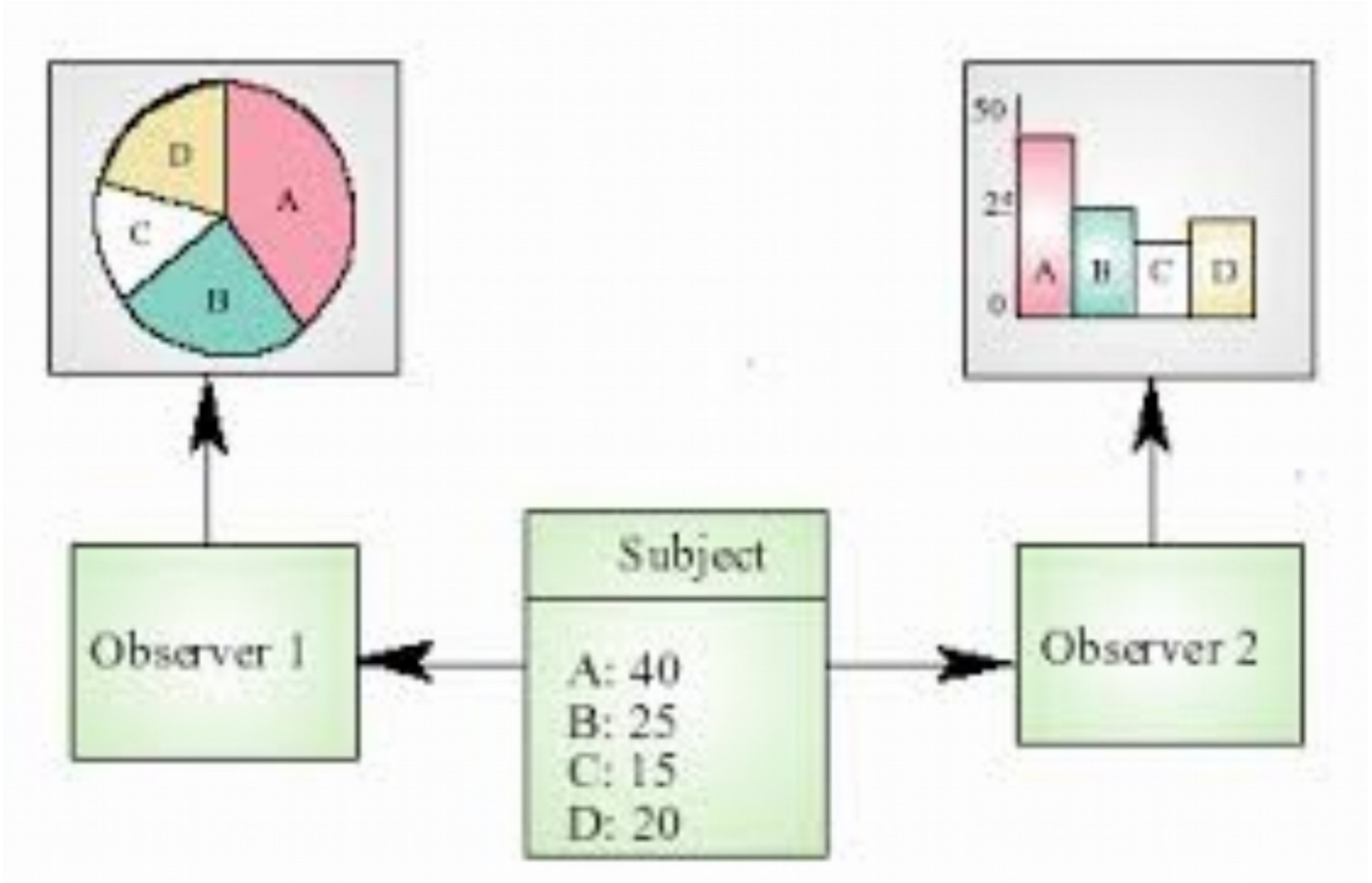
Observer

Instituto Federal do Espirito Santo – Campus Colatina
Disciplina: Projeto de Sistemas
Professor: Giovany Frossard Teixeira

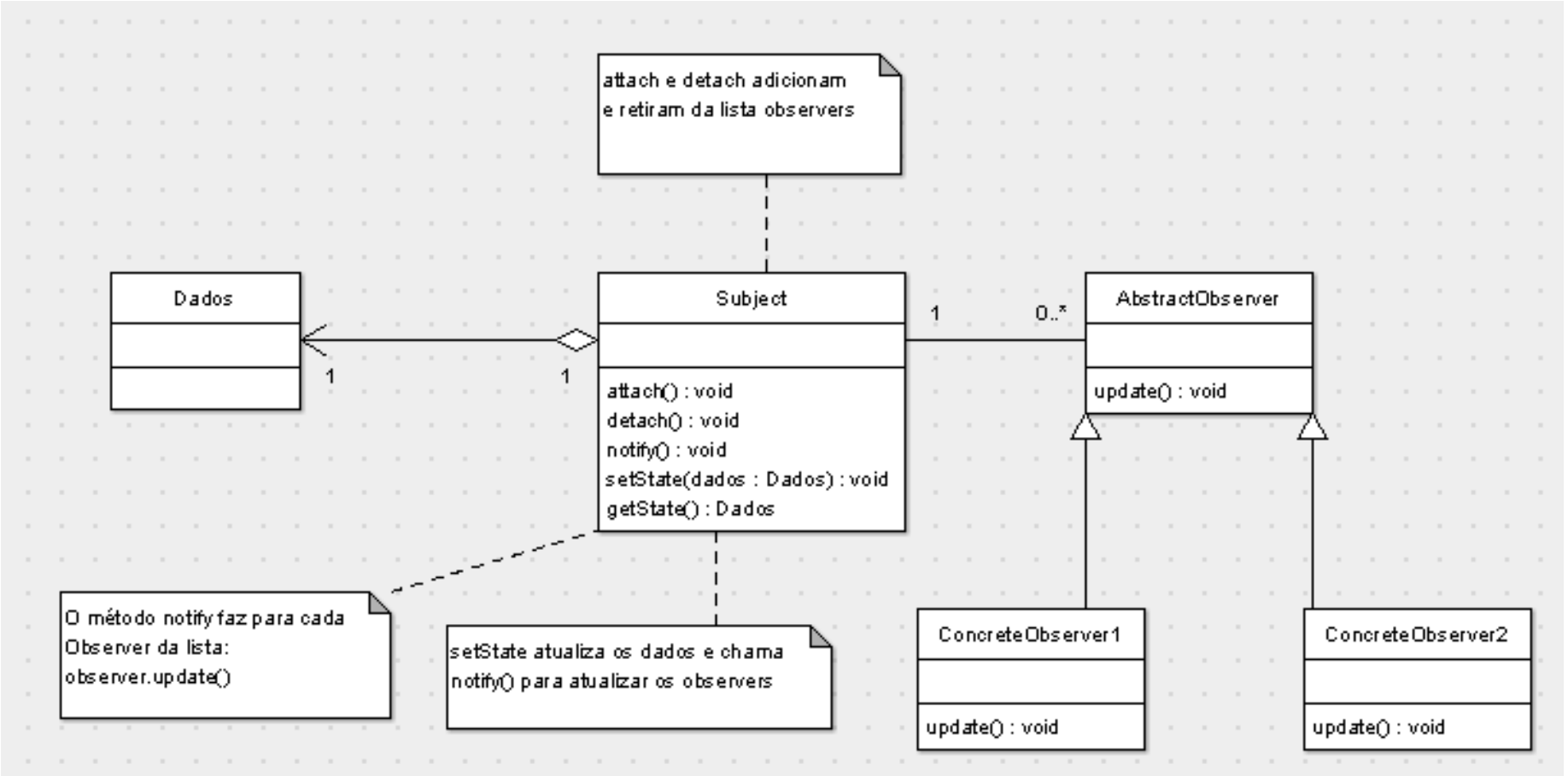
Elementos Essenciais

- **Nome:** Observer (Dependents, Publish-Subscribe)
- **Problema:** Manter objetos visuais (interface gráfica) atualizados quando da mudança de um outro objeto (não visual) relacionado a eles.
- **Solução:** Definir uma dependência um para muitos entre objetos, de maneira que quando um objeto muda de estado todos os seus dependentes são notificados e atualizados automaticamente.
- **Consequências:** Simplificação do processo de atualização de objetos visuais. Possibilidade de alto custo nessa atualização (atualizar várias interfaces gráficas ao mesmo tempo pode ser custoso).

Representação visual



Padrão Observer – Visão Geral



Problema

- Suponha que em um programa é necessário fazer várias representações de um mesmo conjunto de dados. Este conjunto de dados consiste de uma estrutura que contém 3 atributos: valorA, valorB e valorC, como mostra o código a seguir:

```
public class Dados {
    int valorA, valorB, valorC;

    public Dados(int a, int b, int c) {
        valorA = a;
        valorB = b;
        valorC = c;
    }
}
```

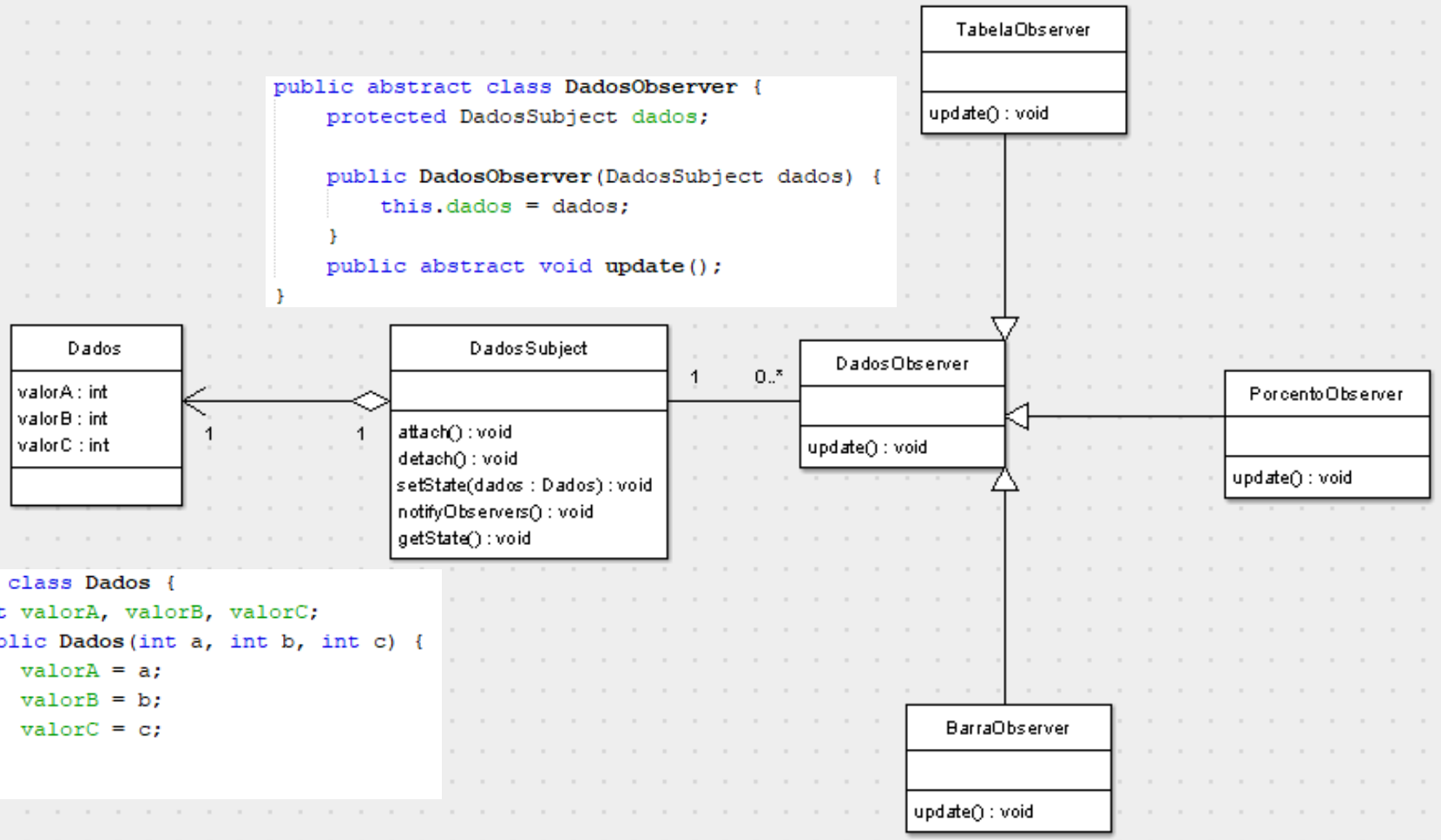
Fonte: https://brizeno.wordpress.com/category/padroes-de-projeto/observer/ - acessado em 17/02/2020

Diagrama do problema

```java
public class TabelaObserver extends DadosObserver {
    public TabelaObserver(DadosSubject dados) {
        super(dados);
    }
    @Override
    public void update() {
        System.out.println("Tabela:\nValor A: " + dados.getState().valorA +
                           "\nValor B: " + dados.getState().valorB +
                           "\nValor C: " + dados.getState().valorC + "\n\n");
    }
}
```

```java
public abstract class DadosObserver {
    protected DadosSubject dados;

    public DadosObserver(DadosSubject dados) {
        this.dados = dados;
    }
    public abstract void update();
}
```




```java
public class Dados {
    int valorA, valorB, valorC;
    public Dados(int a, int b, int c) {
        valorA = a;
        valorB = b;
        valorC = c;
    }
}
```

DadosSubject

```
public class DadosSubject {
    protected ArrayList<DadosObserver> observers;
    protected Dados dados;
    public DadosSubject() {
        observers = new ArrayList<DadosObserver>();
    }
    public void attach(DadosObserver observer) {
        observers.add(observer);
    }
    public void detach(int indice) {
        observers.remove(indice);
    }
    public void setState(Dados dados) {
        this.dados = dados;
        notifyObservers();
    }
    private void notifyObservers() {
        for (DadosObserver observer : observers) {
            observer.update();
        }
    }
    public Dados getState() {
        return dados;
    }
}
```

PorcentoObserver e BarraObserver

```java
public class PorcentoObserver extends DadosObserver {
    public PorcentoObserver(DadosSubject dados) {
        super(dados);
    }
    @Override
    public void update() {
        int somaDosValores = dados.getState().valorA + dados.getState().valorB + dados.getState().valorC;
        DecimalFormat formatador = new DecimalFormat("#.##");
        String porcentagemA = formatador.format((double) dados.getState().valorA / somaDosValores);
        String porcentagemB = formatador.format((double) dados.getState().valorB / somaDosValores);
        String porcentagemC = formatador.format((double) dados.getState().valorC / somaDosValores);
        System.out.println("Porcentagem:\nValor A: " + porcentagemA + "%\nValor B: " + porcentagemB + "%\nValor C: " + porcentagemC + "%" + "\n\n");
    }
}
```

```java
public class BarraObserver extends DadosObserver {
    public BarraObserver(DadosSubject dados) {
        super(dados);
    }
    @Override
    public void update() {
        String barraA = "", barraB = "", barraC = "";
        for (int i = 0; i < dados.getState().valorA; i++) {
            barraA += '=';
        }
        for (int i = 0; i < dados.getState().valorB; i++) {
            barraB += '=';
        }
        for (int i = 0; i < dados.getState().valorC; i++) {
            barraC += '=';
        }
        System.out.println("Barras:\nValor A: " + barraA + "\nValor B: " + barraB + "\nValor C: " + barraC + "\n\n");
    }
}
```

Programa principal

```
public class Observer {
    public static void main(String[] args) {
        DadosSubject dados = new DadosSubject();

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

        dados.setState(new Dados(7, 3, 1));
        dados.setState(new Dados(2, 3, 1));
    }
}
```

```
run:
Tabela:
Valor A: 7
Valor B: 3
Valor C: 1

Barras:
Valor A: =======
Valor B: ===
Valor C: =

Porcentagem:
Valor A: 0,64%
Valor B: 0,27%
Valor C: 0,09%

Tabela:
Valor A: 2
Valor B: 3
Valor C: 1

Barras:
Valor A: ==
Valor B: ===
Valor C: =

Porcentagem:
Valor A: 0,33%
Valor B: 0,5%
Valor C: 0,17%
```

Considerações finais

- Colocar os métodos de getState e setState na mesma classe que os demais métodos (como foi feito no exemplo) pode dificultar a manutenibilidade. Nesse contexto a criação de um Subject como uma classe abstrata (com métodos attach(), detach() e notify()) com classes concretas herdando dessa classe e implementando getState e setState pode ser uma abordagem mais interessante pensando em evoluções/modificações futuras.
- A atualização de todas as interfaces gráficas ao mesmo tempo pode ser uma abordagem custosa. O uso de uma estrutura intermediária para gerenciar subjects e observers pode ser bastante interessante.

Dúvidas?