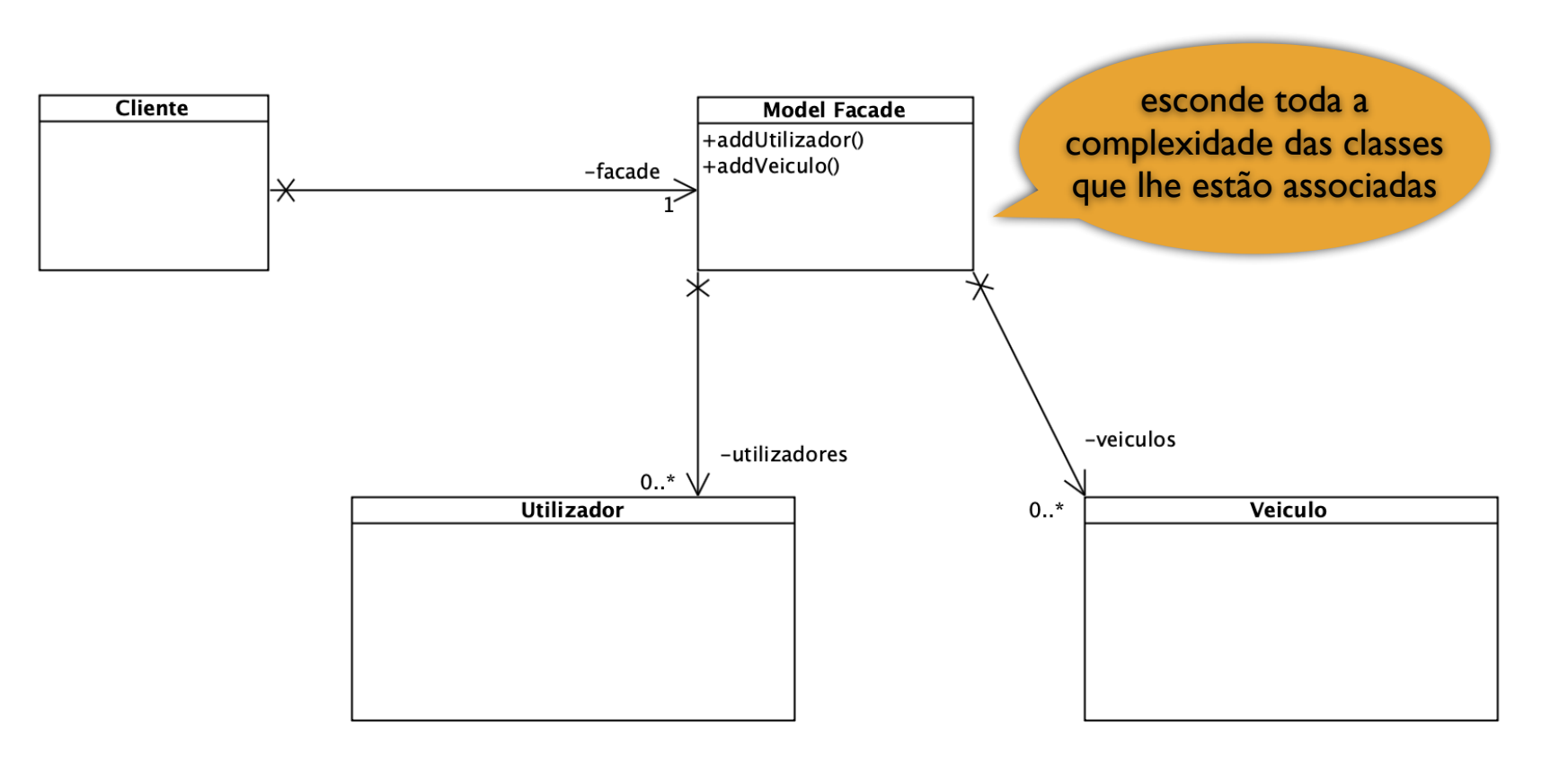
O padrão Facade

- 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).



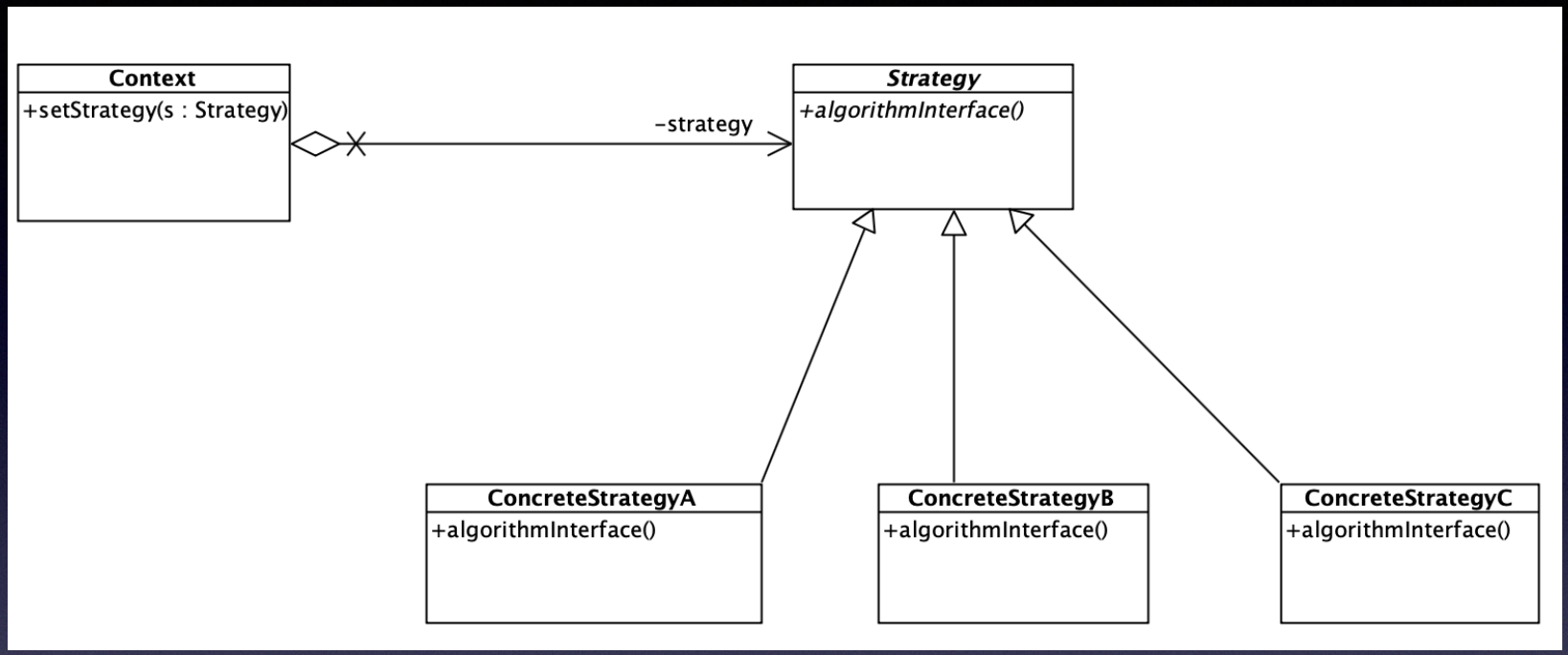
O padrão Strategy

- 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.

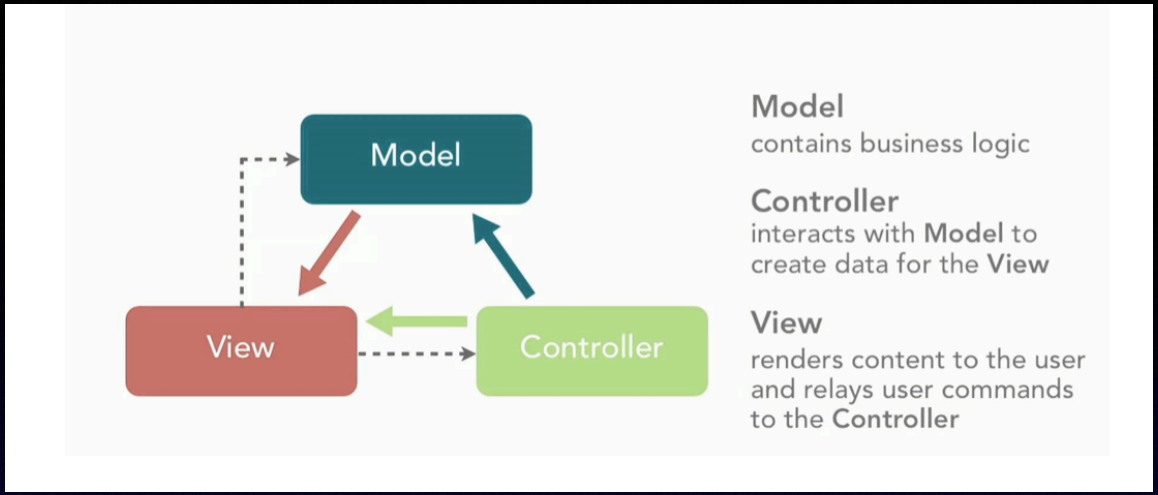
- A `View` deve ter preocupações com:
  - usabilidade
  - ser visualmente agradável, poder mudar layout, etc.
- O `Model` deve preocupar-se em ser:
  - eficiente
  - modular, reutilizável, etc.

- 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 indicia 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;
    private Bank bank;
    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 manipula directamente o Model

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 {
   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);
     }
   }
   …
}
```

o controlador

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

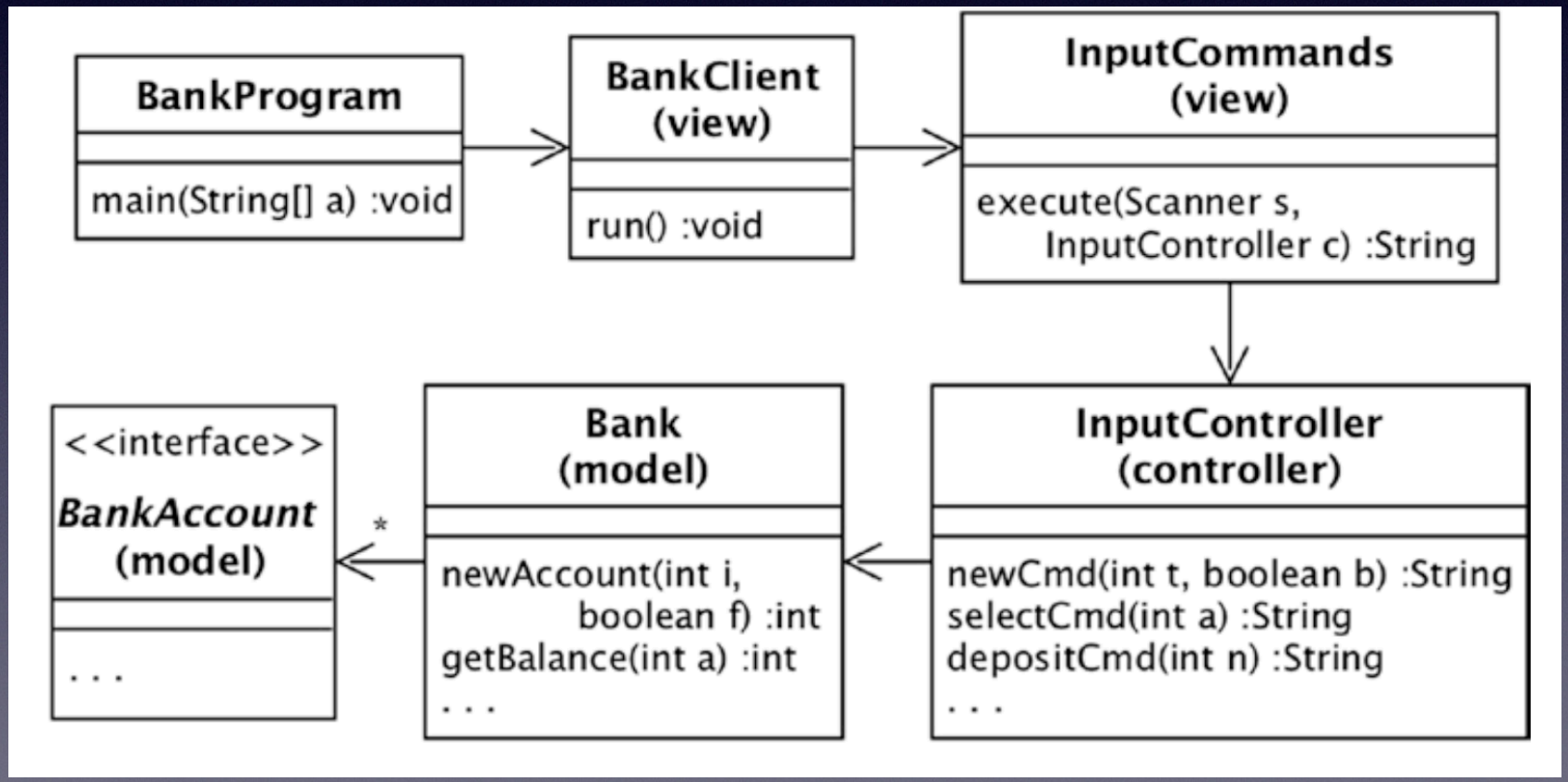
- O programa principal é agora o responsável pela criação das várias camadas e pela interligação das mesmas:
  - deve passar para o `Controller` a referência do `Model`
  - deve fornecer à `View` a referência do `Controller`

- Excerto 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

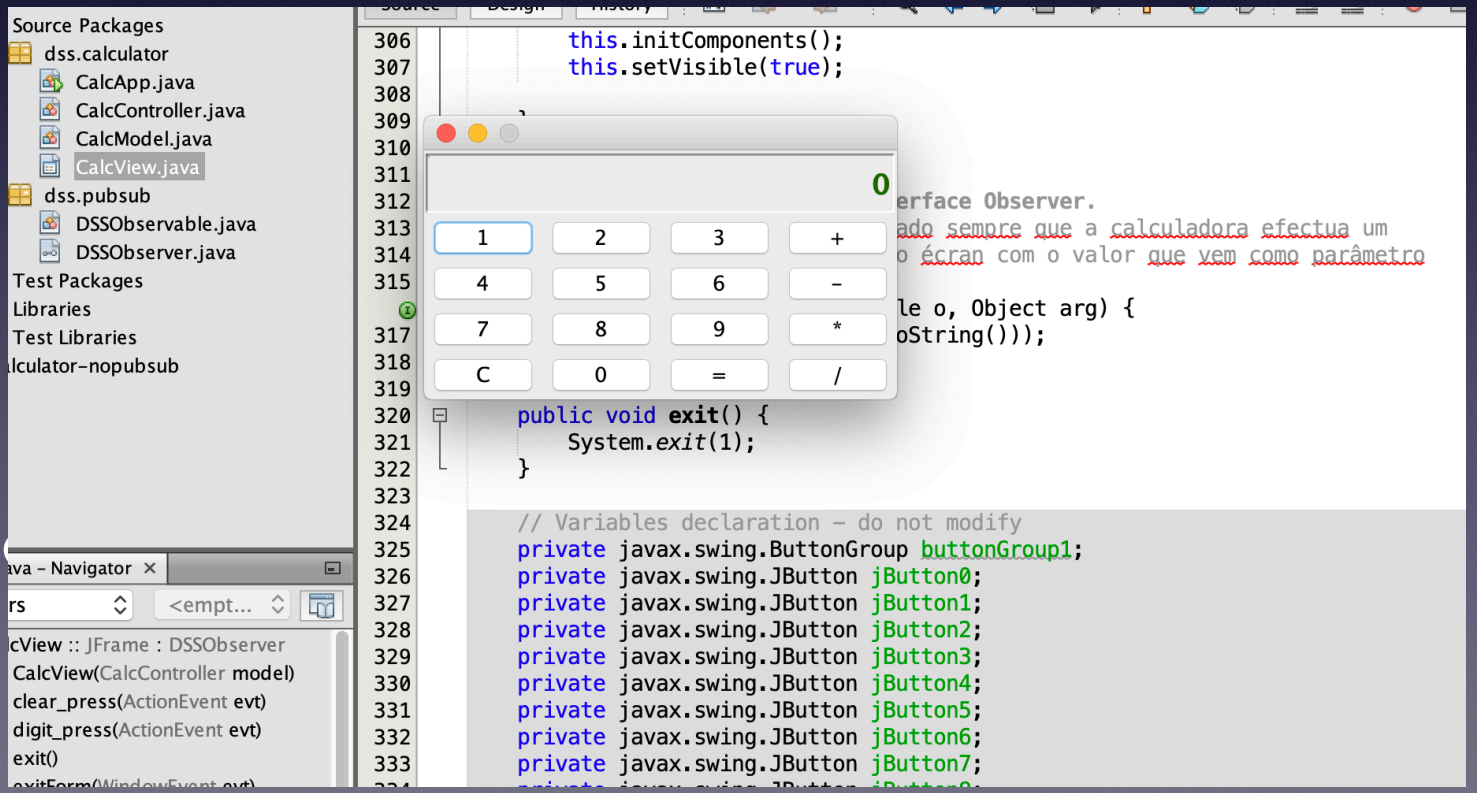
- Do ponto de vista arquitectural, temos o seguinte diagrama:



(*) 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 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;
    }

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

    public void reset() {
        this.value = 0;
        this.notifyObservers(""+value);
    }
```

- 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

- 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();
```