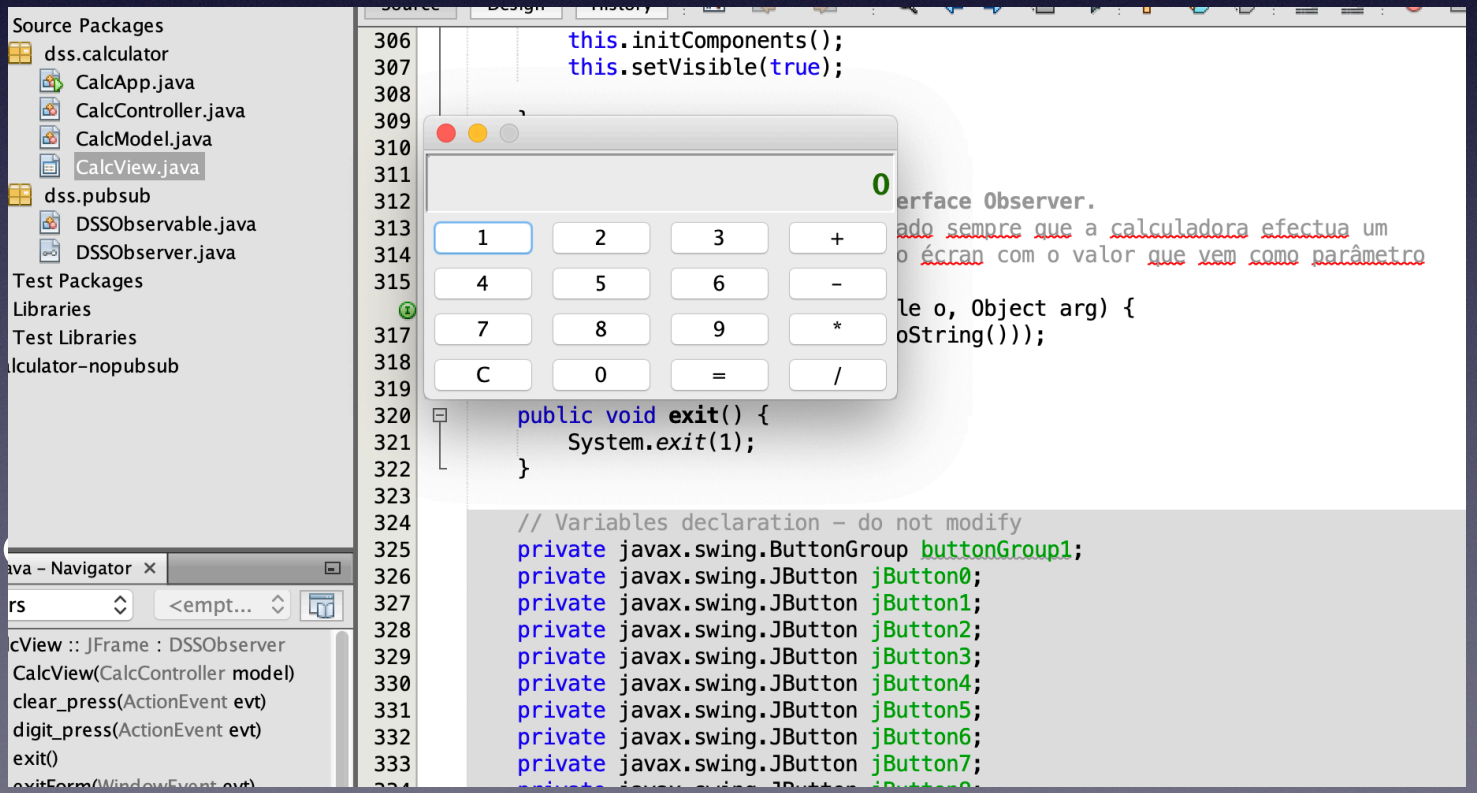
Um exemplo com MVC

- Criação de uma aplicação que é uma calculadora.



- A `View` tem a interface gráfica, onde se desenham os botões e a área onde aparecem os resultados
  - podia ser perfeitamente ser um menu em modo texto
  - até podemos ter mais do que uma `View`!!
- O `Model` é uma classe muito simples, que faz operações matemáticas.

- O `Model` é completamente independente da `View` e do `Controller`
  - recebe invocações de métodos e executa-os

```
public class CalcModel {
    private double value;

    public CalcModel() {
        this.value = 0;
    }

    public void add(double v) {
        this.value += v;
    }

    public void subtract(double v) {
        this.value -= v;

    }

    public void multiply(double v) {
        this.value *= v;

    }

    public void divide(double v) {
        this.value /= v;

    }

    public double getValue() {
        return this.value;
    }

    public void setValue(double v) {
        this.value = v;
    }

    public void reset() {
        this.value = 0;

    }
}
```

(*) retirado de um exemplo da Uc de Desenvolvimento de Sistemas de Software

- O `Controller` **conhece o** `Model` **e faz a gestão dos pedidos recebidos via** `View`

```
public class CalcController extends DSSObservable implements DSSObserver {

    private double screen_value;          // o valor que está a ser lido
    private char lastkey;                 // indica que se vai começar a "ler" um novo número
    private char opr;                     // memória com a operação a aplicar
    private CalcModel model;

    /** Creates a new instance of Calculadora */
    public CalcController(CalcModel model) {...8 lines }

    public void processa(int d) {...10 lines }

    public void processa(char opr) {
        switch (this.opr) {
            case '=': model.setValue(this.screen_value);
                      break;
            case '+': model.add(this.screen_value);
                      break;
            case '-': model.subtract(this.screen_value);
                      break;
            case '*': model.multiply(this.screen_value);
                      break;
            case '/': model.divide(this.screen_value); // Exercício: Acrescente tratamento da divisão por zero!
                      break;
        };
        this.opr = opr;
        this.lastkey = opr;
    }

    public void clear() {
        model.reset();
        this.lastkey = ' ';
    }
```

tem uma variável de instância do tipo do Model

(*) retirado de um exemplo da Uc de Desenvolvimento de Sistemas de Software

- A aplicação principal deve criar a `View`, o `Controller` e o `Model`
  - e colocar a `View` em execução

```
public void run() {
    CalcModel model = new CalcModel();
    CalcController controller = new CalcController(model);
    CalcView view = new CalcView(controller);

    view.run();
```

(*) retirado de um exemplo da Uc de Desenvolvimento de Sistemas de Software

Múltiplas Views

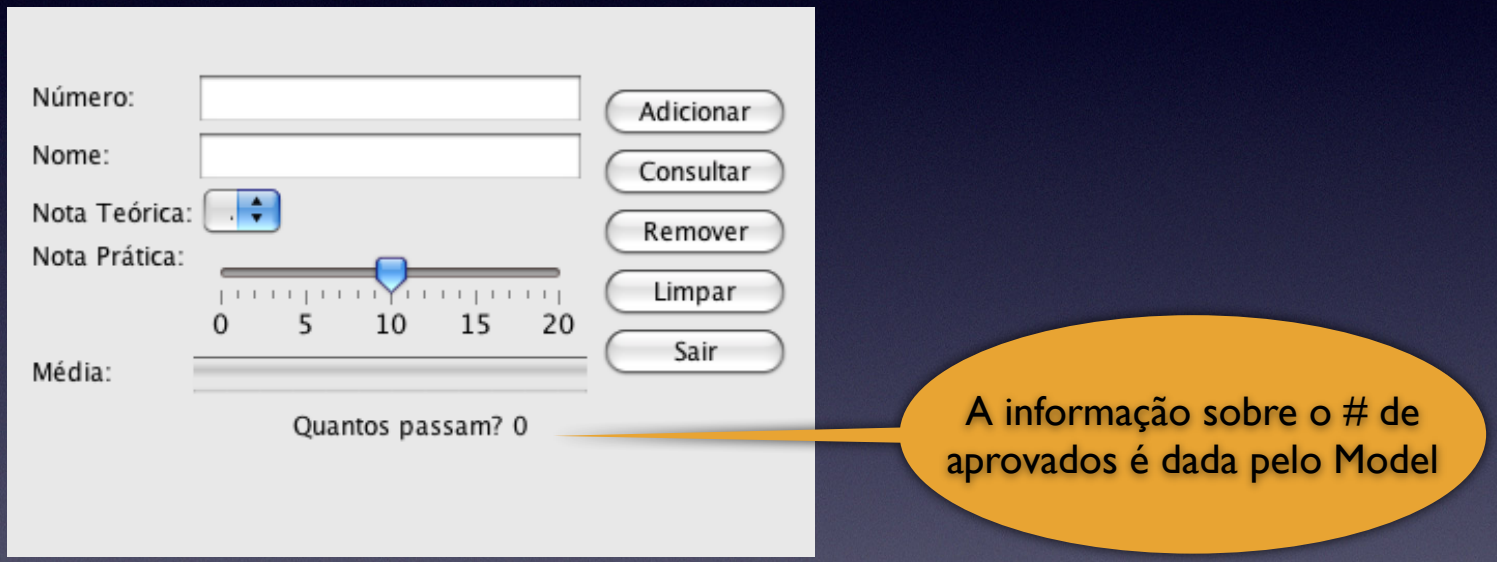
- Uma vantagem de desacoplar o modelo da vista é, além de manter separação do código, permitir ter:
  - várias vistas sobre o modelo
  - várias aplicações cliente sobre a mesma base de funcionalidade
  - se só muda a componente da interacção com o utilizador, o modelo é o mesmo

- No caso da aplicação bancária vista anteriormente, podemos ter o mesmo `Model` e criar:
  - um programa para os clientes
  - um programa para os empregados do banco
  - um programa para a gestão do banco

- O que é necessário criar:
  - view(s) para cada um dos programas
  - controller(s) para cada um dos programas
  - fica facilitada a alteração de programas independentes (principalmente alteração da `View`)

- Coloca-se agora a questão de como fazer reflectir alterações no modelo nas diferentes views
  - pode ser evitado que o `Model` conheça e manipule a `View`
  - não faz sentido a `View` estar sempre a perguntar ao `Model`
  - terá de ser o `Model` a sinalizar que existem alterações e esperar que a `View` queira consultar a informação

- Por exemplo numa aplicação para gestão das notas de uma turma de alunos:



- o `Model` é que possui as regras que determinam em que circustância é que um aluno é aprovado

- Neste caso não faz sentido ser a `View` a tomar a iniciativa de perguntar ao `Model`
  - ... e o `Model` pode ter mais do que uma `View` e não sabe qual delas é que precisa de ser actualizada
  - é melhor ser a `View` a responsável pela actualização (no caso de achar que o deve fazer)

- Mas como é que se pode operacionalizar esta actualização:
  - possibilitando que existam classes que observam o estado de outras
  - criando um mecanismo de notificação quando o estado é alterado
- Recorrendo a um padrão arquitectural designado por Observer

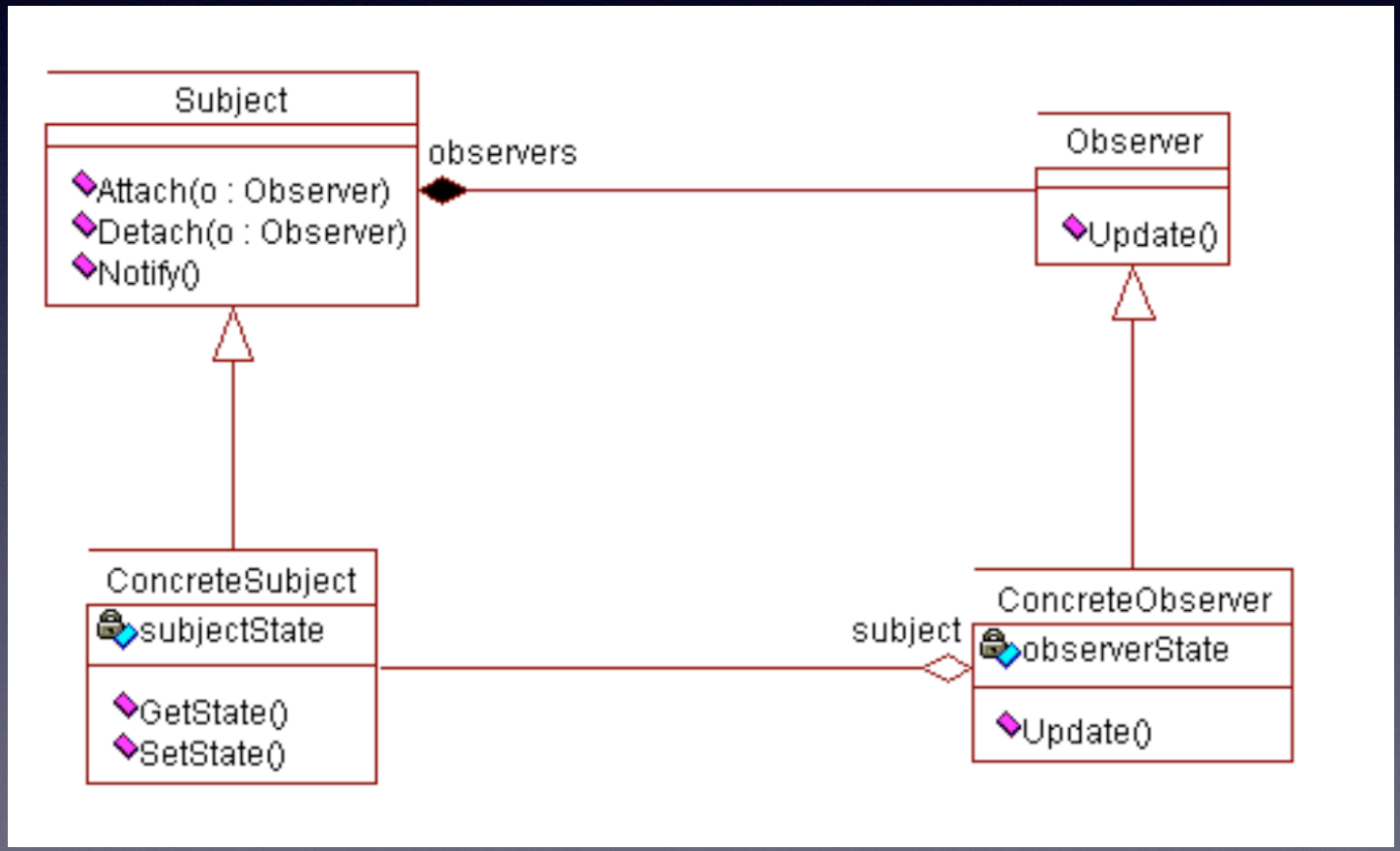
O padrão Observer

- O objectivo deste padrão arquitectural é estruturar a definição de dependências do tipo um para muitos, de modo a que quando um objecto mudar os que dele dependem também mudem.
  - os observadores são notificados da alteração do observado

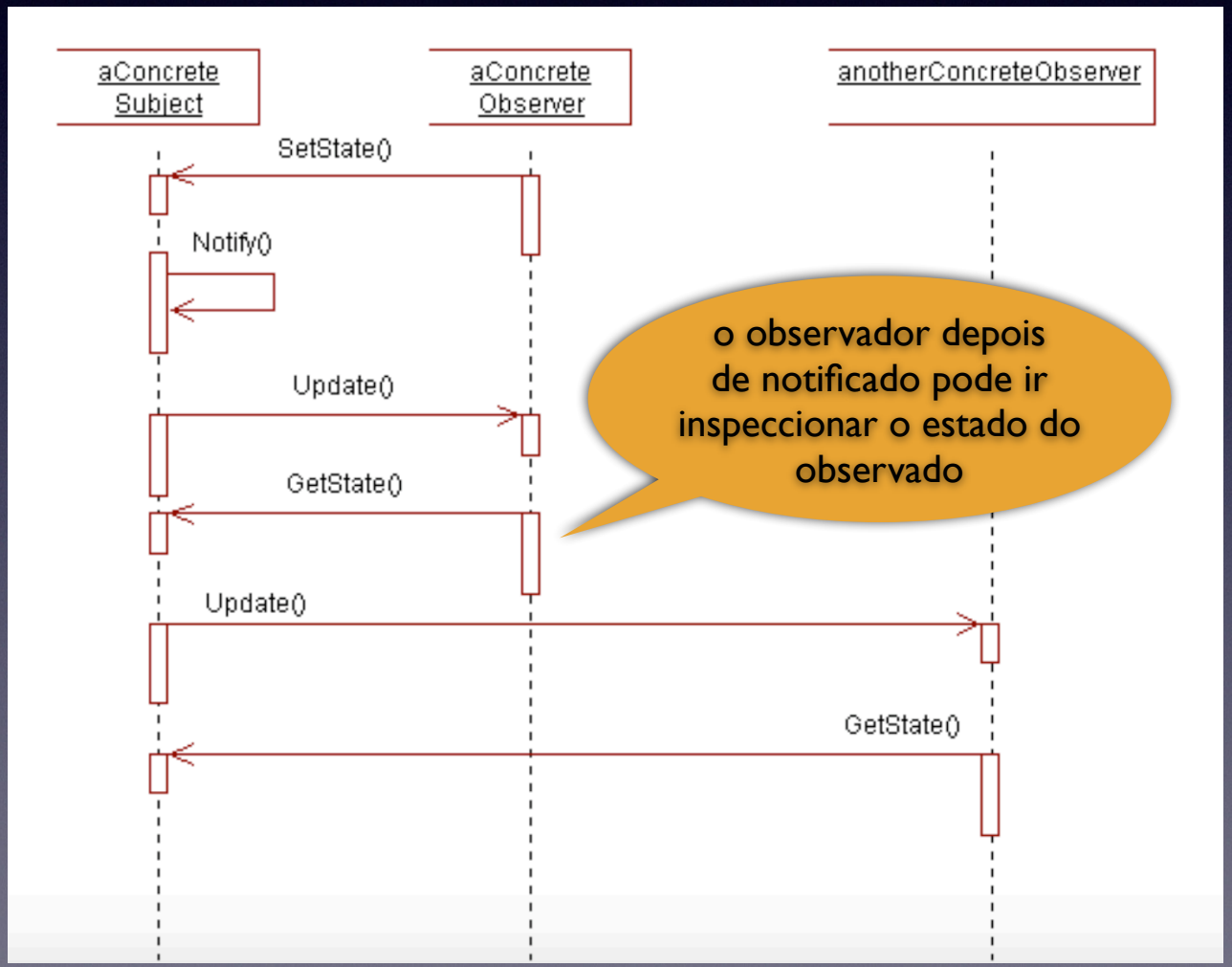
- Esta solução arquitectural pode ser utilizada:
  - quando existe uma dependência um para muitos entre objectos (por exemplo as `View(s)` "associadas" a um `Model`)
  - quando a mudança num objecto implica mudar o estado noutros, mas não se sabe quais (e quantos) objectos mudam
  - quando se precisa que um objecto notifique outros sem saber quem são e como se comportam

- A utilização de observadores (observers) é uma técnica muito utilizada quando se quer implementar programação orientada aos eventos:
  - uma acção na interface (carregar num botão, escolher uma opção) origina uma notificação a quem esteja interessado no evento
  - tem de ser programada esta capacidade de escutar os eventos

- Em termos arquitecturais este padrão tem a seguinte estrutura:



- Em termos de funcionamento estabelecerá interacções do tipo:



- No exemplo do livro "Java Program Design" se quisermos adicionar programas para o marketing e para um auditor podemos fazer:

```
public class Bank {
  private Map<Integer, BankAccount> accounts;
  private int nextacct;
  private MarketingRep rep;
  private Auditor aud;

  public Bank(Map<Integer,BankAccount> accounts, int n,
              MarketingRep r, Auditor a) {
      this.accounts = accounts;
      this.nextacct = n;
      this.rep = r;
      this.aud = a;
  }
```

(*) retirado de Java Program Design, E. Sciore, 2019

- E quando se adiciona uma nova conta ao banco (alterando-se assim o `Model`) os objectos são notificados:

```
public int newAccount(int type, boolean isforeign) {
    int acctnum = this.nextacct++;
    BankAccount ba = AccountFactory.createAccount(type, acctnum);
    ba.setForeign(isforeign);
    rep.update(acctnum, isforeign);
    aud.update(acctnum, isforeign);
    return acctnum;
}
```

rep e aud são objectos observadores. São informados que a flag isforeign tem determinado valor!

(*) retirado de Java Program Design, E. Sciore, 2019

- Como implementar esta lógica de observador/observado?
  - fazer os observadores terem obrigatoriamente um método de `update`, que será invocado pelos observados
  - ter nos observados uma lista com os objectos observadores

- Os observadores como garantidamente devem ter o método `update` (independentemente do que são), devem implementar uma interface que defina esse comportamento

```java
/*
 * DISCLAIMER: Este código foi criado para discussão e edição durante as aulas
 * práticas de DSS, representando uma solução em construção. Como tal, não deverá
 * ser visto como uma solução canónica, ou mesmo acabada. É disponibilizado para
 * auxiliar o processo de estudo. Os alunos são encorajados a testar adequadamente
 * o código fornecido e a procurar soluções alternativas, à medida que forem
 * adquirindo mais conhecimentos.
 */
package dss.pubsub;

/**
 *
 * @author jfc
 */
public interface DSSObserver {
    public void update(DSSObservable source, Object value);
}
```

(*) retirado de um exemplo da Uc de Desenvolvimento de Sistemas de Software

- Os observados devem garantir que sabem quem são os observadores (guardam a referência dos objectos)

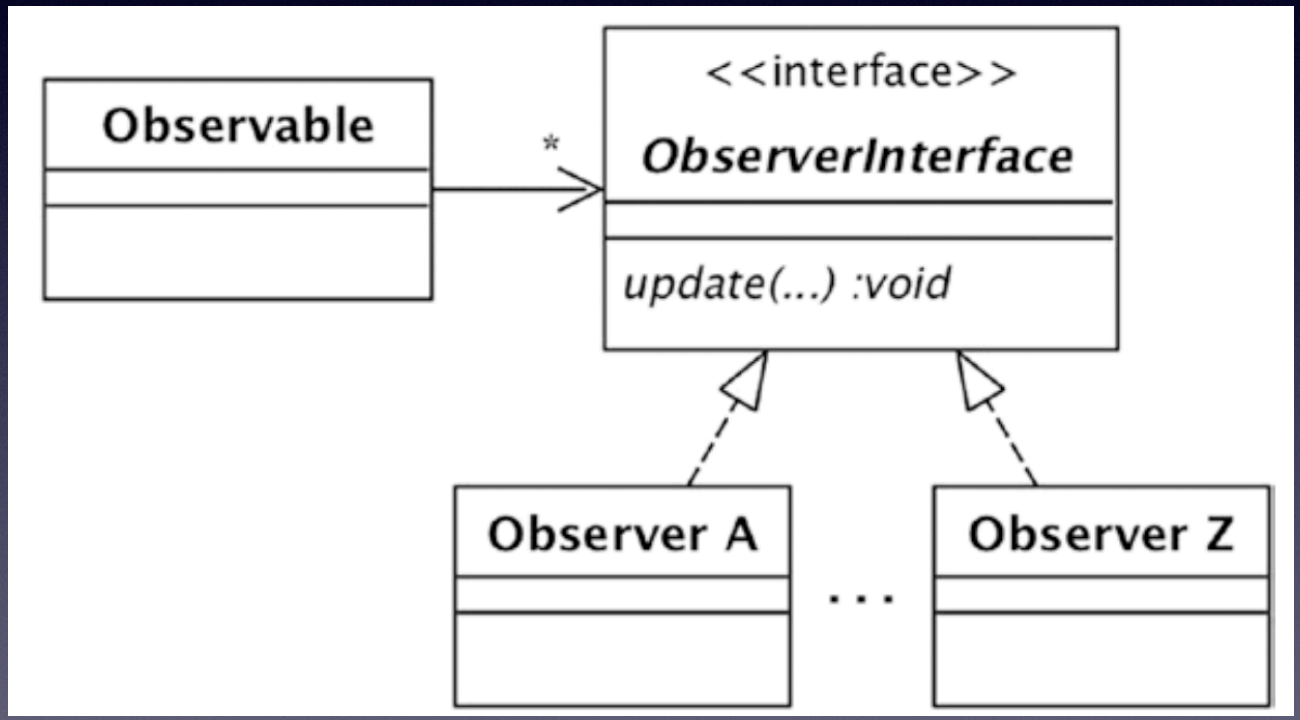
```java
package dss.pubsub;

import java.util.ArrayList;
import java.util.List;

/**
 *
 * @author jfc
 */
public class DSSObservable {

    private List<DSSObserver> observers;

    public DSSObservable() {
        this.observers = new ArrayList<>();
    }

    public void addObserver(DSSObserver o) {
        this.observers.add(o);
    }

    public void notifyObservers(Object value) {
        this.observers.forEach(o -> o.update(this, value));
    }
}
```

envia como parâmetro o observado e um valor que é passado ao observador (depende do programa)

É uma estratégia de pull. O observado envia o valor.

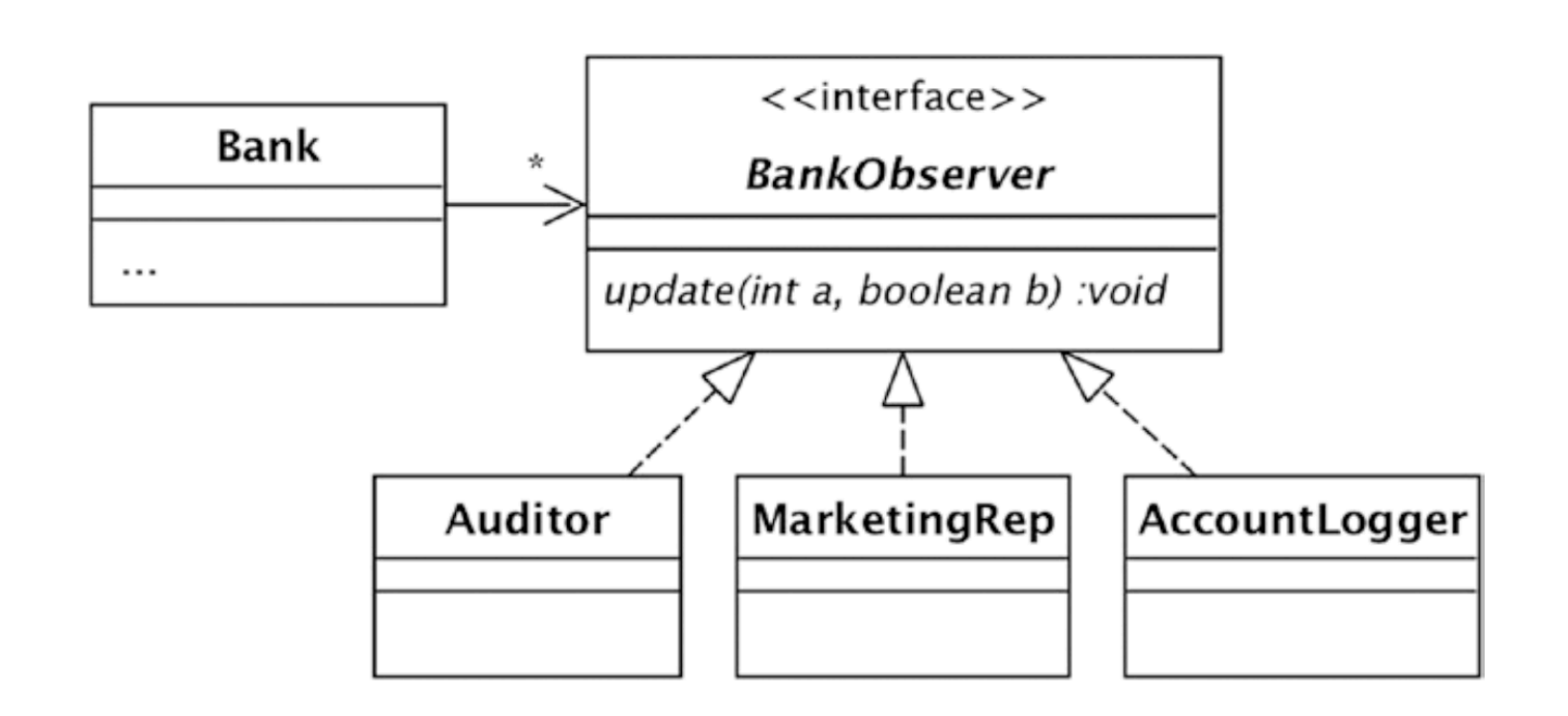
(*) retirado de um exemplo da Uc de Desenvolvimento de Sistemas de Software

- A interface Observer e as classes que a implementam:



(*) retirado de Java Program Design, E. Sciore, 2019

- No caso da aplicação bancária:



(*) retirado de Java Program Design, E. Sciore, 2019

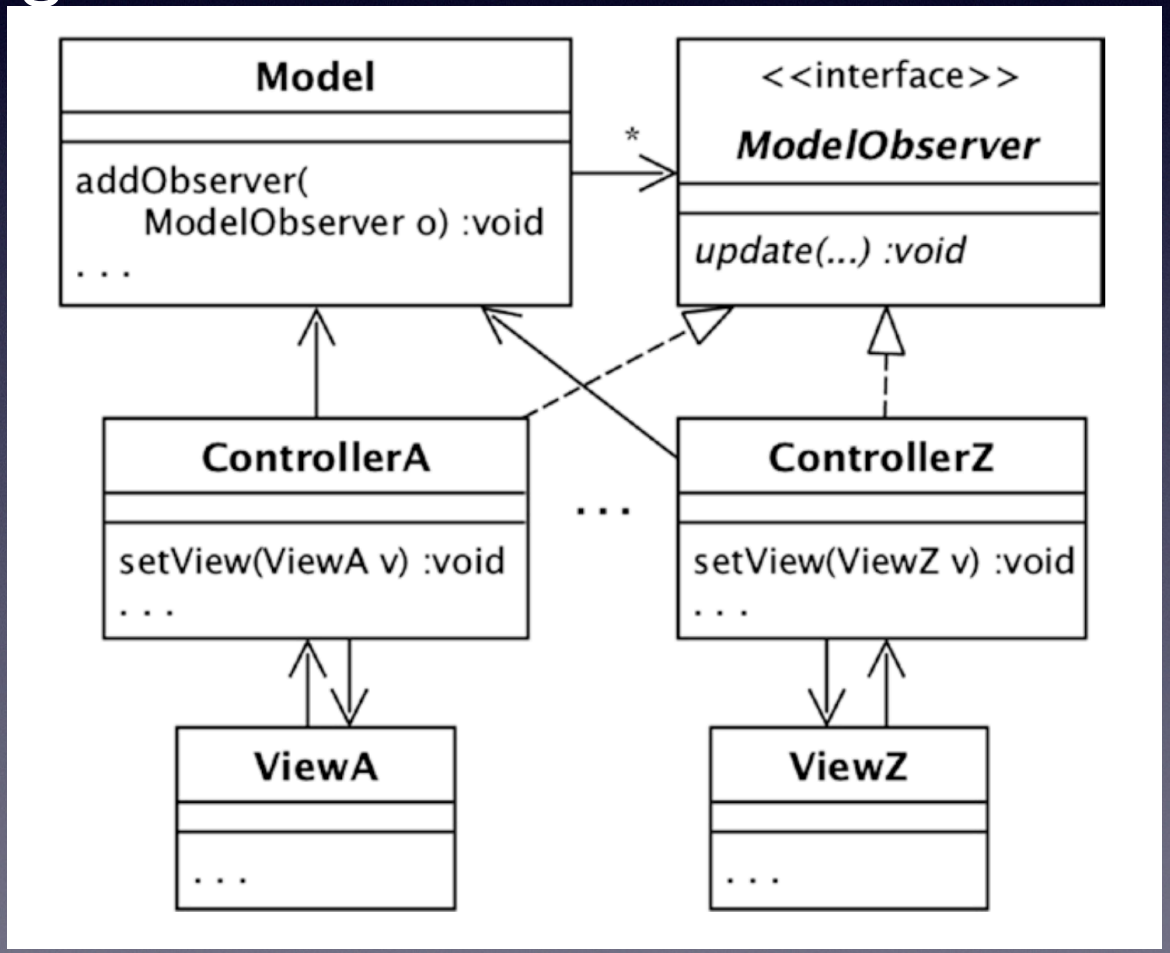
- podemos ter diferentes tipos de observadores que implementam o método update

- Utilizar a funcionalidade disponível na superclasse dos Observados para se colocar como observador:

```
public class BankProgram {
    public static void main(String[] args) {
        ...
        Bank bank = new Bank(accounts, nextacct);
        BankObserver auditor = new Auditor();
        bank.addObserver(auditor);
        ...
    }

}
```

(*) retirado de Java Program Design, E. Sciore, 2019

- Voltando ao padrão arquitectural MVC (Model-View-Controller) a ligação entre os observados e os observadores segue a estratégia:



(*) retirado de Java Program Design, E. Sciore, 2019