

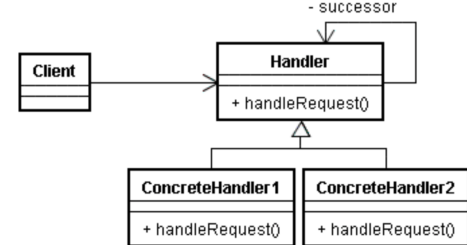
# Padrões Comportamentais Exemplos Práticos

2015/2016

João Alegria | 68861



# Chain of Responsibility



## PlanetHandler.java

```
public abstract class PlanetHandler{
    PlanetHandler sucessor;
    public void setSucessor(PlanetHundler sucessor){
        this.sucessor = sucessor;
    }
    public abstract void handlePedido(PlanetEnum pedido);
}
```

Uso de uma classe abstrata para manipulação das subclasses e para que estas possam implementar o método `setSucessor()`. Essa classe é chamada **PlanetHandler**. As subclasses de terão que implementar `handlePedido()`.

## PlanetEnum.java

```
public enum PlanetEnum{
    Mercury, Venus, Earth, Mars, Jupiter, Saturn,
    Uranus, Nepture;
}
```

Este exemplo irá utilizar um enum de planetas chamado **PlanetEnum**.

## MercuryHandler.java

```
public class MercuryHandler extends PlanetHandler{
    public void handlePedido(PlanetEnum pedido){
        if(pedido == PlanetEnum.Mercury){
            System.out.println("MercuryHandler suporta" + pedido);
            System.out.println("Mercurio é quente");
        } else{
            System.out.print("MercuryHandler não suporta" + pedido);
            if(sucessor != null){
                sucessor.handlePedido(pedido);
            }
        }
    }
}
```

define o próximo sucessor

**MercuryHandler** é subclasse de **PlanetHandler** e implementa o método `handlePedido()`. Se o pedido for um **PlanetEnum.Mercury**, vai processar o pedido. Caso contrario, o pedido é passado ao sucessor, caso exista.

## VenusHandler.java

```
public class VenusHandler extends PlanetHandler{
    public void handlePedido(PlanetEnum pedido){
        if(pedido == PlanetEnum.Venus){
            System.out.println("VenusHandler suporta" + pedido);
            System.out.println("Venus é venenoso");
        } else{
            System.out.print("VenusHandler não suporta" + pedido);
            if(sucessor != null){
                sucessor.handlePedido(pedido);
            }
        }
    }
}
```

**VenusHandler** é semelhante ao **MercuryHandler**, porém lida com pedidos **PlanetEnum.Venus**

## EarthHandler.java

```
public class EarthHandler extends PlanetHandler{
    public void handlePedido(PlanetEnum pedido){
        if(pedido == PlanetEnum.Earth){
            System.out.println("EarthHandler suporta" + pedido);
            System.out.println("Terra é confortável");
        } else{
            System.out.print("EarthHandler não suporta" + pedido);
            if(sucessor != null){
                sucessor.handlePedido(pedido);
            }
        }
    }
}
```

**EarthHandler** procesa pedidos **PlanetEnum.Earth**

### Demo.java

```
public class Demo{
    public static void main(String[] args){
        PlanetHandler chain = setUpChain();
        chain.handlePedido(PlanetEnum.Venus);
        chain.handlePedido(PlanetEnum.Mercury);
        chain.handlePedido(PlanetEnum.Earth);
        chain.handlePedido(PlanetEnum.Jupiter);
    }
    public static PlanetHandler setUpChain(){
        PlanetHandler mercuryHandler = new MercuryHandler();
        PlanetHandler venusHandler = new VenusHandler();
        PlanetHandler earthHandler = new EarthHandler();
        mercuryHandler.setSucessor(venusHandler);
        venusHandler.setSucessor(earthHandler);
        return mercuryHandler;
    }
}
```

Retorna o primeiro da cadeia, neste caso, mercuryHandler, e só depois venusHandler e earthHandler.

Repara que, se um manipulador não suportar o pedido, ele passa o pedido para o próximo manipulador.

MercuryHandler não suporta Venus  
VenusHandler suporta Venus  
Venus é venenoso

MercuryHandler suporta Mercurio  
Mercurio é quente

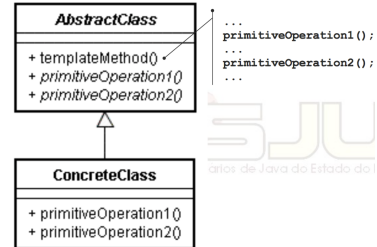
MercuryHandler não suporta Terra  
VenusHandler não suporta Terra  
EarthHandler suporta Terra  
Terra é confortável

MercuryHandler não suporta Jupiter  
VenusHandler não suporta Jupiter  
EarthHandler não suporta Jupiter

A classe **Demo** é a classe cliente. Cria uma cadeia de manipuladores começando por MercuryHandler, depois VenusHandler e EarthHandler. O método setUpChain() retorna a cadeia para a Main() através de uma referência PlanetHandler. São feitos quatro pedidos, onde os pedidos são Mercurio, Vénus, Terra e Júpiter.

Observe que o último pedido feito na cadeia é Júpiter. Este pedido não é suportado por qualquer manipulador, demonstrando que um pedido não tem que ser obrigatoriamente manipulado. Se quisermos, é possível criar um manipulador OtherPlanets e colocá-lo no fim da cadeia para manipular os pedidos que não foram manipulados anteriormente. Isto demonstraria que nós podemos definir manipulador mais concretos no início da cadeia e mais gerais no final.

# Template Method



## Meal.java

```
public abstract class Meal{
    public final void doMeal(){
        prepararIngredientes();
        cook();
        eat();
        cleanUp();
    }
    public abstract void prepararIngredientes();
    public abstract void cook();
    public void eat() {System.out.println("Que bom!");}
    public abstract void cleanUp();
}
```

**Meal** é uma classe abstrata com um Template Method chamado `doMeal()` que define as etapas envolvidas numa refeição. O método é definido como final para não ser substituído (Override). O algoritmo definido por `doMeal()` consiste em 4 etapas: `prepararIngredientes()`, `cook()`, `eat()` e `cleanUp()`. O método `eat()` é implementado porém as subclasses podem fazer Override. `CleanUp()`, `cook()` e `prepararIngredientes()` são métodos declarados como abstratos portanto as suas subclasses precisam de implementá-los.

## HamburgerMeal.java

```
public class HamburgerMeal extends Meal{
    @Override public void prepararIngredientes(){
        System.out.print("Preparar hamburger, pão e batatas");
    }
    @Override public void cook(){
        System.out.println("Cozinhando hamburger e batatas");
    }
    @Override public void cleanUp(){
        System.out.println("Limpendo...");
    }
}
```

A classe **HamburgerMeal** estende **Meal** e implementa os três métodos abstratos.

## TacoMeal.java

```
public class TacoMeal extends Meal{
    @Override public void prepararIngredientes(){
        System.out.print("Preparar carne moída");
    }
    @Override public void cook(){
        System.out.println("Cozinhando carne moída");
    }
    @Override public void eat(){
        System.out.println("Os tacos são deliciosos");
    }
    @Override public void cleanUp(){
        System.out.println("Lavar os pratos...");
    }
}
```

A classe **TacoMeal** implementa três métodos abstratos e substitui o método `eat()`.

**Demo.java**

```
public class Demo{  
    public static void main(String[] args){  
        Meal meal1 = new HamburgerMeal();  
        meal1.doMeal();  
  
        Meal meal2 = new TacoMeal();  
        meal2.doMeal();  
    }  
}
```



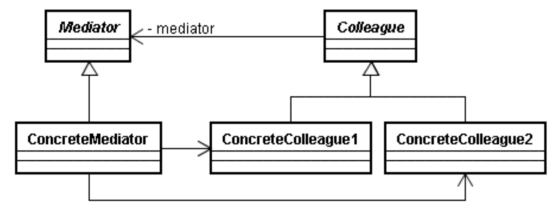
Preparar hamburger, pão e batatas  
Cozinhando hamburger e batatas  
Que bom!  
Limpendo...

Preparar carne moída  
Cozinhando carne moída  
Os tacos são deliciosos  
Lavar os pratos...

A classe **Demo** cria um objeto **HamburgerMeal** e chama o método **doMeal()**. Ele cria um objeto **TacoMeal** e chama **doMeal()** no objeto **TacoMeal**.

O Template Method permite-nos definir as etapas num algoritmo e passar a implementação destas etapas para as subclasses.

# Mediator



Criação de uma classe **Mediator** (sem implementar uma interface de mediador neste exemplo). Este mediador vai intermediar a comunicação entre dois compradores (um comprador sueco e outro francês), um vendedor inglês e um conversor de moedas. O mediador tem como referências os dois compradores, o vendedor e o conversor. Tem métodos para que estes tipos de objetos possam ser registados. Tem também um método `placeBid()` que leva como parâmetros uma licitação (`bid`) e uma unidade de moeda como parâmetros. Converte este montante para um montante dolar através de `dollarConverter()`. Posto isto, pergunta ao vendedor se a oferta foi aceite e retorna a resposta.

## Mediator.java

```
public class Mediator{
    Buyer compradorSueco;
    Buyer compradorFrances;
    AmericanSeller vendedorAmericano;
    DollarConverter conversorDolar;

    public Mediator(){}
    public void registrarCompradorSueco(SwedishBuyer compradorSueco){
        this.compradorSueco = compradorSueco;
    }
    public void registrarCompradorFrances(FrenchBuyer compradorFrances){
        this.compradorFrances = compradorFrances;
    }
    public void registrarVendedorAmericano(AmericanSeller vendedorAmericano){
        this.vendedorAmericano = vendedorAmericano;
    }
    public void registrarConversorDolar(DollarConverter conversorDolar){
        this.conversorDolar = conversorDolar;
    }
    public boolean placeBid(float bid, String tipoMoeda){
        float dollarAmount = conversorDolar.converterParaDolar(bid,tipoMoeda);
        return vendedorAmericano.isBidAccepted(dollarAmount);
    }
}
```

Esta é a class **Buyer**. As classes `SwedishBuyer` e `FrenchBuyer` são subclasses de `Buyer`. O `Buyer` tem um tipo de moeda como parâmetro e também uma referência para o mediador. A classe `Buyer` tem um método `attemptToPurchase()`, este método envia uma licitação para o método `placeBid()` do mediador e retorna a resposta do mediador.

## Buyer.java

```
public class Buyer{
    Mediator mediator;
    String tipoMoeda;
    public Buyer(Mediator mediator, String tipoMoeda){
        this.mediator = mediator;
        this.tipoMoeda = tipoMoeda;
    }
    public boolean attemptToPurchase(float bid){
        System.out.println("Comprador licita" + bid + " " + tipoMoeda);
        return mediator.placeBid(bid,tipoMoeda);
    }
}
```

**SwedishBuyer.java**

```
public class SwedishBuyer extends Buyer{
    public SwedishBuyer(Mediator mediator){
        super(mediator,"coroa");
        this.mediator.registarCompradorSueco(this);
    }
}
```

A classe **SwedishBuyer** é uma subclasse de Buyer. No construtor, definimos o tipo de moeda como "coroa". Registamos também o SwedishBuyer com o mediator para que o mediator tenha conhecimento de SwedishBuyer.

**FrenchBuyer.java**

```
public class FrenchBuyer extends Buyer{
    public FrenchBuyer(Mediator mediator){
        super(mediator,"euro");
        this.mediator.registarCompradorFrances(this);
    }
}
```

A classe **FrenchBuyer** é semelhante à classe SwedishBuyer, porém o tipo de moeda é "euro".

No construtor da classe **AmericanSeller**, a classe recebe a referência para o mediator e o preço em dolares. Este é o preço final de um produto a ser vendido. O vendedor registra o mediator como AmericanSeller. O método isBidAccepted( ) do vendedor leva uma bid (em dolares). Caso o bid supere o preço (em dolares), a oferta é considerada válida e é retornado true. Caso contrário, é retornado false.

**AmericanSeller.java**

```
public class AmericanSeller{
    Mediator mediator;
    float preçoDolares;
    public AmericanSeller(Mediator mediator, float preçoDolares){
        this.mediator = mediator;
        this.preçoDolares = preçoDolares;
        this.mediator.registarVendedorAmericano(this);
    }
    public boolean isBidAccepted(float bidDolares){
        if(bidDolares >= preçoDolares){
            System.out.println("Vendedor aceita licitação de:" + bidDolares + "dolares");
            return true;
        } else{
            System.out.println("Vendedor rejeita licitação de:" + bidDolares + "dolares");
            return false;
        }
    }
}
```



**DollarConverter** obtém uma referência para o mediator e registra o mediator como DollarConverter. Esta classe possui métodos para converter valores em euros, coroas e dólares.

#### **DollarConverter.java**

```
public class DollarConverter{
    Mediator mediator;
    public static final float DOLLAR_UNIT = 1.0f;
    public static final float EURO_UNIT = 0.7f;
    public static final float COROA_UNIT = 8.0f;
    public DollarConverter(Mediator mediator){
        this.mediator = mediator;
        mediator.registarConversorDolar(this);
    }
    private float converterEurosParaDolares(float euros){
        float dolares = euros * (DOLLAR_UNIT/EURO_UNIT);
        System.out.println(euros + "euros=" + dolares + "dolares");
    }
    private float converterCoroasParaDolares(float coroas){
        float dolares = coroas * (DOLLAR_UNIT/COROA_UNIT);
        System.out.println(coroas + "coroas=" + dolares + "dolares");
    }
    private float converterParaDolares(float montante, String tipoMoeda){
        if("coroa.equalsIgnoreCase(tipoMoeda){
            return converterCoroasParaDolares(montante);
        }
        else return converterEurosParaDolares(montante);
    }
}
```

A classe Demo demonstra o padrão Mediator. É criado um objeto SwedishBuyer e um objeto FrenchBuyer e um AmericanSeller e o preço da venda definida em 10 dolares. Em seguida, cria um DollarConverter. Todos estes objetos registam-se com o mediator nos seus construtores. O comprador sueco começa com um lance de 55 coroas e mantém licitações em incrementos de 15 coroas até que seja aceite. O comprador francês começa a licitar em 3 euros e mantém a licitar com incrementos de 1.50 euros até que a oferta seja aceite.

#### **Demo.java**

```
public class Demo{
    public static void main(String[] args){
        Mediator mediator = new Mediator();
        Buyer compradorSueco = new SwedishBuyer(mediator);
        Buyer compradorFrances = new FrenchBuyer(mediator);
        float preçoVendaDolar = 10.0f;

        AmericanSeller vendedorAmericano = new AmericanSeller(mediator, preçoVendaDolar);
        DollarConverter conversorDolar = new DollarConverter(mediator);

        float licitacaoCoroas = 55.0f;
        while(!compradorSueco.attemptToPurchase(licitacaoCoroas){
            licitacaoCoroas += 15.0f;
        }

        float licitacaoEuros = 3.0f;
        while(!compradorFrances.attemptToPurchase(licitacaoEuros){
            licitacaoEuros += 1.5f;
        }
    }
}
```



Comprador licita 55 coroas  
55 coroas = 6.875 dólares  
Vendedor rejeita a licitação de 6.875 dólares

Comprador licita 70 coroas  
70 coroas = 8.75 dólares  
Vendedor rejeita a licitação de 8.75 dólares

Comprador licita 85 coroas  
85 coroas = 10.625 dólares  
Vendedor aceita a licitação de 10.625 dólares

Comprador licita 3 euros  
3 euros = 4.28 dólares  
Vendedor rejeita a licitação de 4.28 dólares

Comprador licita 4.5 euros  
4.5 euros = 6.43 dólares  
Vendedor rejeita a licitação de 6.43 dólares

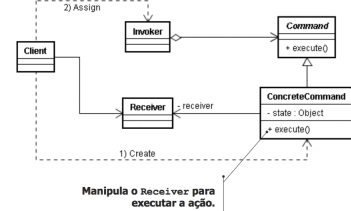
Comprador licita 6 euros  
6 euros = 8.57 dólares  
Vendedor rejeita a licitação de 8.57 dólares

Comprador licita 7.5 euros  
7.5 euros = 10.71 dólares  
Vendedor aceita a licitação de 10.71 dólares

Neste exemplo do padrão Mediator, note-se que todas as comunicações entre os objetos (compradores, vendedor e conversor) ocorrem via mediator. O padrão Mediator ajuda a reduzir o número de referências de objetos necessários (através da composição). As classes proliferam no projeto quando o projeto cresce.

# Command

O padrão Command pode ser usado para executar a funcionalidade de “Undo - Retroceder”. Neste caso, a interface Command deve incluir um método unExecute().



## Command.java

```
public interface Command{
    public void execute();
}
```

Interface **Command** com o método execute().

## LunchCommand.java

```
public class LunchCommand implements Command{
    Lunch lunch;
    public LunchCommand (Lunch lunch){
        this.lunch = lunch;
    }
    @Override public void execute(){
        lunch.makeLunch();
    }
}
```

**LunchCommand** implementa Command. Contém uma referência para almoço, um recetor. O método execute() invoca a ação apropriada no receptor.

## DinnerCommand.java

```
public class DinnerCommand implements Command{
    Dinner dinner;
    public DinnerCommand (Dinner dinner){
        this.dinner = dinner;
    }
    @Override public void execute(){
        dinner.makeDinner();
    }
}
```

**DinnerCommand** é semelhante à classe anterior. Contém uma referência para jantar, um receptor. O método execute() invoca a ação makeDinner() do objeto jantar.

## Lunch.java

```
public class Lunch{
    public void makeLunch(){
        System.out.println("Preparar almoço");
    }
}
```

A classe **Lunch** é um receptor.

## Dinner.java

```
public class Dinner{
    public void makeDinner(){
        System.out.println("Preparar jantar");
    }
}
```

A classe **Dinner** é um receptor.

## MealInvoker.java

```
public class MealInvoker{
    Command command;
    public MealInvoker(Command command){
        this.command = command;
    }
    public void setCommand(Command command){
        this.command = command;
    }
    public void invoke(){
        command.execute();
    }
}
```

**MealInvoker** é a classe Invoker. Contém uma referência para o Command para o invocar. O método invoke() chama o método execute() do Command.

A classe **Demo** demonstra o padrão Command. Instancia um objeto Lunch (receptor) e cria um LunchCommand (Concret Command) com o Lunch. O LunchCommand é referenciado por uma referência de interface de Command. Em seguida, realizamos o mesmo procedimento no jantar e objetos de DinnerCommand. Depois disso, criamos um objeto MealInvoker com lunchCommand e chamamos o método invoke() do mealInvoker. Após isso, define-se o comando do mealInvoker para dinnerCommand e mais uma vez chama-se o invoke() em mealInvoker.

#### **Demo.java**

```
public class Demo {
    public static void main(String[] args) {
        Lunch lunch = new Lunch(); // receiver
        Command lunchCommand = new LunchCommand(lunch); // concrete command

        Dinner dinner = new Dinner(); // receiver
        Command dinnerCommand = new DinnerCommand(dinner); // concrete command

        MealInvoker mealInvoker = new MealInvoker(lunchCommand); // invoker
        mealInvoker.invoke();
        mealInvoker.setCommand(dinnerCommand);
        mealInvoker.invoke();
    }
}
```

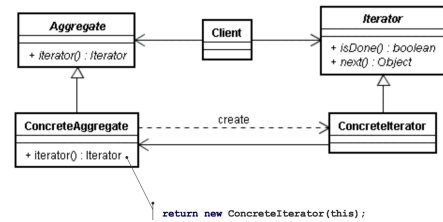
# Iterator

## Item.java

```
public class Item{
    String nome;
    float preço;
    public Item(String nome, float preço){
        this.nome = nome;
        this.preço = preço;
    }
    public String toString(){
        return nome + ":$" + preço;
    }
}
```

## Menu.java

```
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;
public class Menu{
    List<Item> menuItems;
    public Menu(){
        menuItems = new ArrayList<Item>();
    }
    public void addItem(Item item){
        menuItems.add(item);
    }
    public Iterator<Item> iterator(){
        return new MenuItemIterator();
    }
    class MenuItemIterator implements Iterator<Item>{
        int currentIndex = 0;
        @Override public boolean hasNext(){
            if(currentIndex >= menuItems.size()){
                return false;
            } else return true;
        }
        @Override public Item next(){
            return menuItems.get(currentIndex++);
        }
        @Override public void remove(){
            menuItems.remove(--currentIndex);
        }
    }
}
```



Classe **Item**, representa um item num menu. Um item tem um nome e um preço.

Classe **Menu** tem lista de itens do tipo Item. Os itens podem ser adicionados através do método `addItem()`. O método `iterator()` retorna um iterado de itens de menu. A classe `MenuItemIterator` é uma classe interna de `Menu` que implementa a interface de iterado para objetos de `Item`.

### Demo.java

```
public class Demo{
public static void main(String[] args){
    Item i1 = new Item("spaghetti", 7.50f);
    Item i2 = new Item("hamburger", 6.00f);
    Item i3 = new Item("sandesh", 6.50f);

    Menu menu = new Menu();

    menu.addItem(i1);
    menu.addItem(i2);
    menu.addItem(i3);

    System.out.println("Mostrar menu:");
    Iterator<Item> iterator = menu.iterator();
    while(iterator.hasNext()){
        Item item = iterator.next();
        System.out.println(item);
    }

    System.out.println("Removendo o último item retornado:");
    iterator.remove();

    System.out.println("Mostrar menu:");
    iterator = menu.iterator();
    while(iterator.hasNext()){
        Item item = iterator.next();
        System.out.println(item);
    }
}
```



```
Mostrar menu:
spaghetti: $7.5
hamburger: $6.0
sandesh: $6.5
```

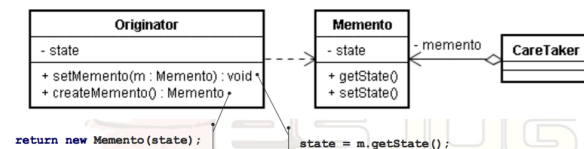
```
Removendo o último item retornado...
```

```
Mostrar menu:
spaghetti: $7.5
hamburger: $6.0
```

A classe **Demo** demonstra o padrão Iterator. Cria três itens e adiciona-os para o objeto menu. Em seguida, ele obtém um iterator de Item do objeto menu e itera sobre os itens no menu. Depois disso, ele chama o remove( ) para remover o último item obtido pelo iterator. Em seguida, obtém um novo objeto iterator do menu e mais uma vez itera sobre os itens de menu.

Nota que, uma vez que o menu utiliza uma coleção Java, poderíamos ter usado um iterator obtido para a lista de menu invés de implementarmos o nosso próprio Iterator.

# Memento



## DietInfo.java

```

// originator - object whose state we want to save

public class DietInfo {
    String personName;
    int dayNumber;
    int weight;
    public DietInfo(String personName, int dayNumber, int
        weight){
        this.personName = personName;
        this.dayNumber = dayNumber;
        this.weight = weight;
    }
    public String toString(){
        return "Name: " + personName + ", day number: " +
            dayNumber + ", weight: " + weight;
    }
    public void setDayNumberAndWeight(int dayNumber, int
        weight){
        this.dayNumber = dayNumber;
        this.weight = weight;
    }
    public Memento save() {
        return new Memento(personName, dayNumber, weight);
    }
    public void restore(Object objMemento) {
        Memento memento = (Memento) objMemento;
        personName = memento.mementoPersonName;
        dayNumber = memento.mementoDayNumber;
        weight = memento.mementoWeight;
    }
}

// memento - object that stores the saved state of the
// originator

private class Memento {
    String mementoPersonName;
    int mementoDayNumber;
    int mementoWeight;
    public Memento(String personName, int dayNumber,
        int weight) {
        mementoPersonName = personName;
        mementoDayNumber = dayNumber;
        mementoWeight = weight;
    }
}
}

```

A **DietInfo** é nossa classe *Originator*. A intenção é ser capaz de salvar e restaurar o seu estado. Ele contém 3 campos: um campo de nome, o número de dias da dieta e o peso no dia especificado da dieta.

Essa classe contém uma classe interna privada chamada Memento. Esta é a nossa classe de recordação que é usado para salvar o estado de DietInfo. Memento tem 3 campos que representam o nome, o número e o peso.

Observe o método save ( ) do DietInfo. Isso cria e retorna um objeto de Memento que retorna o objeto Memento é armazenado pelo caretaker. Note-se que DietInfo.Memento não é visível, então o caretaker não pode fazer referência a DietInfo.Memento. Em vez disso, ele armazena a referência como um objeto.

O método de restore ( ) de DietInfo é usado para restaurar o estado do DietInfo. O caretaker passa no Memento (como um objeto). O Memento é convertido num objeto de Memento... e depois o estado do objeto DietInfo é restaurado, copiando os valores do Memento.

**DietInfoCaretaker** é a classe caretaker que é usada para armazenar o estado (ou seja, o Memento) de um objeto DietInfo (ou seja, o originator). O Memento é armazenado como um objeto uma vez que DietInfo.Memento não é visível pelo caretaker. Isto protege a integridade dos dados armazenados no objeto Memento. O método `saveState()` do caretaker guarda o estado do objeto DietInfo. O método `restoreState()` do caretaker restaura o estado do objeto DietInfo.

```
DietInfoCaretaker.java
// caretaker - saves and restores a DietInfo object's
// state via a memento

// note that DietInfo.Memento isn't visible to the caretaker so
// we need to cast the memento to Object

public class DietInfoCaretaker {
    Object objMemento;
    public void saveState(DietInfo dietInfo) {
        objMemento = dietInfo.save();
    }
    public void restoreState(DietInfo dietInfo) {
        dietInfo.restore(objMemento);
    }
}
```

```
public class MementoDemo {
    public static void main(String[] args) {
        // caretaker
        DietInfoCaretaker dietInfoCaretaker = new
            DietInfoCaretaker();

        // originator
        DietInfo dietInfo = new DietInfo("Fred", 1, 100);
        System.out.println(dietInfo);
        dietInfo.setDayNumberAndWeight(2, 99);

        System.out.println(dietInfo);
        System.out.println("Saving state.");


        dietInfoCaretaker.saveState(dietInfo);
        dietInfo.setDayNumberAndWeight(3, 98);

        System.out.println(dietInfo);
        dietInfo.setDayNumberAndWeight(4, 97);

        System.out.println(dietInfo);
        System.out.println("Restoring saved state.");

        dietInfoCaretaker.restoreState(dietInfo);
        System.out.println(dietInfo);
    }
}
```

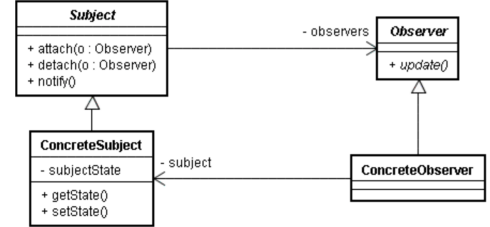
A classe **MementoDemo** demonstra o padrão Memento. Cria um caretaker e, em seguida, um objeto DietInfo. O estado do objeto DietInfo é alterado e exibido. A qualquer momento, o caretaker guarda o estado do objeto DietInfo. Depois disso, o estado do objeto do DietInfo é alterado e exibido. Depois disso, o caretaker restaura o estado do objeto DietInfo. Podemos verificar esta restauração o estado do objeto DietInfo e exibindo.



```
Name: Fred, day number: 1, weight: 100
Name: Fred, day number: 2, weight: 99
Saving state.
Name: Fred, day number: 3, weight: 98
Name: Fred, day number: 4, weight: 97
Restoring saved state.
Name: Fred, day number: 2, weight: 99
```



# Observer



## WeatherSubject.java

```
public interface WeatherSubject{
    public void addObserver(WeatherObserver weatherObserver);
    public void removeObserver(WeatherObserver weatherObserver);
    public void doNotify();
}
```

Interface

## WeatherSubject

declara três métodos:  
addObserver(),  
removeObserver(),  
doNotify().

## WeatherObserver.java

```
public interface WeatherObserver{
    public void doUpdate(int temperatura);
}
```

Também é criada uma interface para os observadores, **WeatherObserver**. Possui um método, doUpdate().

## WeatherStation.java

```
import java.util.HashSet;
import java.util.Iterator;
import java.util.Set;
public class WeatherStation implements WeatherSubject{
    Set<WeatherObserver> weatherObservers;
    int temperatura;
    public WeatherStation(int temperatura){
        weatherObservers = new HashSet<WeatherObserver>();
        this.temperatura = temperatura;
    }
    @Override addObserver(WeatherObserver weatherObserver){
        weatherObservers.add(weatherObserver);
    }
    @Override removeObserver(WeatherObserver weatherObserver){
        weatherObservers.remove(weatherObserver);
    }
    @Override void doNotify(){
        Iterator<WeatherObserver> it = weatherObservers.iterator();
        while(it.hasNext()){
            WeatherObserver weatherObserver = it.next();
            weatherObserver.doUpdate(temperatura);
        }
    }
    public void setTemperatura(int temperatura){
        System.out.println("Estação meteorológica define temperatura para:" + new Temperatura());
        temperatura = new Temperatura();
        doNotify();
    }
}
```

A classe

## WeatherStation

implementa WeatherSubject. É a nossa classe de *Subject*. Ele mantém um conjunto de WeatherObservers que são adicionados via addObserver() e removido através de removeObserver(). Quando o estado WeatherSubject é alterado via setTemperatura(), o método doNotify() é invocado, que entra em contato com todos os WeatherObservers com a temperatura através de seus métodos de doUpdate().

## WeatherCostumer1.java

```
public class WeatherCostumer1 implements WeatherObserver{
    @Override public void doUpdate(int temperatura){
        System.out.println("O WeatherCostumer1 descobriu que a temperatura é: + temperatura);
    }
}
```

## WeatherCustomer1

é um observer que implementa WeatherObserver. O seu método doUpdate() retorna o temperatura atual do WeatherStation e mostra-a.

## WeatherCostumer2.java

```
public class WeatherCostumer2 implements WeatherObserver{
    @Override public void doUpdate(int temperatura){
        System.out.println("O WeatherCostumer2 descobriu que a temperatura é: + temperatura);
    }
}
```

## WeatherCustomer2

é idêntico a WeatherCustomer1.

A classe Demo demonstra o padrão Observer. Ele cria um WeatherStation e, em seguida, um WeatherCustomer1 e um WeatherCustomer2. Os dois clientes são adicionados como observadores para a estação meteorológica. Em seguida, é chamado o método setTemperatura() da estação meteorológica. Isto altera o estado da estação meteorológica e os clientes são notificados sobre esta atualização de temperatura. Em seguida, o objeto WeatherCustomer1 é removido da coleção da estação de observadores. Em seguida, o método setTemperature() é chamado novamente. Isso resulta na notificação do objeto WeatherCustomer2.

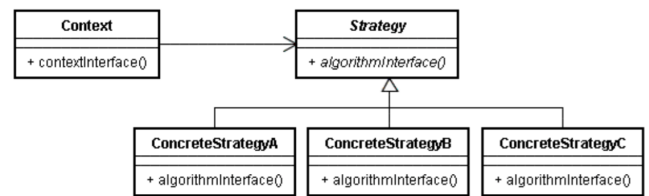
**Demo.java**

```
public class Demo {  
    public static void main(String[] args) {  
        WeatherStation weatherStation = new WeatherStation(33);  
        WeatherCustomer1 wc1 = new WeatherCustomer1();  
        WeatherCustomer2 wc2 = new WeatherCustomer2();  
        weatherStation.addObserver(wc1);  
        weatherStation.addObserver(wc2);  
  
        weatherStation.setTemperature(34);  
        weatherStation.removeObserver(wc1);  
        weatherStation.setTemperature(35);  
    }  
}
```



```
Estação meteorológica define temperatura para: 34  
O WeatherCustomer1 descobriu que a temperatura é:34  
O WeatherCustomer2 descobriu que a temperatura é:34
```

```
Estação meteorológica define temperatura para: 35  
O WeatherCustomer2 descobriu que a temperatura é:35
```



# Strategy

O padrão Strategy é uma maneira no qual a composição que pode ser usada como alternativa à implementação de sub-classes. Ao invés de fornecer diferentes comportamentos através de subclasses, substituindo os métodos na superclasse, o Strategy permite comportamentos diferentes para serem colocados nas classes ConcreteStragy que partilham a mesma interface comum Strategy. A classe Context é uma referência composta para um Strategy.

## Strategy.java

```
public interface Strategy{
    boolean checkTemperatura(int temperaturaF)
```

Interface **Strategy**, declara apenas o método checkTemperatura.

## HikeStrategy.java

```
public class HikeStrategy implements Strategy{
    @Override
    public boolean checkTemperatura(int temperaturaF){
        if((temperaturaF >= 50) && (temperaturaF <= 90)){
            return true;
        } else return false;
    }
}
```

Classe **HikeStrategy** é uma classe concreta de Strategy que implementa a interface Strategy. O método checkTemperatura é implementado para aceitar temperaturas de 50-90°F.

## SkiStrategy.java

```
public class SkiStrategy implements Strategy{
    @Override
    public boolean checkTemperatura(int temperaturaF){
        if((temperaturaF <= 32){
            return true;
        } else return false;
    }
}
```

Classe **SkiStrategy** é uma classe concreta de Strategy que implementa a interface Strategy. Caso temperatura seja 32 ou menos, o método retorna true.

## Context.java

```
public class Context{
    int temperaturaF;
    Strategy strategy;
    public Context(int temperaturaF, Strategy strategy){
        this.temperaturaF = temperaturaF;
        this.strategy = strategy;
    }
    public void setStrategy(Strategy strategy){
        this.strategy = strategy;
    }
    public int getTemperaturaF(){
        return temperaturaF;
    }
    public boolean getResult(){
        return strategy.checkTemperatura(temperaturaF);
    }
}
```

A classe **Context** contém uma temperatura e uma referência para um Strategy. O Strategy pode ser alterado, resultando em um comportamento diferente que opera sobre os mesmos dados no contexto. O resultado disso pode ser obtido do contexto através do método getResult().

**Demo.java**

```
public class Demo{
public static void main(String[] args){
    int temperaturaF = 60;
    Strategy skiStrategy = new SkiStrategy();
    Context context = new Context(temperaturaF, skiStrategy);
    System.out.println("A temperatura" + context.getTemperaturaF() + "F é ideal
        para fazer ski?" + context.getResult());

    Strategy hikeStrategy = new HikeStrategy();
    context.setStrategy(hikeStrategy);
    System.out.println("A temperatura" + context.getTemperaturaF() + "F é ideal
        para fazer hike?" + context.getResult());
}
```



A temperatura 60F é ideal para fazer ski? false  
A temperatura 60F é ideal para fazer hike? true