Engenharia de Sistemas de Informação I (ACH2006)
Atividade 01 - Padrões de Projeto GoF

Mirela Mei - 11208392

Padrões de Projeto

Em Engenharia de Software, um design pattern (padrão de projeto) é uma solução geral para um problema que ocorre com frequência dentro de um determinado contexto em projetos de software. Funciona como uma descrição ou modelo (*template*) de como resolver um problema que pode ser usado em muitas situações diferentes. Ou seja, padrões são 'melhores práticas' formalizadas que o programador pode usar para resolver problemas comuns ao projetar uma aplicação ou sistema. Padrões de projeto orientados a objeto normalmente mostram relacionamentos e interações entre classes ou objetos, sem especificar as classes ou objetos da aplicação final que estão envolvidas. Padrões que implicam orientação a objetos ou estado mutável mais geral, não são tão aplicáveis em linguagens de programação funcional.

Os padrões são divididos em:

- Nome do padrão
- Problema a ser resolvido
- Solução dada pelo padrão
- Consequências

O arquiteto Christopher Alexander, em seus livros estabelece que um padrão deve ter, idealmente, as seguintes características:

- Encapsulamento: encapsula um problema ou solução bem definida. Ele deve ser independente, específico e formulado de maneira a ficar claro onde ele se aplica.
- Generalidade: deve permitir a construção de outras realizações a partir deste padrão.
- Equilíbrio: quando um padrão é utilizado em uma aplicação, o equilíbrio dá a razão, relacionada com cada uma das restrições envolvidas, para cada passo do projeto. Uma análise racional que

envolva uma abstração de dados empíricos, uma observação da aplicação de padrões em artefatos tradicionais, uma série convincente de exemplos e uma análise de soluções ruins ou fracassadas pode ser a forma de encontrar este equilíbrio.
- Abstração: os padrões representam abstrações da experiência empírica ou do conhecimento cotidiano.
- Abertura: um padrão deve permitir a sua extensão para níveis mais baixos de detalhe.
- Combinatoriedade: os padrões são relacionados hierarquicamente. Padrões de alto nível podem ser compostos ou relacionados com padrões que endereçam problemas de nível mais baixo.

Padrões GoF ('Gang of Four')

O movimento ao redor de padrões de projeto só ganhou popularidade em 1995 quando foi publicado o livro *Design Patterns: Elements of Reusable Object-Oriented Software*. Os autores deste livro, Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides, são conhecidos como a "Gangue dos Quatro" (Gang of Four - GoF) e por meio do *Design Patterns*, é possível ter uma base de padrões do campo de design para objetos. O padrão de design nada mais é do que aquela solução que, em algum momento, será reutilizada para resolver um problema que é comumente encontrado em um software, sempre que for necessário, haverá um modelo disponibilizado para solucionar algum empecilho da programação.

- **Padrões de criação**: Os padrões de criação tem como intenção principal abstrair o processo de criação de objetos, ou seja, a sua instanciação. Desta maneira o sistema não precisa se preocupar com questões sobre como o objeto é criado, como é composto, qual a sua representação real. Quando se diz que o sistema não precisa se preocupar com a instanciação do objeto quer dizer que, se ocorrer alguma mudança neste ponto, o sistema em geral não será afetado. Isso é a famosa flexibilidade que os padrões de projeto buscam. Padrões de criação com escopo de classe vão utilizar herança para garantir que a flexibilidade.
- **Padrões estruturais**: Os padrões estruturais se preocupam com a forma como classes e objetos são compostos para formar estruturas maiores. Os de classes utilizam a herança para compor interfaces ou implementações, e os de objeto descrevem maneiras de compor objetos para obter novas funcionalidades. A flexibilidade obtida pela composição de objetos provém da capacidade de mudar a composição

em tempo de execução o que não é possível com a composição estática (herança de classes).

- **Padrões comportamentais**: Os padrões de comportamento se concentram nos algoritmos e atribuições de responsabilidades entre os objetos. Eles não descrevem apenas padrões de objetos ou de classes, mas também os padrões de comunicação entre os objetos. Os padrões comportamentais de classes utilizam a herança para distribuir o comportamento entre classes, e os padrões comportamentais de objeto utilizam a composição de objetos em contrapartida a herança. Alguns descrevem como grupos de objetos que cooperam para a execução de uma tarefa que não poderia ser executada por um objeto sozinho.

Exemplos:

1. Padrões de Criação

Abstract Factory

Este padrão permite a criação de famílias de objetos relacionados ou dependentes por meio de uma única interface e sem que a classe concreta seja especificada. Uma fábrica é a localização de uma classe concreta no código em que objetos são construídos . O objetivo em empregar o padrão é isolar a criação de objetos de seu uso e criar famílias de objetos relacionados sem ter que depender de suas classes concretas, isso permite novos tipos derivados serem introduzidos sem qualquer alteração ao código que usa a classe base.

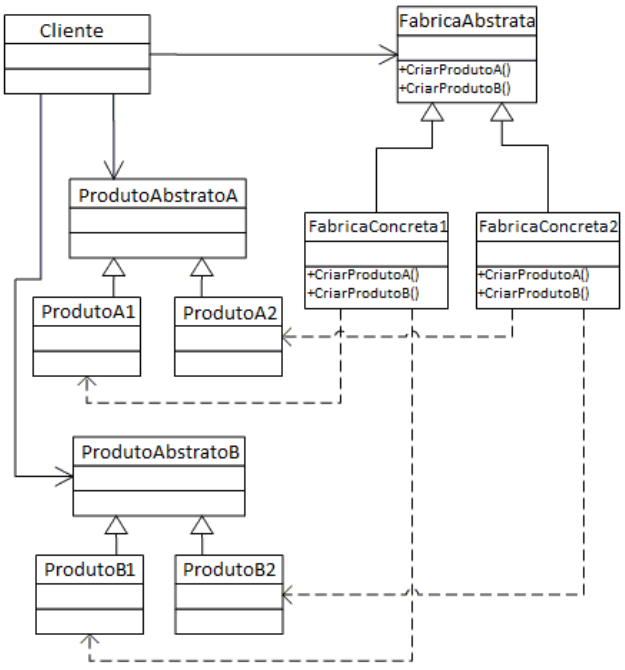

Diagrama UML do padrão

Neste exemplo, a classe abstrata WidgetFactory possui duas especializações: MotifWidgetFactory para *widgets* Motif e QtWidgetFactory para *widgets* Qt. Essas especializações são classes concretas capazes de "produzir" os elementos da interface gráfica. O cliente do *toolkit* obtém os elementos gráficos de que necessita por meio da classe (interface) WidgetFactory sem ter conhecimento das classes concretas. Da mesma maneira, o cliente somente interage com as interfaces que representam os elementos produzidos pela Abstract Factory (no exemplo, a classe Janela e a classe Botao). O código imprimiria na tela o texto "Eu sou um botao Motif!" ou "Eu sou um botao Qt!" dependendo do valor retornado pelo método Configuracao.obterInterfaceGraficaAtual(), que descobre a interface gráfica, Motif ou Qt, utilizada pelo sistema.

```
abstract class WidgetFactory{
    public static WidgetFactory obterFactory(){

        if( Configuracao.obterInterfaceGraficaAtual() == Configuracao.MotifWidget ){
            return new MotifWidgetFactory();
        }
        else{
            return new QtWidgetFactory();
        }
   }

   public abstract Botao criarBotao();
}

class MotifWidgetFactory extends WidgetFactory{
    public Botao criarBotao(){
        return new BotaoMotif();
    }
}

class QtWidgetFactory extends WidgetFactory{
    public Botao criarBotao(){
        return new BotaoQt();
    }
}

abstract class Botao{
    public abstract void desenhar();
}

class BotaoMotif extends Botao{
    public void desenhar(){
       System.out.println("Eu sou um botao Motif!");
    }
}

class BotaoQt extends Botao{
    public void desenhar(){
       System.out.println("Eu sou um botao Qt!");
    }
}

public class Cliente{
    public static void main(String[] args){
        WidgetFactory factory = WidgetFactory.obterFactory();
        Botao botao = factory.criarBotao();
        botao.desenhar();
    }
}
```

Builder

Builder é um padrão de projeto de software criacional que permite a separação da construção de um objeto complexo da sua representação, de forma que o mesmo processo de construção possa criar diferentes representações.

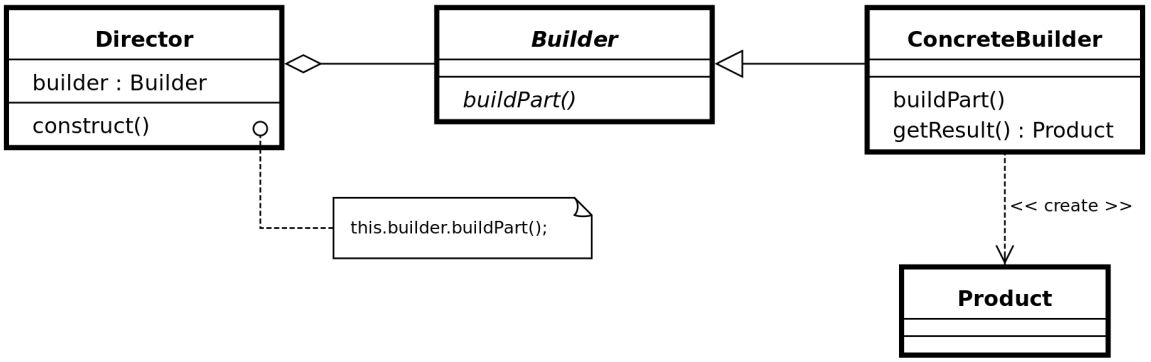

Diagrama UML do padrão

O padrão Builder pode ser utilizado em uma aplicação que converte o formato de texto RTF para uma série de outros formatos e que permite a inclusão de suporte para conversão para outros formatos, sem a alteração do código fonte do leitor de RTF.
Neste exemplo, o método lerRTF() (da classe LeitorRTF) percorre uma lista com os *tokens* encontrados no texto de entrada (formato RTF) e, para cada tipo de *token*, chama um método do objeto de tipo ConversorTexto. Dependendo do formato escolhido para o texto de destino, será escolhida uma implementação da classe ConversorTexto: ConversorPDF, ConversorTeX ou ConversorASCII. Cada uma destas classes implementa os métodos de acordo com as características do formato relacionado. Note que a classe ConversorASCII não implementa os métodos converteParagrafo() nem converteFonte() porque o formato ASCII não possui elementos de estilo.

```java
abstract class ConversorTexto {
	public void converterCaractere(char c) {
		// vazio
	}

	public void converterParagrafo() {
		// vazio
	}

	public void converterFonte(Fonte f) {
		// vazio
	}
}

class ConversorPDF extends ConversorTexto {
	public void converterCaractere(char c) {
		System.out.print("Caractere PDF");
	}

	public void converterParagrafo() {
		System.out.print("Parágrafo PDF");
	}

	public void converterFonte(Fonte f) {
		System.out.print("Fonte PDF");
	}
}

class ConversorTeX extends ConversorTexto {
	public void converterCaractere(char c) {
		System.out.print("Caractere Tex");
	}

	public void converterParagrafo() {
		System.out.print("Paragrafo Tex");
	}

	public void converterFonte(Fonte f) {
		System.out.print("Fonte Tex");
	}
}

class ConversorASCII extends ConversorTexto {
	public void converterCaractere(char c) {
		System.out.print("Caractere ASCII");
	}
}

```

```
49  class LeitorRTF {
50
51          private ConversorTexto conversor;
52
53          LeitorRTF(ConversorTexto c) {
54                  this.conversor = c;
55          }
56
57          public void lerRTF() {
58
59                  List<Token> tokens = obterTokensDoTexto();
60
61                  for (Token t : tokens) {
62                          if (t.getTipo() == Token.Tipo.CARACTERE) {
63                                  conversor.converterCaractere(t.getCaractere());
64                          }
65                          if (t.getTipo() == Token.Tipo.PARAGRAFO) {
66                                  conversor.converterParagrafo();
67                          }
68                          if (t.getTipo() == Token.Tipo.FONTE) {
69                                  conversor.converterFonte(t.getFonte());
70                          }
71                  }
72          }
73  }
74
75  public class Cliente {
76
77          public static void main(String[] args) {
78
79                  ConversorTexto conversor;
80                  if (args[0].equals("pdf")) {
81                          conversor = new ConversorPDF();
82                  } else if (args[0].equals("tex")) {
83                          conversor = new ConversorTeX();
84                  } else {
85                          conversor = new ConversorASCII();
86                  }
87                  LeitorRTF leitor = new LeitorRTF(conversor);
88
89                  leitor.lerRTF();
90          }
91  }
```

Prototype

O padrão Prototype é aplicado quando existe a necessidade de clonar, literalmente, um objeto. Ou seja, quando a aplicação precisa criar cópias exatas de algum objeto em tempo de execução este padrão é altamente recomendado. Este padrão pode ser utilizado em sistemas que precisam ser independentes da forma como os seus componentes são criados, compostos e representados.

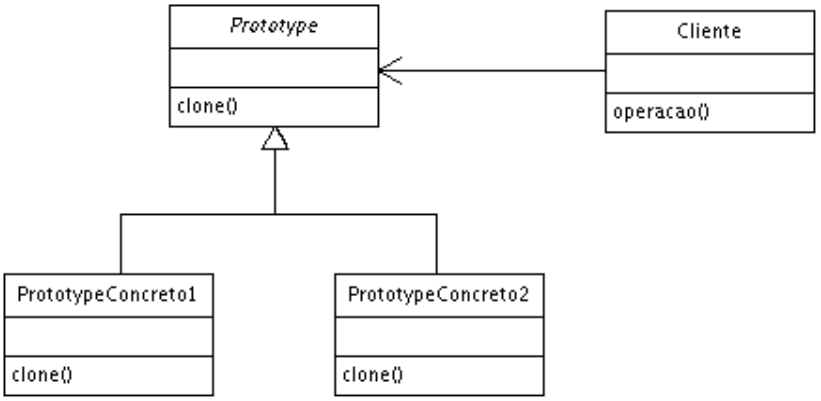

Diagrama UML do padrão

Neste exemplo é mostrado uma hierarquia de classes representando documentos de formato ASCII e PDF que são criados através da classe Cliente. A partir de duas instâncias prototípicas, ascii e pdf, o método criarDocumento cria clones de documentos de acordo com o tipo desejado. A tarefa de realizar a criação da instância é implementada na classe Documento e herdada por suas classes filhas, ASCII e PDF.

```
public interface ICloneable {
        public Object clone();
}

abstract class Documento implements ICloneable {

    protected Documento clone() {

        Object clone = null;

        try {
             clone = super.clone();
        } catch (CloneNotSupportedException ex) {
             ex.printStackTrace();
        }

        return (Documento) clone;
    }
}

class ASCII extends Documento { }

class PDF extends Documento { }

class Cliente {

    static final int DOCUMENTO_TIPO_ASCII = 0;

    static final int DOCUMENTO_TIPO_PDF = 1;

    private Documento ascii = new ASCII();

    private Documento pdf = new PDF();

    public Documento criarDocumento(int tipo) {

        if (tipo == Cliente.DOCUMENTO_TIPO_ASCII) {
            return ascii.clone();
        } else {
            return pdf.clone();
        }
    }
}
```

2. Padrões Estruturais

Adapter

De forma exemplificável por um adaptadores de cabos, o padrão **Adapter** converte a interface de uma classe para outra interface que o cliente espera encontrar, "traduzindo" solicitações do formato requerido pelo usuário para o formato compatível com a classe *adaptee* e as redirecionando. Dessa forma, o Adaptador permite que classes com interfaces incompatíveis trabalhem juntas.

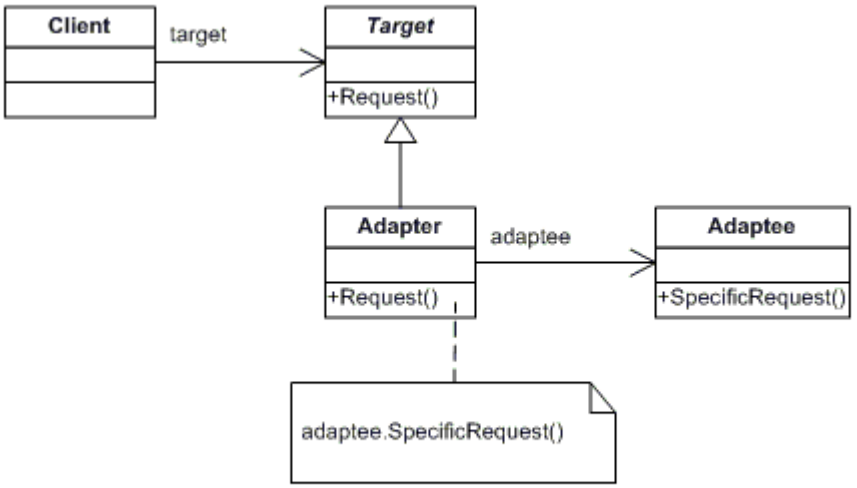

Diagrama UML do padrão

Neste exemplo

- *Target (Alvo): d*efine a interface do domínio específico que o cliente utiliza.
- *Adapter (Adaptador): a*dapta a interface Adaptee para a interface da classe Target.
- *Adaptee (Adaptada): d*efine uma interface existente que necessita ser adaptada.
- *Client (Cliente): c*olabora com os objetos em conformidade com a interface Target.

```java
//Classe adaptada (Adaptee)
class SensorXbox2 {
    //Solicitação Especifica
    public void conectarXbox2() {
        System.out.println("Um novo controle foi conectado ao sensor do Xbox.");
    }
}

//Classe alvo (Target)
class SensorPS5 {
    //Solicitação
    public void conectarPS5() {
        System.out.println("Um novo controle foi conectado ao sensor do PS5.");
    }
}

//Classe adaptador (Adapter)
class AdaptadorPS5ParaXbox2 extends SensorPS5 {
    private SensorXbox2 adaptee;
    public AdaptadorPS5ParaXbox2(SensorXbox2 adaptee) {
        this.adaptee = adaptee;
    }

    //Override da solicitação
    public void conectarPS5() {
        adaptee.conectarXbox2();
        System.out.println("But stadia wins!");
    }
}

//Classe Cliente(Client)
public class ControlePS5 {

    private SensorPS5 sensorAQueSeConecta;

    public void Conectar(SensorPS5 sensor){
        this.sensorAQueSeConecta = sensor;
        sensorAQueSeConecta.conectarPS5();
    }

//}
//public class Teste{
    public static void main(String[] args) {

        //Tem-se um Xbox2 e mas deseja-se usar um controle ps5:
        SensorXbox2 adaptee = new SensorXbox2();
        ControlePS5 target = new ControlePS5();
        //Cria-se um falso sensor de PS5 que, na verdade, traduz e repassa os comandos para o Xbox em questão:
        AdaptadorPS5ParaXbox2 adapter = new AdaptadorPS5ParaXbox2(adaptee);
        //Conecta-se o controle ao adapter:
        target.Conectar(adapter);
    }
}
```

Decorator

É um padrão que permite adicionar um comportamento a um objeto já existente em tempo de execução, ou seja, agrega dinamicamente responsabilidades adicionais a um objeto. Decorators oferecem uma alternativa flexível ao uso de herança para estender uma funcionalidade, com isso adiciona-se uma responsabilidade ao objeto e não à classe.

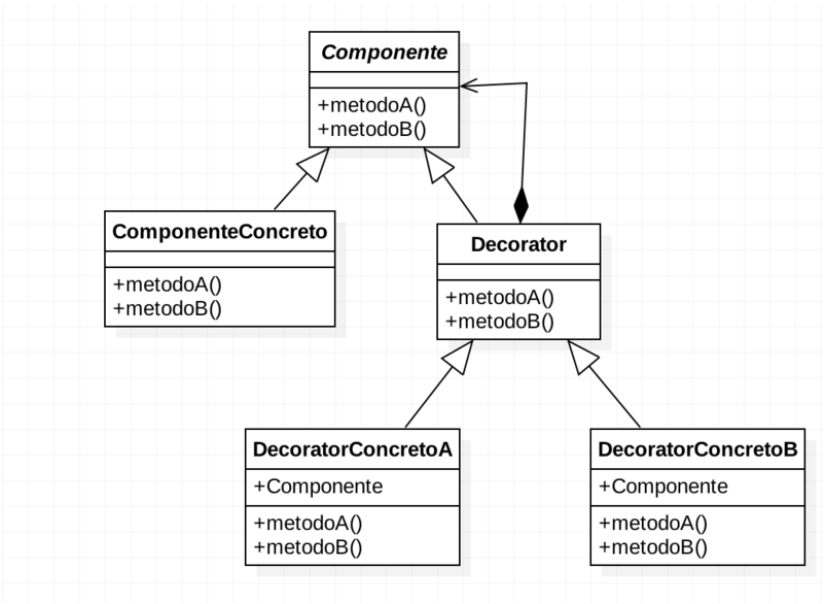

Diagrama UML do padrão

Um exemplo simples e prático da aplicação do Decorator seria colocar acessórios em uma arma, como miras e silenciadores. Um modo de se contornar esse problema seria criar uma interface e criar armas que implementam essa interface, porém isso faria com que tivéssemos muitas classes que implementam essa interface, para apenas dois acessórios teríamos que criar 4 classes (Arma sem acessórios, arma com silenciador, arma com mira, arma com mira e silenciador) tornando essa uma alternativa ruim. Além disso, essa alternativa viola os princípios de baixa acoplação e alta coesão. Outra solução seria utilizar o *Design Pattern Decorator*, com esse *Design Pattern* será possível contornar esse problema de uma forma simples e que não viole nenhum princípio de *design*. Além disso, ele possibilita que sejam adicionados acessórios durante o tempo de execução.

```java
public interface Arma{
  public void montar();
}

public class ArmaBase implements Arma{
  @Override
  public void montar(){
    System.out.println("Essa é uma arma base");
  }
}

public class ArmaDecorator implements Arma{
  public Arma arma;
  public ArmaDecorator(Arma arma){
    this.arma = arma;
  }
  @Override
  public void montar(){
    this.arma.montar();
  }
}

public class Mira extends ArmaDecorator{
  public Mira(Arma arma){
    super(arma);
  }
  @Override
  public void montar(){
    super.montar();
    System.out.println("Adicionando mira a arma");
  }
}

public class Silenciador extends ArmaDecorator{
  public Silenciador(Arma arma){
    super(arma);
  }
  @Override
  public void montar(){
    super.montar();
    System.out.println("Adicionando silenciador a arma");
  }
}
```

Facade

É usado quando um sistema é muito complexo ou difícil de entender, já que possui um grande número de classes independentes ou se os trechos de código fonte estão indisponíveis. Este padrão esconde as complexidades de um sistema maior e provê uma interface simplificada ao cliente.
Tipicamente envolve uma única classe responsável por englobar uma série

de membros requeridos pelo cliente. Estes membros acessam o sistema em nome do *Facade* e escondem os detalhes de implementação.

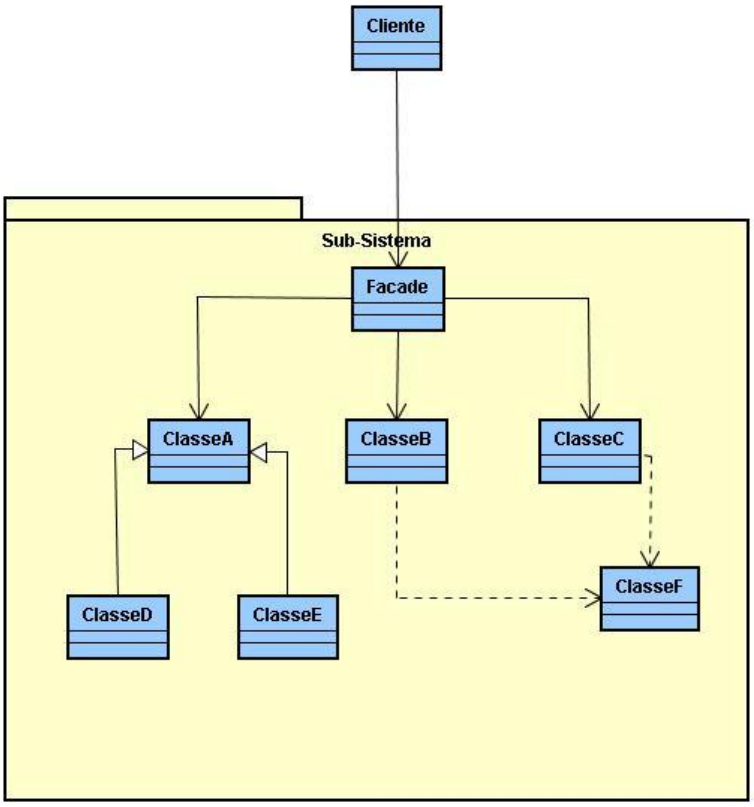

Diagrama UML do padrão

Esse é um exemplo abstrato de como um cliente ("você") interage com um *Facade* (o "computador") para um sistema complexo (as partes internas do computador como o processador e o disco rígido)

```java
class CPU {
  public void freeze() { ... }
  public void jump(long position) { ... }
  public void execute() { ... }
}

class Memory {
  public void load(long position, byte[] data) { ... }
}

class HardDrive {
  public byte[] read(long lba, int size) { ... }
}

class Computer {
  private CPU cpu;
  private Memory memory;
  private HardDrive hardDrive;

  public Computer() {
    this.cpu = new CPU();
    this.memory = new Memory();
    this.hardDrive = new HardDrive();
  }

  public void startComputer() {
    cpu.freeze();
    memory.load(BOOT_ADDRESS, hardDrive.read(BOOT_SECTOR, SECTOR_SIZE));
    cpu.jump(BOOT_ADDRESS);
    cpu.execute();
  }
}

class You {
  public static void main(String[] args) {
    Computer facade = new Computer();
    facade.startComputer();
  }
}
```

3. Padrões Comportamentais

Iterator

um iterador se refere tanto ao objeto que permite ao programador percorrer um container, (uma coleção de elementos) particularmente listas,quanto ao padrão de projetos *Iterator*, no qual um iterador é usado para percorrer um container e acessar seus elementos. O padrão Iterator desacopla os algoritmos dos recipientes, porém em alguns casos, os algoritmos são necessariamente específicos dos containers e, portanto, não podem ser desacoplados.

```java
// digamos que coleção seja uma coleção de BlaBleBli
for (Iterator it = colecao.iterator() it.hasNext()) {
  BlaBleBli obj = (BlaBleBli) it.next();
}

//Em Java 5, o "iterator" pode até ficar escondido:
for (BlaBleBli obj : colecao) {
```