



Aula – 5

Padrão Observer

Disciplina: COM221 – Computação Orientada a Objetos II

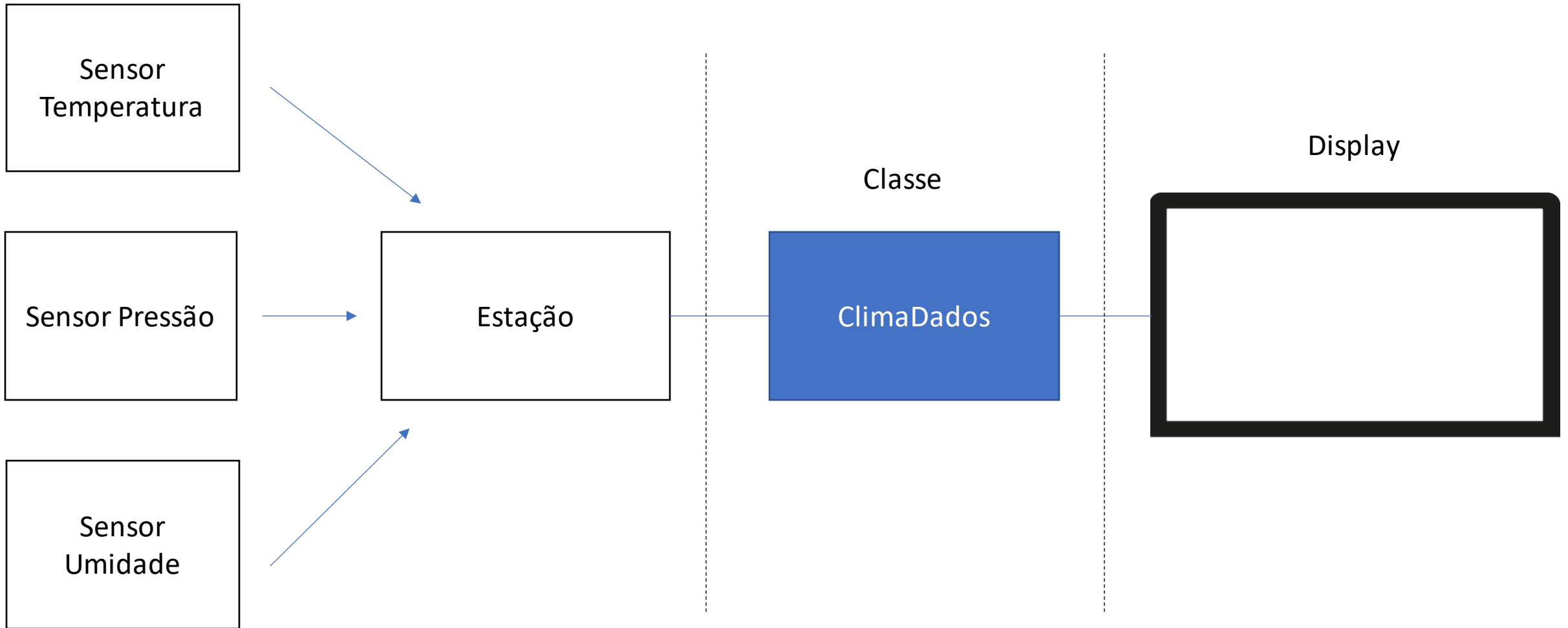
Prof: Phyllipe Lima
phyllipe@unifei.edu.br

Universidade Federal de Itajubá – UNIFEI
IMC – Instituto de Matemática e Computação



Monitoramento do Clima 2.0





Qual a tarefa?

Criar um aplicativo que usa a classe
ClimaDados para atualizar o display com as novas
medidas



Um modelo da classe ClimaDados já foi fornecido

ClimaDados
getTemperatura() getPressao() getUmidade() novasMedidas(); //outros métodos

```
//Esse método é chamado sempre quando  
//os valores medidos se modificam  
public void novasMedidas() {  
    //Coloque o código para atualizar o display aqui!  
}
```

```
public class ClimaDados {  
  
    //Membros da classe  
  
    //Esse método é chamado sempre que os valores  
    //medidos se modificam  
    public void novasMedidas() {  
  
        //Lembrando que os getters já estão implementados  
        double temperatura = getTemperatura();  
        double pressao = getPressao();  
        double umidade = getUmidade();  
  
        condicoesAtuaisDisplay.atualiza(temperatura,pressao,umidade);  
        previsaoDisplay.atualiza(temperatura,pressao,umidade);  
        estatisticaDisplay.atualiza(temperatura,pressao,umidade);  
    }  
  
    //outros métodos omitidos
```

```
public class ClimaDados {
```

```
    //Membros da classe
```

```
    //Esse método é chamado sempre que os valores  
    //medidos se modificam
```

```
    public void novasMedidas() {
```

```
        //Lembrando que os getters já estão implementados
```

```
        double temperatura = getTemperatura();
```

```
        double pressao = getPressao();
```

```
        double umidade = getUmidade();
```

```
        condicoesAtuaisDisplay.atualiza(temperatura,pressao,umidade);
```

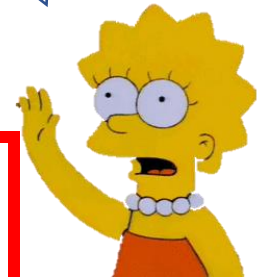
```
        previsaoDisplay.atualiza(temperatura,pressao,umidade);
```

```
        estatisticaDisplay.atualiza(temperatura,pressao,umidade);
```

```
    }
```

```
    //outros métodos omitidos
```

Esse bloco parece ser
muito suscetível a
mudança!
Deveríamos encapsular
esse trecho




```
public class ClimaDados {
```

```
    //Membros da classe
```

```
    //Esse método é chamado sempre que os valores  
    //medidos se modificam
```

```
    public void novasMedidas() {
```

```
        //Lembrando que os getters já estão implementados
```

```
        double temperatura = getTemperatura();
```

```
        double pressao = getPressao();
```

```
        double umidade = getUmidade();
```

```
        condicoesAtuaisDisplay.atualiza(temperatura,pressao,umidade);
```

```
        previsaoDisplay.atualiza(temperatura,pressao,umidade);
```

```
        estatisticaDisplay.atualiza(temperatura,pressao,umidade);
```

```
    }
```

```
    //outros métodos omitidos
```

Usando instâncias
concretas não
conseguimos adicionar
ou remover displays sem
modificar o programa



```
public class ClimaDados {
```

```
//Membros da classe
```

```
//Esse método é chamado sempre que os valores  
//medidos se modificam
```

```
public void novasMedidas() {
```

```
//Lembrando que os getters já estão implementados
```

```
double temperatura = getTemperatura();
```

```
double pressao = getPressao();
```

```
double umidade = getUmidade();
```

```
condicoesAtuaisDisplay.atualiza(temperatura,pressao,umidade);
```

```
previsaoDisplay.atualiza(temperatura,pressao,umidade);
```

```
estatisticaDisplay.atualiza(temperatura,pressao,umidade);
```

```
}
```

```
//outros métodos omitidos
```

Temos uma interface
comum para o display!
Todos possuem o
método "atualiza". Isso é
bom 😊



Vamos conhecer mais um ***design pattern*** para nos auxiliar nessa situação!





Considere a assinatura de uma revista!

- A editora começa a produzir revistas!
- Pessoas podem **assinar** revistas
- Cada vez que uma nova revista é **publicada, todos os assinantes recebem** uma cópia.
- Enquanto for um assinante, irá receber.
- Caso não deseje receber, pode **cancelar a assinatura**. Assim deixará de **receber revistas** quando o editor lançar uma nova versão.



EGM^[ii]
THE DIGITAL MAGAZINE



Subject!/Observável



Notifica



Observador
Capiroto!

Notifica

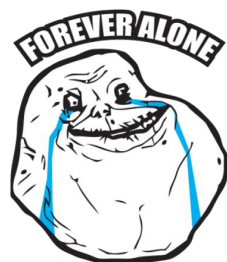


Observadora!
Capirota

Notifica

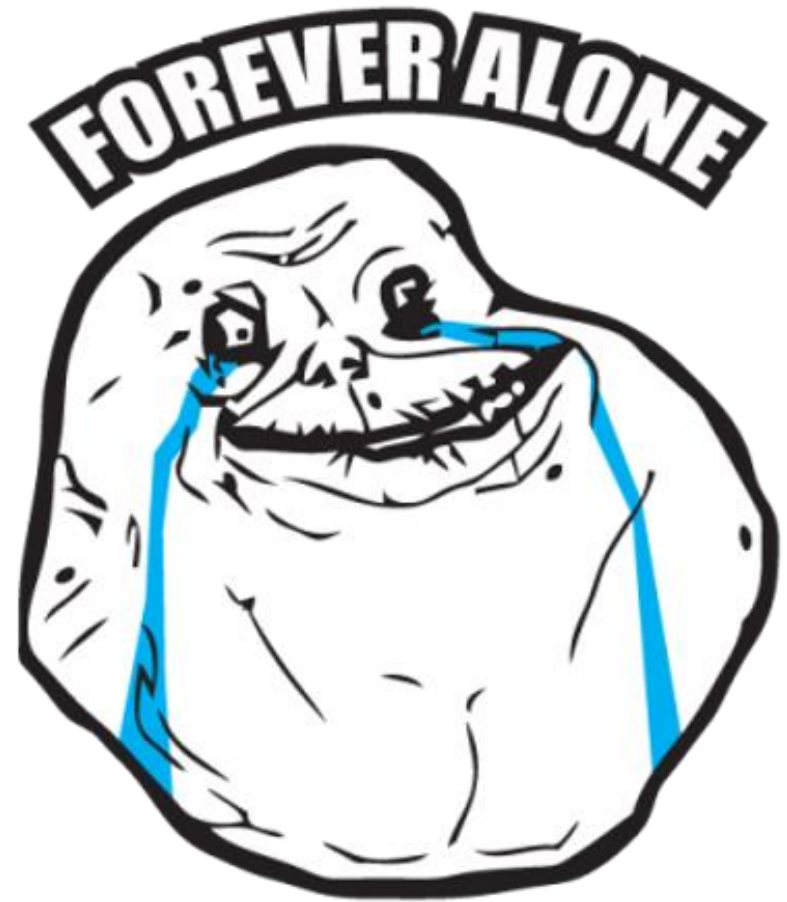


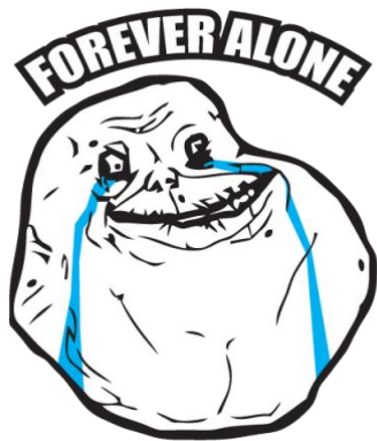
Observadora!
Tinhosa



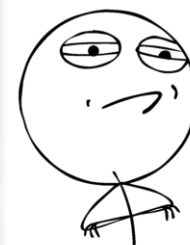
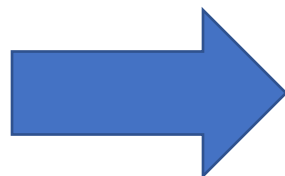
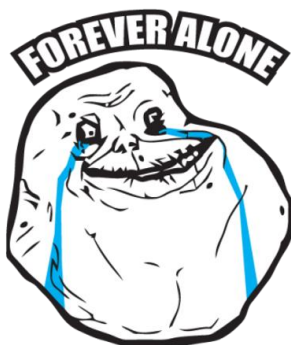
Não é observador!
Forever Alone

Forever Alone também quer ser notificado! Isto é, quer se tornar um ***observador***





Assinar/Registrar



Observador!
Tinhoso!



Notifica



Observador
Capiroto!

Notifica



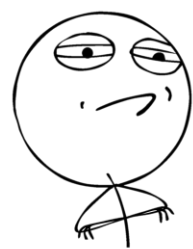
Observadora!
Capirota

Notifica

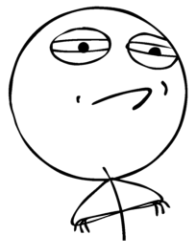


Observadora!
Tinhosa

Notifica

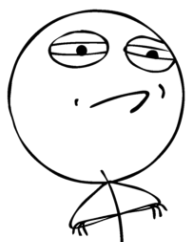


Observador!
Tinhoso

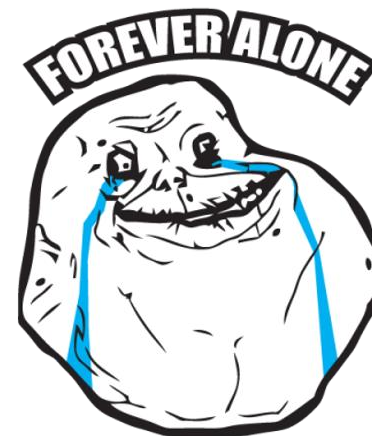
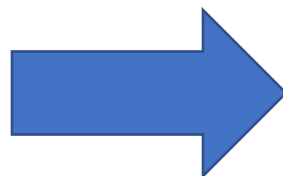


Observador
Capiroto!

Cancelar a Assinatura



Observador
Capiroto!



Não será mais
notificado

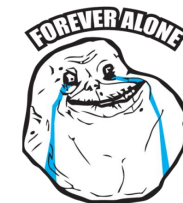
Subject!



Notifica

Notifica

Notifica



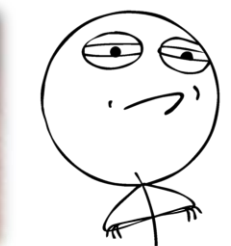
Capiroto não
receberá mais
notificações!



Observadora!
Capirota



Observadora!
Tinhosa



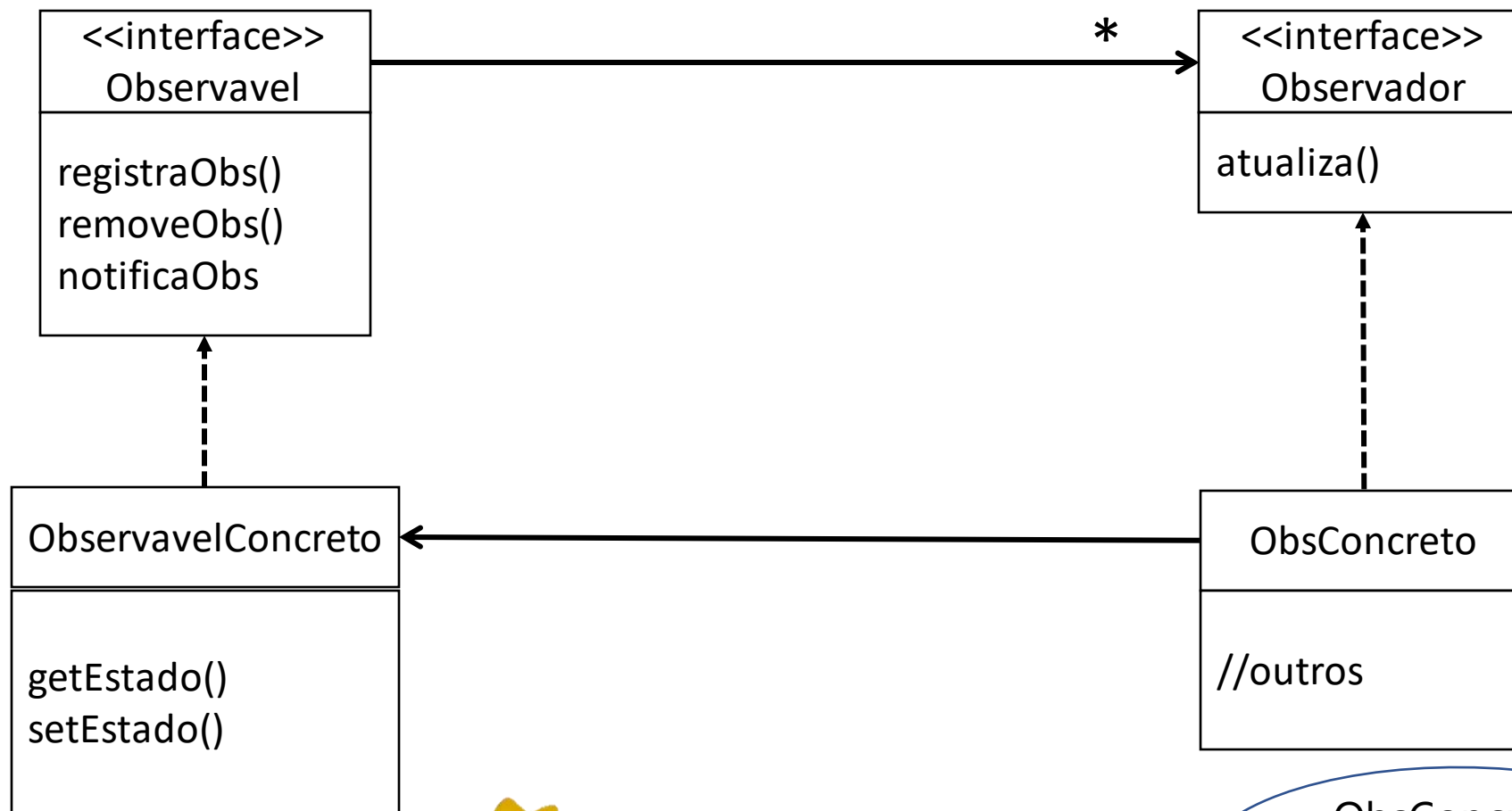
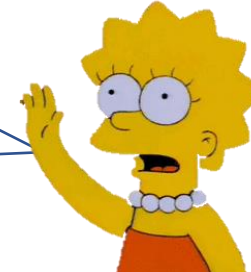
Observador!
Tinhoso

Observer:

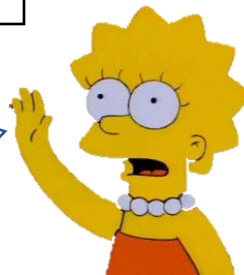
Define uma dependência “um para muitos”. Quando um objeto mudar de estado (subject/observável), todos os seus dependentes são notificados automaticamente!



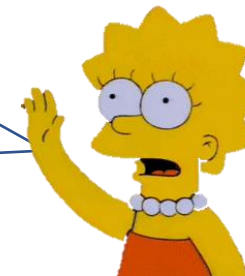
Observável pode
conter vários
observadores



Pode ter getter
e setters para
seu estado



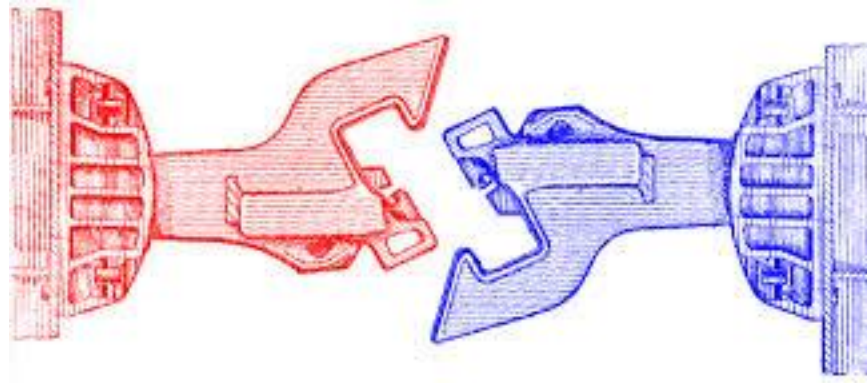
ObsConcreto
registra em um
SubjectConcreto






Princípio de Design:

Busque acoplamento
fraco(aceitável) entre objetos!

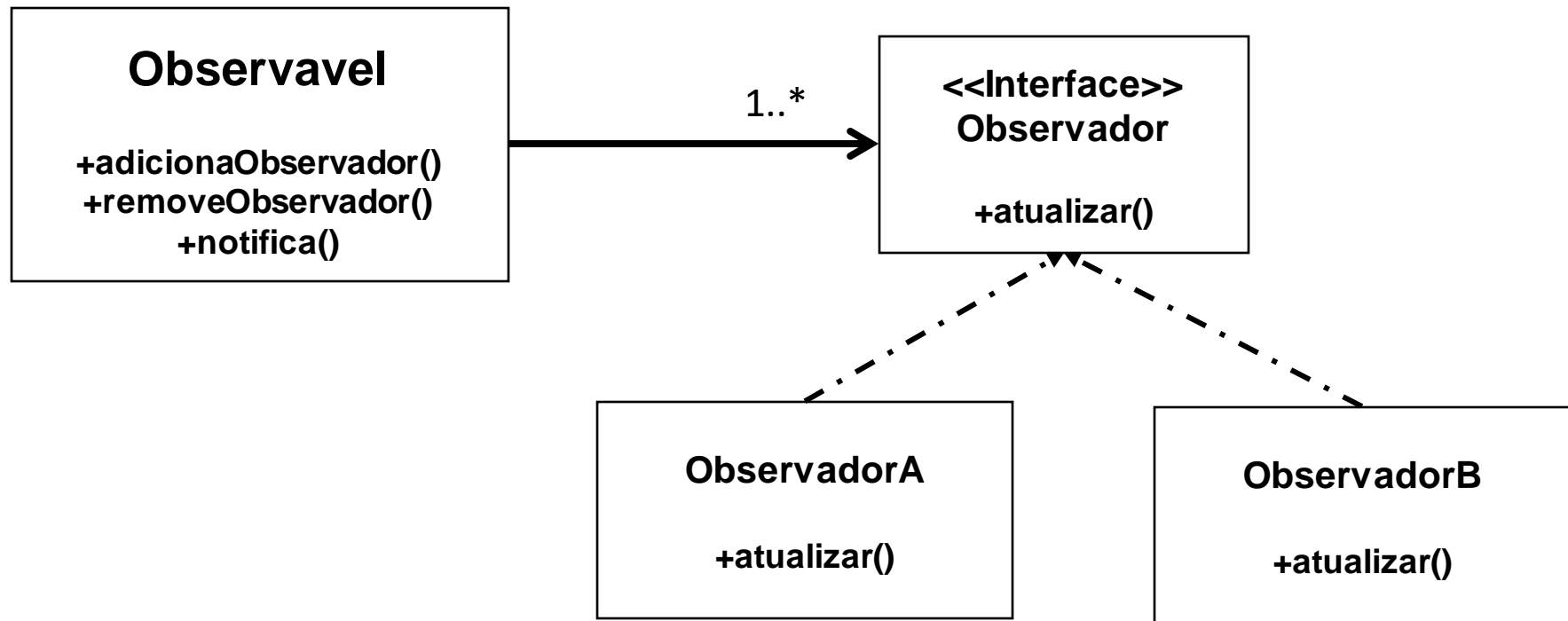
Observável



Observador

-
-  O Observável conhece apenas a **interface** do observador
 -  Novos **observadores** podem ser adicionados a qualquer momento
 -  O Observável não precisa ser modificado para armazenar novos **observadores**

Outros Exemplos!



```
public class Jogador {  
  
    private boolean estouVivo = true;  
  
    //Metodo chamado pelo Framework/Game Engine  
    public void Atualiza() {  
  
        if(!estouVivo) {  
            //Alem de executar a logica do jogador morto  
            jogadorMorreu();  
            //Avisar UI que o jogador morreu  
            //Avisar o servidor que o jogador morreu  
            //Avisar o gerenciador que o jogador morreu  
            //.....  
        }  
  
    }  
}
```

```
public interface ObservaJogador {  
  
    public void notifica();  
  
}
```

```
public class UI implements ObservaJogador {  
  
    @Override  
    public void notifica() {  
        //Faz alguma coisa quando o jogador morrer  
    }  
  
}
```

```
public class Jogador {
```

```
    private boolean estouVivo = true;
```

```
    private List<ObservaJogador> observadores;
```

```
    public void addObserver(ObservaJogador observador) {  
        observadores.add(observador);  
    }
```

```
    public void removeObservador(ObservaJogador observador) {  
        observadores.remove(observador);  
    }
```

```
    //Metodo chamado pelo Framework/Game Engine
```

```
    public void Atualiza() {
```

```
        if(!estouVivo) {  
            jogadorMorreu();
```

```
            for (ObservaJogador observaJogador : observadores) {  
                observaJogador.notifica();//Notificando a turma!  
            }
```

```
        }
```

```

public class Jogador {

    private boolean estouVivo = true;
    private List<ObservaJogador> observadores;

    public void addObserver(ObservaJogador observador) {
        observadores.add(observador);
    }

    public void removeObservador(ObservaJogador observador) {
        observadores.remove(observador);
    }

    //Metodo chamado pelo Framework/Game Engine
    public void Atualiza() {

        if(!estouVivo) {
            jogadorMorreu();
            for (ObservaJogador observaJogador : observadores) {
                observaJogador.notifica();//Notificando a turma!
            }
        }
    }
}

```



```

public class Jogador {

    private boolean estouVivo = true;

    //Metodo chamado pelo Framework/Game Engine
    public void Atualiza() {

        if(!estouVivo) {
            //Alem de executar a logica do jogador morto
            jogadorMorreu();
            //Avisar UI que o jogador morreu
            //Avisar o servidor que o jogador morreu
            //Avisar o gerenciador que o jogador morreu
            //....
        }
    }
}

```



Monitoramento do Clima 2.0



Exercício Proposto 1 – Contando Palavras

Crie um aplicativo que recebe uma frase e quebre em palavras (String). A aplicação deve contar as palavras de acordo com os seguintes critérios!

- Contar todas as palavras
- Contar palavras com quantidades pares de caracteres
- Contar palavras começadas com maiúsculas
- Utilize o padrão observer



Referência

¹ This was once revealed to me in a dream.

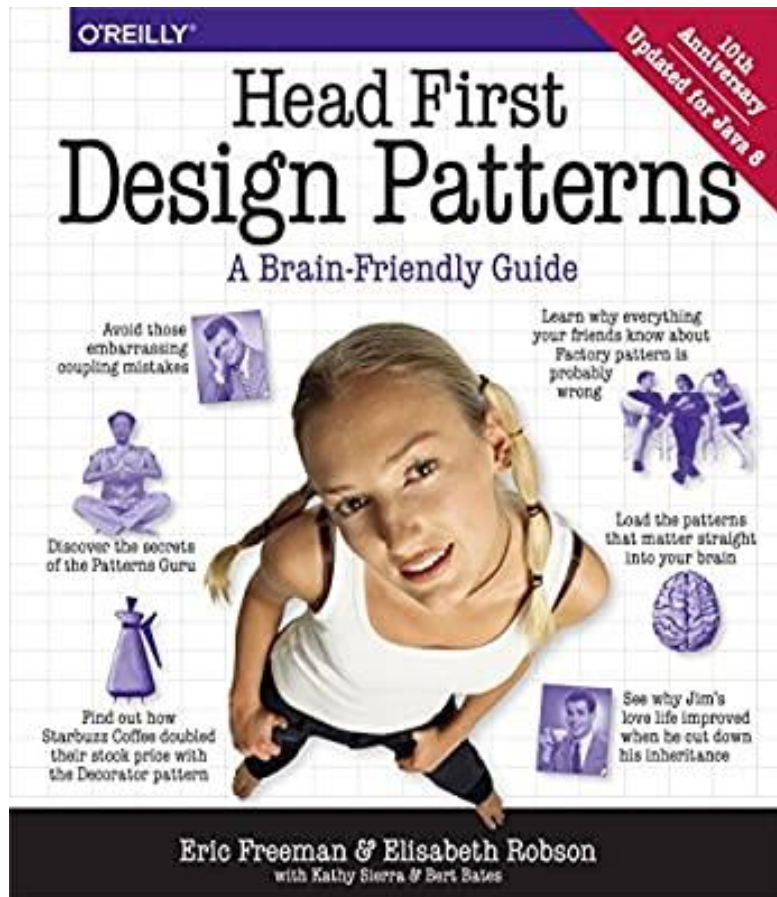


- Capítulo 6 do livro Engenharia de Software Moderna
 - Padrões de Projeto
 - <https://engsoftmoderna.info/cap6.html>

Referência - Complementar

¹ This was once revealed to me in a dream.

- Head First Design Patterns
- Cap 2



Referência - Complementar

¹ This was once revealed to me in a dream.

Design Patterns com Java

Projeto orientado a objetos guiado por padrões



 Casa do
Código

EDUARDO GUERRA

- Design Patterns com Java
- Cap 3

Implementações

- ❑ <https://github.com/phillima-unifei/COM221>





Aula – 5

Padrão Observer

Disciplina: COM221 – Computação Orientada a Objetos II

Prof: Phyllipe Lima
phyllipe@unifei.edu.br

Universidade Federal de Itajubá – UNIFEI
IMC – Instituto de Matemática e Computação