Múltiplas Views

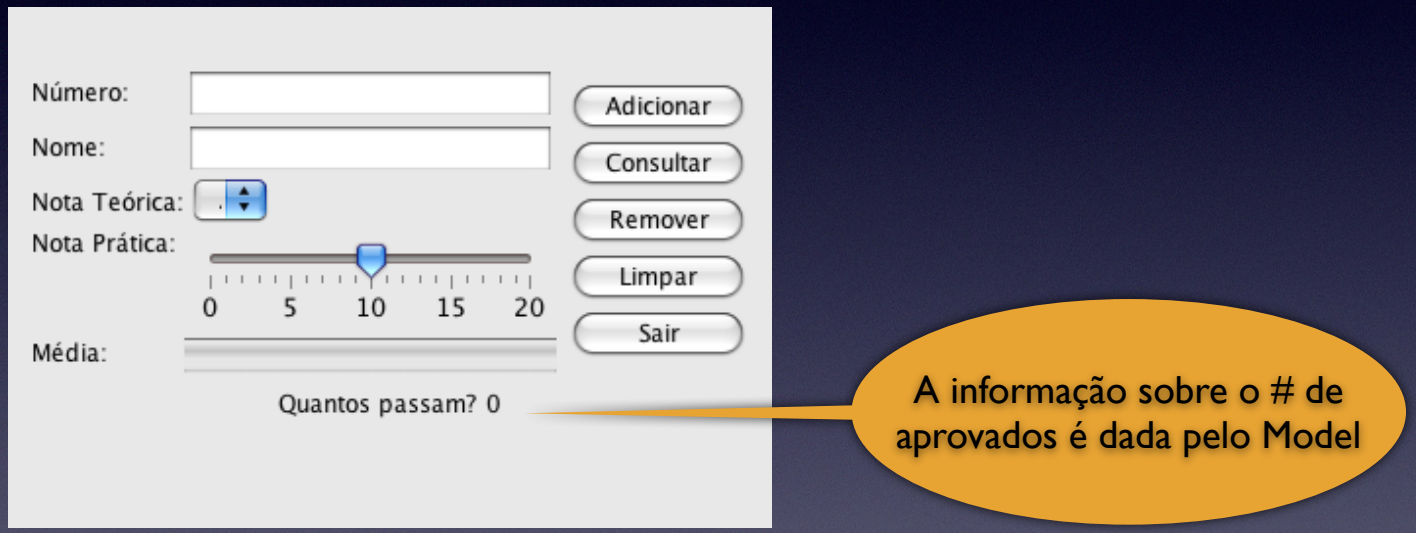
- Uma vantagem de desacoplar o modelo da vista é, além de manter separação do código, permitir ter:
  - várias vistas sobre o modelo
  - várias aplicações cliente sobre a mesma base de funcionalidade
  - se só muda a componente da interacção com o utilizador, o modelo é o mesmo

- 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 as alterações que se fazem 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

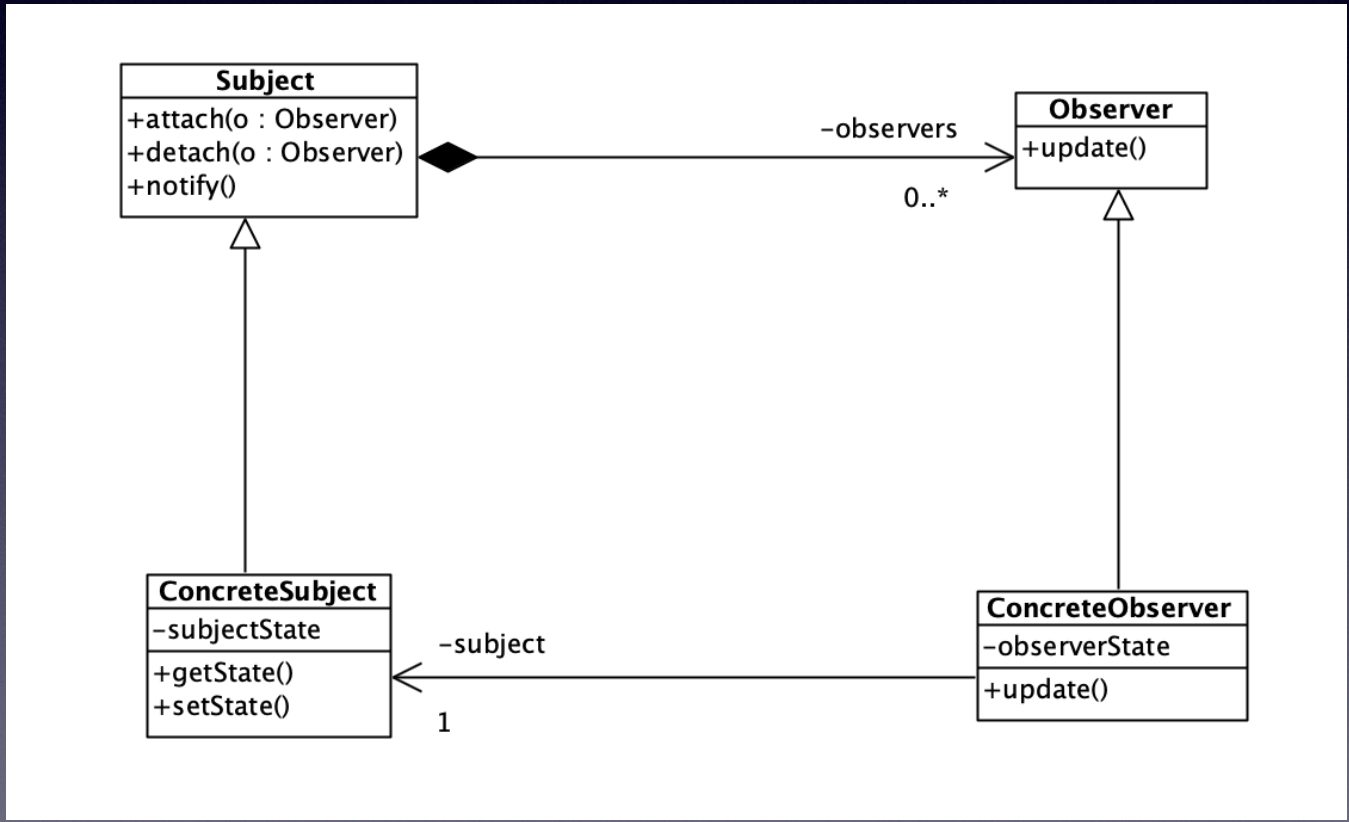
O padrão Observer

- 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

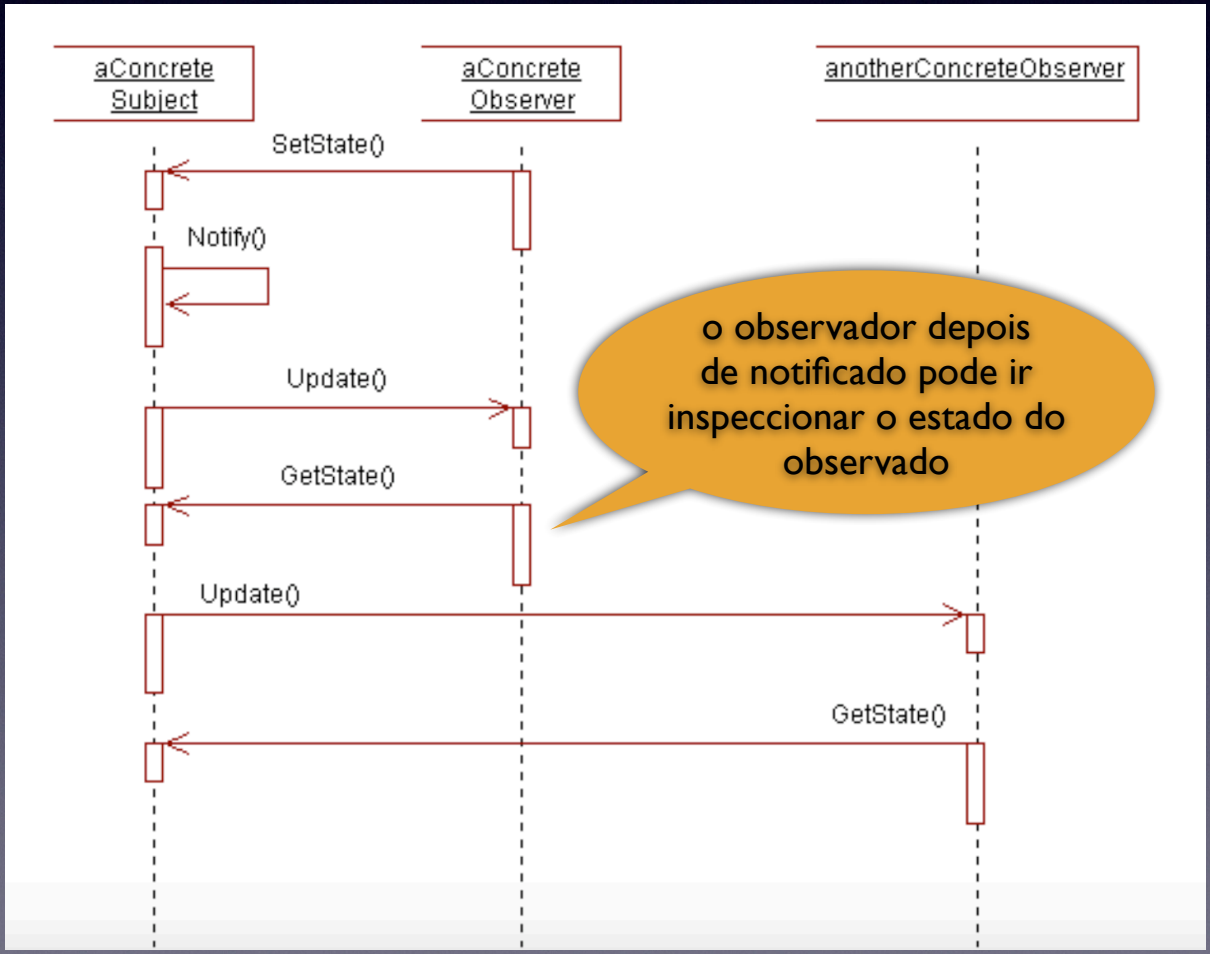
- Esta solução arquitectural pode ser utilizada:
  - quando existe uma dependência *um para muitos* entre objectos (por exemplo as `View(s)` "associadas" a um `Model`)
  - quando a mudança num objecto implica mudar o estado noutros, mas não se sabe quais (e quantos) objectos mudam
  - quando se precisa 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;
  }
```

os observadores!

(*) 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 colecção 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

```
/*
 * 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)

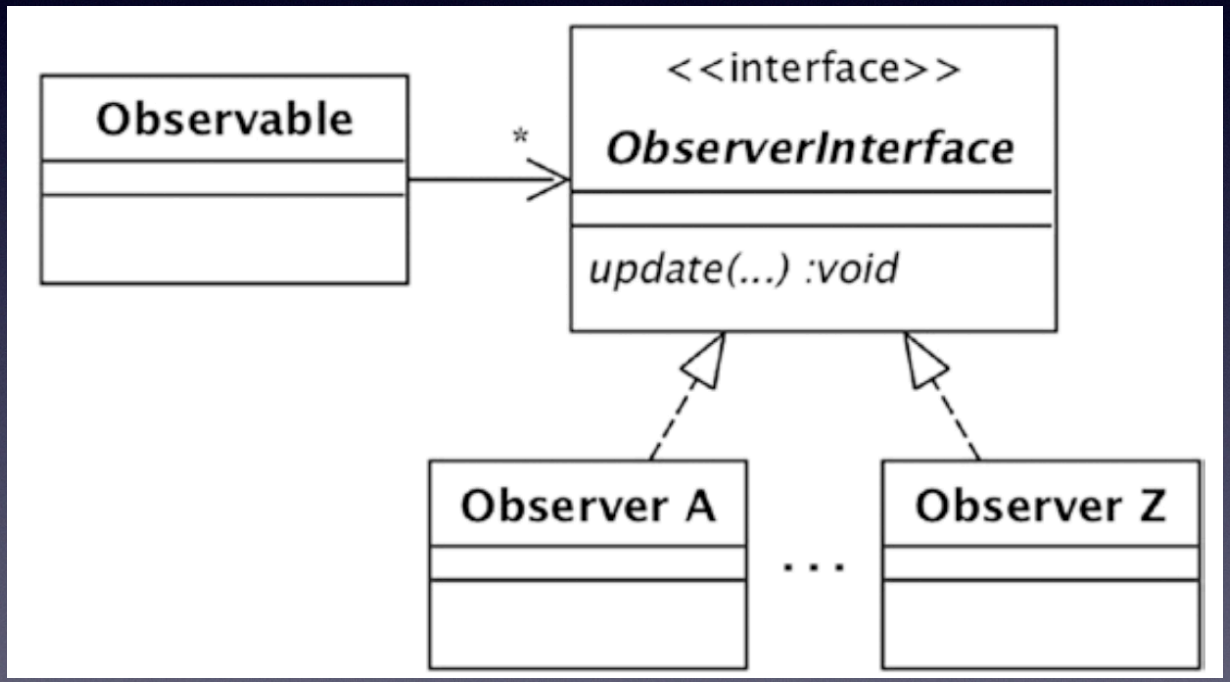
```java
package dss.pubsub;

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 (é dependente do programa!!)

É uma estratégia de *push*. O observado envia o valor.

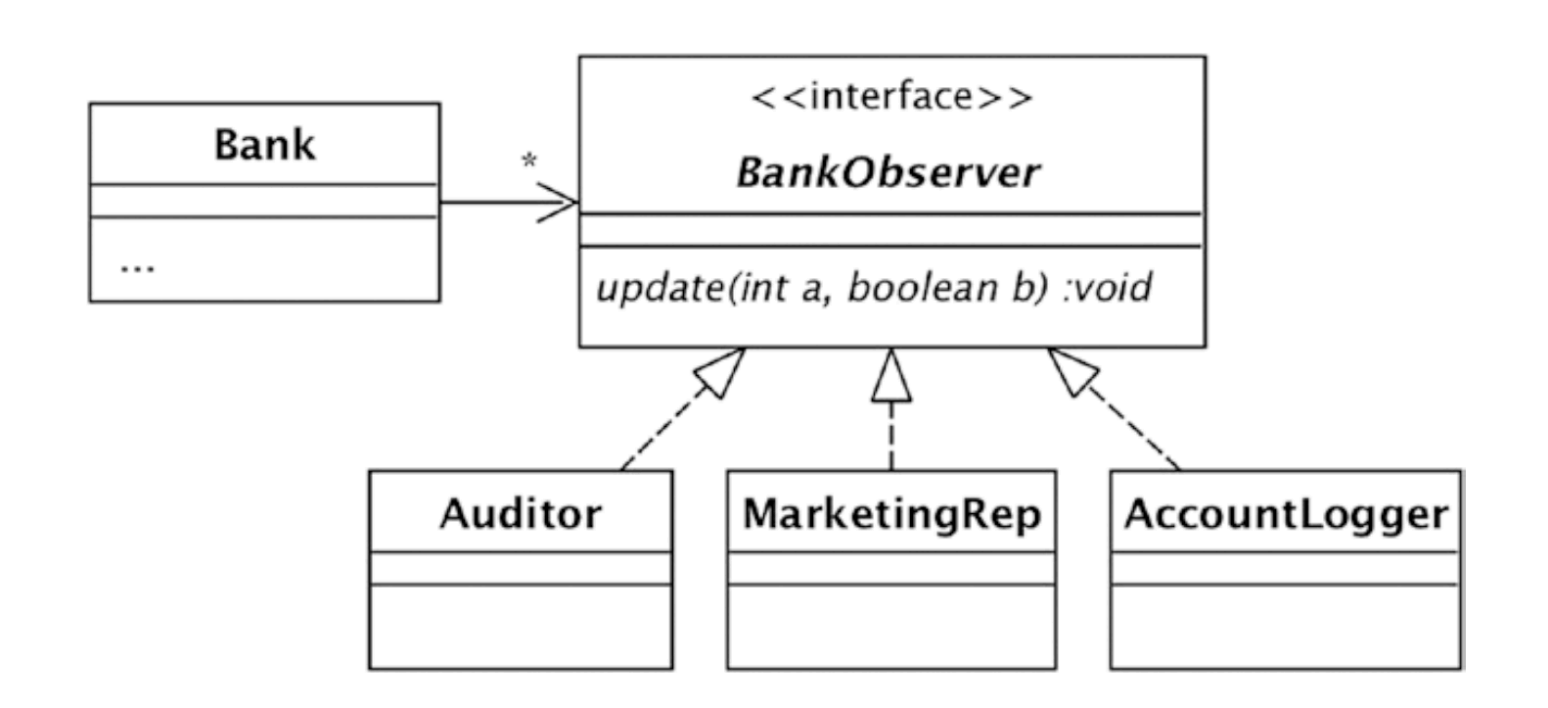
(*) retirado de um exemplo da Uc de Desenvolvimento de Sistemas de Software

- 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

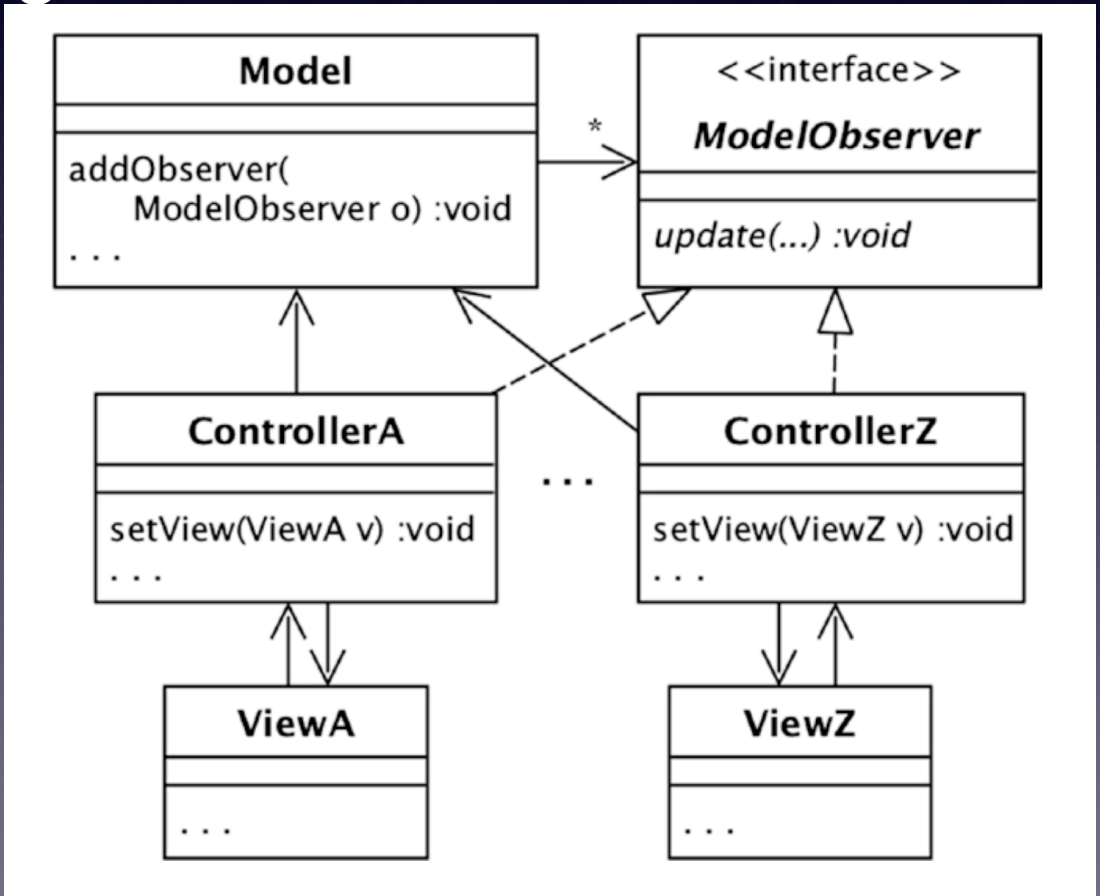
- podemos ter diferentes tipos de observadores que implementam o método update

- Utilizar a funcionalidade disponível 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);
        ...
    }

}
```

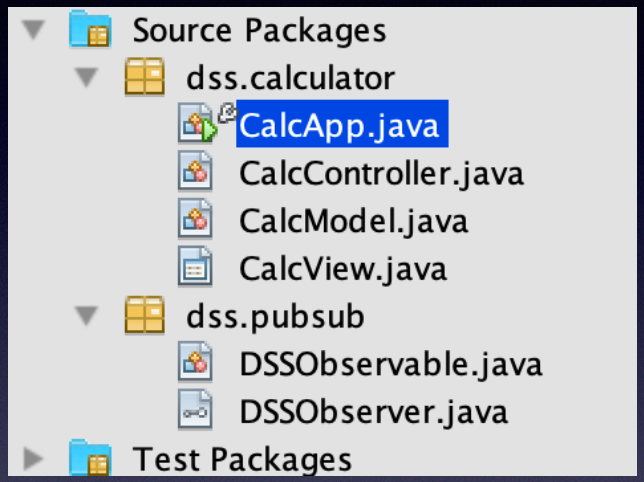
(*) retirado de Java Program Design, E. Sciore, 2019

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



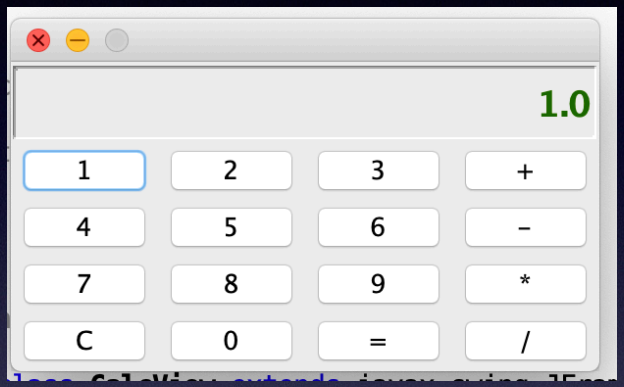
(*) retirado de Java Program Design, E. Sciore, 2019

- Outro exemplo: no projecto,



- CalcApp é a classe que cria `Model`, `Controller` e `View` e coloca tudo a correr.

- Seja um exemplo de uma aplicação que é uma calculadora.



  - A `View` tem o layout de uma calculadora (com botões e campo de texto)
  - O `Model` implementa as operações (+,-,* e /)

- A classe principal (com o main)

```
public class CalcApp {

    private CalcApp() {
    }

    public static void main(String args[]) {
        SwingUtilities.invokeLater(new Runnable() {

            @Override
            public void run() {
                CalcModel model = new CalcModel();
                CalcController controller = new CalcController(model);
                CalcView view = new CalcView(controller);
                /** view registada como observador do controller para poder actualizar o écran
                 * durante a construção do número no controller  */
                controller.addObserver(view);

                /** controller registado como observador do model para poder actualizar o valor após operações no model */
                model.addObserver(controller);
                view.run();
            }
        });
    }
```

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

- O Model:

```
public class CalcModel extends DSSObservable {
    private double value;

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

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

    public void subtract(double v) {
        this.value -= v;
        this.notifyObservers(""+value);
    }

    public void multiply(double v) {
        this.value *= v;
        this.notifyObservers(""+value);
    }

    public void divide(double v) {
        this.value /= v;
        this.notifyObservers(""+value);
    }

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

O valor calculado foi alterado e notifica os observadores!

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

- O Controller:

```
public class CalcController extends DSSObservable implements DSSObserver {

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

    /** Creates a new instance of Calculadora */
    public CalcController(CalcModel model) {
        this.screen_value = 0;
        this.lastkey = ' ';
        this.opr = '=';
        this.model = model;
        /** o notifyObservers serve para comunicar o novo valor da calculadora */
        this.notifyObservers(this.screen_value);
    }

    public void processa(int d) {
        if (this.lastkey != 'd') {
            this.screen_value = d;
            this.lastkey = 'd';
        } else {
            this.screen_value = this.screen_value*10+d;
        }
        /** o notifyObservers serve para comunicar o novo valor da calculadora */
        this.notifyObservers(this.screen_value);
    }
```

Nota: no caso do Java Swing o código do controller e da View usualmente está no mesmo ficheiro. Mas podia ser separado usando a estratégia apresentada…

notifica o observador, neste caso a View

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

- continuação…

```
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 d
                  break;
    };
    this.opr = opr;
    this.lastkey = opr;
}

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

@Override
public void update(DSSObservable source, Object value) {
    this.screen_value = Double.parseDouble(value.toString());
}
```

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

- A `View` (neste caso em Java Swing):

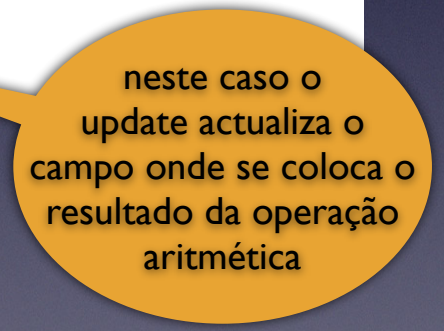
```
public class CalcView extends javax.swing.JFrame implements DSSObserver {

    /** A calculadora que vai fazer as contas... */
    private CalcController controller;

    /** Creates new form JCalculadora */
    public CalcView(CalcController ctl) {
        this.controller = ctl;
    }

    /** This method is called from within the constructor to
     * initialize the form.
     * WARNING: Do NOT modify this code. The content of this method is
     * always regenerated by the Form Editor.
     */
    Generated Code

    private void opr_press(java.awt.event.ActionEvent evt) {
        // Add your handling code here:
        this.controller.processa(evt.getActionCommand().charAt(0));
    }
```

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

- A `View`, implementa a interface Observer, e tem de fornecer um implementação para o método update:

```
/**
 * Método correspondente à interface Observer.
 * Este é o método que é invocado sempre que a calculadora efectua um
 * notifyObservers, actualiza o écran com o valor que vem como parâmetro
 */
public void update(DSSObservable o, Object arg) {
    this.screen.setText((arg.toString()));
}
```

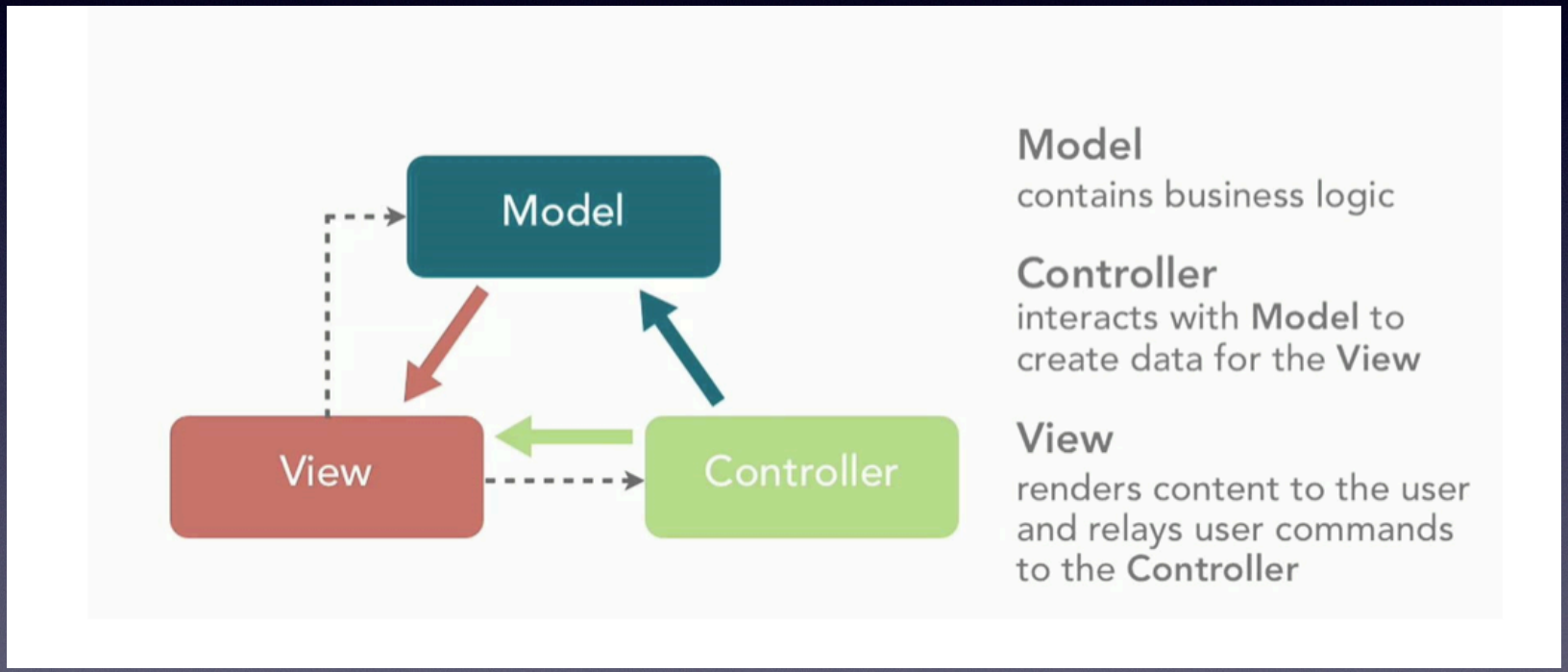
(*) retirado de um exemplo da Uc de Desenvolvimento de Sistemas de Software



- Existem várias abordagens ao padrão MVC:
  - na comunidade não há um consenso claro de qual é a abordagem mais correcta (deve a view observar o model? a view deve interrogar directamente o model?)
  - o que se apresentou atrás é apenas uma visão de separação entre `Model`, `View` e `Controller`

- A definição base em que todos podemos concordar é que: os controllers fazem pedidos ao model e o model notifica os observadores quando o seu estado muda
  - idealmente devem ser os controllers os observadores do model, mas encontram-se abordagens em que é a view
- Mais importante é a noção de desacoplamento que a utilização de MVC e Observer proporcionam.

Ainda sobre MVC...



Ainda sobre MVC...

MVC in Swing

Swing actually uses a simplified variant of the MVC design called the *model-delegate* . This design combines the view and the controller object into a single element, the *UI delegate* , which draws the component to the screen and handles GUI events. Bundling graphics capabilities and event handling is somewhat easy in Java, since much of the event handling is taken care of in AWT. As you might expect, the communication between the model and the UI delegate then becomes a two-way street, as shown in Figure 1-8.

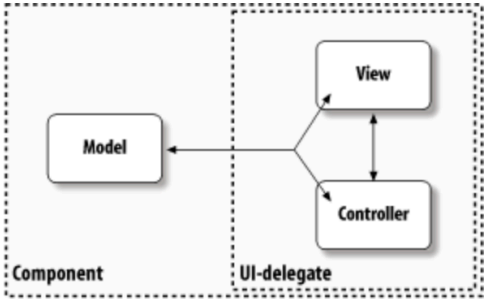


Figure 1-8. With Swing, the view and the controller are combined into a UI-delegate object

Uma abordagem com o MVC delegate

```
/**
 * Classe APPNewGen - APP New Generation, que utiliza as classes dos menús com
 * handlers de comportamento (View) e a classe que implementa a TextUI (Controller).
 */


public class APPNewGen {

  public static void main(String[] args) {
    new TextUI().run();    // coloca o delegate em execução
  }

}
```

- A classe que implementa a UI em modo texto:

```
public class TextUI {
  // o modelo
  private DriveIt model;

  //scanner
  private Scanner sc;

  /**
   * Construtor que cria os menus e o model
   */

  public TextUI() {
    this.model = new DriveItPL8();
    sc = new Scanner(System.in);
  }
```

o Model (de uma empresa de aluguer de veículos)

```java
/**
 * Método que executa o menu principal.
 * Coloca a interface em execução.
 */

public void run() {
  NewMenu menu = new NewMenu(new String[] {
          " Gestão de Veículos",
          " Dar Nome à Empresa"
  });

  //pré-condição: só se altera o nome da empresa se este for vazio!!
  menu.setPreCondition(2, () -> this.model.getNome().equals(""));
  //registar os métodos de tratamento, passados como
  //expressões lambda (interface funcional)
  menu.setHandler(1, () -> gestaoVeiculos());
  menu.setHandler(2, () -> alterarNomeEmpresa());

  //colocar o menú em visualização e esperar leitura de opção
  menu.run();
}
```

uma expressão lambda com a pré-condição

uma expressão lambda com o comportamento a executar

```
private void gestaoVeiculos() {
  NewMenu menuVeiculos = new NewMenu(new String[] {"Adicionar Veiculo",
                             "Listar Veiculo",
                             "Remover Veiculo"});

  //registar os handlers
  menuVeiculos.setHandler(1, () -> addVeiculo());
  menuVeiculos.setHandler(2, () -> listVeiculo());
  menuVeiculos.setHandler(3, () -> remVeiculo());

  menuVeiculos.run();
}


private void alterarNomeEmpresa() {
  System.out.println("Novo nome da Empresa: ");
  String nome = sc.nextLine();
  this.model.setNome(nome);
}
```

A classe Menu com handlers

```
public class NewMenu {

    // Interfaces auxiliares

    /** Functional interface para handlers. */
    public interface Handler {  // método de tratamento
        public void execute();
    }

    /** Functional interface para pré-condições. */
    public interface PreCondition {
        public boolean validate();
    }

    // Varíável de classe para suportar leitura

    private static Scanner is = new Scanner(System.in);

    // Variáveis de instância

    private List<String> opcoes;             // Lista de opções
    private List<PreCondition> disponivel;   // Lista de pré-condições
    private List<Handler> handlers;          // Lista de handlers
```

Podemos também utilizar Predicate<T>

```java
/**
 * Correr o NewMenu.
 *
 * Termina com a opção 0 (zero).
 */
public void run() {
    int op;
    do {
        show();
        op = readOption();
        // testar pré-condição
        if (op>0 && !this.disponivel.get(op-1).validate()) {
            System.out.println("Opção indisponível! Tente n
        } else if (op>0) {
            // executar handler
            this.handlers.get(op-1).execute();
        }
    } while (op != 0);
}
```

Executar o lambda da pré-condição.

Executar o lambda da acção a executar.

```java
/** Apresentar o NewMenu */
private void show() {
    System.out.println("\n *** NewMenu *** ");
    for (int i=0; i<this.opcoes.size(); i++) {
        System.out.print(i+1);
        System.out.print(" - ");
        System.out.println(this.disponivel.get(i).validate()?this.opcoes.get(i):"---");
    }
    System.out.println("0 - Sair");
}

/** Ler uma opção válida */
private int readOption() {
    int op;
    //Scanner is = new Scanner(System.in);

    System.out.print("Opção: ");
    try {
        String line = is.nextLine();
        op = Integer.parseInt(line);
    }
    catch (NumberFormatException e) { // Não foi escrito um int
        op = -1;
    }
    if (op<0 || op>this.opcoes.size()) {
        System.out.println("Opção Inválida!!!");
        op = -1;
    }
    return op;
}
```

No caso da pré-condição não ser válida não apresenta a opção de menu