# Tratamento de Erros

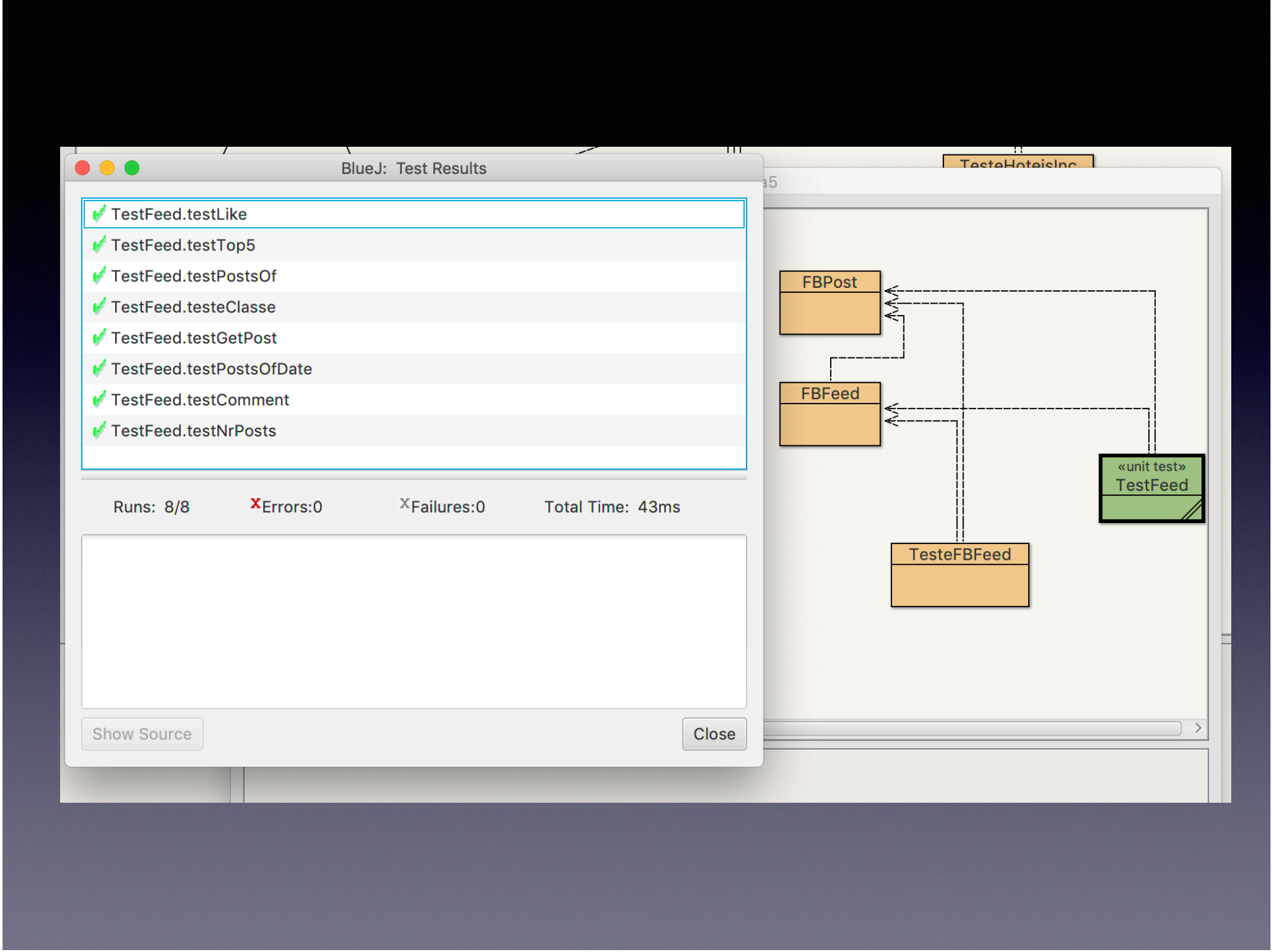
- Nesta altura já se apresentou uma forma de efectuar testes unitários, isto é fazer programas que fazem asserções sobre o comportamento exibido pelos programas em determinadas situações.
- Esses testes podem ser executados sempre que se altera o código como medida preventiva de detecção de erros antes de passarmos o componente (classe, módulo, etc.) para terceiros.

- Criar o teste, construir os objectos…

```
public TestFeed() {

    FBPost post0 = new FBPost(0, "User 1", LocalDateTime.of(2018,3,10,10,30,0), "Teste 1", 0, new ArrayList<
    FBPost post1 = new FBPost(1, "User 1", LocalDateTime.of(2018,3,12,15,20,0), "Teste 2", 0, new ArrayList<
    FBPost post2 = new FBPost(2, "User 2", LocalDateTime.now(), "Teste 3", 0, new ArrayList<>());
    FBPost post3 = new FBPost(3, "User 3", LocalDateTime.now(), "Teste 4", 0, new ArrayList<>());
    FBPost post4 = new FBPost(4, "User 4", LocalDateTime.now(), "Teste 5", 0, new ArrayList<>());

    List<FBPost> tp = new ArrayList<>();
    tp.add(post0);
    tp.add(post1);
    tp.add(post2);
    tp.add(post3);
    tp.add(post4);
    //tp.add(post5);
    feed.setPosts(tp);
}
```

```java
@Test
public void testNrPosts() {
    int np = feed.nrPosts("User 1");
    assertEquals(np,2);
    //assertTrue(np == 2);
}

@Test
public void testPostsOf() {
    List<FBPost> posts = feed.postsOf("User 2");
    assertNotNull(posts);
    assertEquals(posts.size(),1);
    FBPost p = feed.postsOf("User 2").get(0);
    assertNotNull(p);
    assertEquals("User 2",p.getUsername());
}
```

```java
@Test
public void testGetPost() {
    FBPost p = feed.getPost(3);
    assertEquals(p.getUsername(),"User 3");
}

@Test
public void testComment() {
    FBPost p = feed.getPost(3);
    feed.comment(p, "Primeiro comentario");
    assertTrue(p.getComentarios().size() == 1);
    assertEquals(p.getComentarios().get(0), "Primeiro comentario");
}
```


BlueJ: Test Results
TestFeed.testLike
TestFeed.testTop5
TestFeed.testPostsOf
TestFeed.testeClasse
TestFeed.testGetPost
TestFeed.testPostsOfDate
TestFeed.testComment
TestFeed.testNrPosts
Runs: 8/8
Errors:0
Failures:0
Total Time: 43ms
Show Source
Close
FBPost
FBFeed
«unit test»
TestFeed
TesteFBFeed

- Existem outro tipo de testes que se designam por testes de integração, onde é suposto fazermos uma avaliação qualitativa da orquestração dos vários objectos no meu programa.
  - poderá acontecer que uma classe não apresente problemas quando testada de forma unitária, mas ao ser integrada numa outra classe apresente problemas.

- Podemos olhar para cada um dos métodos que são oferecidos pelas classes (e pelas interfaces) como sendo contratos. E neles poder especificar:
  - as condições em que podem ser invocados
  - o que realizam (o algoritmo)
  - o que acontece depois de serem executados, isto é, o que aconteceu ao estado

- Podemos determinar asserções sobre:
  - as pré-condições, o que tem de ser garantido para que o método possa ser executado
  - as pós-condições, a validação das alterações ao estado em caso de sucesso da execução correcta do método

- Nem sempre é possível executar um método. Por exemplo:
  - criar um círculo de raio negativo
  - levantar dinheiro de uma conta sem saldo suficiente
  - efectuar uma viagem com distância superior à autonomia do veículo

- Nessas circunstâncias, o método deve enviar um sinal de erro.
  - numa lógica diferente dos erros do C
  - que obriguem o erro a ser verificado
  - que se possam efectuar operações de gestão do erro (acções de recuperação).

- (das fichas…) Temos escrito código e comentado situações em que o comportamento pode não ser o esperado.

e se V não existe?

classe que implementa Map.Entry

```
import static java.util.AbstractMap.SimpleEntry;
import static java.util.Map.Entry;
...
  // Dá erro se vértice não existe
  Set<Entry<String, String>> fanOut (String v) {
     Set<Entry<String, String>> res = new HashSet<>();   // SimpleEntry não é Comparable!

     for (String vout: this.adj.get(v)) {
        res.add(new SimpleEntry<>(v, vout));
     }
     return res;
  }

  Set<Entry<String, String>> fanIn(String v) {
     Set<Map.Entry<String, String>> res = new HashSet<>(); // SimpleEntry não é Comparable!

     for (Entry<String,Set<String>> e: this.adj.entrySet()) {
        if (e.getValue().contains(v)) {
           res.add(new SimpleEntry<>(e.getKey(), v));
        }
     }
     return res;
  }
```

# Tratamento de Erros

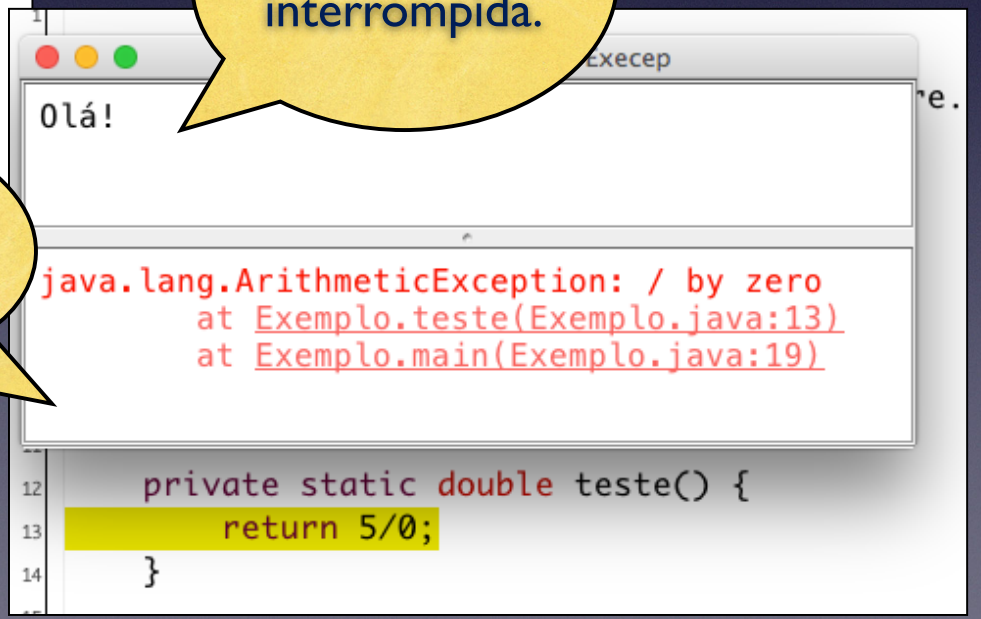
- Java usa a noção de *excepções* para realizar tratamento de erros
- Uma excepção é um *evento* que ocorre durante a execução do programa e que interrompe o fluxo normal de processamento
  - garante-se assim que o surgimento de um erro obriga o programador a criar código para o tratar. Em vez de apenas o ignorar…

# Excepções

```
10 public class Exemplo {
11
12     private static double teste() {
13         return 5/0;
14     }
15
16
17     public static void main(String[] args) {
18         System.out.println("Olá!");
19         System.out.println(teste());
20         System.out.println("Até logo!");
21
22     }
23 }
```

A execução é interrompida.

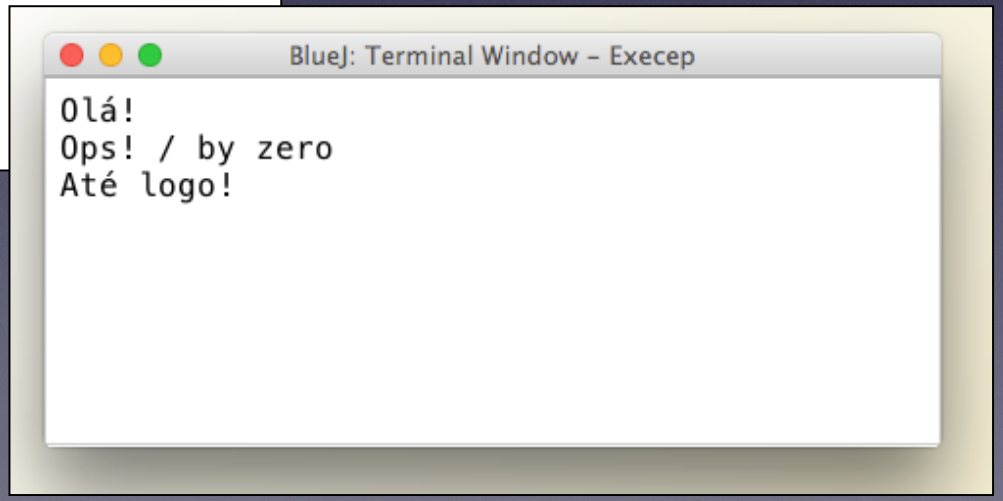
O erro é propagado para trás pela stack de invocações de métodos.



# *try* e *catch*

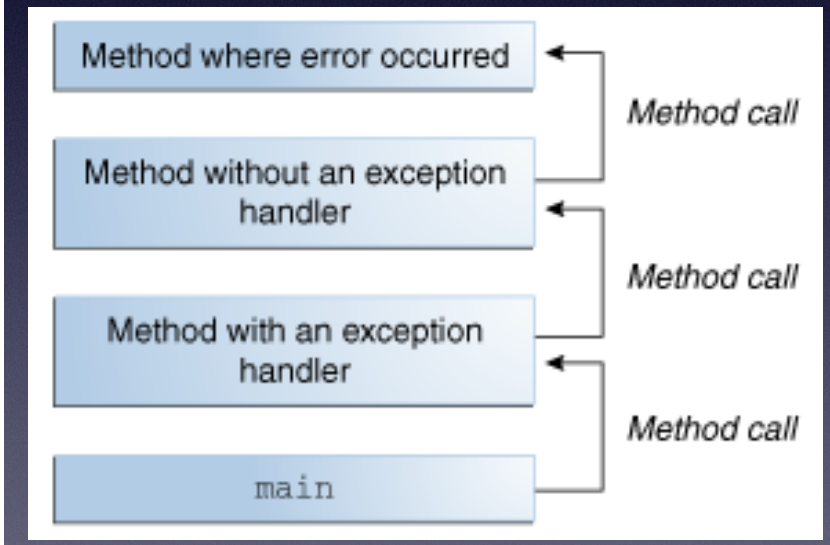
```
10 public class Exemplo {
11
12     private static double teste() {
13         return 5/0;
14     }
15
16
17     public static void main(String[] args) {
18         System.out.println("Olá!");
19         try {
20             System.out.println(teste());
21         }
22         catch (ArithmeticException e) {
23             System.out.println("Ops! "+e.getMessage());
24         }
25         System.out.println("Até logo!");
26     }
27 }
```

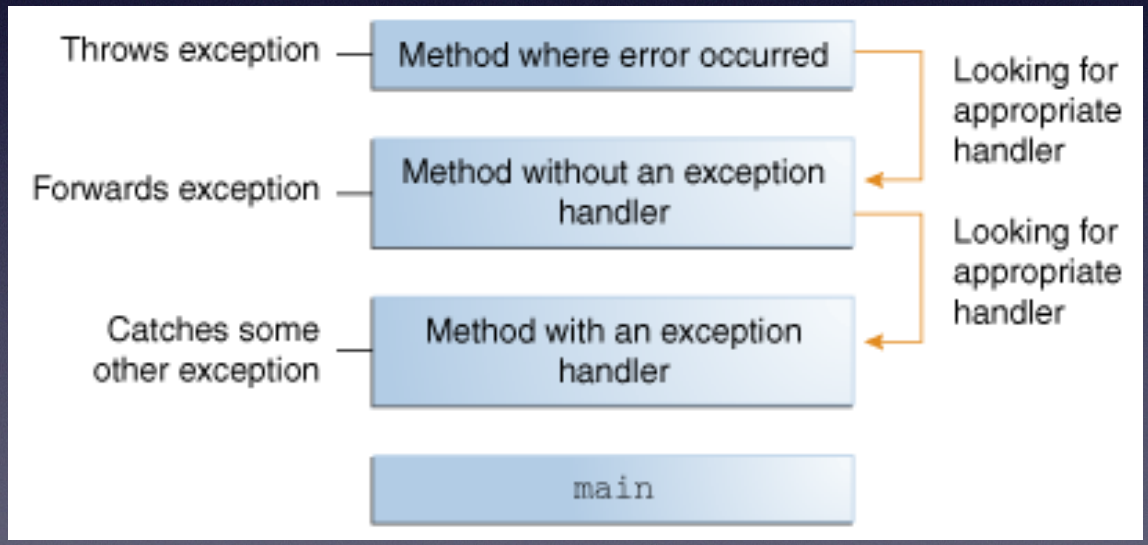
A execução retoma no **catch**.



# Excepções

- Modelo de funcionamento:



# Criar Excepções

```java
public class AlunoNaoExisteException extends Exception {
    public AlunoNaoExisteException(String msg) {
        super(msg);
    }
}
```

```java
/**
 * Obter o aluno da turma com número num
 *
 * @param num o número do aluno pretendido
 * @return uma cópia do aluno com o número n
 * @throws AlunoException
 */

public Aluno getAluno(int num) throws AlunoNaoExisteException {
    Aluno a = this.alunos.get(num);
    if (a == null)
      throw new AlunoNaoExisteException(""+num);
    return a.clone();
}
```

Obrigatório declarar que lança excepção.

Lança uma excepção.

```java
public static void main(String[] args) {

    // ...
    int num;
    do {
      op = lerOpcao();
      switch (op) {
          CONSULTAR:
            num = leNumero();
            try {
              a = turma.getAluno(num);
              System.out.println(a.toString());
            }
            catch (AlunoNaoExisteException e) {
              System.out.println("Oops, não existe  " + e.getMessage());
            }
            break;
          INSERIR:
            // ...
      }
    } while (op != SAIR);
}
```

Vai tentar um getAluno...

Apanha e trata a excepção.

# Tipos de Excepções

- Excepções de *runtime*
  - Condições excepcionais interna à aplicação - ou seja, erros nossos!!
  - **RuntimeException** e suas subclasses
  - Exemplo: **NullPointerException**
- Erros
  - Condições excepcionais externas à aplicação
  - **Error** e suas subclasses
  - Exemplo: **IOError**
- Checked Exceptions
  - Condições excepcionais que aplicações bem escritas deverão tratar
  - Obrigadas ao requisito *Catch or Specify*
  - Exemplo: **FileNotFoundException**

# Modelo de utilização das excepções

- Os métodos onde são detectadas as excepções devem sinalizar isso (`throws ...Exception`)
  - recomenda-se que para cada tipo de excepção se crie uma classe de Excepção
- métodos que invocam métodos que libertam excepções devem decidir se as tratam ou fazem passagem das mesmas (`throws ...Exception`)

- Se não for feito antes, o tratamento das excepções chega ao método main()
  - aí pode ser feita toda a gestão da comunicação com o utilizador (`out.println` ou outras)
  - métodos de outras classes, que não a classe de teste, não devem enviar informação de erro para o écran.

# Vantagens do uso de Excepções

- Separam código de tratamento de erros do código *regular*
- Propagação dos erros pela stack de invocações de métodos
- Junção e diferenciação de tipos de erros

# Exemplo
# Leitura/Escrita em ficheiros

- Gravar em modo texto:

```
/**
 * Método que guarda o estado de uma instância num ficheiro de texto.
 *
 * @param nome do ficheiro
 */

public void escreveEmFicheiroTxt(String nomeFicheiro) throws IOException {
    PrintWriter fich = new PrintWriter(nomeFicheiro);
    fich.println("------- HotéisInc --------");
    fich.println(this.toString()); // ou fich.println(this);
    fich.flush();
    fich.close();
 }
```

- Gravação modo binário:
  - obrigatório decidir que classes são persistidas através da implementação da interface `Serializable`
  - utilização de `java.io.ObjectOutputStream`

necessita de `implements Serializable`

```
/**
 * Método que guarda em ficheiro de objectos o objecto que recebe a mensagem.
 */

public void guardaEstado(String nomeFicheiro) throws FileNotFoundException,IOException {
    FileOutputStream fos = new FileOutputStream(nomeFicheiro);
    ObjectOutputStream oos = new ObjectOutputStream(fos);
    oos.writeObject(this); //guarda-se todo o objecto de uma só vez
    oos.flush();
    oos.close();
}
```

- Leitura em modo binário
  - utilização de `java.io.ObjectInputStream`

```
/**
 * Método que recupera uma instância de HoteisInc de um ficheiro de objectos.
 * Este método tem de ser um método de classe que devolva uma instância já
 * construída de HoteisInc.
 *
 * @param nome do ficheiro onde está guardado um objecto do tipo HoteisInc
 * @return objecto HoteisInc inicializado
 */

public static HoteisInc carregaEstado(String nomeFicheiro) throws FileNotFoundException,
                                        IOException,
                                        ClassNotFoundException {
    FileInputStream fis = new FileInputStream(nomeFicheiro);
    ObjectInputStream ois = new ObjectInputStream(fis);
    HoteisInc h = (HoteisInc) ois.readObject();
    ois.close();
    return h;
}
```

# Utilização na classe de teste

```
//Gravar em ficheiro de texto

try {
   osHoteis.escreveEmFicheiroTxt("estadoHoteisTXT.txt");
}
catch (IOException e) {System.out.println("Erro a aceder a ficheiro!");}


//Gravar em ficheiro de objectos

try {
   osHoteis.guardaEstado("estadoHoteis.obj");
}
catch (FileNotFoundException e) {
   System.out.println("Ficheiro não encontrado!");}
catch (IOException e) {
   System.out.println("Erro a aceder a ficheiro!");}
```

O erro acontece em HoteisInc. O tratamento do erro é feito na classe de teste!!

```
public static void main(String[] args)

    // carregar informação
    do {
        menumain.executa();
        switch (menumain.getOpcao()) {
            case 1  // invocar método 1
                    break;
            case 2  // invocar método 2
                    break;
            case 3  // invocar método 3
                    break;
            case 4  // invocar método 4
                    break;
            case 5  // invocar método 5
                    break;
            case 6  // invocar método 6
        }
    } while (menumain.getOpcao()!=0);
    try {
        tab.gravaObj("estado.tabemp");
        tab.log("log.txt", true);
    }
    catch (IOException e) {
        System.out.println("Não consegui gravar os dados!");
    }
    System.out.println("Até breve!...");
}
```

Carregar dados no início (erros são tratados dentro do método de carregamento).

Gravar dados (e log) no fim (erros são tratados aqui).

# A abordagem do nio

- As classes apresentadas atrás permitem, de forma simples, ter uma estratégia de utilização das inúmeras streams para persistência de informação
  - para gravar em texto: `PrintWriter`
  - para gravar em modo binário: `ObjectOutputStream`
  - para ler em modo binário: `ObjectInputStream`

# A classe Files

- Na classe Files (`nio.File.Files`) encontram-se muitos métodos disponíveis para operações sobre ficheiros (e gestão do sistema de ficheiros)
- Possui métodos de âmbito mais geral sobre ficheiros, possibilitando operações de mais alto nível, quer na leitura quer no acesso à informação.

# A classe Files

- Exemplo disso é a utilização de `lines(Path p)` ou de `readAllLines(Path p)` para leitura em bulk de dados de um ficheiro de texto.
  - a estratégia é depois utilizar-se um mecanismo de parsing das `String` obtidas para encontrar a informação pretendida

- O método `readAllLines` devolve uma `List<String>`
- Em `Path` deve ser passado o caminho para o ficheiro
- 

```
public List<String> lerFicheiro(String nomeFich) {
      List<String> lines = new ArrayList<>();
      try { lines = Files.readAllLines(Paths.get(nomeFich), StandardCharsets.UTF_8); }
      catch(IOException exc) { System.out.println(exc.getMessage()); }
      return lines;
}
```

- Obtêm-se uma `List<String>` com o resultado das várias linhas do ficheiro (que tem de ter linefeed) e depois interpreta-se cada linha sabendo qual o separador (“:”)

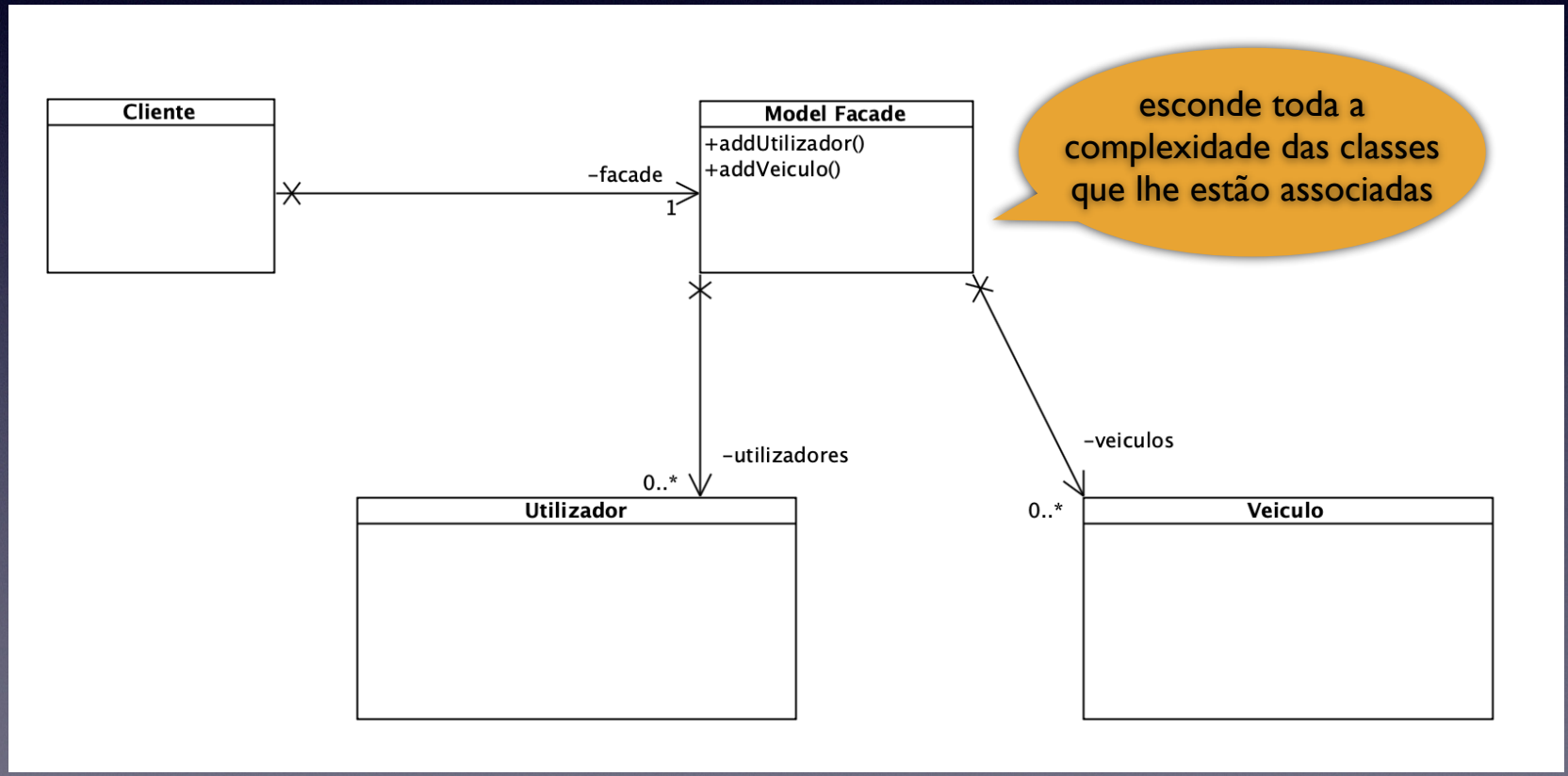
Exemplo de um ficheiro de texto com elementos do exercício da Casa Inteligente

```
public static void parse() throws LinhaIncorretaException {
    List<String> linhas = lerFicheiro("C:\\Path\\To\\file.csv");
    Map<String, Fornecedor> fornecedores = new HashMap<>();
    Map<Integer, CasaInteligente> casas = new HashMap<>();
    String[] linhaPartida;
    CasaInteligente c = null; SmartDevice sd = null;
    String divisao = null;
    for (String linha : linhas) {
        linhaPartida = linha.split(":", 2);
        switch(linhaPartida[0]){
            case "Fornecedor":
                Fornecedor f = Fornecedor.parse(linhaPartida[1]);
                fornecedores.put(f.getFornecedor(), f);
                break;
            case "Casa":
                c = CasaInteligente.parse(linhaPartida[1]);
                casas.put(c.getNif(), c);
                divisao = null;
                break;
            case "Divisao":
                if (c == null) throw new LinhaIncorretaException();
                divisao = linhaPartida[1];
                c.addRoom(divisao);
                break;
            case "SmartBulb":
                if (divisao == null) throw new LinhaIncorretaException();
                sd = SmartBulb.parse(linhaPartida[1]);
                c.addDevice(sd);
                c.addToRoom(divisao, sd.getId());
                break;
            case "SmartCamera":
                if (divisao == null) throw new LinhaIncorretaException();
                sd = SmartCamera.parse(linhaPartida[1]);
                c.addDevice(sd);
                c.addToRoom(divisao, sd.getId());
                break;
```

# 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).



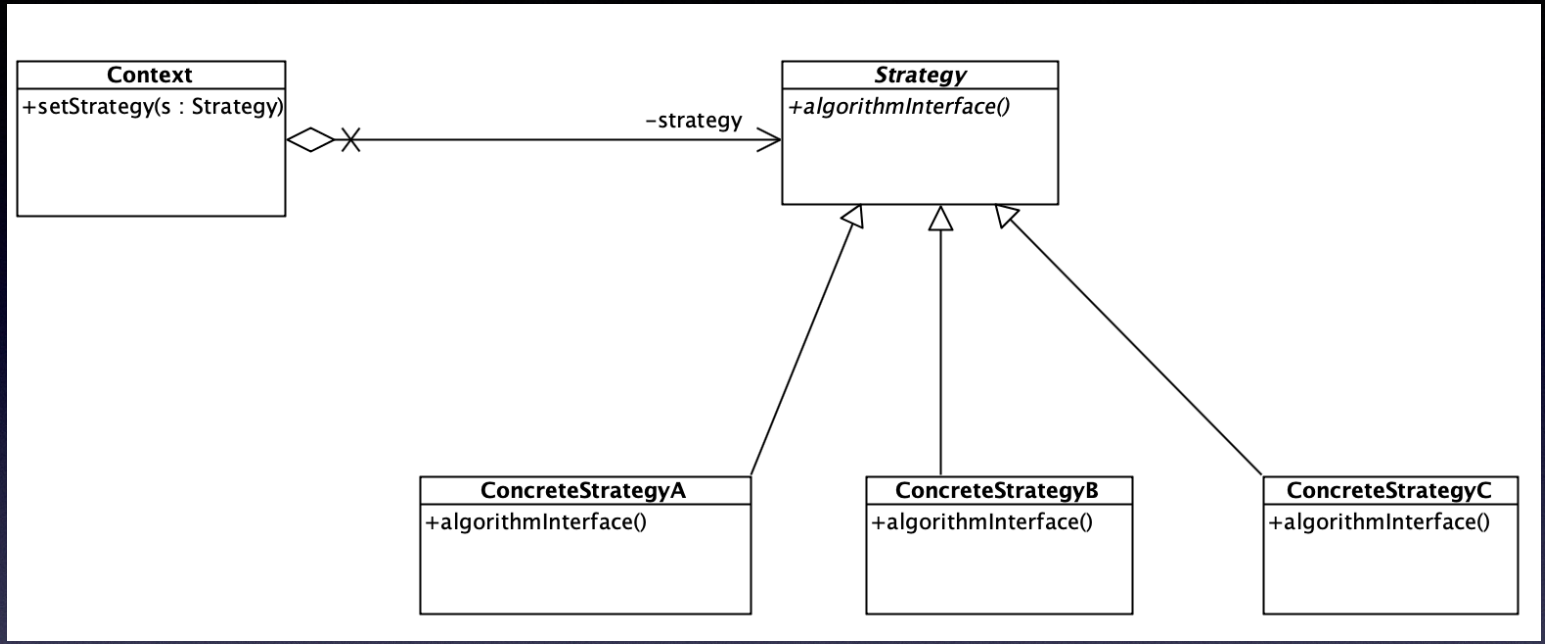
# O padrão Strategy

- 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/semelhantes 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.

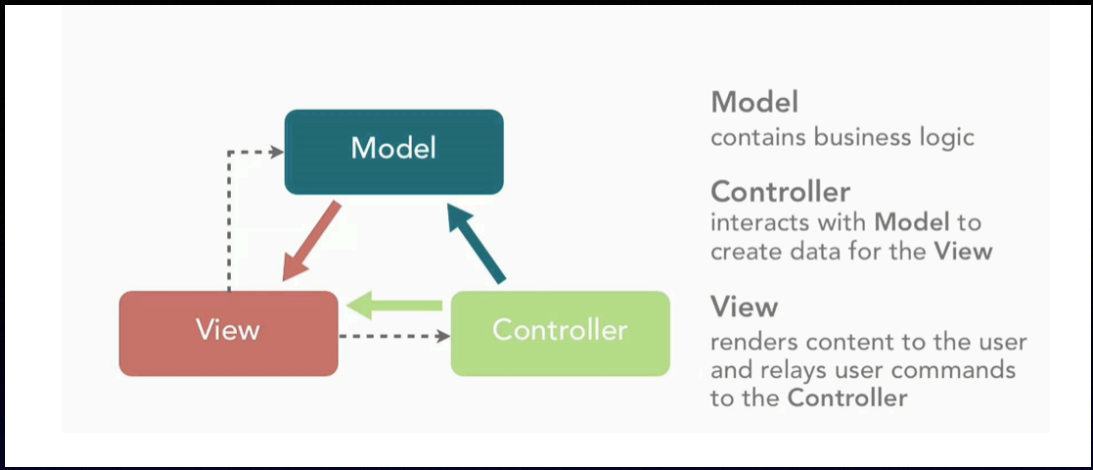
- A `View` deve ter preocupações com:
  - usabilidade
  - ser visualmente agradável, poder mudar layout, etc.
- O `Model` deve preocupar-se em ser:
  - eficiente
  - modular, reutilizável, etc.

- 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

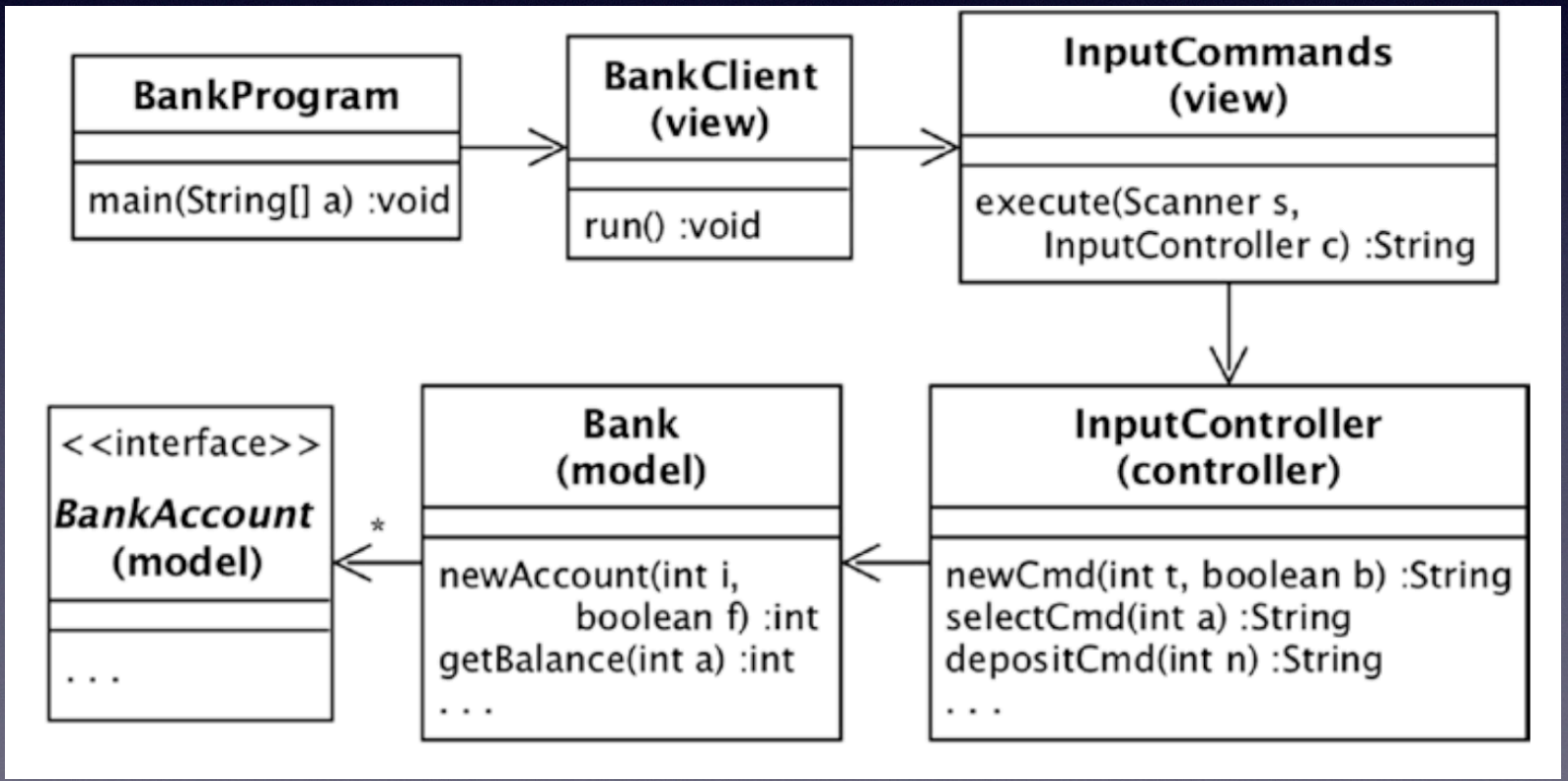
- O programa principal é agora o responsável pela criação das várias camadas e pela interligação das mesmas:
  - deve passar para o `Controller` a referência do `Model`
  - deve fornecer à `View` a referência do `Controller`

- 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

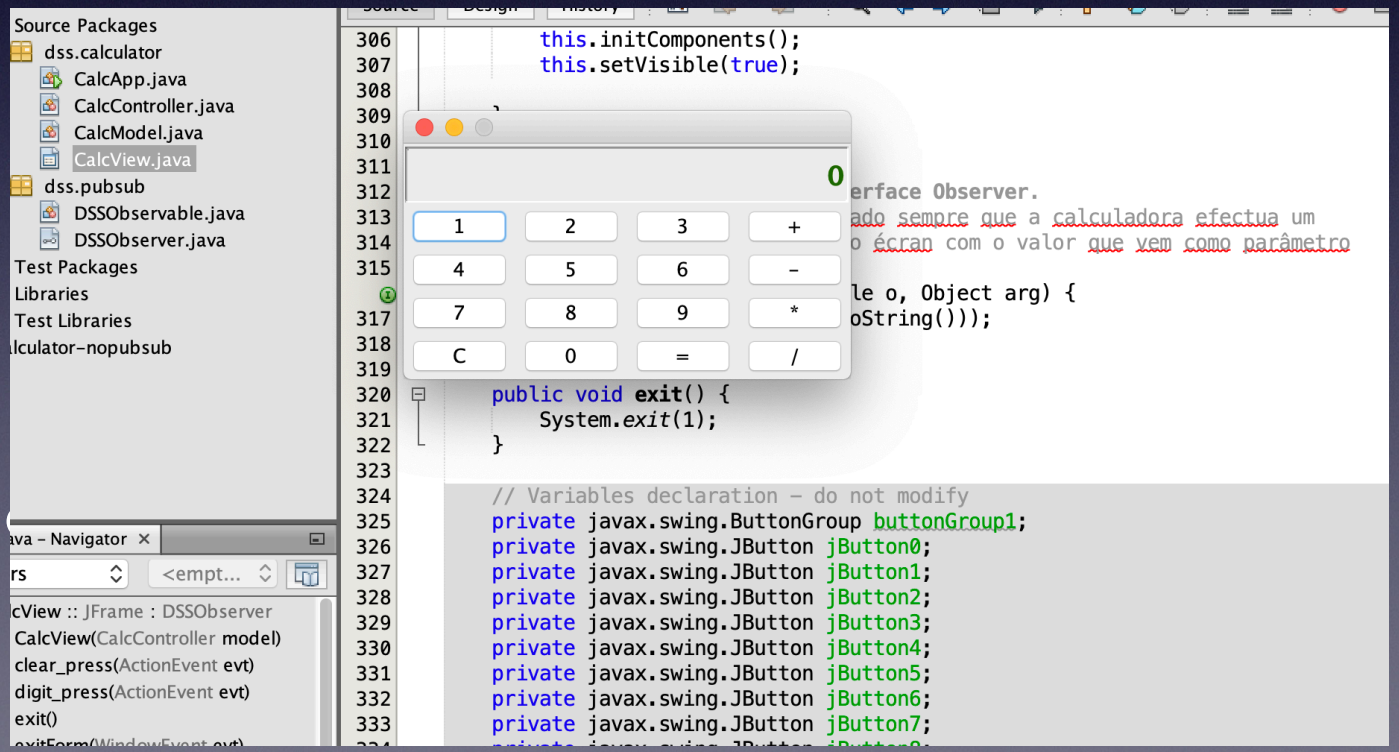
- Do ponto de vista arquitectural, temos o seguinte diagrama:




(*) 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

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

```
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;

    }
}
```

- 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

- View em modo Texto: 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() {

    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:

```java
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…

```java
/**
 * 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");
}
```

- ler uma opção - e tratar a excepção!

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

    System.out.print("Opção: ");
    try {
        op = is.nextInt();
    }
    catch (InputMismatchException e) {
        op = -1;
    }
    if (op<0 || op>this.opcoes.size()) {
        System.out.println("Opção Inválida!!!");
        op = -1;
    }
    return op;
}
```