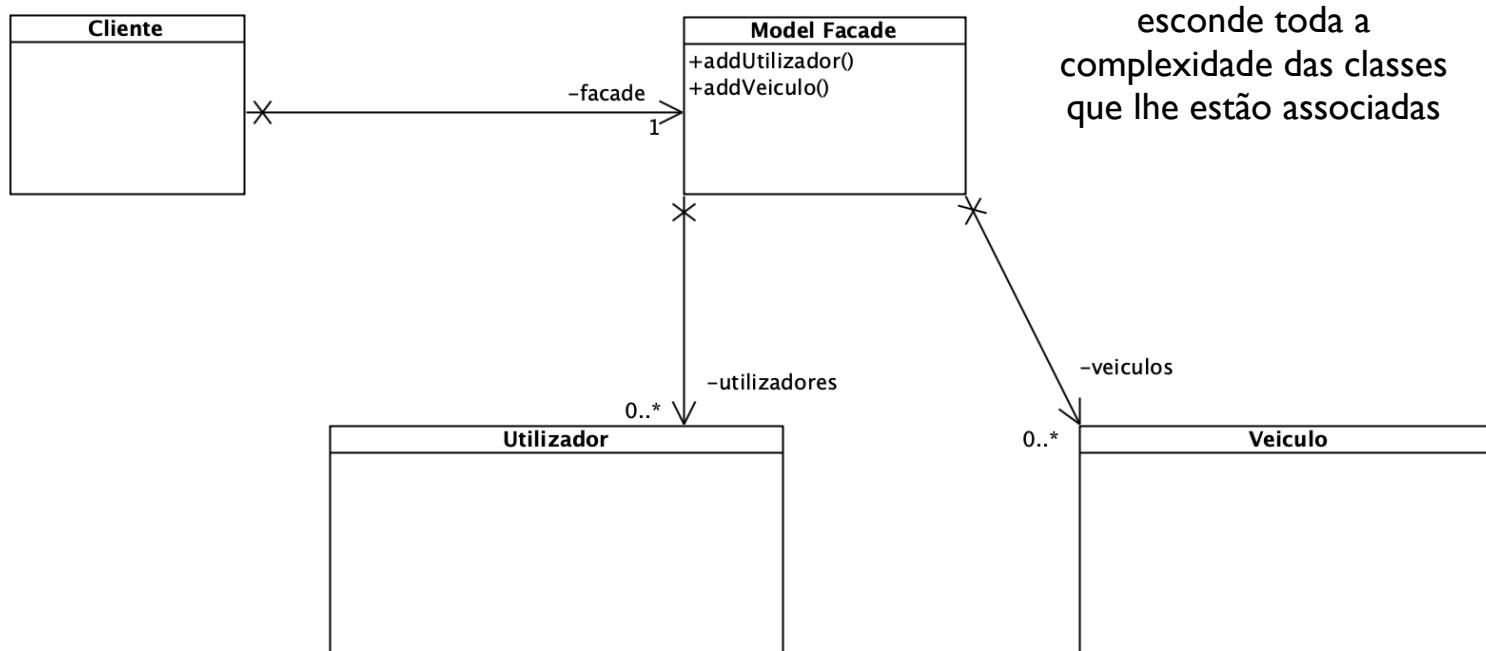


O padrão Facade

- Este padrão determina que podemos ter uma classe a fornecer serviços para os clientes, permitindo:
 - diminuir as dependências entre classes
 - encapsular e esconder classes que estão para trás do facade
 - permitir evoluir de forma autónoma as entidades “escondidas”

- Por vezes temos uma classe a fazer este papel de “fachada”, mas podemos ter também uma interface (uma API).



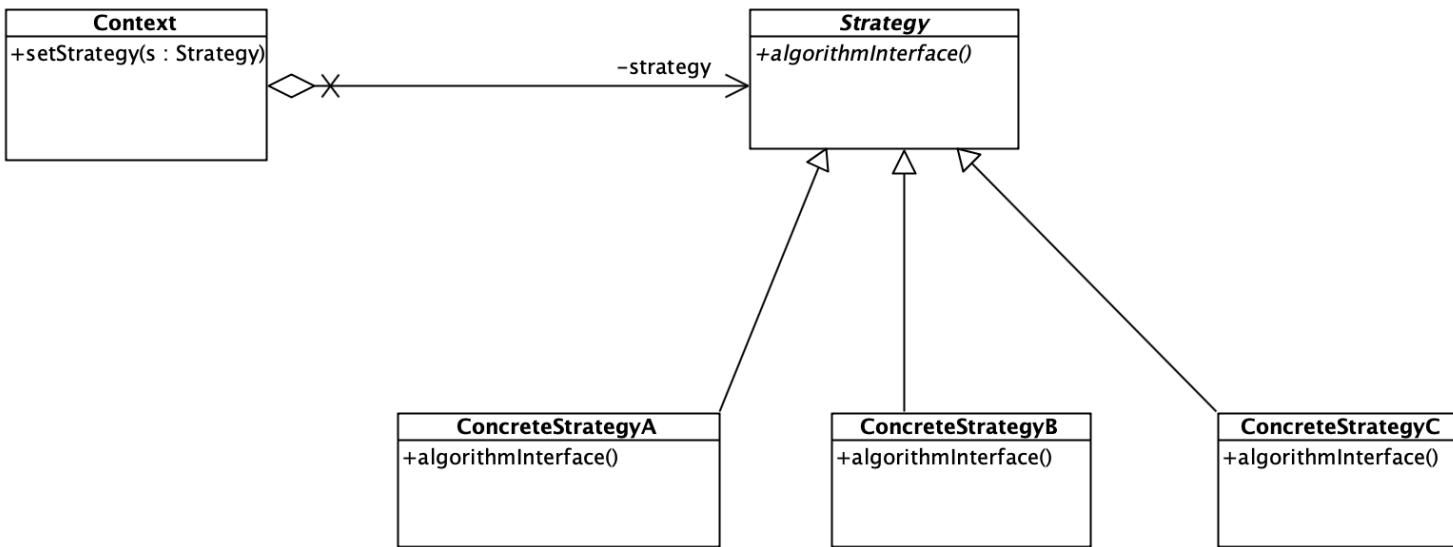
O padrão Strategy

- Este padrão de concepção permite autonomizar o comportamento, possibilitando que este seja passado como parâmetro.
- várias das nossas operações sobre estruturas de dados podem assim ser refeitas, permitindo diminuir o código e evitando repetições
- torna também os programas mais flexíveis a alterações de comportamento

- O objectivo é definir uma família de algoritmos, encapsular cada um deles num objecto, tornando-os assim reutilizáveis em mais do que uma situação.
- Possibilita-se assim que aplicações cliente diferentes possam utilizar algoritmos (estratégias) diferentes.
- Diversas operações de transformação dos elementos de estruturas de dados podem ser revistas à luz deste padrão.

- Aplicação:
 - quando se necessita de variações de um algoritmo e não se quer reflectir isso na escrita dos métodos (criar muitas estruturas do tipo if...then...else)
 - quando o algoritmo usa dados que não devem ser conhecidos da aplicação cliente
 - muitas classes relacionadas são diferentes a nível de comportamento e podemos retirar essa complexidade passando-a como parâmetro

● O padrão Strategy



- no modelo acima usa-se uma classe abstracta mas poderia também ser uma interface.
- o método **setStrategy** pode ser invocado para alterar o algoritmo

- Já vimos anteriormente uma situação que decorre da utilização deste padrão.

```
/**  
 * Método que recebe uma Consumer<T> e aplica a todos os  
 * hóteis existentes.  
 */  
  
public void aplicaTratamento(Consumer<Hotel> c) {  
    this.hoteis.values().forEach(h -> c.accept(h));  
}
```

```
Consumer<Hotel> downgradeEstrelas = h -> h.setEstrelas(h.getEstrelas()-1);  
osHoteis.aplicaTratamento(downgradeEstrelas);
```

- Permite detectar funcionalidades semelhantes e factorizá-las. Favorece a criação de uma família de algoritmos
- Apresenta uma alternativa ao esquema natural de herança - as alterações/variantes são passados como parâmetros
- permite eliminar expressões condicionais na escolha do algoritmo
- compatível com a utilização de `java.util.function`

Model, View, Controller

- Quando construímos aplicações somos condicionados a não confundir código de interacção com o utilizador com o código da chamada camada computacional.
- porque tem tempos de alteração e construção diferentes
- porque normalmente o tipo de código, e mesmo tecnologia, é diferente

- Chamamos View ao código da componente que faz a interacção com o utilizador
- Chamamos Model ao código que assegura a parte das regras e camada computacional
 - que sempre definimos que não fazia nenhuma interacção de I/O para poder ser reutilizável

- A regra básica exprime-se como “Separar o Model (o modelo) da View (a vista)”
- em OO para alcançar este desiderato é necessário ter:
 - classes dedicadas à codificação da vista
 - classes dedicadas à codificação do modelo
 - não devemos ter classes que tenham ambas as competências.

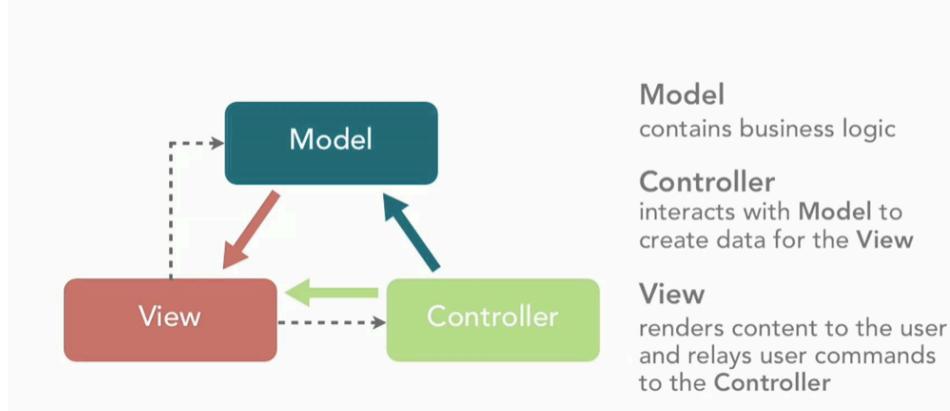
- A View deve ter preocupações com:
 - usabilidade
 - ser visualmente agradável, poder mudar layout, etc.
- O Model deve preocupar-se em ser:
 - eficiente
 - modular, reutilizável, etc.

- Para manter esta separação deve existir um componente (uma classe) que faz a ligação entre a View e o Model
 - Essa classe chama-se Controller
 - O controller faz a mediação entre a View e o Model
 - Sabe qual é o método do Model que tem de ser invocado para satisfazer o requisito da View

- Este padrão arquitectural designa-se por **Model-View-Controller (MVC)**
- este padrão indica que não deve existir uma classe que faça mais do que um papel ao mesmo tempo.

The Model-View-Controller Rule

A program should be designed so that its model, view,
and controller code belong to distinct classes.



- O controller recebe os pedidos da View e encaminha para o Model
- As respostas do Model são enviadas para a View, sendo mediadas pelo Controller
- existe a possibilidade de serem enviadas directamente desde que não se conheça a View (existem variantes do MVC!)

- No livro Java Program Design (ver bibliografia da UC), apresenta-se um exemplo de uma aplicação bancária em que se pode verificar uma situação de não separação de camadas.
- O Model é representado pela classe Bank
- A classe que implementa a interacção com o utilizador e faz render da View é a classe BankClient

```

public class BankClient {
    private Scanner scanner;
    private boolean done = false;           a View manipula
    private Bank bank;                   directamente o Model
    private int current = 0;

    ...

    private void processCommand(int cnum) {

        inputCommand cmd = commands[cnum];
        current = cmd.execute(scanner, bank, current);
        if (current < 0)
            done = true;

    }
}

```

a View faz a gestão do
que é lido e invoca os
métodos no Model

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

- Como se vê a View conhece o Model e faz a gestão da invocação dos métodos

- Este mecanismo de construção não salvaguarda a independência de camadas e não possibilita o desacoplamento
- é necessário criar um mecanismo de *middleware*, o Controller, que seja conhecido da View e que conheça o Model

```
public class BankClient {                                o controlador
    private Scanner scanner;
    private InputController controller;
    private InputCommand[] commands = InputCommands.values();

    public BankClient(Scanner scanner, InputController cont) {
        this.scanner = scanner;
        this.controller = cont;
    }

    public void run() {
        String usermessage = construtcMessage();
        String response = "";

        while (!response.equals("Goodbye!")) {
            System.out.println(usermessage);
            int cnum = scanner.nextInt();
            InputCommand cmd = commands[cnum];
            response = cmd.execute(scanner, controller);
            System.out.println(response);
        }
    }

    ...
}
```

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

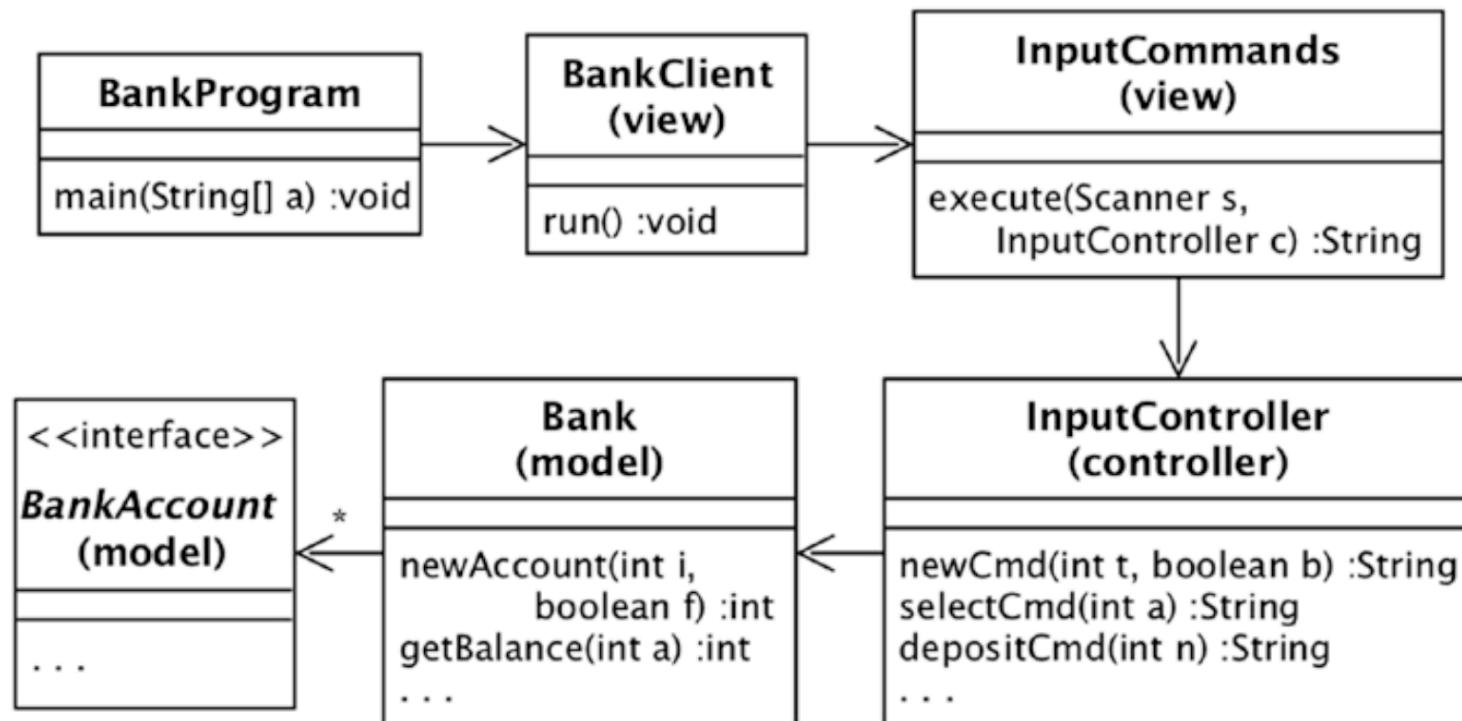
- O programa principal é agora o responsável pela criação das várias camadas e pela interligação das mesmas:
 - deve passar para o Controller a referência do Model
 - deve fornecer à View a referência do Controller

- Exerto do arranque do programa com a interligação das camadas

```
Map<Integer,BankAccount> accounts = info.getAccounts();  
int nextacct = info.nextAcctNum();  
Bank bank = new Bank(accounts, nextacct);  
...  
InputController controller = new InputController(bank);  
Scanner scanner = new Scanner(System.in);  
BankClient client = new BankClient(scanner, controller);  
client.run();  
info.saveMap(accounts,bank.nextAcctNum());
```

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

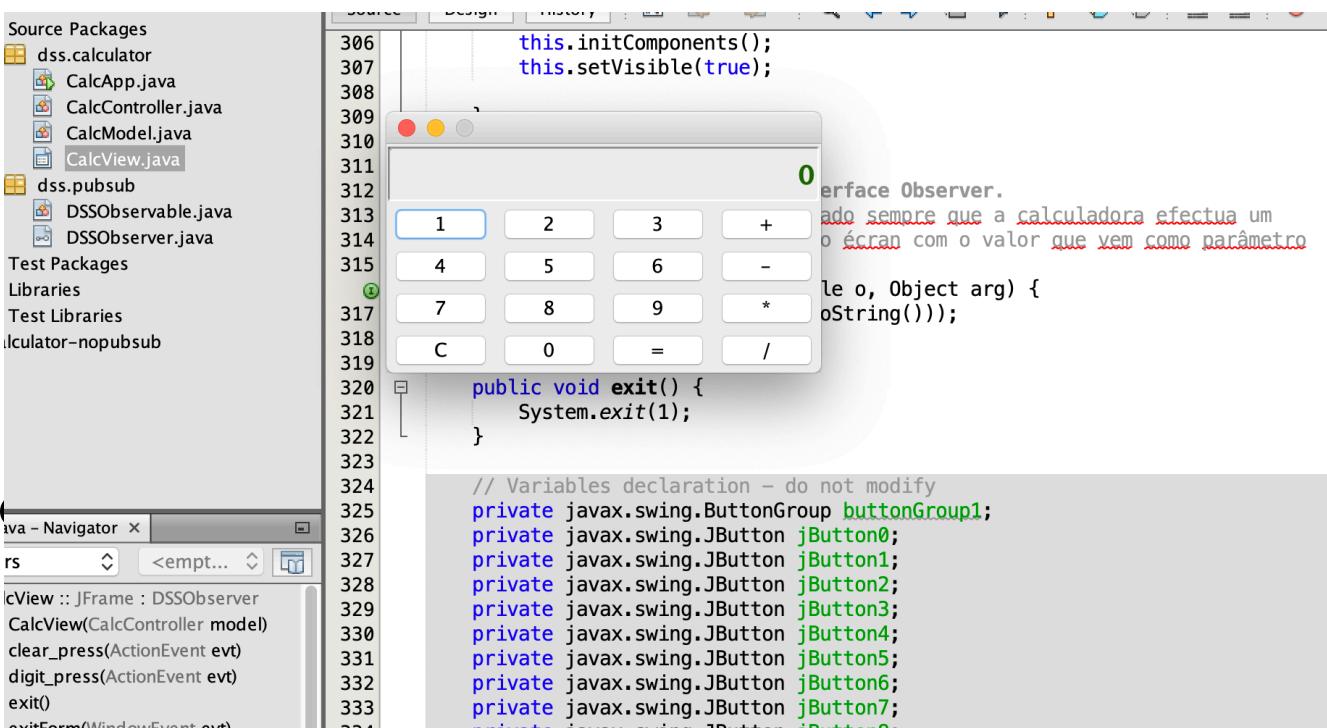
- Do ponto de vista arquitectural, temos o seguinte diagrama:



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

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 DSS0bservable implements DSS0bserver {

    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:

The screenshot shows a window titled 'Nota Teórica' (Theoretical Grade) with the following fields and controls:

- Número: (Number) - An input field.
- Nome: (Name) - An input field.
- Nota Teórica: (Theoretical Grade) - A numeric input field with up/down arrows.
- Nota Prática: (Practical Grade) - A horizontal slider scale from 0 to 20, with a blue marker at 10.
- Média: (Average) - A horizontal slider scale from 0 to 20, with a blue marker at 10.
- Buttons on the right: Adicionar (Add), Consultar (Search), Remover (Remove), Limpar (Clear), and Sair (Exit).
- Text at the bottom: Quantos passam? 0

A informação sobre o # de aprovados é dada pelo Model

- o Model é que possui as regras que determinam em que circunstância 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

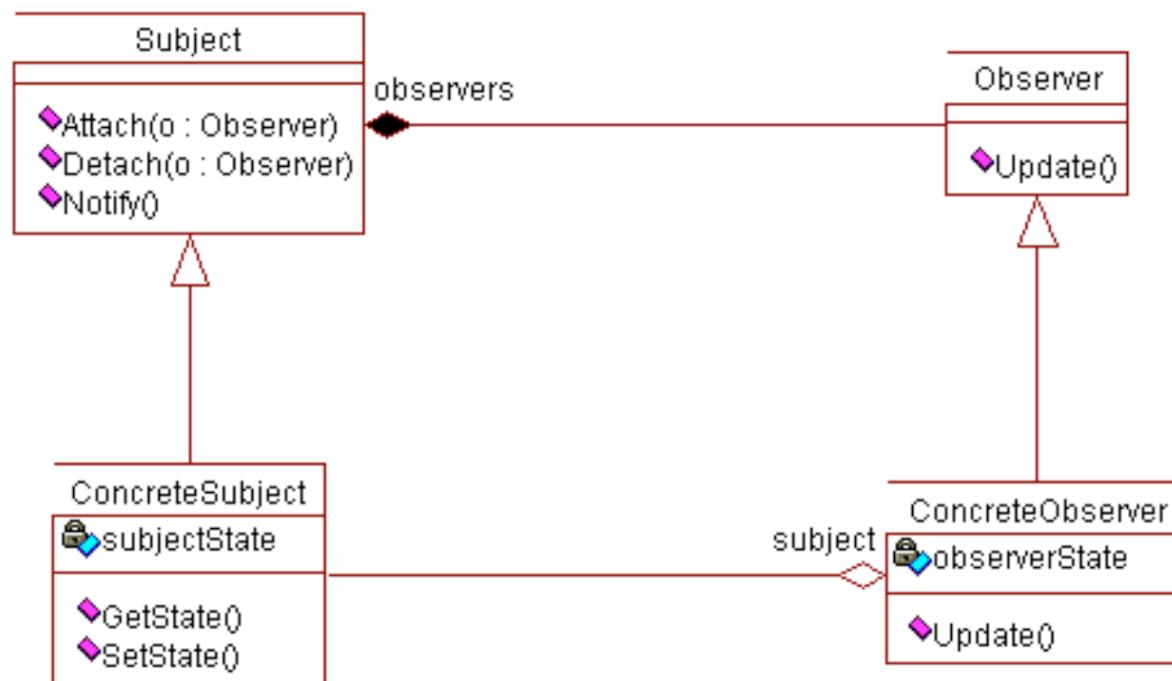
O padrão Observer

- 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

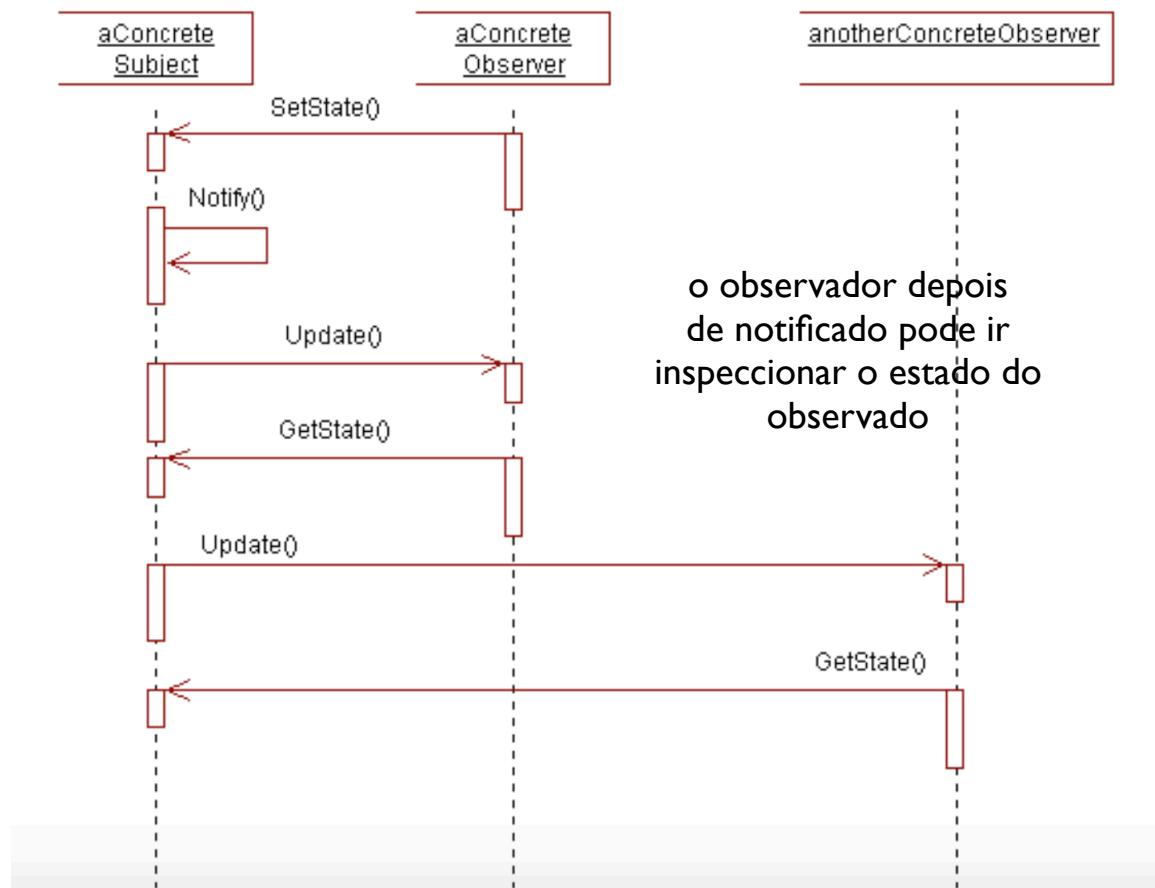
- Esta solução arquitectural pode ser utilizada:
 - quando existe uma dependência um para muitos entre objectos (por exemplo as View(s) “associadas” a um Model)
 - quando a mudança num objecto implica mudar o estado noutras, mas não se sabe quais (e quantos) objectos mudam
 - quando se precisa que um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

```
/*
 * 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)

```

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));
    }
}

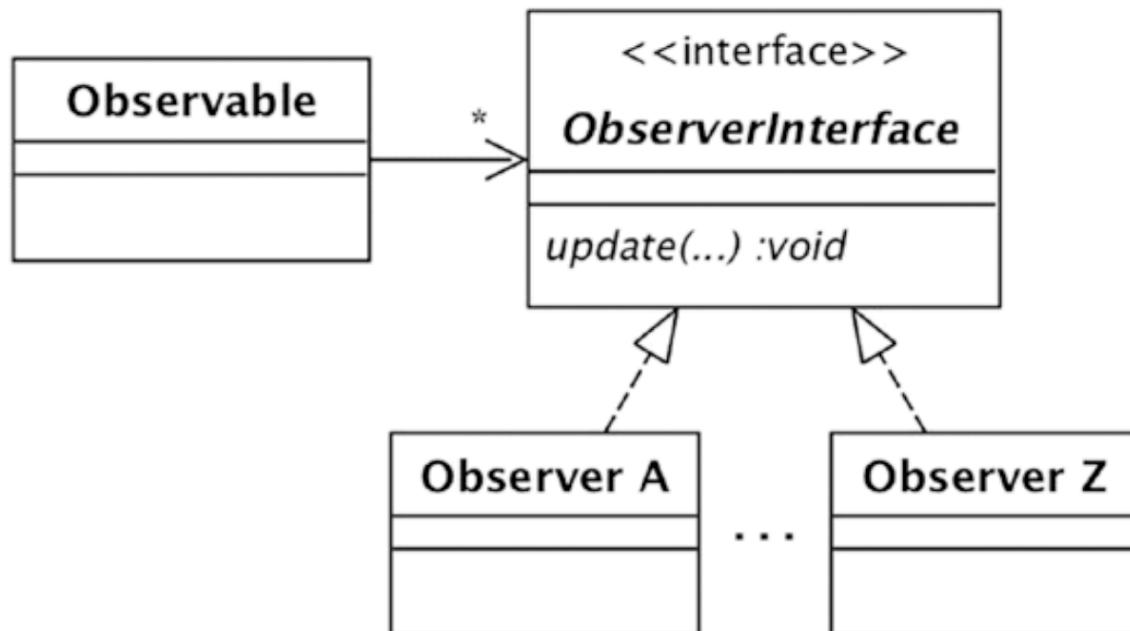
```

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

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

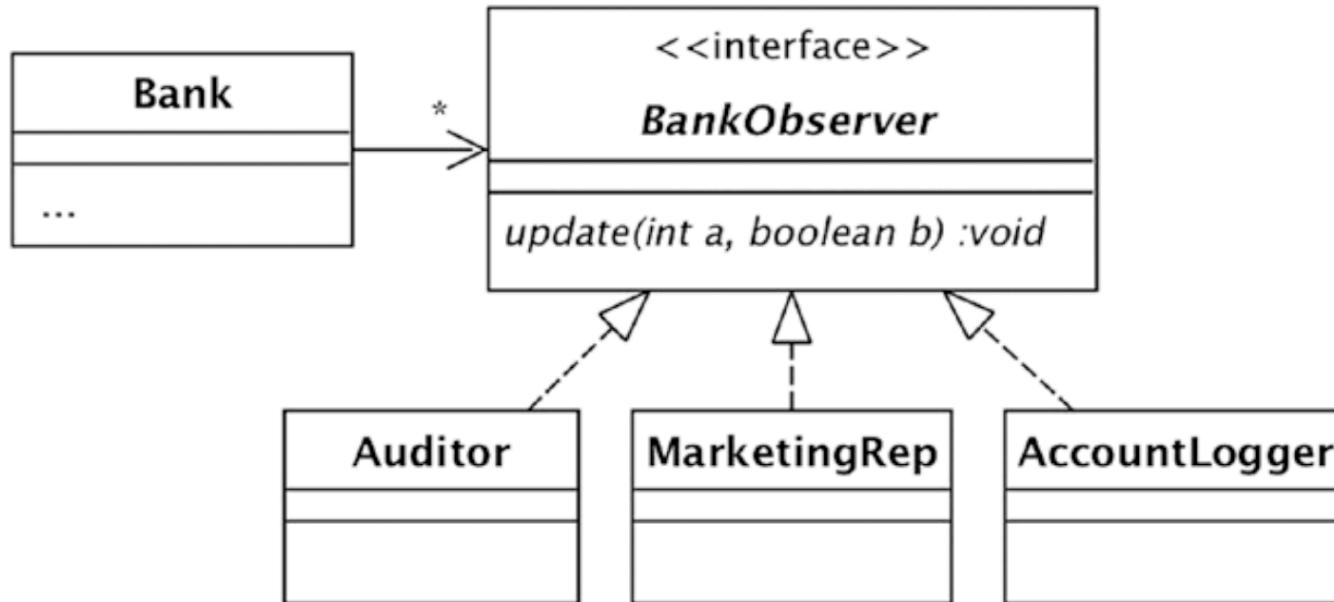
(*) retirado de um exemplo da Uc de Desenvolvimento de Sistemas de Software

- 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

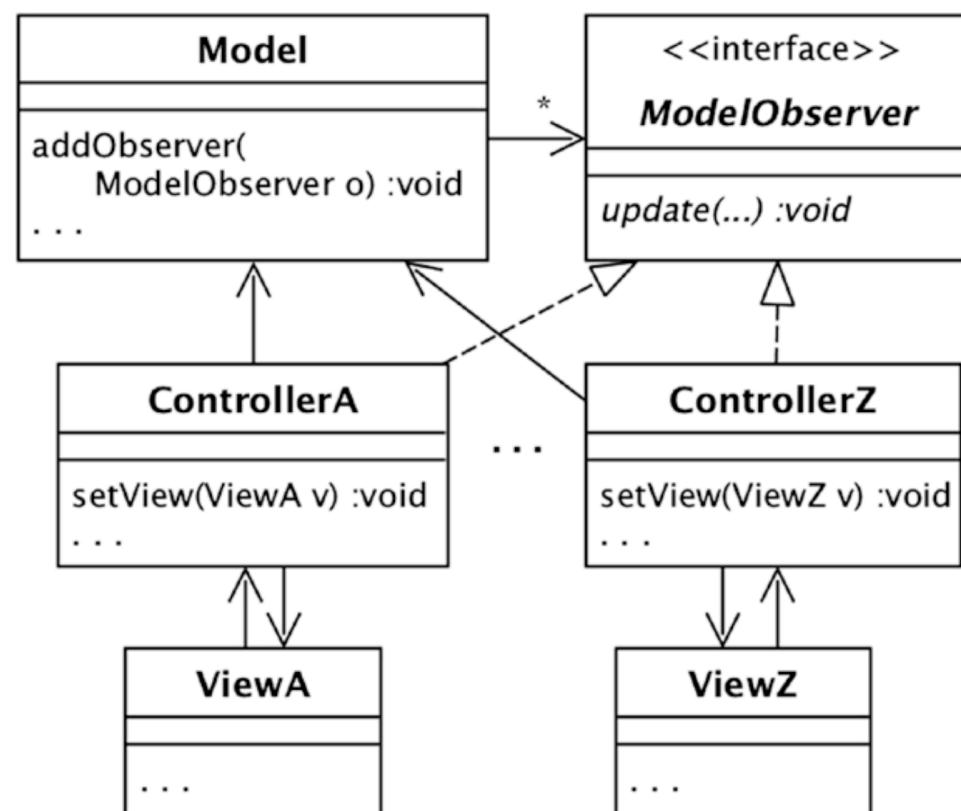
- podemos ter diferentes tipos de observadores que implementam o método update

- Utilizar a funcionalidade disponível 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);  
        ...  
    }  
}
```

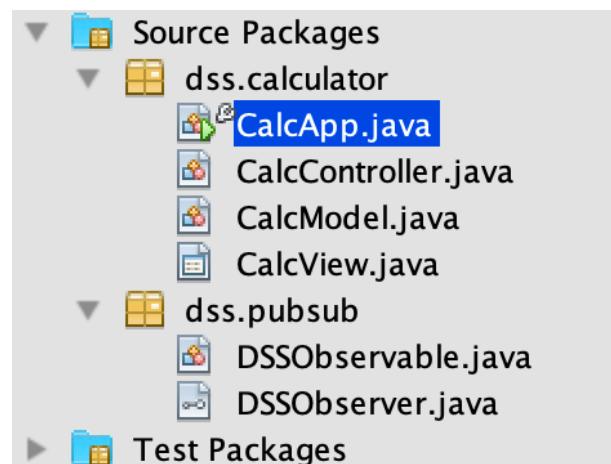
(*) retirado de Java Program Design, E. Sciore, 2019

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



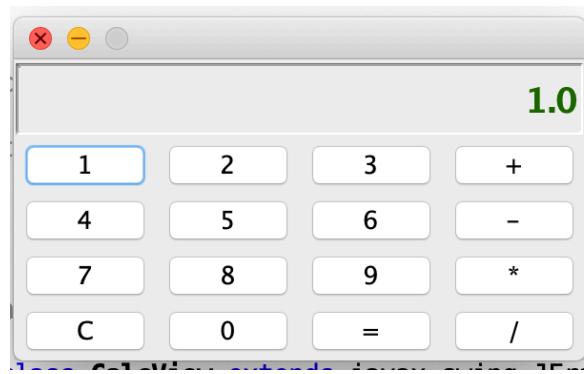
(*) retirado de Java Program Design, E. Sciore, 2019

- Outro exemplo: no projeto,



- **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

- ○ Model:

```
public class CalcModel extends Observable {  
    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;  
    }  
}
```

(*) 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 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) {  
        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...

(*) 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 de divisão por zero
                    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 ecran 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

neste caso o
update actualiza o
campo onde se coloca o
resultado da operação
aritmética

- Exemplo de uma aplicação com View em modo texto (exemplo dos Hotéis visto anteriormente):

```
public class HoteisIncApp {  
  
    // A classe HoteisInc tem a 'lógica de negócio'.  
    private HoteisInc logNegocio;  
  
    // Menus da aplicação  
    private Menu menuPrincipal, menuHoteis;  
  
    /**  
     * O método main cria a aplicação e invoca o método run()  
     */  
    public static void main(String[] args) {  
        new HoteisIncApp().run();  
    }  
}
```

- Esta classe também implementa o Controller:

```
private void run() {  
    System.out.println(this.logNegocio.toString());  
    do {  
        menuPrincipal.executa();  
        switch (menuPrincipal.getOpcao()) {  
            case 1: System.out.println("Escolheu adicionar");  
                      break;  
            case 2: //trataConsultarHotel();  
            case 3: //outro método  
        }  
    } while (menuPrincipal.getOpcao() != 0); // A opção 0 é usada para sair  
    try {  
        this.logNegocio.guardaEstado("estado.obj");  
    }  
    catch (IOException e) {  
        System.out.println("Ops! Não consegui gravar os dados!");  
    }  
    System.out.println("Até breve!...");  
}
```

- A View:

```
public class Menu {  
    // variáveis de instância  
    private List<String> opcoes;  
    private int op;  
  
    /**  
     * Constructor for objects of class Menu  
     */  
    public Menu(String[ ] opcoes) {  
        this.opcoes = Arrays.asList(opcoes);  
        this.op = 0;  
    }  
}
```

- continuação...

```
/**  
 * Método para apresentar o menu e ler uma opção.  
 */  
  
public void executa() {  
    do {  
        showMenu();  
        this.op = lerOpcao();  
    } while (this.op == -1);  
}  
  
/** Apresentar o menu */  
private void showMenu() {  
    System.out.println("\n *** Menu *** ");  
    for (int i=0; i<this.opcoes.size(); i++) {  
        System.out.print(i+1);  
        System.out.print(" - ");  
        System.out.println(this.opcoes.get(i));  
    }  
    System.out.println("0 - Sair");  
}
```