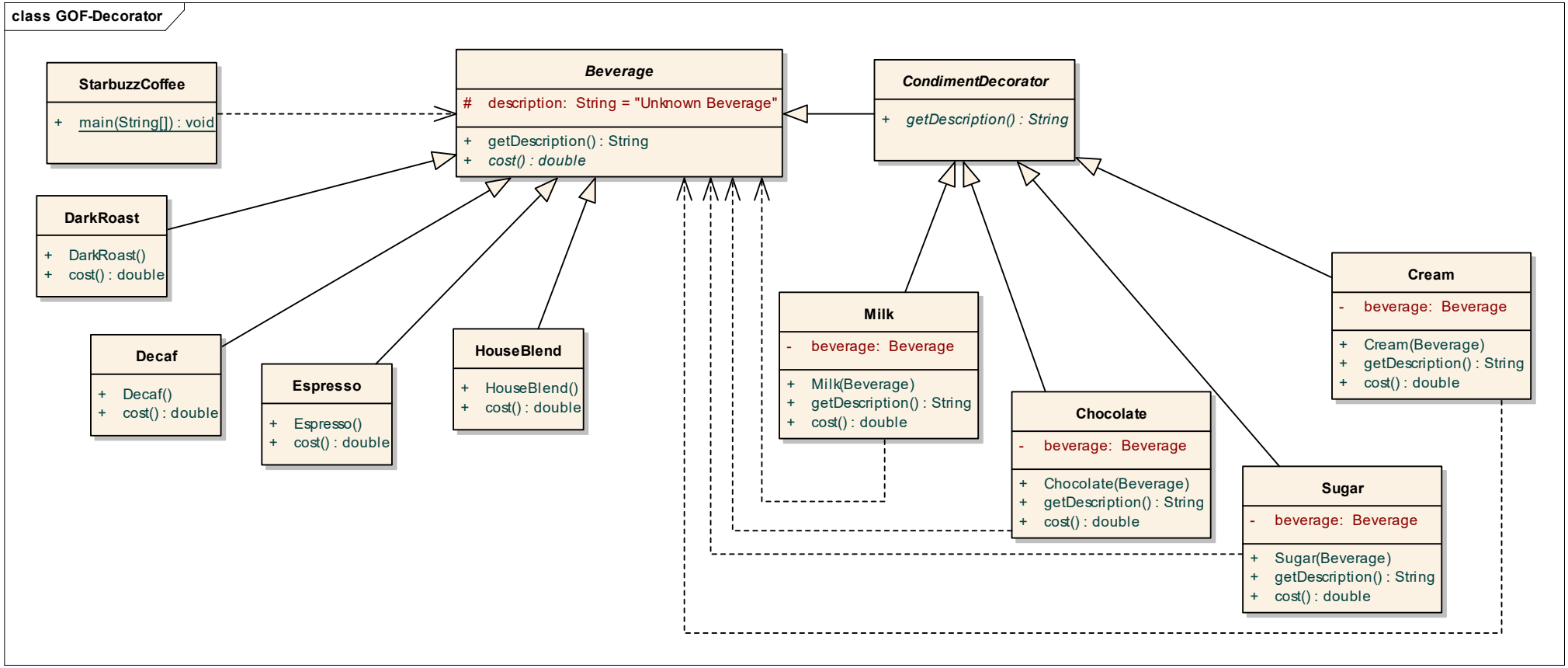


Decorator

- Objetivo
  - Permite adicionar responsabilidades a um objeto de forma dinâmica
  - Desta forma não é necessário criar subclasses a fim de estender a funcionalidade dos objetos
- Motivação
  - Em alguns casos deseja-se adicionar responsabilidades a um objeto e não a uma classe inteira
  - Exemplo
    - Considere a modelagem de um cardápio de cafés onde é possível acrescentar diversos acompanhamentos a um café
    - Como calcular o custo de cada item disponível no cardápio? Criar uma classe para cada opção não é melhor alternativa
    - Neste caso o Decorator pode ser uma opção para a modelagem

Decorator Exemplo

- Baseado Capítulo 3, do livro “Head First Design Pattern”
- Neste exemplos as bedidas (DarkRoast, Decaf, Espresso, HouseBlend) pode ser decoradas com diferentes acompanhamentos (Milk, Chocolate, Sugar, Cream)



Decorator
Exemplo - Código

```
public abstract class Beverage {
	protected String description = "Unknown Beverage“;
	public String getDescription() {
		return description;
	}
	public abstract double cost();
}
public abstract class CondimentDecorator extends Beverage {
	public abstract String getDescription();
}
```

Decorator
Exemplo - Código

```
public class DarkRoast extends Beverage {
      public DarkRoast() { description = "Dark Roast Coffee“;    }
      public double cost() { return .99; }
}
public class Decaf extends Beverage {
      public Decaf() {description = "Decaf Coffee";  }
      public double cost() { return 1.05;  }
}

public class Espresso extends Beverage {
      public Espresso() { description = "Espresso“; }
      public double cost() { return 1.99; }
}
public class HouseBlend extends Beverage {
      public HouseBlend() { description = "House Blend Coffee";  }
      public double cost() { return .89; }
}
```

Decorator
Exemplo - Código

```
public class Milk extends CondimentDecorator {
      private Beverage beverage;
      public Milk(Beverage beverage) { this.beverage = beverage;  }
      public String getDescription() {return beverage.getDescription() + ", Milk";  }
      public double cost() { return .10 + beverage.cost();  }
}
public class Chocolate extends CondimentDecorator {
      private Beverage beverage;
      public Chocolate(Beverage beverage) {  this.beverage = beverage; }
      public String getDescription() { return beverage.getDescription() + ", Chocolate“;  }
      public double cost() { return .20 + beverage.cost();  }
}
public class Sugar extends CondimentDecorator {
      private Beverage beverage;
      public Sugar(Beverage beverage) {   this.beverage = beverage;  }
      public String getDescription() { return beverage.getDescription() + ", Sugar";   }
      public double cost() { return .15 + beverage.cost(); }
}
```

Decorator
Exemplo - Código

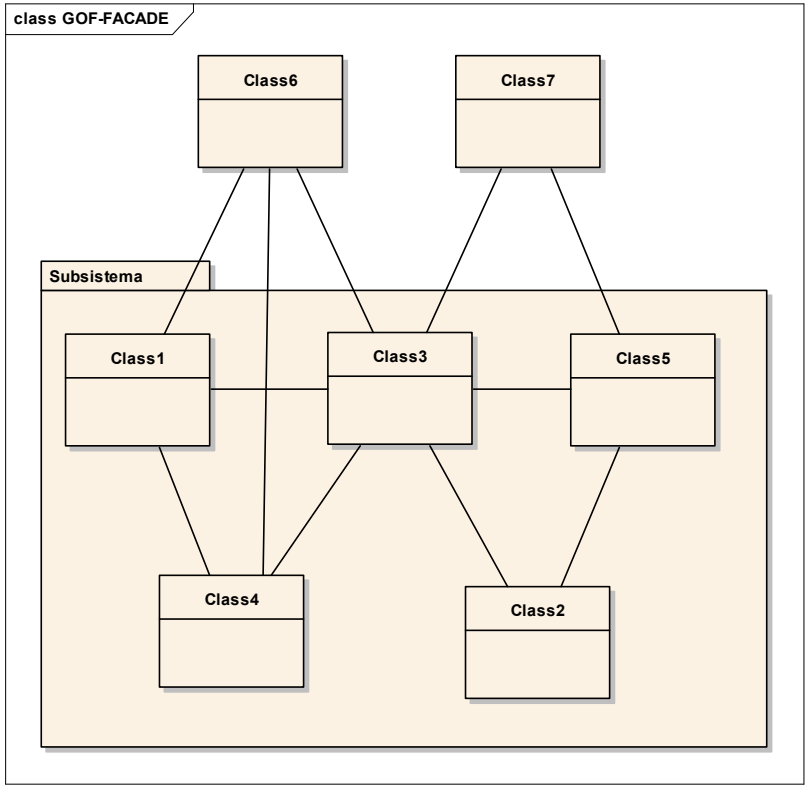
```
public class Cream extends CondimentDecorator {
	private Beverage beverage;

	public Cream(Beverage beverage) {
		this.beverage = beverage;
	}

	public String getDescription() {
		return beverage.getDescription() + ", Cream";
	}

	public double cost() {
		return .10 + beverage.cost();
	}
}
```

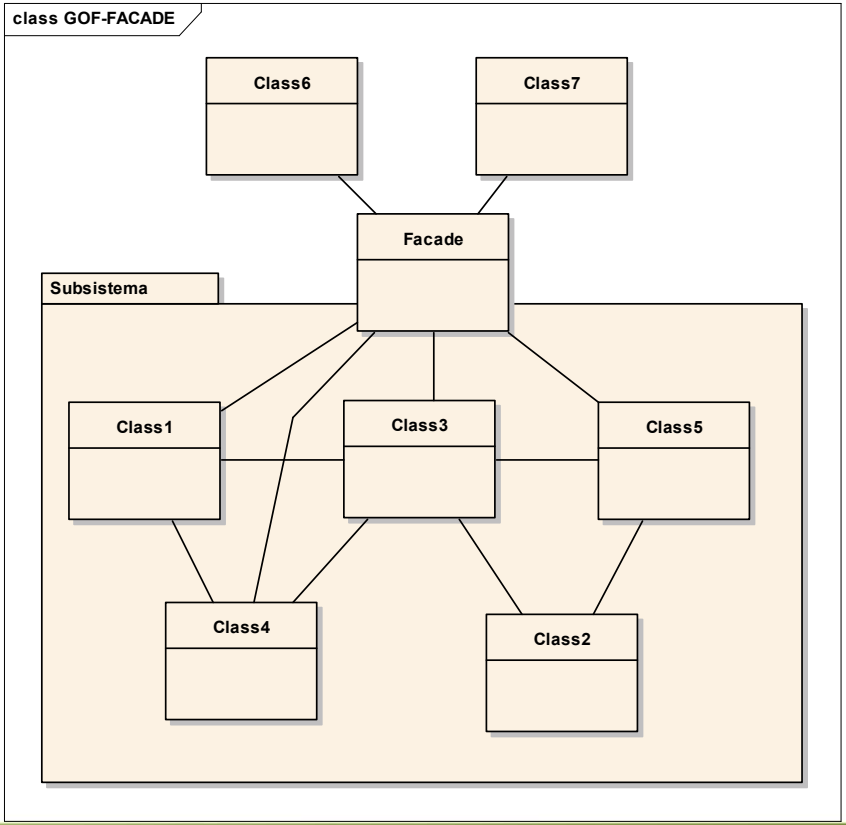
Decorator
Exemplo - Código

```
public class StarbuzzCoffee {
    public static void main(String args[]) {
        Beverage beverage = new Espresso();
        System.out.println(beverage.getDescription()  + " $" + beverage.cost());
        Beverage beverage2 = new DarkRoast();
        beverage2 = new Chocolate(beverage2);
        beverage2 = new Milk(beverage2);
        beverage2 = new Cream(beverage2);
        System.out.println(beverage2.getDescription()+ " $" + beverage2.cost());

        Beverage beverage3 = new HouseBlend();
        beverage3 = new Sugar(beverage3);
        beverage3 = new Chocolate(beverage3);
        beverage3 = new Cream(beverage3);
        System.out.println(beverage3.getDescription()  + " $" + beverage3.cost());
    }
}
```

Facade

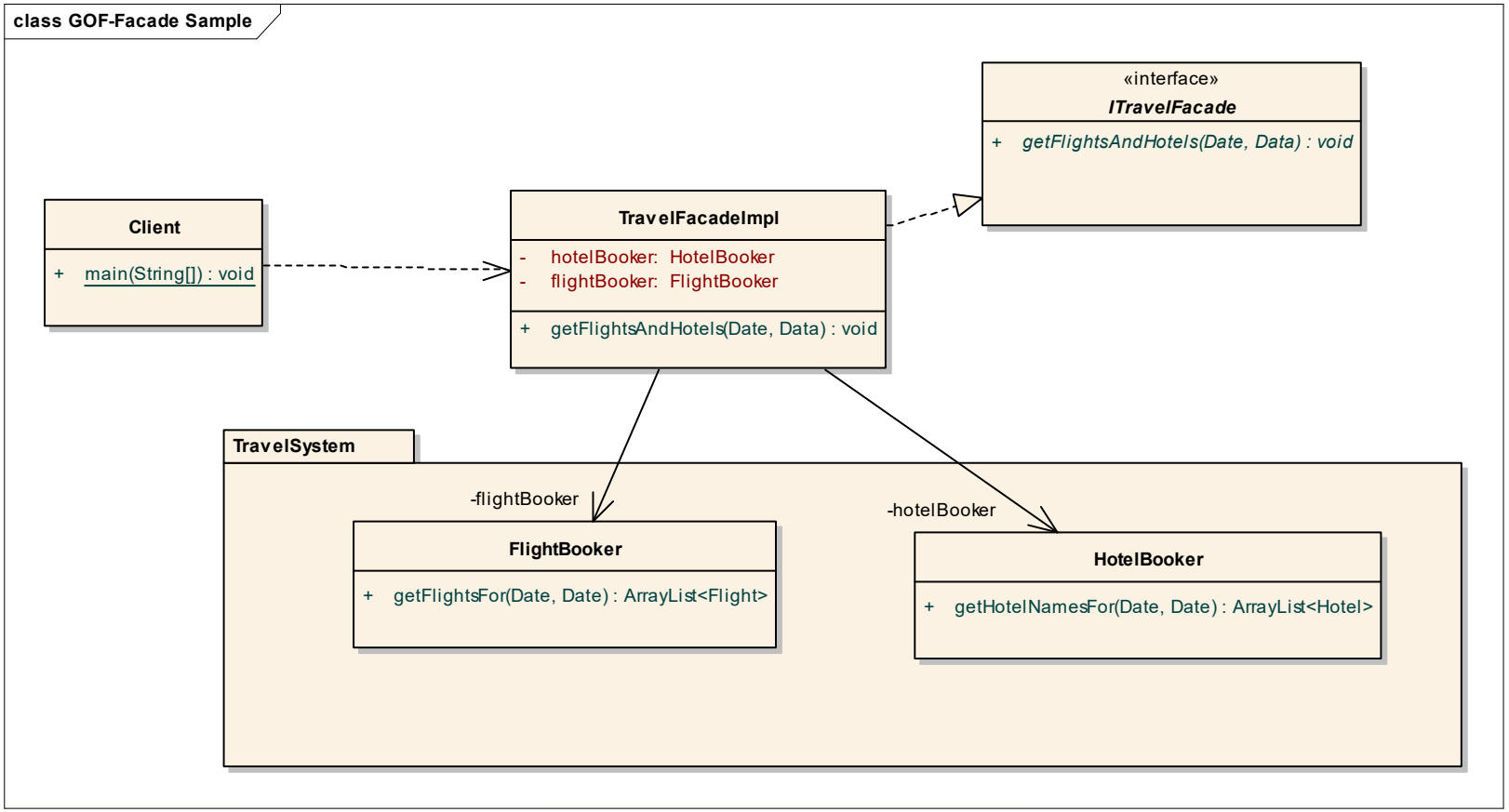
- Objetivo
  - Fornece uma interface comum para um grupo de classes de um subsistema, facilitando o seu uso
- Motivação
  - Reduzir o acomplamento entre sistemas





Facade
Exemplo - UML

- A facade (TravelFacade) esconde a complexidade para integrar com o sistema de viagem (TravelSystem)
- Facade implementa todo o código necessário para integrar com o sistema
- O ideal é definir a fachada como uma interface e então criar sua implementação concreta



Façade
Exemplo - Código

```
public class HotelBooker{
       public ArrayList<Hotel> getHotelNamesFor(Date from, Date to) {
        //returns hotels available in the particular date range
      }
}
public class FlightBooker{
      public ArrayList<Flight> getFlightsFor(Date from, Date to){
      //returns flights available in the particular date range
      }
}

public interface ITravelFacade {
  public void getFlightsAndHotels(Date from, Data to);
}
```

Façade
Exemplo - Código

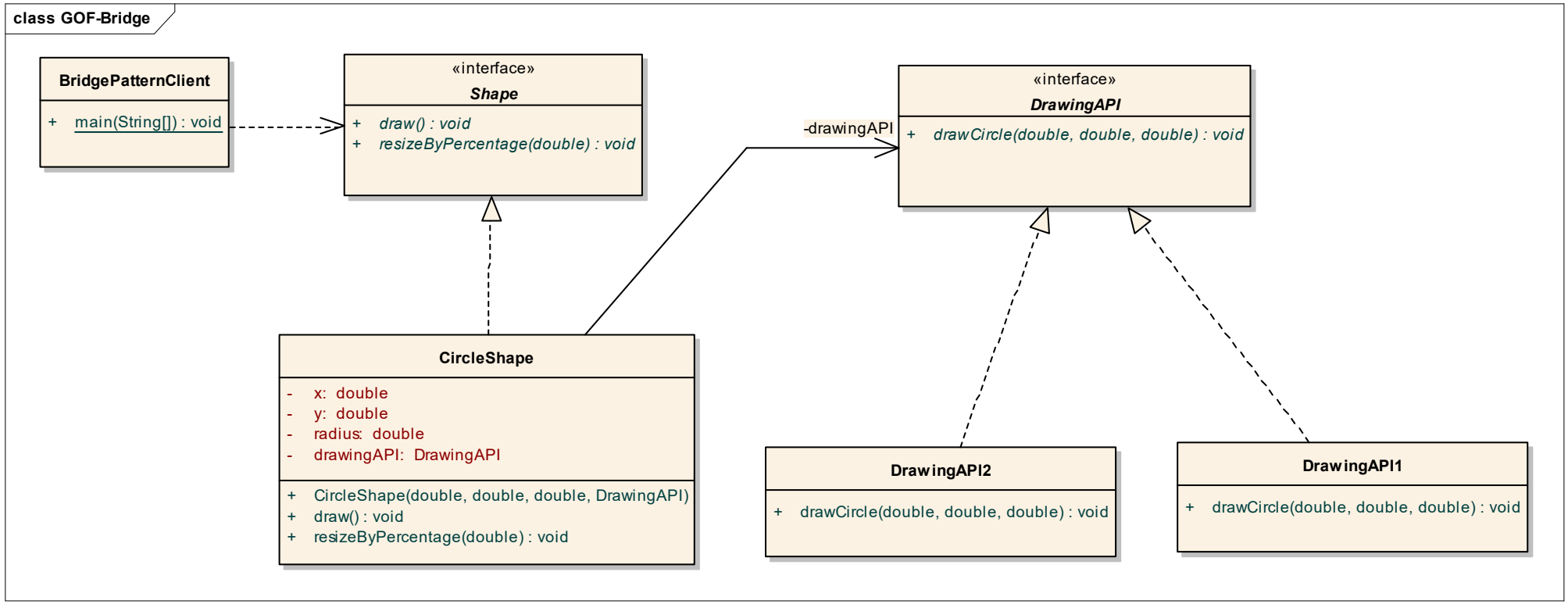
```
public class TravelFacadeImpl implements ITravelFacade{
      private HotelBooker hotelBooker;
      private FlightBooker flightBooker;
      public void getFlightsAndHotels(Date from, Data to)        {
            ArrayList<Flight> flights = flightBooker.getFlightsFor(from, to);
            ArrayList<Hotel> hotels = hotelBooker.getHotelsFor(from, to);
         //process and return
      }
}

public class Client {
      public static void main(String[] args) {
        TravelFacadeImpl facade = new TravelFacade();
        facade.getFlightsAndHotels(from, to);
      }
}
```

Bridge

- Objetivo
  - Desacoplar a abstração da implementação
  - Cada um pode ser estendido de forma independente
- Motivação
  - Quando é possível a presença de mais de uma implementação para uma determinada abstração
- Aplicação
  - Evitar uma ligação forte entre a abstração e a implementação
  - Permitir que uma implementação seja escolhida em tempo de execução

Bridge
Exemplo

- Abstração (Shape) e implementação (DrawingAPI) podem evoluir de forma independente



Bridge
Exemplo - Código

```
/** "Implementor" */
interface DrawingAPI {
    public void drawCircle(double x, double y, double radius);
}

/** "ConcreteImplementor" 1/2 */
class DrawingAPI1 implements DrawingAPI {
    public void drawCircle(double x, double y, double radius) {
        System.out.printf("API1.circle at %f:%f radius %f\n", x, y, radius);
    }
}

/** "ConcreteImplementor" 2/2 */
class DrawingAPI2 implements DrawingAPI {
    public void drawCircle(double x, double y, double radius) {
        System.out.printf("API2.circle at %f:%f radius %f\n", x, y, radius);
    }
}

/** "Abstraction" */
```

Bridge
Exemplo - Código

```
/** "Implementor" */
interface DrawingAPI {
    public void drawCircle(double x, double y, double radius);
}
 /** "ConcreteImplementor" 1/2 */
class DrawingAPI1 implements DrawingAPI {
   public void drawCircle(double x, double y, double radius) {
        System.out.printf("API1.circle at %f:%f radius %f\n", x, y, radius);
   }
}
 /** "ConcreteImplementor" 2/2 */
class DrawingAPI2 implements DrawingAPI {
   public void drawCircle(double x, double y, double radius) {
        System.out.printf("API2.circle at %f:%f radius %f\n", x, y, radius);
   }
}
```

Bridge
Exemplo - Código

```
interface Shape {/** "Abstraction" */
   public void draw();                                  // low-level
   public void resizeByPercentage(double pct);          // high-level
}
 class CircleShape implements Shape {/** "Refined Abstraction" */
   private double x, y, radius;
   private DrawingAPI drawingAPI;
   public CircleShape(double x, double y, double radius, DrawingAPI drawingAPI) {
       this.x = x;  this.y = y;  this.radius = radius;
       this.drawingAPI = drawingAPI;
   }
   public void draw() {// low-level i.e. Implementation specific
        drawingAPI.drawCircle(x, y, radius);
   }
   public void resizeByPercentage(double pct) {// high-level i.e. Abstraction specific
        radius *= pct;
   }
}
```

Bridge
Exemplo - Código

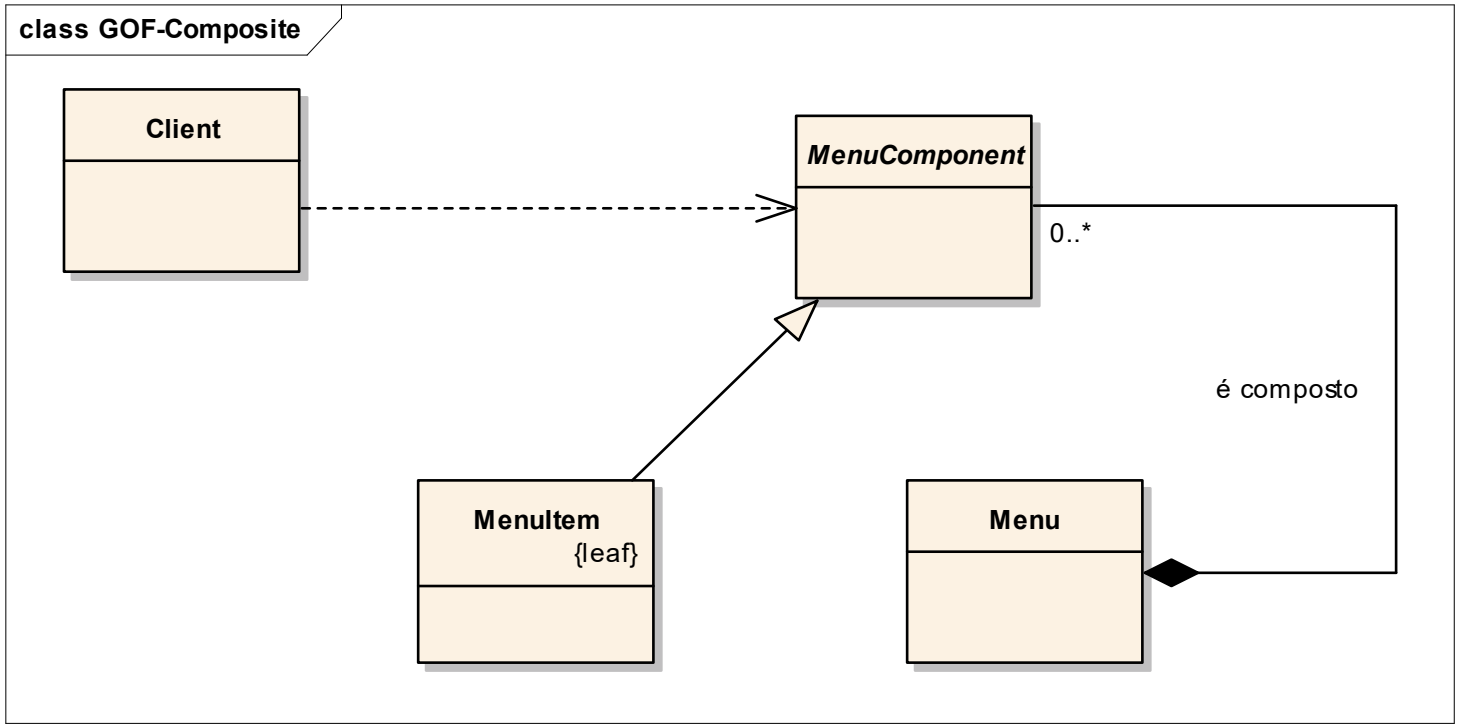
```
/** "Client" */
class BridgePatternClient {
   public static void main(String[] args) {
      Shape[] shapes = new Shape[] {
         new CircleShape(1, 2, 3, new DrawingAPI1()),
         new CircleShape(5, 7, 11, new DrawingAPI2()),
      };

      for (Shape shape : shapes) {
         shape.resizeByPercentage(2.5);
         shape.draw();
      }
   }
}
```

Composite

- Objetivo
  - Compor objetos de forma que partes e estruturas, formadas por estas partes possam ser tratadas de maneira uniforme
- Motivação
  - Em muitas situações objetos podem ser compostos para gerar outros objetos
  - Caso o código trate as partes e os objetos compostos de forma diferenciada acarretando em uma aplicação mais complexa
- Uso
  - Representar hierarquias do tipo “todo-parte” de objetos
  - Tratar tanto objetos individuais, quanto composições destes objetos de maneira uniforme

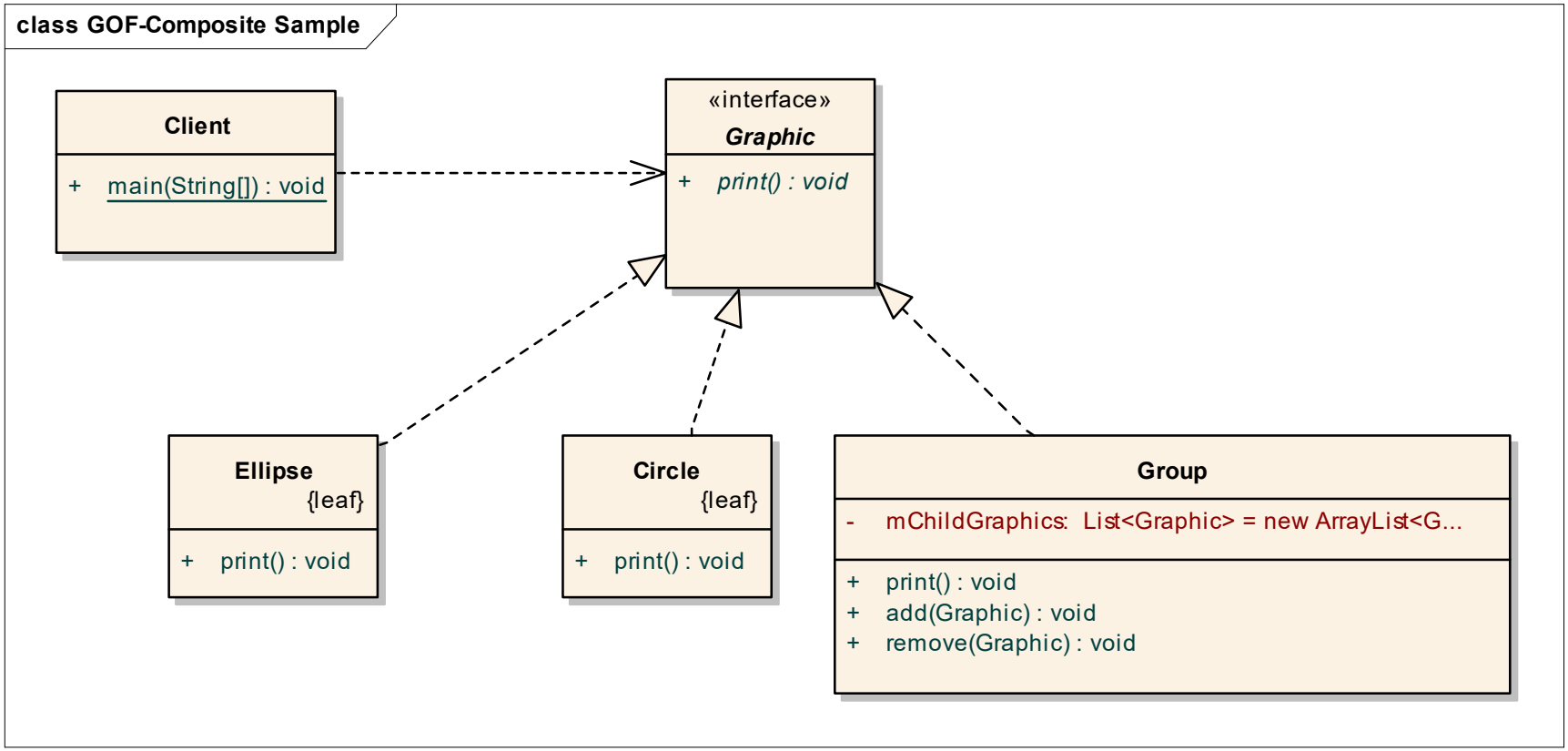
Composite
Exemplo

- O diagrama mostra o uso deste padrão de projeto
  - Um Menu é composto de elementos MenuItem, porém o tratamento do todo (Menu) como das suas partes (MenuItem) será mesmo pois ambos representam herança de MenuComponent



Composite Exempo

- Aplicação Gráfica
  - Uma Ellipse ou um Circle é um elemento gráfico
  - É possível ter grupos de elementos gráficos que devem ser tratados de maneira uniforme



Composite
Exemplo – Código

```
import java.util.List;
import java.util.ArrayList;
interface Graphic {/** "Component" */
    //Prints the graphic.
    public void print();
}
public final class Ellipse implements Graphic {/** "Leaf" */
    public void print() {
        System.out.println("Ellipse");
    }
}
public final class Circle implements Graphic {/** "Leaf" */
    public void print() {
        System.out.println("Circle");
    }
}
```

Composite
Exemplo – Código

```
class Group implements Graphic {/** "Composite" */
    //Collection of child graphics.
    private List<Graphic> mChildGraphics = new ArrayList<Graphic>();
    public void print() {
        for (Graphic graphic : mChildGraphics) {
            graphic.print();
        }
    }
    public void add(Graphic graphic) {//Adds the graphic to the composition.
        mChildGraphics.add(graphic);
    }
    public void remove(Graphic graphic) {//Removes the graphic from the composition.
        mChildGraphics.remove(graphic);
    }
}
```

Composite
Exemplo – Código

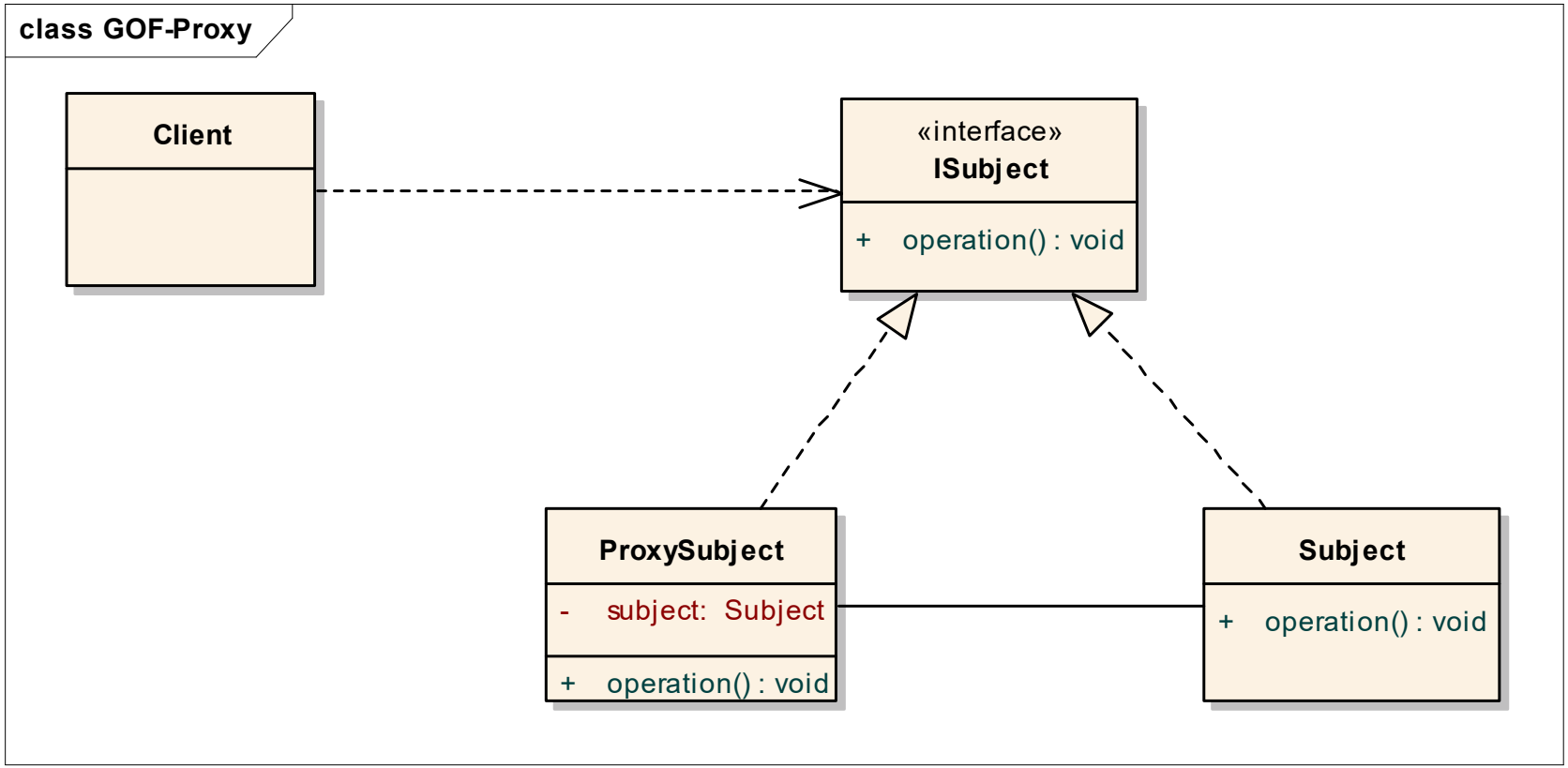
```
public class Client {/** Client */
    public static void main(String[] args) {
        Ellipse ellipse1 = new Ellipse(); //Initialize single graphics
        Ellipse ellipse2 = new Ellipse();
        Circle circle1 = new Circle ();
        Circle circle2 = new Circle ();
        Group = new Group(); //Initialize three composite graphics
        Group graphic1 = new Group();
        Group graphic2 = new Group();
        graphic1.add(ellipse1); //Composes the graphics
        graphic1.add(ellipse2);
        graphic1.add(circle1);
        graphic2.add(circle2);
        graphic.add(graphic1);
        graphic.add(graphic2);
         //Prints the complete graphic components
        graphic.print();
    }
}
```

Proxy

- Objetivo
  - Fornece um substituto para um outro objeto permitindo o controle de acesso a este objeto
- Motivação
  - Em muitas situações é necessário controlar o acesso a um objeto a fim de adiar o custo de sua criação e inicialização
  - Por exemplo:
    - Um documento que contém várias fotos. Abrir todos os arquivos de forma simultânea poderia ser ineficiente
    - O proxy fornece uma representação para a imagem e sua exibição somente ocorrerá no momento em que a página que contiver a respectiva foto seja exibida
- Uso
  - Proxy Remoto – Oferece uma representação local para um objeto remoto, por exemplo, em outro espaço de endereçamento
  - Proxy Virtual – Oferece uma representação para um objeto cuja criação será feita por demanda
  - Protection Proxy – Controla o acesso a um determinado objeto

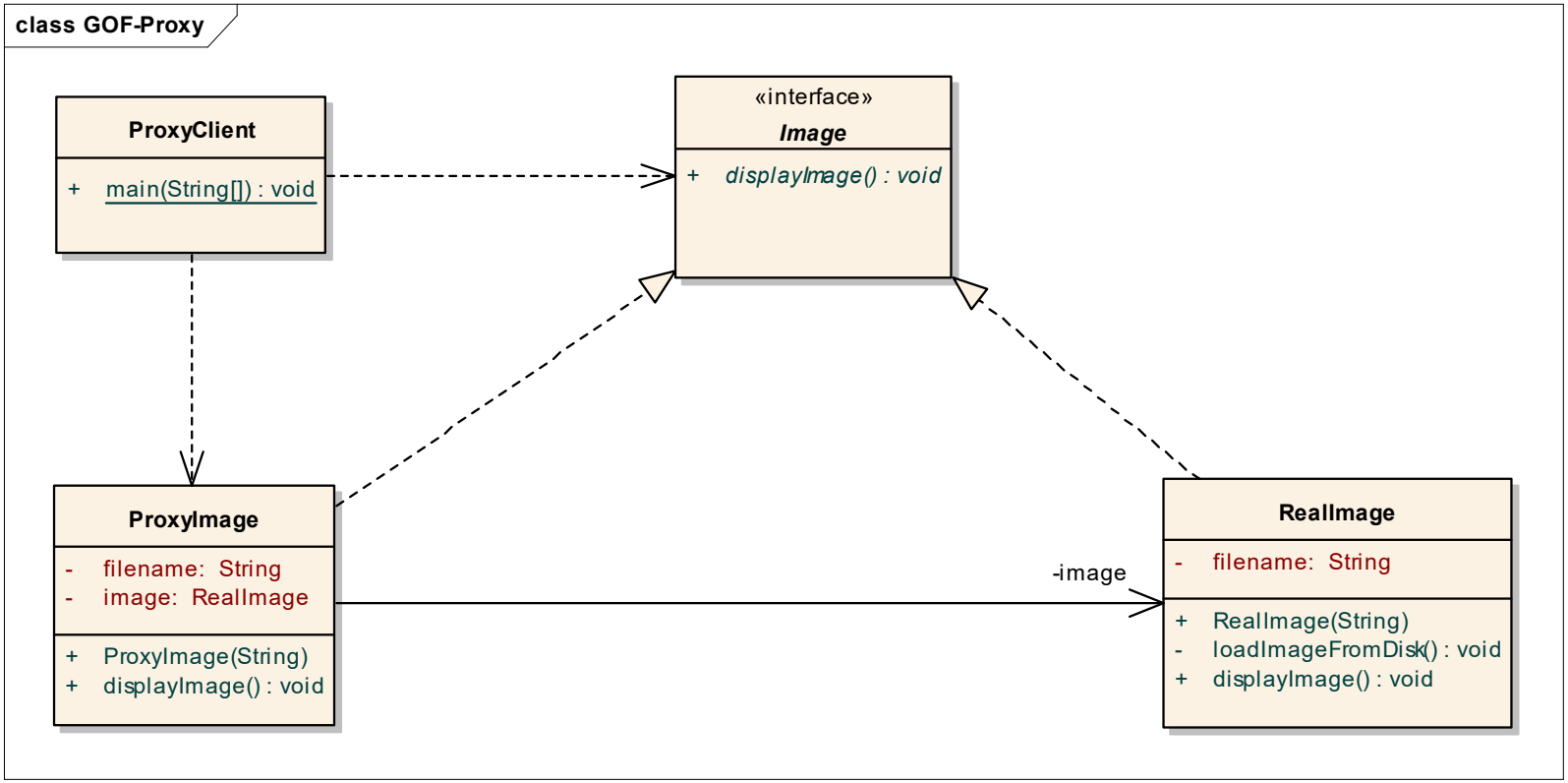
Proxy
Estrutura

- A classe ProxySubject representa a classe Subject. Neste caso o cliente (Client) utilizará uma instância de ProxySubject e não de subject
- A classe ProxySubject pode por exemplo: restringir o acesso ao subject; ser uma representação local; conter um cache para um subject remoto



Proxy
Exemplo

- A classe ProxyImage representa uma imagem
- Neste caso trata-se de um proxy virtual, o cliente do proxy (ProxyClient) não precisa criar objetos da classe RealImage, mas apenas da classe ProxyImage que representará um objeto da classe RealImage



Proxy
Exemplo – Código

```
interface Image {
    void displayImage();
}

class RealImage implements Image {
    private String filename;
    public RealImage(String filename) {
        this.filename = filename;
        loadImageFromDisk();
    }
    private void loadImageFromDisk() {
        System.out.println("Loading   " + filename);
    }
    public void displayImage() {
        System.out.println("Displaying " + filename);
    }
}
```

Proxy
Exemplo – Código

```
class ProxyImage implements Image {
    private String filename;
    private RealImage image;
    public ProxyImage(String filename) {
        this.filename = filename;
    }
    public void displayImage() {
        if (image == null) {
           image = new RealImage(filename);
        }
        image.displayImage();
    }
}
```

Proxy
Exemplo – Código

```
class ProxyClient  {
    public static void main(String[] args) {
        Image image1 = new ProxyImage("HiRes_10MB_Photo1");
        Image image2 = new ProxyImage("HiRes_10MB_Photo2");

        image1.displayImage(); // loading necessary
        image2.displayImage(); // loading necessary
        image1.displayImage(); // loading unnecessary
    }
}
```