Padrões de projeto

Fernando C. Garcia Filho[1], **Geovane Camargo**[1], **Guilherme Fagundes**[1], **Igor Fachini**[1], **Rafael Chewinski**[1]

[1]Bacharelado em Sistemas de Informação – Católica de Santa Catarina (UNERJ)
R. dos Imigrantes, 500 – 89254-430 – Jaragua do sul – SC – Brazil

fernando.garcia@unerj.br, geovane.camargo@unerj.br,guilherme.fagundes@unerj.br

igor.fachini@unerj.br, rafael.chewinski@unerj.br

Abstract. *A design pattern generally describes a reusable general solution for a problem that occurs frequently within a given context in software design. The main purpose of design patterns available to developers today, and to make certain reusable components, facilitating standardization, allowing more flexibility to solve problems encountered in the development of the system, structuring its more flexible ways applications, understandable and easy to keep. The original idea came with Christopher Alexander, because it proposed the creation of standards for architecture catalogs. "Design patterns make it easier to reuse successful designs and architectures"[Gamma et al. 2000].*
This article presents the main concepts of the design patterns Builder, Decorator, Observer and Proxy, demonstrating its functionality through theoretical and practical examples.

Resumo. *Um padrão de projeto, de modo geral, descreve uma solução geral reutilizável para um problema que ocorre com frequência dentro de um determinado contexto no projeto de software. O principal objetivo dos padrões de projeto disponíveis para os desenvolvedores atualmente e de tornar certos componentes reutilizáveis, facilitando a padronização, permitindo mais agilidade para as soluções de problemas encontrados no desenvolvimento do sistema, estruturando os seus aplicativos de maneiras mais flexíveis, compreensíveis e fáceis de manter. A ideia original surgiu com Christopher Alexander, pois o mesmo propôs a criação de catálogos de padrões para arquitetura. "Os padrões de projeto tornam mais fácil reutilizar projetos e arquiteturas bem sucedidas" [Gamma et al. 2000].*
Este artigo apresenta os principais conceitos dos padrões de projeto Builder, Decorator, Observer e Proxy, demonstrando sua funcionalidade através de embasamento teórico e exemplos práticos.

1. Padrão Builder

1.1. Intenção

O builder é um padrão de criação, usado para separar a construção de um objeto complexo de sua representação podendo que o mesmo possa apresentar diferentes representações.

Resumindo o usuário apenas escolhera o que ele ira querer que seja feito, sem saber os passos que são feitos para construir tal produto.

1.2. Motivação

Cada tipo de classe conversora implementa o mecanismo para criação e montagem de um objeto complexo, colocando-o atrás de uma interface abstrata.Cada classe conversora é chamada de builder no padrão, e o leitor é chamado de director. O Builder separa o algoritmo para interpretar um formato de texto de como um formato convertido é criado e representado.[de Almeida Ribeiro 2013]

1.3. Aplicabilidade

O padrão builder pode ser utilizado nas seguintes situações:

- Quando o processo de construção de um objeto é considerado complexo e também é adequado quando se trata da construção de representações múltiplas de uma mesma classe.
- Quando ha necessidade representar o mesmo objeto de varias formas.

1.4. Estrutura

O cliente chamara o método main, que assim iniciara as classes Builder e Director.A classe Builder representa o nosso objeto complexo.O construtor da classe Director recebe um objeto Builder como sendo um parâmetro através do cliente e é responsável por chamar os métodos apropriados da classe Builder. A fim de fornecer a classe cliente com uma interface para todos os construtores concretos, a classe Builder deve ser uma classe abstrata. Desta forma, podemos adicionar novos tipos de objetos apenas definindo a estrutura e reutilizando a lógica para o processo real da construção. O cliente é o único que precisa saber sobre os novos tipos, já a classe Director, precisa saber apenas quais os métodos que precisará chamar.

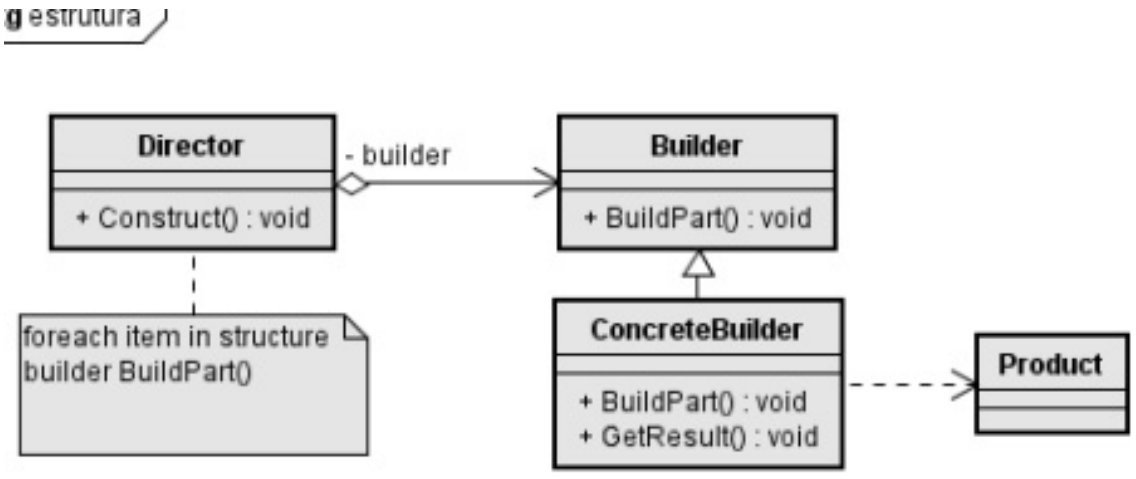

Figura 1. Modelo UML da estrutura teorica

1.5. Participantes

O padrão builder é composto por 4 componentes básicos que são a Interface (ou classe abstrata) Builder, o concrete builder (construtor concreto), o Director (Diretor) e o product (produto).

- Classe Builder – esta classe especifica uma interface ou uma classe abstrata para a criação das partes (ou podemos considerar como sendo os passos que serão realizados) de um objeto a fim de criar corretamente o produto (Product). Cada um desses passos são, geralmente, abstrações das funcionalidades realizadas nas subclasses concretas.
- Concrete Builder – esta classe é responsável pela construção e pela montagem das partes do produto por meio da implementação da classe builder. Ela define e mantém o controle da representação que a classe cria, além de fornecer uma interface para recuperação do produto.
- Director – esta é a classe que controla o algoritmo responsável por gerar o objeto do produto final. Um objeto Director é instanciado e seus métodos construtores são chamados. O método inclui um parâmetro para capturar objetos específicos do tipo Concrete Builder que serão então utilizados para gerar o produto (product). Dessa forma, o director, chama os métodos do concrete builder na ordem correta para gerar o objeto produto.
- Product – o product representa o objeto complexo que está sendo construído. O concrete builder então constrói a representação interna do produto e define o processo pelo qual essa classe será montada. Na classe product são incluídas outras classes que definem as partes que a constituem, dentre elas, as interfaces para a montagem das partes no resultado final.[José 2013]

1.6. Colaboração

O ConcreteBuilder trás apenas suas partes realmente importantes para o objeto cliente, isolando toda parte responsável pela construção.

1.7. Consequências

1.7.1. Vantagens

Este é o tipo de padrão que permite que um objeto cliente seja capaz de construir um objeto complexo, especificando apenas o seu tipo e o seu conteúdo, sendo então protegido dos detalhes relacionados com a representação do objeto, entrando aqui o conceito de encapsulamento.

1.7.2. Desvantagens

Um problema do padrão é que seja necessário sempre chamar o método de construção para depois utilizar o produto em si. Dando assim essa responsabilidade ao cliente.

1.8. Implementação

A ideia original da prova de conceito é de [Brizeno 2011a].Abaixo esta a modelagem UML da prova de conceito, criada utilizando a ferramenta Astah.

O código foi implementado utilizando o ambiente de desenvolvimento integrado Netbeans e baseado no modelo mostrado acima.Nesta classe teremos o nosso produto, com todas as variáveis que o carro ira ter.

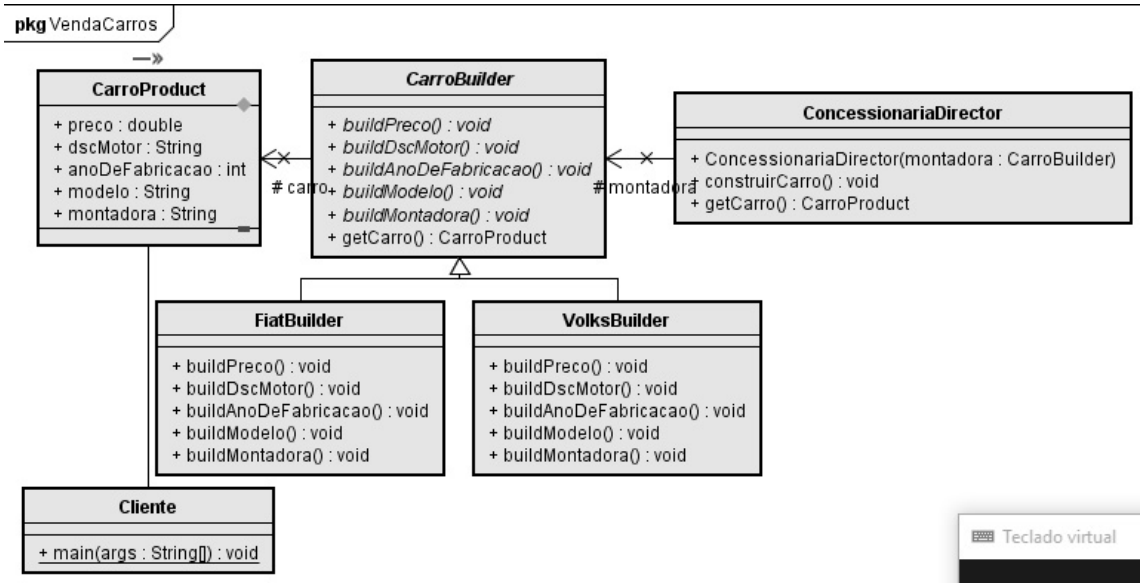

Figura 2. Modelo UML da prova de conceito

```
public class CarroProduct {
   double preco;
   String dscMotor;
   int anoDeFabricacao;
   String modelo;
   String montadora;
}
```

Nesta classe abstrata temos o carro que será construído, os passos para sua construção e um método que devolve o carro construído.

```
public abstract class CarroBuilder {

   protected CarroProduct carro;

   public CarroBuilder() {
      carro = new CarroProduct();
   }

   public abstract void buildPreco();

   public abstract void buildDscMotor();

   public abstract void buildAnoDeFabricacao();

   public abstract void buildModelo();

   public abstract void buildMontadora();
```

```
    public CarroProduct getCarro() {
        return carro;
    }
}
```

Na classe FiatBuilder nós “personalizamos” o carro, com as informações da Fiat. Note os comentários “// Operação complexa”. Antes da atribuição do preço, por exemplo, poderíamos realizar todo o cálculo necessário,buscando o valor no banco de dados, calcular impostos, desvalorização, entre outras operações. Essa é a ideia principal do padrão Builder, dividir em pequenos passo a construção do objeto.

```
public class FiatBuilder extends CarroBuilder {

    @Override
    public void buildPreco() {
        // Operao complexa.
        carro.preco = 25000.00;
    }

    @Override
    public void buildDscMotor() {
        // Operao complexa.
        carro.dscMotor = "Fire Flex 1.0";
    }

    @Override
    public void buildAnoDeFabricacao() {
        // Operao complexa.
        carro.anoDeFabricacao = 2011;
    }

    @Override
    public void buildModelo() {
        // Operao complexa.
        carro.modelo = "Palio";
    }

    @Override
    public void buildMontadora() {
        // Operao complexa.
        carro.montadora = "Fiat";
    }
}
```

Dado um Builder, a classe vai executar os métodos de construção, definindo assim o algoritmo de construção do carro, e depois devolve o carro. O código cliente vai lidar apenas com o Director, toda a estrutura e algoritmos utilizados para construir o carro ficarão por debaixo dos panos.

```
public class ConcessionariaDirector {
   protected CarroBuilder montadora;

   public ConcessionariaDirector(CarroBuilder montadora) {
      this.montadora = montadora;
   }

   public void construirCarro() {
      montadora.buildPreco();
      montadora.buildAnoDeFabricacao();
      montadora.buildDscMotor();
      montadora.buildModelo();
      montadora.buildMontadora();
   }

   public CarroProduct getCarro() {
      return montadora.getCarro();
   }
}
```

Aqui mostramos como o cliente ira manipular esses dados, so ha necessidade de utilizar um novo Builder para construir o produto que quiser.

```
public static void main(String[] args) {
   ConcessionariaDirector concessionaria = new
      ConcessionariaDirector(
         new FiatBuilder());

   concessionaria.construirCarro();
   CarroProduct carro = concessionaria.getCarro();
   System.out.println("Carro: " + carro.modelo + "/" +
      carro.montadora
         + "\nAno: " + carro.anoDeFabricacao + "\nMotor: "
         + carro.dscMotor + "\nValor: " + carro.preco);

   System.out.println();

   concessionaria = new ConcessionariaDirector(new
      VolksBuilder());
   concessionaria.construirCarro();
   carro = concessionaria.getCarro();
   System.out.println("Carro: " + carro.modelo + "/" +
      carro.montadora
         + "\nAno: " + carro.anoDeFabricacao + "\nMotor: "
         + carro.dscMotor + "\nValor: " + carro.preco);
}
```

1.9. Usos Conhecidos

- Conversor de RTF.
- O Framework Service Configurator do Adaptive Communications Enviroment usa um builder para construir componentes de serviços de rede que são "linkeditados" a um servidor em tempo de execução.[de Almeida Ribeiro 2013]

1.10. Padrões relacionados

Abstract Factory e Composite são muito semelhantes ao padrão de projeto builder.

2. Padrão Decorator

2.1. Intenção

Fornecer uma alternativa flexível ao uso de subclasses para estender as funcionalidades. De forma mais simples podemos dizer que o padrão decorator permite estender (decorar) dinamicamente as características de uma classe usando a composição [Macoratti 2013].

2.2. Motivação

"Decorator ou wrapper, é um padrão de projeto de software que permite adicionar um comportamento a um objeto já existente em tempo de execução, ou seja, agrega dinamicamente responsabilidades adicionais a um objeto [Wikipedia 2011]. As vezes queremos adicionar funcionalidades especiais a objetos e não em toda uma classe, por exemplo, uma maneira de adicionar essa responsabilidade é usar herança. Uma abordagem mais flexível é embutir o componente em outro objeto que acrescente a borda. O objeto que envolve o primeiro é chamado de decorator. O decorator segue a interface que decora, desse modo a sua presença é transparente para os clientes do componente. Essa transparência permite encaixar decoradores recursivamente, dessa forma possibilitando um número ilimitado de responsabilidade adicionais.

As classes ScrollDecorator e BorderDecorator são subclasses de Decorator, ou seja, uma classe abstrata que possui componentes visuais que decoram outros componentes visuais.

VisualComponent é a classe abstrata para objetos visuais, é ela que define suas interfaces de desenho e tratamento de eventos. Repare como a classe Decorator simplesmente repassa as solicitações de desenho e como as subclasses de Decorator podem estender essa opração.

2.3. Aplicabilidade

- Para adicionar funcionalidades especiais a objetos de forma dinâmica e transparente; - Responsabilidades podem ser removidas; - Quando a extensão através do uso de subclasses não é prática.

2.4. Estrutura (UML)

2.5. Participantes

- Component (Visual Component): Define a interface para os objetos que podem ou não ter responsabilidades acrescentadas dinamicamente. - ConcreteComponent (TextView): Define a qual objeto as responsabilidades serão atribuídas - Decorator Mantem uma referência para um objeto Component e define uma interface que segue a interface de Component. - ConcreteDecorator (BorderDecorator, ScrollDecorator) Acrescenta responsabilidades ao componente.

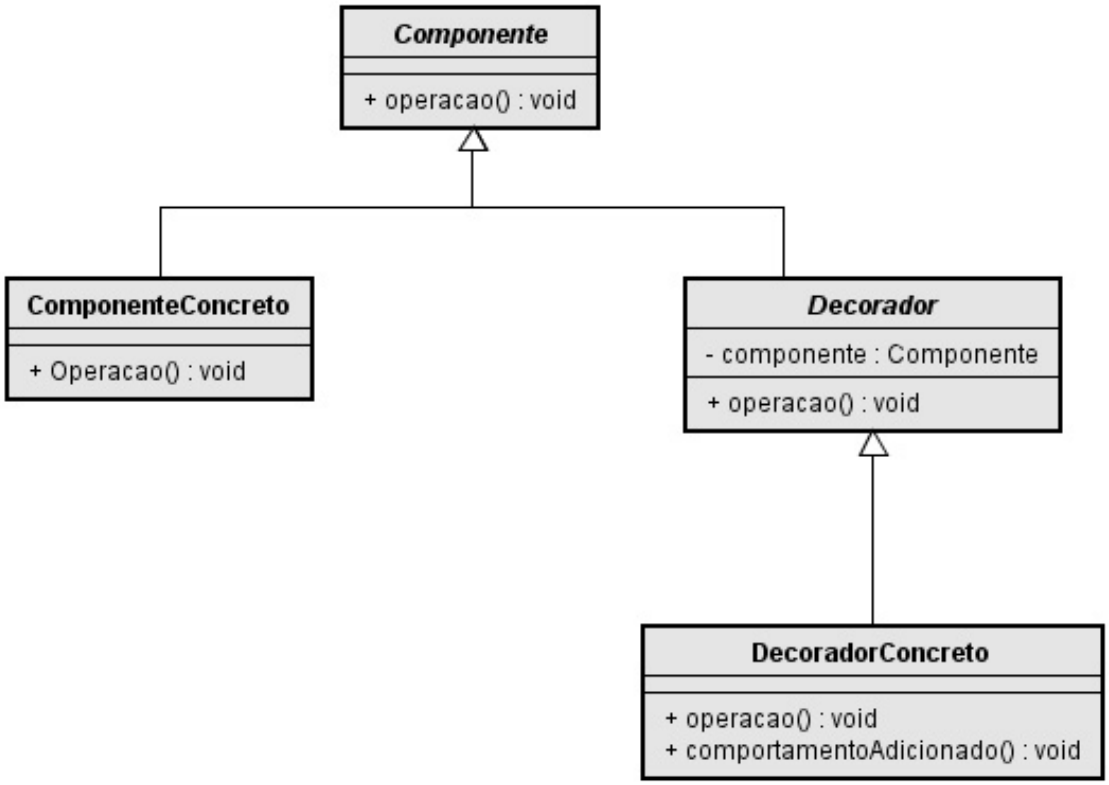

Figura 3. Modelo UML da estrutura teorica

2.6. Colaboração

- Decorator repassa solicitações para o seu objeto Component, desse modo, ele pode ou não executar operações adicionais antes e depois de repassar a solicitação.

2.7. Consequencias

1-Maior flexibilidade do que a herança estática. Eles também tornam fácil acrescentar uma propriedade duas vezes 2-Evita classes sobrecarregadas de caracteristicas na parte superior da hierarquia. Também é facil definir novas espécies de decorators independente das classes de objetos que eles estendem, mesmo no caso de extensões nao prevista. 3-Um Decorador e o seu componente não são identicos, ou seja ele funciona como um envoltório transparente. 4-Grande quantidade de pequenos objetos.

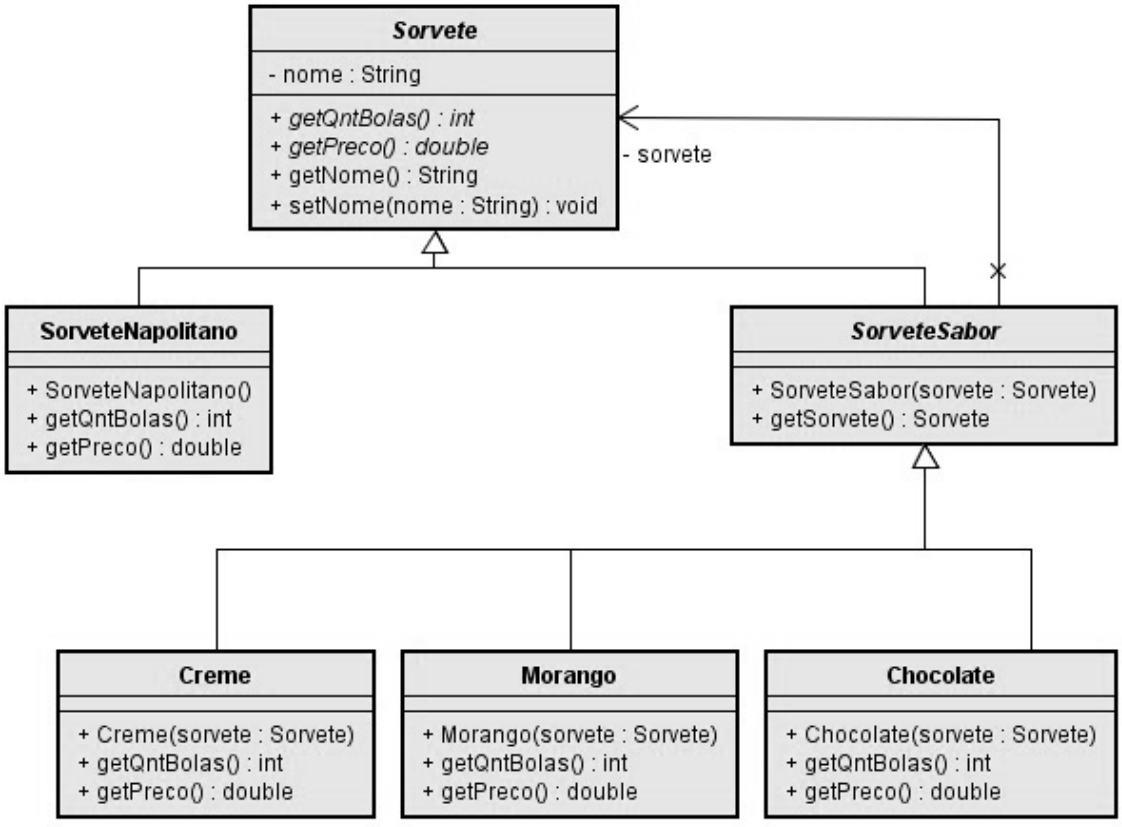

Figura 4. Modelo UML da estrutura pratica

2.8. Implementação

A classe Sorvete e Abstract para não ser instanciada. Possui os métodos abstratos que serão implementados pelas classes concretas, que servirão basicamente para contagem de valores.

```
public abstract class Sorvete {

   private String nome;
   public abstract int getQntBolas();
   public abstract double getPreco();

   public String getNome() {
      return nome;
   }

   public void setNome(String nome) {
      this.nome = nome;
   }


}
```

SorveteSabor será a classe Decorator. Percebe-se que no construtor do decorador é necessário passar um objeto Sorvete qualquer, este objeto pode ser tanto um sabor quanto outro decorador. O parâmetro recebido e atribuído ao valor da variável sorvete desta classe decorador. Ai está o conceito chave para o padrão Decorator, pois se acrescenta vários decoradores ou objetos em qualquer ordem em um só sorvete.

```
public abstract class SorveteSabor extends Sorvete{

   private Sorvete sorvete;

   public SorveteSabor(Sorvete sorvete) {
      this.sorvete = sorvete;
      sorvete.setNome("");
   }

   public Sorvete getSorvete() {
      return sorvete;
   }

}
```

A classe Chocolate extende a SorveteSabor(Decorator) e implementa tudo que é necessário da herança. Esta classe, bem como a Morango e Creme, irão ser instanciadas na criação dos objetos que serão adicionados posteriormente a um mesmo sorvete, atribuindo comportamentos diferentes ao mesmo.

```
public class Chocolate extends SorveteSabor {

   public Chocolate(Sorvete sorvete) {
      super(sorvete);
   }

   @Override
   public int getQntBolas() {
      return 1 + this.getSorvete().getQntBolas();
   }

   @Override
   public double getPreco() {
      return 1.50 + this.getSorvete().getPreco();
   }

}
```

O programa então se prontifica a criar um sorvete (no exemplo abaixo foi criado um Sorvete Napolitano, outra classe concreta criada para ter a representatividade do sorvete que queremos) que pode ”adotar”mais de um comportamento. Esta acão é percebida na instância de objetos das classes de sabores no mesmo sorvete criado. Ao final, imprime-se

o nome do sorvete, a quantidade de bolas e o preço total.

```
public class Main {
   public static void main(String[] args) {

      Sorvete sorvete = new SorveteNapolitano();

      retornaInformacoes(sorvete);

      sorvete = new Creme(new Morango(new Chocolate(sorvete)));


      retornaInformacoes(sorvete);


   }
   public static void retornaInformacoes(Sorvete sorvete){
      System.out.println(sorvete.getNome() + " - " +
            sorvete.getQntBolas() + " preco: R$" +
            sorvete.getPreco());
   }

}
```

2.9. Vantagens

Conformidade de interface Omissão de classe abstrata Decorator Mantendo leves as classes Component. Mudar o exterior de um objeto versus mudar o seu interior. Teremos mais flexibilidade se outros objetos da UI não souberem que está havendo embelezamento [Gamma et al. 2000].

2.10. Desvantagens

Uma desvantagem do padrão é que teremos inúmeras classes pequenas que pode ser bastante complicado para um desenvolvedor que está tentando entender o funcionamento da aplicação Permite que o adaptador sobrescreva algumas funções do adaptado É mais dificil sobrescrever funções do adaptado

2.11. Usos conhecidos

Muitos toolkits orientados a objetos utilizam decoradores para acrescentar adornos gráficos e widgets. Alguns exemplos: InterViews[LVC89,LCI*92], a ET++[WGM88] e a biblioteca de classes ObjectWorks/SmallTalk [Par90] Um DebuggingGlyph imprime uma depuração (debbugging) antes e depois de repassar uma solicitação de layout para seu componente, essa informação de rastreio pode ser usada para analisar e depurar comportamentos de layout. Streams são uma abstração fundamental em muitos recursos de programas de entrada e saida, o diagrama abaixo apresenta uma maneira elegante de acrescentar estas responsabilidades a streams.

A classe abstrata stream mantém um buffer interno e fornece operações para armazenar dados no stream (PutInt, PutString) Nesse diagrama acima, a classe abstrata é a

StreamDecorator, a qual mantem uma referencia para um componente stream que repassa solicitações para o mesmo.

3. Padrão Observer

3.1. Intenção

O Observer (observador) é um padrão de comportamento, usado para definir dependência entre objetos, de forma que quando o estado de um objeto é modificado, todos seus dependentes são avisados e atualizados automaticamente.

3.2. Motivação

Ao criar ou ampliar um sistema que possua diversas classes as quais se comunicam, é necessário manter a consistência entre os objetos que estiverem relacionados, mas ao mesmo tempo não relacionar fortemente estes objetos, de forma que sejam o mais reutilizável possível.

Por exemplo, em conjuntos de ferramentas para criação de interfaces gráficas de usuário, os dados para serem apresentados e os aspectos de apresentação da interface são separados, mas trabalham em conjunto. Ao modificar os dados, eles são atualizados automaticamente e corretamente na interface. Diferentes objetos de interface também podem mostrar o mesmo dado.

Como estabelecer esse comportamento e relacionamento é descrito pelo padrão Observer. Os objetos são divididos em subject (assunto) e observer (observador). Um subject pode ter qualquer quantidade de observadores, e todos eles são notificados ao mudar o estado do subject. Em seguida, cada observer perguntara ao subject seu estado para se sincronizar.

O subject notifica seus observers sem ter que saber quem eles são. Um número qualquer de observers pode se inscrever para receber notificações. Essa interação também é conhecida como publish-subscribe.

3.3. Aplicabilidade

O padrão Observer pode ser utilizado nas seguintes situações:

- Quando uma abstração possui dois aspectos interdependentes. Ao encapsular estes aspectos em objetos diferentes, é possível modificá-los e reutilizá-los de forma independente.
- O mais comum, que é quando uma mudança em um objeto exige mudanças em outros objetos, sem saber quantos objetos necessitam ser modificados.
- Quando um objeto necessita enviar notificações a outros objetos sem ter informações sobre eles, ou seja, não é desejável que os objetos estejam fortemente acoplados entre eles.

3.4. Estrutura

De forma teórica no padrão Observer, temos uma classe abstrata Subject, a qual possui uma lista de observadores e possui mecanismos para adicionar e remover observadores de sua lista, além de poder notificar eles se existir uma mudança em seu estado. Temos a classe abstrata Observer, a qual possui o método abstrato update.

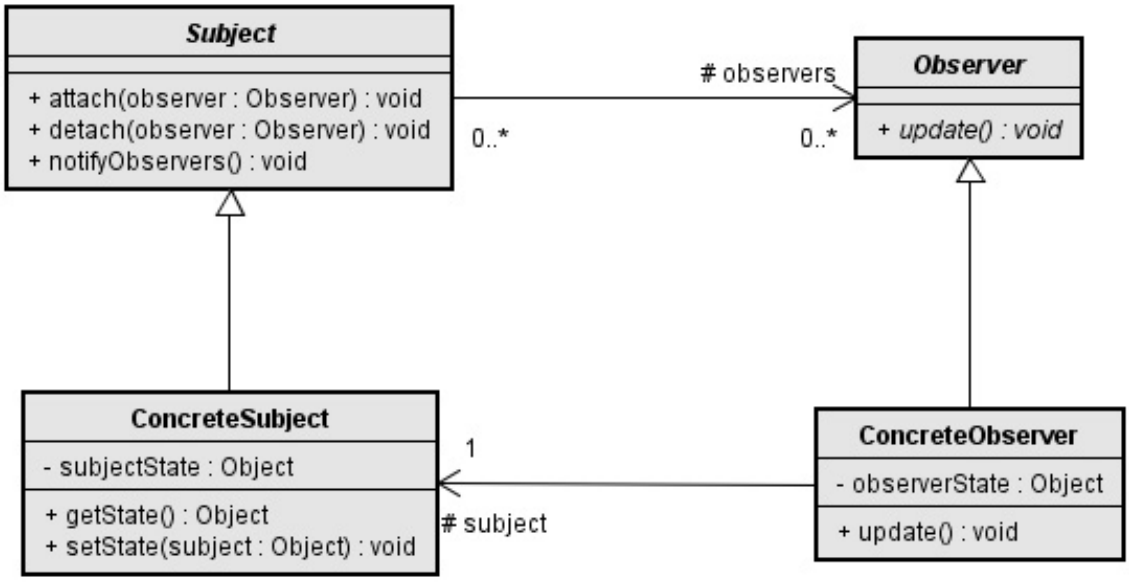

Figura 5. Modelo UML da estrutura teórica do padrão Observer

Adicionalmente temos as implementações de ambas classes abstratas, neste caso chamadas de ConcreteSubject e ConcreteObserver. O ConcreteSubject é uma implementação do Subject, e se encarrega de guardar um estado, e de chamar o método de notificação criado no Subject quando apropriado, por exemplo, ao ser modificado seu estado. O ConcreteObserver é uma implementação do Observer, o qual conhece o ConcreteSubject, possui um estado interno e implementa o método update para atualizar seu estado quando notificado pelo Subject.

3.5. Participantes

Em geral são 4 participantes:

- Subject
  - O qual conhece seus observadores, dos quais pode se ter um número qualquer.
  - Apresenta uma interface para adicionar e remover observadores.
- Observer
  - Define uma interface para atualizar objetos que devem ser avisados sobre mudanças no subject.
- ConcreteSubject
  - Armazena estados que sejam de interesse para o ConcreteObserver.
  - Envia uma notificação a todos seus observadores ao mudar seu estado.
- ConcreteObserver
  - Possui uma referência ao ConcreteSubject.
  - Armazena estados que devem ser sincronizados com os estados do Subject.
  - Implemente a interface de atualização do Observer, para manter seu estado consistente.

3.6. Colaboração

- O ConcreteSubject envia uma notificação a seus ConcreteObservers (já seja aos relevantes ou a todos) ao receber uma mudança de estado que poderia tornar os dados entre eles inconsistentes.
- Após receber essa notificação, os ConcreteObservers consultam o ConcreteSubject para obter informações para voltar a deixar seu estado consistente.
- A mudança de estado não é sempre feita pelo Subject ou pelo Observer, e a notificação não sempre precisa ser chamada pelo Subject, pode ser chamado por algum observador ou por outros objetos.

3.7. Consequências

3.7.1. Vantagens

- Permite modificar e reutilizar subjects e observadores de forma independente, sem ter que usar ao outro.
- Permite acrescentar novos observadores, sem modificar os subjects e observadores existentes.
- O acoplamento entre subject e observadores é abstrato e mínimo. Devido a isso eles podem estar em camadas diferentes do sistema.
- O subject envia suas notificações a todos seus Observers interessados, sem se importar com quem e quantos eles são. É responsabilidade dos Observers tratar ou ignorar as notificações.

3.7.2. Desvantagens

- Se Subject e Observer forem agrupados, o objeto resultante vai ter que ou residir em ambas camadas ou perder abstração para conseguir residir em uma delas.
- Já que os observadores não conhecem os outros observadores, eles desconhecem o custo total para a mudança no subject. Isso significa que uma cascata de mudanças pode ser gerada por uma operação aparentemente insignificante.
- Critérios de dependência que não tenham sido bem definidos ou mantidos podem levar a atualizações falsas ou não necessárias e que podem ser de difícil detecção.
- Notificações simples de atualização não indicam o que foi mudado no estado. Sem ajuda adicional pode ser difícil para os observadores descobrirem o que mudou, o que pode aumentar o trabalho necessário para descobrir essas mudanças, ou a atualizações desnecessárias.

3.8. Implementação

3.8.1. Modelagem e Código

A ideia original da prova de conceito é de [Brizeno 2011b], com algumas alterações feitas para ficar mais de acordo a teoria de [Gamma et al. 2000]. Abaixo está a modelagem UML da prova de conceito, criada utilizando a ferramenta Astah.

O código foi implementado utilizando o ambiente de desenvolvimento integrado Netbeans e baseado no modelo mostrado acima.

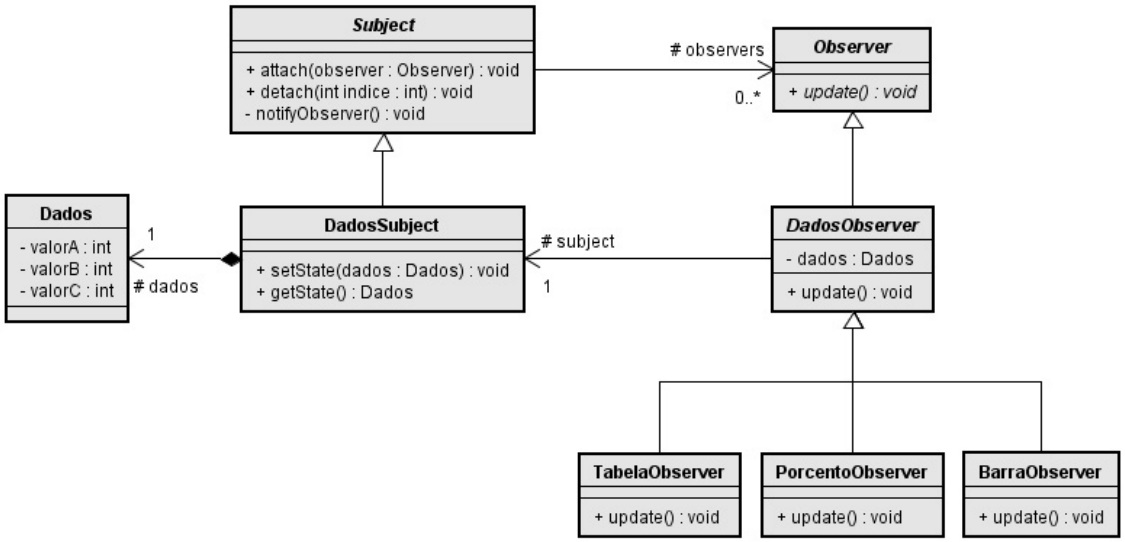

Figura 6. Modelo UML da prova de conceito do padrão Observer

Dados é a classe que contem diferentes valores que são observados pelos observadores. Os métodos equals e hashcode foram implementados para a realização dos testes unitários.

```
public class Dados {

   private int valorA;
   private int valorB;
   private int valorC;

   public Dados() {
      super();
   }

   public Dados(int valorA, int valorB, int valorC) {
      this();
      this.valorA = valorA;
      this.valorB = valorB;
      this.valorC = valorC;
   }

   public int getValorA() {
      return valorA;
   }

   public void setValorA(int valorA) {
      this.valorA = valorA;
   }

   public int getValorB() {
      return valorB;
```

```
    }

    public void setValorB(int valorB) {
        this.valorB = valorB;
    }

    public int getValorC() {
        return valorC;
    }

    public void setValorC(int valorC) {
        this.valorC = valorC;
    }

    @Override
    public boolean equals(Object obj) {
        boolean igual = false;
        if (obj instanceof Dados) {
            Dados dados = (Dados) obj;
            igual = this.getValorA() == dados.getValorA() &&
                this.getValorB() == dados.getValorB() &&
                this.getValorC() == dados.getValorC();
        }
        return igual;
    }

    @Override
    public int hashCode() {
        int hash = 7;
        hash = 31 * hash + this.valorA;
        hash = 31 * hash + this.valorB;
        hash = 31 * hash + this.valorC;
        return hash;
    }

}
```

Subject é a classe abstrata que implementa como adicionar e remover observadores, além de notificar eles. Foi modificado o nome do método notify a notifyObservers, devido a que o nome original já é usado na classe Object no Java.

```
public abstract class Subject {

    protected List<IObserver> observers;

    public Subject() {
        observers = new ArrayList<>();
    }
```

```
    public void attach(IObserver observer) {
        observers.add(observer);
    }

    public void detach(int index) {
        observers.remove(index);
    }

    public void notifyObservers() {
        for (IObserver o : observers) {
            o.update();
        }
    }

}
```

A interface IObserver foi criada para especificar que os observadores devem implementar o método update.

```
public interface IObserver {

    public abstract void update();

}
```

DadosSubject é a especialização da classe Subject, a qual especifica quais serão os dados observados, nesse caso uma instancia da classe Dados e implementa os métodos getState, utilizado no método update dos nossos observadores, e setState, utilizado para modificar o valor do objeto do tipo Dados e chamar o método notifyObservers.

```
public class DadosSubject extends Subject {

    private Dados dados;

    public void setState(Dados dados) {
        this.dados = dados;
        notifyObservers();
    }

    public Dados getState() {
        return dados;
    }

}
```

DadosObservers é uma classe abstrata que implementa a interface IObserver. Ela possui seu estado interno, nesse caso uma instancia da classe Dados e também conhece quem é seu subject, o qual deve ser informado a traves de seu próprio construtor. Além

disso ela implementa o método update para atualizar sua própria informação e o método getDados, neste caso utilizado para realizar os testes unitários.

```
public abstract class DadosObserver implements IObserver {

   protected Dados dados;
   protected final DadosSubject subject;

   public DadosObserver(DadosSubject subject) {
      this.subject = subject;
   }

   @Override
   public void update() {
      dados = subject.getState();
   }

   public Dados getDados() {
      return dados;
   }
}
```

BarraObserver, PorcentoObserver e TabelaObserver são três especializações diferentes da classe DadosObservers, cada uma realizando uma operação diferente ao ser chamado seu método update.

BarraObserver mostra no console uma barra para cada um dos valores do objeto do tipo Dados, feitas pelo caractere '=', ao ser chamado seu método update.

```
public class BarraObserver extends DadosObserver {

   public BarraObserver(DadosSubject subject) {
      super(subject);
   }

   @Override
   public void update() {
      super.update();
      String barraA = "";
      String barraB = "";
      String barraC = "";

      for (int i = 0; i < dados.getValorA(); i++) {
         barraA += '=';
      }

      for (int i = 0; i < dados.getValorB(); i++) {
         barraB += '=';
      }
```

```
        for (int i = 0; i < dados.getValorC(); i++) {
            barraC += '=';
        }

        System.out.println(String.format("Barras:\nValor A:
            %s\nValor B: %s\nValor C: %s",
                barraA, barraB, barraC));
    }
}
```

PorcentoObserver mostra a quantidade que cada valor do objeto dados representa do total de seus valores.

```
public class PorcentoObserver extends DadosObserver {

    public PorcentoObserver(DadosSubject subject) {
        super(subject);
    }

    @Override
    public void update() {
        super.update();
        int soma = dados.getValorA() + dados.getValorB() +
            dados.getValorC();
        final double porcentagemA = 100.0 * dados.getValorA() /
            soma;
        final double porcentagemB = 100.0 * dados.getValorB() /
            soma;
        final double porcentagemC = 100.0 * dados.getValorC() /
            soma;
        System.out.println(String.format("Porcentagem:\nValor A:
            %.2f%%\nValor B: %.2f%%\nValor C: %.2f%%",
                porcentagemA, porcentagemB, porcentagemC));
    }
}
```

TabelaObserver simplesmente mostra cada valor de nosso objeto dados.

```
public class TabelaObserver extends DadosObserver {

    public TabelaObserver(DadosSubject subject) {
        super(subject);
    }

    @Override
    public void update() {
        super.update();
        System.out.println(String.format("Tabela:\nValor A:
            %d\nValor B: %d\nValor C: %d",
```

```
                dados.getValorA(), dados.getValorB(),
                    dados.getValorC()));
    }

}
```

Finalmente a classe cliente contém um método main, e é utilizada para mostrar nossas classes em funcionamento. Nela é criado um DadosSubject, o qual adere três observadores diferentes, um de cada tipo especificado. Posteriormente é modificado o estado do nosso subject duas vezes, para demostrar a execução dos métodos criados.

```
public class Cliente {

    public static void main(String[] args) {
        DadosSubject subject = new DadosSubject();

        subject.attach(new TabelaObserver(subject));
        subject.attach(new PorcentoObserver(subject));
        subject.attach(new BarraObserver(subject));

        subject.setState(new Dados(1, 2, 3));
        subject.setState(new Dados(10, 7, 4));
    }
}
```

3.8.2. Considerações

Segundo Gamma et al, existem vários aspectos relacionados à implementação deste padrão que merecem atenção. Eles são:

1. É possível utilizar outros mecanismos, por exemplo, hash table para mapear subjects e observers, útil ao existir poucos observadores e muitos subjects mas que aumenta o custo de acesso aos observers.
2. Caso um observer tenha mais do que um subject tambem pode ser interessante que o subject se passe a si mesmo como parâmetro em seu método notify para a operação update, para que o observer saiba qual subject analizar.
3. Alem disso, em casos onde podem haver muitas mudanças consecutivas, pode ser desejável que chamar o notify seja responsabilidade do cliente em lugar de ser chamado automaticamente, o que aumenta a eficiência, mas aumenta a chance de erros.
4. É importante garantir que se um subject for deletado, ele seja desassociado de seus observers, para evitar referências ao vazio (null pointers).
5. Antes de chamar o metodo notify, é importante que o subject garante que seu estado seja auto consistente. Para fazer isso pode ser usado o Template Method.
6. Existem dois extremos de informação que o subject pode passar como argumento no metodo update. De um lado temos o push model, no qual o subject envia toda a informação possível aos observadores, graças a isso ele é mais eficiente, mas

menos reutilizável. No outro lado temos o pull model, no qual o subject envia o mínimo possível de informação e os observers devem pedir detalhes, o que torna este método mais reutilizável, mas menos eficiente.

7. É possível melhorar a eficiência do processo ao modificar a implementação de inscrição no subject e permitir que os observadores se registrem somente a eventos de seu interesse. Assim, ao ocorrer um evento, o subject só informa aos observers registrado a ele.
8. As melhorias anteriores podem ser aplicadas utilizando um ChangeManager, que seria uma instancia do padrão Mediator e a qual também utiliza o padrão singleton.
9. Em casos de linguagens sem heranças múltiplas, pode ser interessante combinar a interface da classe Subject e Observer em uma, para definir um objeto que funciona tanto como um como o outro, como no caso do Smalltalk.

3.9. Usos Conhecidos

Um uso bem conhecido para quem trabalha no mundo Java é o listener utilizado no Java Swing. Nele listeners são registrados a objetos visuais da interface de usuário, de acordo ao seu tipo e a ação da qual desejam ser notificados. Este comportamento também pode ser visto em outros toolkits de criação de GUI (Graphic User Interface – Interface Gráfica de Usuário), como no Visual Studio, já seja para a criação de Windows Form ou Universal Windows Applications ou, de acordo a [Gamma et al. 2000], no MVC (Model / View / Controller) da Smalltalk.

3.10. Padrões relacionados

Mediator e Singleton podem ser usados para criar um ChangeManager, o qual atua como mediador entre subjects e observadores, e é único e globalmente acessível.

Template Method pode ser usado para manter consistência interna dentro dos subjects.

4. Padrão Proxy

Proxy também conhecido como “Surrogate”, fornece um substituo ou um marcador de outro objeto para poder marca-lo e controlar o seu acesso.

4.1. Motivação

A razão de controlar um objeto é adiar a o custo da sua criação até o momento em que for realmente necessário o usá-lo. Como por exemplo um documento que possui uma grande representação gráfica, causem um grande custo computacional. Tendo como objetivo que tais documentos, devessem ser abertos rapidamente ao invés de abri-los de uma vez só que ira causar um grande custo.

Para reduzir o custo a solução é usar um objeto Proxy que substitui temporária mente o objeto real, que funciona exatamente como a imagem real quando for necessária. O proxy apenas cria a imagem real quando o editor de documentos solicita ao mesmo exibir a si próprio invocando sua operação Draw. O proxy repassa as solicitações para imagem, portanto deve manter referência para imagem após cria-la. Mesmo que as imagens sejam de arquivos separados o proxy usa o nome do arquivo como referência do objeto real. O proxy também armazena sua extensão como altura e largura.[Gamma et al. 2000, **?**]

4.2. Aplicabilidade

O padrão proxy será aplicável sempre que o houver a necessidade de uma referência mais versátil, ou sofisticada, listadas aqui algumas situações onde o padrão proxy é comum.

- Remote proxy, fornece um representante local para objeto em espaço de endereça mento diferente. Podendo ocultar o fato de um objeto reside em um espaço de e endereçamento diferente.
- Virtual proxy, cria objetos sob demanda.
- Protection proxy, controla o acesso ao objeto original.
- Smart reference, é um substituto para um simples pointer que executa ações adicionais quanto um objeto é acessado.

4.3. Estrutura

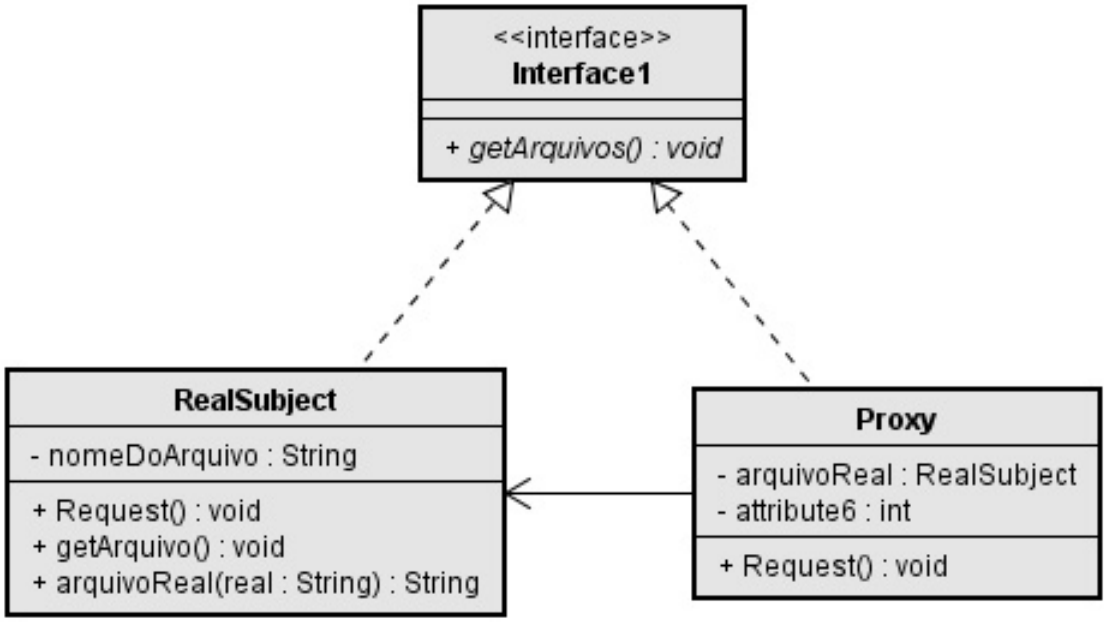

Figura 7. Modelo UML da estrutura teórica do padrão Proxy

4.4. Participantes

- Proxy
  - Proxy mantém uma referência que permite o proxy acessar o objeto real.
  - Proxy fornece uma interface idêntica ao Subject, de modo que o proxy possa substituir o objeto real.
  - controla o acesso ao objeto, sendo responsavel pela sua criação e exclusão.
- Subject
  - é a interface comum para o RealSubject e o proxy, sendo assim o Proxy pode ser chamado em qualquer lugar que um RealObject é esperado
- RealSubject
  - é o objeto real que o proxy representa

4.5. Colaboração

- Dependendo do tipo, o Proxy repassa para RealSubject quando for necessário

4.6. Consequências

O padrão proxy possui um nivel de referência indireta no acesso ao objeto. Essa referência adicional tem diversos usos, dependendo do tipo de proxy:

- Um proxy remoto pode ocultar o fato de um objeto estar endereçado em um local diferente.
- O proxy virtual pode criar um objeto sob demanda.
- O protect proxy quanto Smarte reference tem adicionais de organização quando um objeto é acessado

Uma outra otimização do padrão proxy que pode ser ocultada do cliente é o copy – on - write. Ela está associado à criação de objetos sob demanda, ou seja, ele pega um objeto caro no sentido computacional e o adia assim o processo de cópia somente é usado quando ele for modicado.

Para o copy-on-write, funcionar o objeto deve ter suas referências contadas. Como copiar o proxy não fará nada mais do que incrementar está contagem. Então somente quando o cliente solicitar uma operação que modifique o objeto o proxy fará a cópia. Sendo assim o proxy tende a diminuir a contagem de referências, se contagem chegar a zero objeto é deletado.

4.7. Implementação

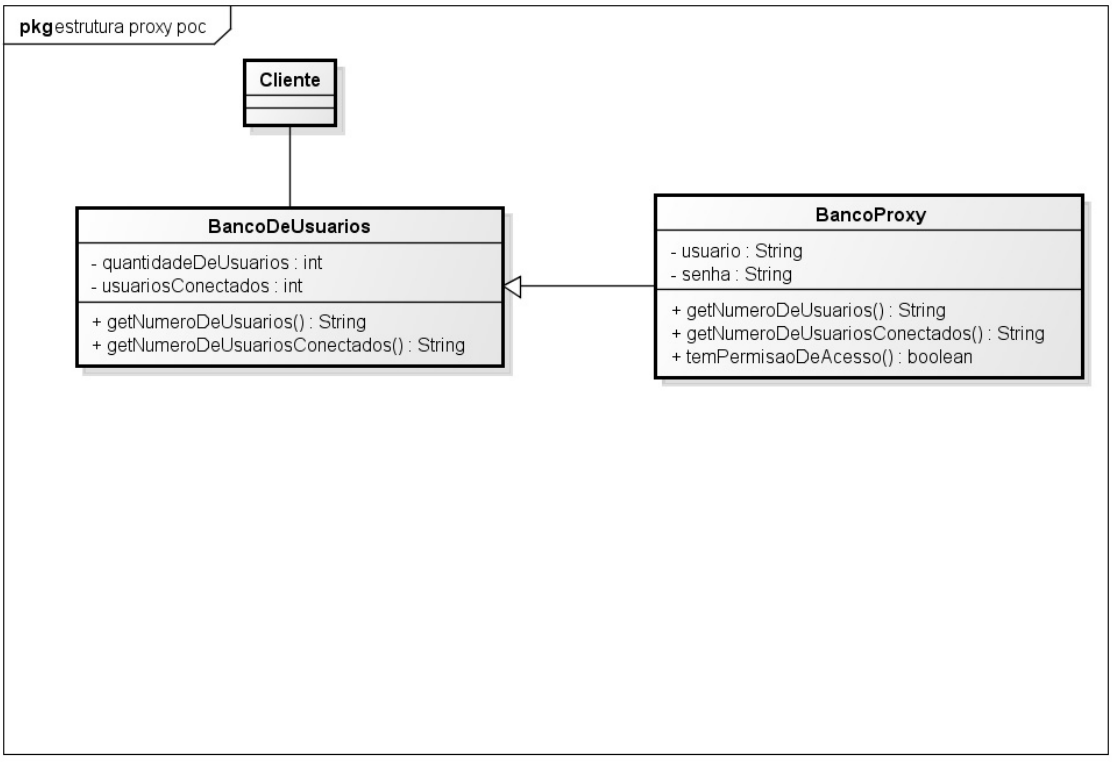

Figura 8. Modelo UML da prova de conceito do padrão Proxy

O código foi implementado utilizando o ambiente de desenvolvimento integrado Netbeans e baseado na ideia apresentada no UML feita primeiramente por [Brizeno 2011b, **?**]. Esta classe possui as caracterisicas, as quais o padrão proxy vai controlar o acesso, indica quantos usuarios estão cadastrados e conectados.

```
 public class BancoUsuarios {

   private int quantidadeDeUsuarios;
   private int usuariosConectados;

   public BancoUsuarios() {
      quantidadeDeUsuarios = (int) (Math.random() * 100);
      usuariosConectados = (int) (Math.random() * 10);
   }

   public String getNumeroDeUsuarios() {
      return new String("Total de usurios: " +
         quantidadeDeUsuarios);
   }

   public String getUsuariosConectados() {
      return new String("Usurios conectados: " +
         usuariosConectados);
   }

}
```

A classe BancoProxy (Esta classe ultiliza o padrão protection proxy), controla o acesso a classe acima (BancoUsuarios) ele ira o substituir até verificar o acesso a ela, criando referências no caso ”Usuário”e ”senha”caso referências forem de acordo a classe real será acessada caso contrario é retornado nulo.

```
public class BancoProxy extends BancoUsuarios{
    protected String usuario, senha;

   public BancoProxy(String usuario, String senha) {
      this.usuario = usuario;
      this.senha = senha;
   }

   @Override
   public String getNumeroDeUsuarios() {
      if (temPermissaoDeAcesso()) {
         return super.getNumeroDeUsuarios();
      }
      return null;
   }

   @Override
   public String getUsuariosConectados() {
      if (temPermissaoDeAcesso()) {
         return super.getUsuariosConectados();
      }
      return null;
```

```
    }

    private boolean temPermissaoDeAcesso() {
        return "admin".equals(usuario) &&
            "admin".equals(senha);
    }
```

E por fim a classe TestBancoProx, que contém o método main onde monstra a classe funcionando. Nela é criado um novo objeto com usuário e senha distintas, das geradas pelo método BancoUsuario assim aplicando um teste de falha ao padrão proxy.

```
public class TesteBancoProxy {

    public static void main(String[] args) {

        System.out.println("Hacker acessando");
        BancoUsuarios banco = new BancoProxy("Hacker",
            "1234");
        System.out.println(banco.getNumeroDeUsuarios());
        System.out.println(banco.getUsuariosConectados());

        System.out.println("\nAdministrador acessando");
        banco = new BancoProxy("admin", "admin");
        System.out.println(banco.getNumeroDeUsuarios());
        System.out.println(banco.getUsuariosConectados());
    }
}
```

4.7.1. Vantagens

- Incersão de funcionalidade não aclopada.
- Possibilita a troca de objetos em tempo de execução.
- É flexível.

4.7.2. Desvantagens

- O objeto proxy deve ser criado em algum momento.
- Possiveis impactos na performace

4.8. Usos Conhecidos

O Proxy É muito utilizado pela tecnologia J2EE, pelo objeto 'javax.ejb.EJBObject', que representa uma referência remota ao EJB. O Framework Hibernate também utiliza Proxy em uma técnica utilizada para acessar o banco de dados somente quando for necessário.

4.9. Padrões relacionados

Os padrões relacionados ao proxy são o Adapter, que fornece uma interface para o objeto que adapta.Entretanto o proxy, fornece essa mesma interface mas como seu objeto.

A também o decorator, que mesmo tendo implementações semelhantes ao proxy, mas contudo ele adiciona responsabilidades ao objeto enquanto o proxy controla seu acesso

Referências

Brizeno, M. (2011a). Mão na massa: Builder. https://brizeno.wordpress.com/2011/10/17/mao-na-massa-builder/. Accessed: 2016-05-11.

Brizeno, M. (2011b). Mão na massa: Observer. https://brizeno.wordpress.com/2011/10/17/mao-na-massa-observer/. Accessed: 2016-05-11.

de Almeida Ribeiro, A. A. (2013). Padrão de projeto: Builder. http://www.dpi.ufv.br/projetos/apri/?page_id=695. Accessed: 2016-05-15.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (2000). *Padrões de projeto: soluções reutilizáveis de software*. Bookman.

José, E. (2013). Aplicando os padrões builder. http://www.devmedia.com.br/design-patterns-aplicando-os-padroes\ \-builder-singleton-e-prototype/31023. Accessed: 2016-05-15.

Macoratti, J. C. (2013). O padrão de projeto decorator. http://www.macoratti.net/13/02/net_decor1.htm. Accessed: 2016-05-02.

Wikipedia (2011). Decorator. https://pt.wikipedia.org/wiki/Decorator. Accessed: 2016-05-4.