



Exercicios do 3 capitulo Java para iniciantes

Giuliano Lemes Pereira – 2ano Big data e inteligência analítica.

1.

```
import java.util.Scanner;
public class exercicios98 {

    public static void main(String args[]) {
        char s = ' ';
        String texto = "";
        Scanner captura = new Scanner(System.in);
        captura.useDelimiter("");

        while(s != '.'){
            s = captura.next().charAt(0);
            texto += s;
        }
        long spaces = texto.chars().filter(c -> c == (int)' ').count();
        System.out.format("Foram digitados %d espaços.", spaces);
        captura.close();
    }
}
```

2.

```
if(condição) instrução;
else if(condição) instrução;
else if(condição) instrução;
... else
    instrução;
```

3. O if(y>100)

4.

```
for(int i = 1000; i >= 0; i-=2){.....
```

5.

Não é válido a variável “i” esta no contexto do “for”

6.

O break sai de uma estrutura de repetição, no java ele serve como um goto, que pode navegar estruturas.

7.

0 1

2 3

4 5

6 7

8 9

8.

```
for (int i = 1; i < 10000; i*=2) {
```

```

        System.out.print(i + " ");
        System.out.println();
    }

```

9.

```

public static void main(String args[]) {
    char s = ' ';
    Scanner captura = new Scanner(System.in);
    captura.useDelimiter("");
    int count = 0;
    while (s != '.') {
        System.out.println("Entre com uma letra");
        char c = captura.next().charAt(0);
        if(c == '.') break;
        if(c >= 'a' && c <= 'z') {
            c = (char)(c - 32);
        } else if(c >= 'A' && c <= 'Z') {
            c = (char)(c + 32);
        }
        count++;
        System.out.println(c);
    }

    System.out.format("Número de alterações: %d .", count);
    captura.close();
}

```

2. Laço infinito é uma estrutura de repetição que não tem fim, não foi estipulado um momento de saída da instrução.

11.

Não o goto pode ser especificado para qualquer bloco.

Os 4 pilares da Programação Orientada a Objetos

Conheça nesse artigo os 4 principais pilares, bem como as diferenças para programação estruturada e as principais vantagens da POO.

O [desenvolvimento de software](#) é extremamente amplo. Nesse mercado, existem diversas linguagens de programação, que seguem diferentes paradigmas. Um desses paradigmas é a Orientação a Objetos, que atualmente é o mais difundido entre todos. Isso acontece porque se trata de um padrão que tem evoluído muito, principalmente em questões voltadas para segurança e reaproveitamento de código, o que é muito importante no desenvolvimento de qualquer aplicação moderna.

A Programação Orientada a Objetos (POO) diz respeito a um padrão de desenvolvimento que é seguido por muitas linguagens, como C# e [Java](#). A seguir, iremos entender as diferenças entre a POO e a Programação Estruturada, que era muito utilizada há alguns anos, principalmente com a [linguagem C](#). Esse padrão se baseia em quatro pilares que veremos ao longo desse artigo. Além disso, a POO diversas vantagens em sua utilização, que também serão vistas e explicadas.

Saiba mais sobre [Orientação a Objetos](#)

Programação Estruturada vs Programação Orientada a Objetos

Na **Figura 1** vemos uma comparação muito clara entre a programação estruturada e a programação orientada a objetos no que diz respeito aos dados. Repare que, no paradigma estruturado, temos procedimentos (ou funções) que são aplicados globalmente em nossa aplicação. No caso da orientação a objetos, temos métodos que são aplicados aos dados de cada objeto. Essencialmente, os procedimentos e métodos são iguais, sendo diferenciados apenas pelo seu escopo.

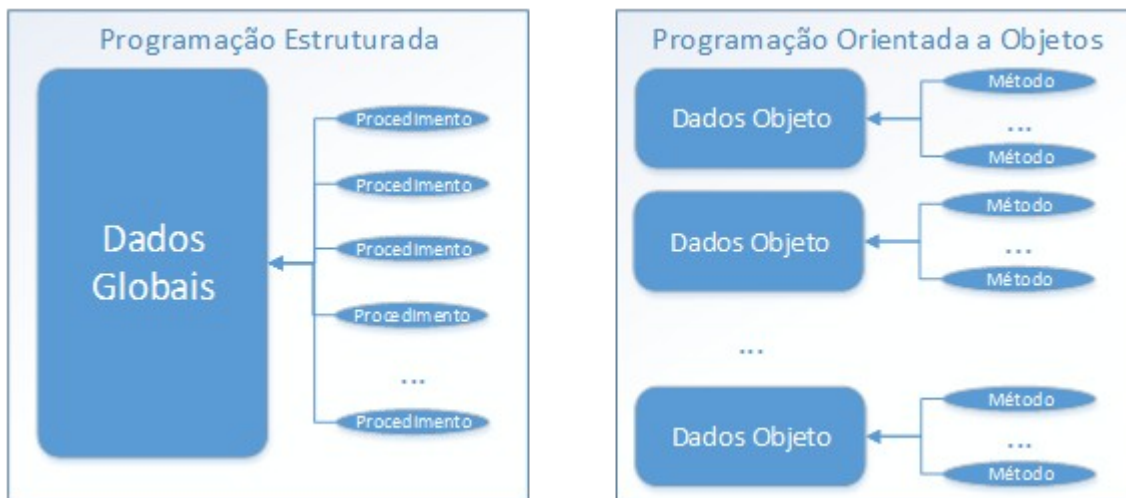


Figura 1. Estruturada x Orientação a Objetos

A linguagem C é a principal representante da programação estruturada. Se trata de uma linguagem considerada de baixo nível, que atualmente não é utilizada para projetos muito grandes. A sua principal utilização, devido ao baixo nível, é em programação para sistemas embarcados ou outros em que o conhecimento do hardware se faz necessário para um bom programa.

Essa colocação nos traz a um detalhe importante: a programação estruturada, quando bem feita, possui um desempenho superior ao que vemos na

programação orientada a objetos. Isso ocorre pelo fato de ser um paradigma sequencial, em que cada linha de código é executada após a outra, sem muitos desvios, como vemos na POO. Além disso, o paradigma estruturado costuma permitir mais liberdades com o hardware, o que acaba auxiliando na questão desempenho.

Entretanto, a programação orientada a objetos traz outros pontos que acabam sendo mais interessantes no contexto de aplicações modernas. Como o desempenho das aplicações não é uma das grandes preocupações na maioria das aplicações (devido ao poder de processamento dos computadores atuais), a programação orientada a objetos se tornou muito difundida. Essa difusão se dá muito pela questão da reutilização de código e pela capacidade de representação do sistema muito mais perto do que veríamos no mundo real.

Veremos em detalhes esses e outros pontos que dizem respeito a programação orientada a objetos. Como desenvolvedores, é nossa missão entender quais são as vantagens e desvantagens de cada um dos paradigmas de programação e escolhermos o melhor para nossa aplicação. A escolha da linguagem também deve estar presente nessa escolha.

DevCast: [Por que adotamos Orientação a Objetos?](#)

Os 4 pilares da Programação Orientada a Objetos

Para entendermos exatamente do que se trata a orientação a objetos, vamos entender quais são os requerimentos de uma linguagem para ser considerada nesse paradigma. Para isso, a linguagem precisa atender a quatro tópicos bastante importantes:

Abstração

A abstração consiste em um dos pontos mais importantes dentro de qualquer linguagem Orientada a Objetos. Como estamos lidando com uma representação de um objeto real (o que dá nome ao paradigma), temos que imaginar o que esse objeto irá realizar dentro de nosso sistema. São três pontos que devem ser levados em consideração nessa abstração.

O primeiro ponto é darmos uma identidade ao objeto que iremos criar. Essa identidade deve ser única dentro do sistema para que não haja conflito. Na maior parte das linguagens, há o conceito de pacotes (ou namespaces). Nessas linguagens, a identidade do objeto não pode ser repetida dentro do

pacote, e não necessariamente no sistema inteiro. Nesses casos, a identidade real de cada objeto se dá por ..

A segunda parte diz respeito a características do objeto. Como sabemos, no mundo real qualquer objeto possui elementos que o definem. Dentro da programação orientada a objetos, essas características são nomeadas propriedades. Por exemplo, as propriedades de um objeto “Cachorro” poderiam ser “Tamanho”, “Raça” e “Idade”.

Por fim, a terceira parte é definirmos as ações que o objeto irá executar. Essas ações, ou eventos, são chamados métodos. Esses métodos podem ser extremamente variáveis, desde “Acender()” em um objeto lâmpada até “Latir()” em um objeto cachorro.

Saiba mais sobre [Abstração e Polimorfismo](#)

Encapsulamento

O encapsulamento é uma das principais técnicas que define a programação orientada a objetos. Se trata de um dos elementos que adicionam segurança à aplicação em uma programação orientada a objetos pelo fato de esconder as propriedades, criando uma espécie de caixa preta.

A maior parte das linguagens orientadas a objetos implementam o encapsulamento baseado em propriedades privadas, ligadas a métodos especiais chamados getters e setters, que irão retornar e setar o valor da propriedade, respectivamente. Essa atitude evita o acesso direto a propriedade do objeto, adicionando uma outra camada de segurança à aplicação.

Para fazermos um paralelo com o que vemos no mundo real, temos o encapsulamento em outros elementos. Por exemplo, quando clicamos no botão ligar da televisão, não sabemos o que está acontecendo internamente. Podemos então dizer que os métodos que ligam a televisão estão encapsulados.

Saiba mais sobre [Encapsulamento em Java](#)

Herança

O reuso de código é uma das grandes vantagens da programação orientada a objetos. Muito disso se dá por uma questão que é conhecida como herança. Essa característica otimiza a produção da aplicação em tempo e linhas de código.

Para entendermos essa característica, vamos imaginar uma família: a criança, por exemplo, está herdando características de seus pais. Os pais, por sua vez, herdam algo dos avós, o que faz com que a criança também o faça, e assim sucessivamente. Na orientação a objetos, a questão é exatamente assim, como mostra a **Figura 2**. O objeto abaixo na hierarquia irá herdar características de todos os objetos acima dele, seus “ancestrais”. A herança a partir das características do objeto mais acima é considerada herança direta, enquanto as demais são consideradas heranças indiretas. Por exemplo, na família, a criança herda diretamente do pai e indiretamente do avô e do bisavô.

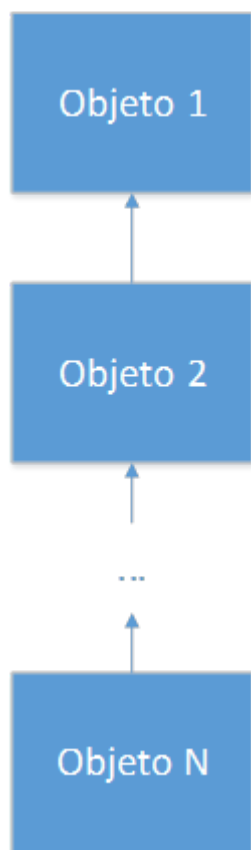


Figura 2. Herança na orientação a objetos

A questão da herança varia bastante de linguagem para linguagem. Em algumas delas, como C++, há a questão da herança múltipla. Isso, essencialmente, significa que o objeto pode herdar características de vários “ancestrais” ao mesmo tempo diretamente. Em outras palavras, cada objeto pode possuir quantos pais for necessário. Devido a problemas, essa prática não foi difundida em linguagens mais modernas, que utilizam outras artimanhas para criar uma espécie de herança múltipla.

Outras linguagens orientadas a objetos, como C#, trazem um objeto base para todos os demais. A classe `object` fornece características para todos os objetos em C#, sejam criados pelo usuário ou não.

Polimorfismo

Outro ponto essencial na programação orientada a objetos é o chamado polimorfismo. Na natureza, vemos animais que são capazes de alterar sua forma conforme a necessidade, e é dessa ideia que vem o polimorfismo na orientação a objetos. Como sabemos, os objetos filhos herdam as características e ações de seus “ancestrais”. Entretanto, em alguns casos, é necessário que as ações para um mesmo método seja diferente. Em outras palavras, o polimorfismo consiste na alteração do funcionamento interno de um método herdado de um objeto pai.

Como um exemplo, temos um objeto genérico “Eletrodoméstico”. Esse objeto possui um método, ou ação, “`Ligar()`”. Temos dois objetos, “Televisão” e “Geladeira”, que não irão ser ligados da mesma forma. Assim, precisamos, para cada uma das classes filhas, reescrever o método “`Ligar()`”.

Com relação ao polimorfismo, valem algumas observações. Como se trata de um assunto que está intimamente conectado à herança, entender os dois juntamente é uma boa ideia. Outro ponto é o fato de que as linguagens de programação implementam o polimorfismo de maneiras diferentes. O C#, por exemplo, faz uso de métodos virtuais (com a palavra-chave `virtual`) que podem ser reimplementados (com a palavra-chave `override`) nas classes filhas. Já em Java, apenas o atributo “`@Override`” é necessário.

Esses quatro pilares são essenciais no entendimento de qualquer linguagem orientada a objetos e da orientação a objetos como um todo. Cada linguagem irá implementar esses pilares de uma forma, mas essencialmente é a mesma coisa. Apenas a questão da herança, como comentado, que pode trazer variações mais bruscas, como a presença de herança múltipla. Além disso, o encapsulamento também é feito de maneiras distintas nas diversas linguagens, embora os getters e setters sejam praticamente onipresentes.

Saiba mais sobre [Polimorfismo em Java](#)

Principais vantagens da POO

A programação orientada a objetos traz uma ideia muito interessante: a representação de cada elemento em termos de um objeto, ou classe. Esse tipo de representação procura aproximar o sistema que está sendo criado ao que é observado no mundo real, e um objeto contém características e ações, assim como vemos na realidade. Esse tipo de representação traz algumas vantagens muito interessantes para os desenvolvedores e também para o usuário da aplicação. Veremos algumas delas a seguir.

A reutilização de código é um dos principais requisitos no desenvolvimento de software atual. Com a complexidade dos sistemas cada vez maior, o tempo de desenvolvimento iria aumentar exponencialmente caso não fosse possível a reutilização. A orientação a objetos permite que haja uma reutilização do código criado, diminuindo o tempo de desenvolvimento, bem como o número de linhas de código. Isso é possível devido ao fato de que as linguagens de programação orientada a objetos trazem representações muito claras de cada um dos elementos, e esses elementos normalmente não são interdependentes. Essa independência entre as partes do software é o que permite que esse código seja reutilizado em outros sistemas no futuro.

Outra grande vantagem que o desenvolvimento orientado a objetos traz diz respeito a leitura e manutenção de código. Como a representação do sistema se aproxima muito do que vemos na vida real, o entendimento do sistema como um todo e de cada parte individualmente fica muito mais simples. Isso permite que a equipe de desenvolvimento não fique dependente de uma pessoa apenas, como acontecia com frequência em linguagens estruturadas como o C, por exemplo.

A criação de bibliotecas é outro ponto que é muito mais simples com a orientação a objetos. No caso das linguagens estruturadas, como o C, temos que as bibliotecas são coleções de procedimentos (ou funções) que podem ser reutilizadas. No caso da POO, entretanto, as bibliotecas trazem representações de classes, que são muito mais claras para permitirem a reutilização.

Entretanto, nem tudo é perfeição na programação orientada a objetos. A execução de uma aplicação orientada a objetos é mais lenta do que o que vemos na programação estruturada, por exemplo. Isso acontece devido à complexidade do modelo, que traz representações na forma de classes. Essas representações irão fazer com que a execução do programa tenha muitos

desvios, diferente da execução sequencial da programação estruturada. Esse é o grande motivo por trás da preferência pela linguagem C em hardware limitado, como sistemas embarcados. Também é o motivo pelo qual a programação para sistemas móveis como o Google Android, embora em Java (linguagem orientada a objetos), seja feita o menos orientada a objetos possível.

No momento atual em que estamos, tecnologicamente essa execução mais lenta não é sentida. Isso significa que, em termos de desenvolvimento de sistemas modernos, a programação orientada a objetos é a mais recomendada devido as vantagens que foram apresentadas. Essas vantagens são derivadas do modelo de programação, que busca uma representação baseada no que vemos no mundo real.

Saiba mais sobre [vantagens e desvantagens da Orientação a Objetos](#)

Exemplos de Linguagens Orientadas a Objetos

Há uma grande quantidade de linguagens de programação orientada a objetos no mercado atualmente. Nesse artigo, iremos apresentar 3 das mais utilizadas no momento: Java, C# e C++. Cada uma delas possui uma abordagem diferente do problema que as torna muito boas para alguns tipos de aplicações e não tão boas para outros.

Java

O Java é, muito provavelmente, a linguagem de programação mais utilizada no mercado atual. Auxiliado pela presença do JRE (Java Runtime Environment), ou variações dele, em quase todos os dispositivos eletrônicos do momento, a linguagem Java é um grande sucesso entre os desenvolvedores. O sucesso da linguagem aumentou ainda mais com o Google Android, que escolheu o Java como linguagem preferencial de desenvolvimento de aplicações.

O Java implementa os quatro pilares de forma bastante intuitiva, o que facilita o entendimento por parte do desenvolvedor. A abstração, o primeiro pilar, é implementado através de classes, que contém propriedades e métodos, de forma bastante simples. Já o encapsulamento é realizado através de propriedades privadas, auxiliadas por métodos especiais getters e setters, como mostra a **Listagem 1**. Vale ressaltar a palavra-chave “this” mostrada no método SetId(). Essa palavra-chave funciona como um representante da classe atual, uma auto-referência ao próprio objeto.

```
private int id;
public int GetId()
{
    return id;
}

public void SetId(int id)
{
    this.id = id;
}
```

Listagem 1. Encapsulamento em Java

As questões de herança e polimorfismo no Java são um pouco mais complexas. O Java possui herança simples, o que significa que cada classe pode herdar de apenas uma outra. Entretanto, o Java possui as chamadas Interfaces, que possuem propriedades e assinaturas de métodos. Essas interfaces precisam ser implementadas para funcionar, o que significa que uma classe pode implementar várias interfaces e herdar de apenas uma classe. Na questão de polimorfismo, o atributo `@Override` é responsável por informar ao Java que o método em questão está sendo reescrito.

C#

O C#, por sua vez, é outra das linguagens mais utilizadas no mercado. Como os computadores pessoais no mundo, em sua maioria, possuem o sistema operacional Windows, da Microsoft, o C# se popularizou. Isso porque o Windows implementa o Framework .NET, ao qual o C# está associado. O C# é uma linguagem de uso geral e especialmente criada para utilização com a orientação a objetos. Vale ressaltar que, em C#, tudo é um objeto (herda da classe `object`).

A abstração é muito simples, e segue o modelo do Java. A questão de encapsulamento é um pouco diferente devido a implementação dos métodos `getter` e `setter`. A nomenclatura também é um pouco diferente. A variável que realmente guarda o valor do dado é chamada atributo, enquanto a propriedade é o elemento que realmente acessa aquele dado do mundo externo. Isso está mostrado na **Listagem 2**. Além disso, o C# faz uso de duas palavras-chave especiais: `get` e `set`.

```
// Atributo
private int id;

// Propriedade
public int Id
{
    get;
    set;
}
```

Listagem 2. Encapsulamento em C#

A questão da herança em C# também segue o modelo do Java: herança simples e a possibilidade de utilização de interfaces. A importância das interfaces é muito grande, uma vez que elas podem dar o tipo dos dados, que somente posteriormente serão associados a um tipo real, como mostra a **Listagem 3**. Isso também é válido para o Java. Por padrão, as identidades das interfaces começam com a letra “I”. O polimorfismo, por sua vez, é baseado em métodos virtuais (com a palavra-chave virtual) na classe pai e reescritos com a palavra-chave override na classe filha.

```
IExemploInterface exemplo;
exemplo = new ImplementacaoIExemploInterface();
```

Listagem 3. Interfaces em C#

C++

O C++, por sua vez, é uma linguagem um pouco mais primitiva, e permite muito mais liberdades com o hardware. Como ele foi derivado imediatamente do C, o C++ permite a utilização de ponteiros, por exemplo, que irão trabalhar diretamente com a memória. Além disso, o C++ pode utilizar todas as bibliotecas C que existem diretamente.

Em termos de abstração, o C++ implementa classes, assim como qualquer linguagem orientada a objetos. Ele também possui o sentido de privado e público, que é utilizado para encapsulamento. Esse encapsulamento é realizado através de métodos getter e setter, muito similar ao visto em Java, como mostra a **Listagem 4**. Repare que a listagem mostra somente a assinatura dos métodos especiais, sendo que sua implementação é a mesma

que em Java. Esse tipo de adaptação é muito comum em C++, onde a classe é guardada em um arquivo .h e sua implementação em um arquivo .cpp.

```
private:
    int id;
public:
    int GetId() const;
    void SetId(int const id);
```

Listagem 4. Encapsulamento em C++

A questão da herança no C++ é um pouco diferente. A linguagem permite a herança múltipla, o que significa que cada classe pode herdar de quantas classes desejar. Isso pode causar problemas de métodos que possuem o mesmo nome, portanto o desenvolvedor precisa estar atento. O polimorfismo é baseado em métodos virtuais, da mesma forma como o C#. A complexidade, entretanto, é maior, uma vez que temos que cuidar de detalhes de mais baixo nível, como acesso a memória.

Além dessas exemplificadas, existem outras linguagens que merecem ser citadas. Entre elas, podemos elencar: [Python](#), linguagem de script orientada a objetos que é muito utilizada em pesquisas científicas devido a sua velocidade; Object Pascal (também conhecida como [Delphi](#), devido ao nome de sua IDE), apesar do grande número de sistemas mais antigos que a utilizam; [Objective-C](#), que é a linguagem de preferência para desenvolvimento de aplicações para os sistemas da Apple, como iPhone e iPad; Ruby, voltada para o desenvolvimento web; e Visual Basic [.NET](#), muito utilizada até pouco tempo, mas também caindo em desuso, principalmente devido ao avanço do C# em popularidade.

Ao longo desse artigo, procuramos elencar os elementos que fazem da programação orientada a objetos um sucesso no momento. Vimos os quatro pilares desse paradigma e entendemos como eles são implementados em algumas das linguagens mais utilizadas no mercado de desenvolvimento. Além disso, entendemos algumas das vantagens que tornaram a programação orientada a objetos um grande sucesso para o desenvolvimento de sistemas modernos.

A quem se destina

Gerentes de TI, desenvolvedores, analistas de sistemas, documentadores, auditores e outros relacionados à organização de empresas de T.I

Qual a razão de saber convenções de código

Inúmeras razões, algumas delas:

- 80% do tempo programadores, deveriam estar focados em regras de negócio, entendimento, estudos e outros 20% programando, sendo que ainda faltaria tempo para melhora do código;
- Dificilmente um código será mantido "eternamente" pelo seu criador original;
- Códigos bem escritos, bem descritivos, bem anotados aumentam a produtividade, diminuem a quantidade de treinamentos, facilita a leitura e agrada visualmente;
- Se você vende seu código como produto, você deve ter certeza que ele é um produto bem testado, empacotado para ser entregue e que faz o que promete, entregando valor ao seu cliente;
- Convenções padronizam métodos de usabilidade com boas práticas baseado no conhecimento e experiência de um corpo especializado na área;

Para que as convenções funcionem corretamente no seu ambiente de trabalho é ideal que todos sigam as conformidades dos padrões. Todos, isso deve ser levado como uma regra.

Comentários

```
/*
 * Classname                (nome da classe Java)
 *
 * Version information      (versionamento)
 *
 * Date                    (Data e Hora)
 *
 * author                  (autor(res) da criação)
 * Copyright notice        (informações do método, pra que serve, idéia principal)
 */
```

Package e Import Statements

Primeiro os pacote e depois os imports em seguida

```
package java.awt;
```

```
import java.awt.peer.CanvasPeer;
```

Note que o primeiro, pacote é um componente único, escrito em minúsculas e claro que você poderá fazer o uso da *(todos) em casos específicos, mas o ideal é que seja único diminuindo assim a gama de bibliotecas necessárias para futuros Deploys (implantações). Atualmente com, edu, gov, mil, net, org, ou uso em inglês de duas letras que identificam os códigos de países, conforme especificado na norma ISO 3166, 1981.

Composição de uma classe e Interfaces

Part of Class/Interface Declaration

Notas

1	Class/interface documentation comment (<code>/**...*/</code>)	
2	class or interface statement	
3	Class/interface documentation comment (<code>/**...*/</code>)	Este comentário deve conter todas as informações da classe ou interface de forma ampla visando sua documentação.
4	Class (static) variables	Primeiro as variáveis de classe <code>publics</code> , em seguida, os <code>protecteds</code> , o nível de pacote, em seguida, (sem modificador de acesso), e então as <code>privates</code> .
5	Instance variables	Idem item 4
6	Constructors	
7	Methods	Aqui os métodos devem ser agrupados pelo seu nível de funcionalidade.

Alinhamento do código

Inicie com 4 espaços ou um tab. A construção exata do recuo (espaços vs tabs) é indeterminado. Tabs deve ser definido exatamente cada 8 espaços (não 4).

Evite linhas com mais de 80 caracteres, uma vez que em alguns terminais fica mais difícil de ler.

Nota: os exemplos para utilização em documentação deve ter um menor comprimento de linha, geralmente não mais do que 70 caracteres.

Quando uma expressão não couber em uma única linha, quebrá-lo de acordo com estes princípios gerais:

- Quebre após uma vírgula.
- Quebre antes de um operador.
- Prefira quebras de nível superior para diminuir o nível-breaks.
- Se as regras acima expostas levarem a um código confuso ou ao código que está esmagado contra a margem direita, apenas coloque 8 espaços em seu lugar.

Aqui estão alguns exemplos de quebra de chamadas de método:

```
someMethod(longExpression1, longExpression2, longExpression3,
           longExpression4, longExpression5);
```

```
var = someMethod1(longExpression1,
                  someMethod2(longExpression2,
                              longExpression3));
```

Exemplo com códigos usados em métodos aritméticos:

```
longName1 = longName2 * (longName3 + longName4 - longName5)
              + 4 * longname6; // PREFER
```

```
longName1 = longName2 * (longName3 + longName4
                        - longName5) + 4 * longname6; // AVOID
```

Exemplo de códigos alinhando métodos. o Primeiro de forma convencional e o segundo ele mudaria as segunda e terceira linhas para a extrema direita se utilizado recuo convencional, então em vez disso, recua apenas 8 espaços

```
//DON'T USE THIS INDENTATION
if ((condition1 && condition2)
```

Exemplo de 3 formas aceitáveis de métodos ternários

```
alpha = (aLongBooleanExpression) ? beta : gamma;
```

```
alpha = (aLongBooleanExpression) ? beta
      : gamma;
```

```
alpha = (aLongBooleanExpression)
      ? beta
      : gamma;
```

Comentários

Programas em Java podem ter dois tipos de comentários: comentários de implementação do código e comentários de documentação. Observações de aplicação são os encontrados no mesmo modo C++, que são delimitados por `/* ... */`, e `/** ... */`. Comentários de documentação (conhecido como "comentários doc") são Java-only, e são delimitados por `/** ... */`. Comentários de documentação podem ser extraídos para arquivos HTML usando a ferramenta javadoc.

Comentários de implementação significa comentar o código ,ou parte dele ,ou para comentários sobre uma aplicação particular, um método de uma classe por exemplo.

Comentários de documentação são destinadas a descrever a especificação do código, a partir de uma perspectiva de implementação livre para ser lido por desenvolvedores que não necessariamente possuem o código fonte em mãos.

Os comentários devem ser usados para dar uma visão geral do código e fornecer informações adicionais que não está prontamente disponível no próprio código. Os comentários devem conter apenas informações relevantes para a leitura e compreensão do programa. Por exemplo, informações sobre como o pacote correspondente é construído ou em qual diretório ele reside não deve ser incluído como um comentário.

Discussão de decisões de projeto não triviais ou não óbvia é inapropriado, sempre evitar a duplicação de informação que está presente em (e claro) o código. Em geral, evite quaisquer comentários que possam sair do cronograma do código (quando usar datas, fixe-se somente a data de criação) como o código evolui.

Comentários nunca devem incluir caracteres especiais, nem deve ser usado em demasia, mas sempre com parcimônia.

Alguns exemplos de códigos com comentários:

Um comentário de bloco deve ser precedido por uma linha em branco para separá-la do resto do código.

```
/*
 * Aqui está um bloco de comentário...
 */
```

Comentários de bloco podem começar com `/* -`, reconhecido pelo travessão (1) como o início de um bloco de comentário que não deve ser reformatado, ou seja é um alerta de que, quem o fez quer que mantenha nessa forma. Exemplo:

```
/*-
 * Here is a block comment with some very special
 * formatting that I want indent(1) to ignore.
 *
 *     one
 *         two
 *             three
 */
```

Ainda temos os comentários simples, exemplo:

```
if (condition) {

    /* Handle the condition. */
    ...
}
```

Trailing Comments

Comentários muito curtos podem aparecer na mesma linha do código que está inserida, mas deve ser deslocado o suficiente para separá-los das declarações. Se mais de um breve comentário aparecer em um trecho de código, todos eles deverão ser recuados para a definição mesma guia, deixando sempre alinhados.

Exemplo:

```
if (a == 2) {
    return TRUE;           /* special case */
} else {
    return isPrime(a);     /* works only for odd a */
}
```

Comentários de fim de linha `"/"`

O delimitador `//` como comentário pode comentar uma linha completa ou apenas parcial. Ele não deve ser utilizado em linhas consecutivas para comentários de texto, no entanto, pode ser utilizado em linhas consecutivas para comentando secções de código. Exemplos de todos os estilos:

```
if (foo > 1) {  
    // Do a double-flip.  
    ...  
}  
else {  
    return false;           // Explain why here.  
}  
//if (bar > 1) {  
//  
//    // Do a triple-flip.  
//    ...  
//}  
//else {  
//    return false;  
//}
```

Declarações

Uma declaração por linha é recomendado desde que o a menção ao código seja justificada por seu tipo e (ou) referência, em outras palavras:

```
int level; // indentation level  
int size;  // size of table
```

é preferível que, seja:

```
int level, size;
```

Nunca coloque tipos diferentes numa mesma linha. Exemplo:

```
int foo,  fooarray[]; //errado!
```

No exemplo abaixo foi usado um espaço entre o tipo e o variável identificada, outra forma aceitável é o uso de tabs.

```
int level;           // indentation level  
int size;            // size of table  
Object  currentEntry; // currently selected table entry
```

Classes e Interfaces

Classes Java e interfaces seguem algumas regras de formatação, veja:

- Sem espaço entre um método e o parênteses e o nome do método "(" início de lista de parâmetros;
- Abertura da chaves "{" aparece no fim da mesma linha que foi declarado o código ;
- Fechamento da chaves "}" começa uma linha alinhada no conjunto do método a qual foi criada, exceto quando há códigos em parte em branco(vazio) ou nulo }"devendo aparecer imediatamente depois de aberto com "{"
- Métodos são sempre separados por uma linha em branco.

```
class Sample extends Object {  
    int ivar1;  
    int ivar2;  
  
    Sample(int i, int j) {  
        ivar1 = i;  
        ivar2 = j;  
    }  
  
    int emptyMethod() {}  
  
    ...  
}
```

Convenção para nomes

As convenções de nomenclatura tem como objetivo tornar os programas mais compreensíveis, tornando-os mais fáceis de ler. Eles podem também fornecer informações sobre a função do identificador, por exemplo, quer se trate de um pacote, constante, ou de classe que pode ser útil na compreensão do código.

Identifier Type	Rules for Naming	Exemplos
Packages	O prefixo do nome do pacote deve ser único, deve sempre ser escrito em letras minúsculas todo-ASCII e deve ser um dos nomes de domínio de nível superior, atualmente com, edu, gov, mil, net, org, códigos de duas letras identificando os países, tal como especificado na norma ISO 3166, 1981. Componentes subseqüentes do nome do pacote varia de acordo com uma organização próprias convenções de nomenclatura internos. Tais convenções podem especificar que certos componentes do nome do diretório haver divisão, departamento, projeto, máquina, ou nomes de login.	com.sun.eng com.apple.quicktime.v2 edu.cmu.cs.bovik.cheese
Classes	Os nomes de classe devem ser substantivos, em	class Raster; class ImageSprite;

maiúsculas e minúsculas com a primeira letra de cada palavra interna em maiúscula. Tente manter seus nomes de classe simples e descritivo. Sempre evite palavras-ligadas, evite todas siglas e abreviaturas, seja semântico.

Interfaces	Nomes de interfaces devem ser usadas com as primeiras letras em maiúsculas como nome de classes.	<code>interface RasterDelegate;</code> <code>interface Storing;</code>
Methods	Métodos devem ser verbos, com a letra minúscula em primeiro lugar, com a primeira letra de cada palavra interna em maiúscula.	<code>run(); runFast();</code> <code>getBackground();</code>
Variables	Os nomes de variáveis não deve começar com underscore _ ou sinal de dólar \$ personagens, mesmo que ambos não são permitidos. Os nomes de variáveis devem ser curtos, mas significativo. A escolha de um nome variável deve ser mnemônico, isto é, concebidos para indicar ao observador casual a intenção da sua utilização. Um personagem nomes de variáveis devem ser evitadas, exceto para temporários "descartáveis" variáveis. Os nomes comuns para variáveis temporárias são i, j, k, m, n e para inteiros, c, d, e e para caracteres	<code>int i; char c; float myWidth;</code>
Constants	Os nomes de variáveis declaradas constantes de classes e de constantes ANSI deve ser todo em letras maiúsculas com palavras separadas por sublinhados ("_").	<code>static final int MIN_WIDTH = 4;</code> <code>static final int MAX_WIDTH = 999;</code> <code>static final int GET_THE_CPU = 1;</code>

Nessa área de boas práticas em documentação do código temos ainda um amplo estudo e normas à seguir para se implantar as boas práticas no seu ambiente de trabalho, aconselho acessar o link de [documentação](#).

Herança, reescrita e polimorfismo

"O homem absurdo é aquele que nunca muda." -- Georges Clemenceau

Ao término desse capítulo, você será capaz de:

- dizer o que é herança e quando utilizá-la;
- reutilizar código escrito anteriormente;
- criar classes filhas e reescrever métodos;
- usar todo o poder que o polimorfismo dá.

9.1 Repetindo código?

Como toda empresa, nosso Banco possui funcionários. Vamos modelar a classe `Funcionario`:

```
public class Funcionario {  
    private String nome;  
    private String cpf;
```

```
        private double salario;
        // métodos devem vir aqui
    }
```

Além de um funcionário comum, há também outros cargos, como os gerentes. Os gerentes guardam a mesma informação que um funcionário comum, mas possuem outras informações, além de ter funcionalidades um pouco diferentes. Um gerente no nosso banco possui também uma senha numérica que permite o acesso ao sistema interno do banco, além do número de funcionários que ele gerencia:

```
public class Gerente {
    private String nome;
    private String cpf;
    private double salario;
    private int senha;
    private int numeroDeFuncionariosGerenciados;

    public boolean autentica(int senha) {
        if (this.senha == senha) {
            System.out.println("Acesso Permitido!");
            return true;
        } else {
            System.out.println("Acesso Negado!");
            return false;
        }
    }
}

// outros métodos
}
```

Precisamos mesmo de outra classe?

Poderíamos ter deixado a classe `Funcionario` mais genérica, mantendo nela senha de acesso, e o número de funcionários gerenciados. Caso o funcionário não fosse um gerente, deixaríamos estes atributos vazios.

Essa é uma possibilidade, porém podemos começar a ter muito atributos opcionais, e a classe ficaria estranha. E em relação aos métodos? A classe `Gerente` tem o método `autentica`, que não faz sentido existir em um funcionário que não é gerente.

Se tivéssemos um outro tipo de funcionário que tem características diferentes do funcionário comum, precisaríamos criar uma outra classe e copiar o código novamente!

Além disso, se um dia precisarmos adicionar uma nova informação para todos os funcionários, precisaremos passar por todas as classes de funcionário e adicionar esse atributo. O problema acontece novamente por não centralizarmos as informações principais do funcionário em um único lugar!

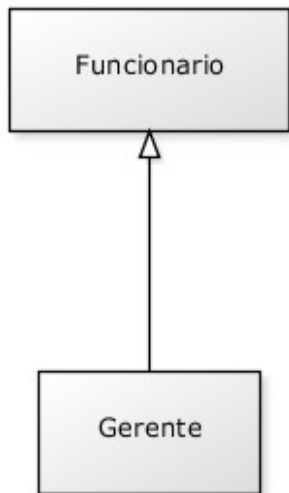
Existe um jeito, em Java, de relacionarmos uma classe de tal maneira que uma delas herda tudo que a outra tem. Isto é uma relação de classe mãe e classe filha. No nosso caso, gostaríamos de fazer com que o Gerente tivesse tudo que um Funcionario tem, gostaríamos que ela fosse uma extensão de Funcionario. Fazemos isto através da palavra chave `extends`.

```
public class Gerente extends Funcionario {
    private int senha;
    private int numeroDeFuncionariosGerenciados;

    public boolean autentica(int senha) {
        if (this.senha == senha) {
            System.out.println("Acesso Permitido!");
            return true;
        } else {
            System.out.println("Acesso Negado!");
            return false;
        }
    }

    // setter da senha omitido
}
```

Em todo momento que criarmos um objeto do tipo Gerente, este objeto possuirá também os atributos definidos na classe Funcionario, pois um Gerente é um Funcionario:



```
public class TestaGerente {
    public static void main(String[] args) {
        Gerente gerente = new Gerente();

        // podemos chamar métodos do Funcionario:
        gerente.setNome("João da Silva");

        // e também métodos do Gerente!
        gerente.setSenha(4231);
    }
}
```

Executar código

Dizemos que a classe `Gerente` herda todos os atributos e métodos da classe mãe, no nosso caso, a `Funcionario`. Para ser mais preciso, ela também herda os atributos e métodos privados, porém não consegue acessá-los diretamente. Para acessar um membro privado na filha indiretamente, seria necessário que a mãe expusesse um outro método visível que invocasse esse atributo ou método privado.

Super e Sub classe

A nomenclatura mais encontrada é que `Funcionario` é a superclasse de `Gerente`, e `Gerente` é a subclasse de `Funcionario`. Dizemos também que todo `Gerente` é um `Funcionário`. Outra forma é dizer que `Funcionario` é classe mãe de `Gerente` e `Gerente` é classe filha de `Funcionario`.

E se precisamos acessar os atributos que herdamos? Não gostaríamos de deixar os atributos de `Funcionario`, `public`, pois dessa maneira qualquer um poderia alterar os atributos dos objetos deste tipo. Existe um outro modificador de acesso, o `protected`, que fica entre o `private` e o `public`. Um atributo `protected` só pode ser acessado (visível) pela própria classe, por suas subclasses, e pelas classes que se encontram no mesmo pacote.

```
public class Funcionario {  
    protected String nome;  
    protected String cpf;  
    protected double salario;  
    // métodos devem vir aqui  
}
```

Sempre usar `protected`?

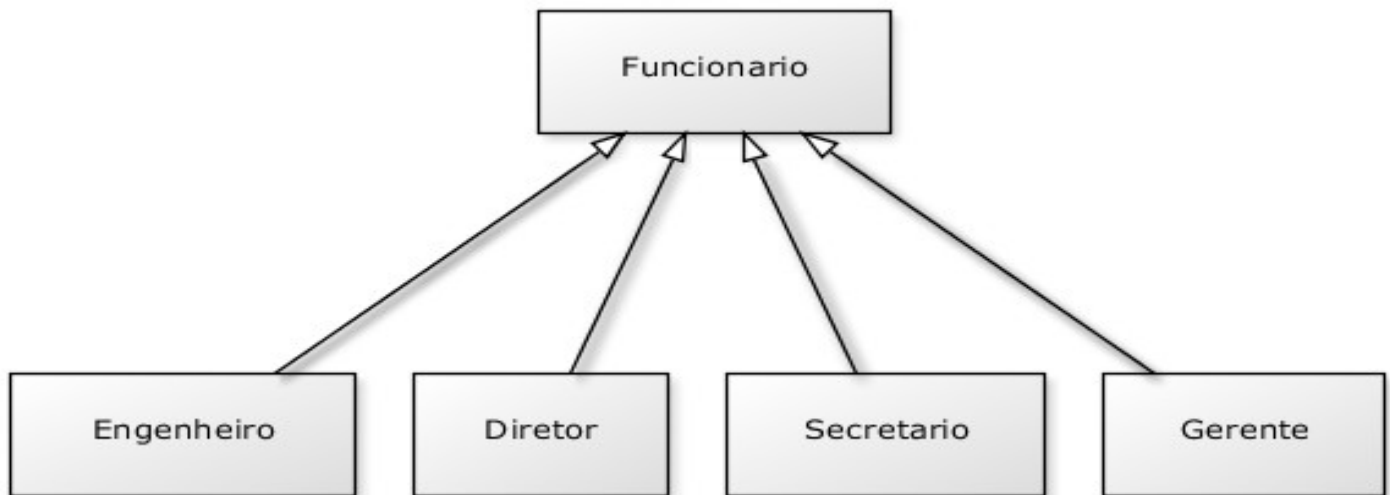
Então por que usar `private`? Depois de um tempo programando orientado a objetos, você vai começar a sentir que nem sempre é uma boa ideia deixar que a classe filha acesse os atributos da classe mãe, pois isso quebra um pouco a ideia de que só aquela classe deveria manipular seus atributos. Essa é uma discussão um pouco mais avançada.

Além disso, não só as subclasses, mas também as outras classes que se encontram no mesmo pacote, podem acessar os atributos `protected`. Veja outras alternativas ao `protected` no exercício de discussão em sala de aula juntamente com o instrutor.

Da mesma maneira, podemos ter uma classe `Diretor` que estenda `Gerente` e a classe `Presidente` pode estender diretamente de `Funcionario`.

Fique claro que essa é uma decisão de negócio. Se `Diretor` vai estender de `Gerente` ou não, vai depender se, para você, `Diretor` é um `Gerente`.

Uma classe pode ter várias filhas, mas pode ter apenas uma mãe, é a chamada herança simples do java.



Editora Casa do Código com livros de uma forma diferente



Editoras tradicionais pouco ligam para ebooks e novas tecnologias. Não dominam tecnicamente o assunto para revisar os livros a fundo. Não têm anos de experiência em didáticas com cursos.

Conheça a Casa do Código, uma editora diferente, com curadoria da Caelum e obsessão por livros de qualidade a preços justos.

[Casa do Código, ebook com preço de ebook.](#)

9.2 Reescrita de método

Todo fim de ano, os funcionários do nosso banco recebem uma bonificação. Os funcionários comuns recebem 10% do valor do salário e os gerentes, 15%.

Vamos ver como fica a classe `Funcionario`:

```
public class Funcionario {
    protected String nome;
    protected String cpf;
    protected double salario;

    public double getBonificacao() {
        return this.salario * 0.10;
    }
    // métodos
}
```


Se deixarmos a classe `Gerente` como ela está, ela vai herdar o método `getBonificacao`.

```
Gerente gerente = new Gerente();
gerente.setSalario(5000.0);
System.out.println(gerente.getBonificacao());
```

O resultado aqui será 500. Não queremos essa resposta, pois o gerente deveria ter 750 de bônus nesse caso. Para consertar isso, uma das opções seria criar um novo método na classe `Gerente`, chamado, por exemplo, `getBonificacaoDoGerente`. O problema é que teríamos dois métodos em `Gerente`, confundindo bastante quem for usar essa classe, além de que cada um dá uma resposta diferente.

No Java, quando herdamos um método, podemos alterar seu comportamento. Podemos reescrever (reescrever, sobrescrever, override) este método:

```
public class Gerente extends Funcionario {
    int senha;
    int numeroDeFuncionariosGerenciados;

    public double getBonificacao() {
        return this.salario * 0.15;
    }
    // ...
}
```

Agora o método está correto para o `Gerente`. Refaça o teste e veja que o valor impresso é o correto (750):

```
Gerente gerente = new Gerente();
gerente.setSalario(5000.0);
System.out.println(gerente.getBonificacao());
```

Executar código

a anotação `@Override`

Há como deixar explícito no seu código que determinado método é a reescrita de um método da sua classe mãe. Fazemos isso colocando `@Override` em cima do método. Isso é chamado anotação. Existem diversas anotações e cada uma vai ter um efeito diferente sobre seu código.

```
@Override
public double getBonificacao() {
```

```
        return this.salario * 0.15;
    }
```

Repare que, por questões de compatibilidade, isso não é obrigatório. Mas caso um método esteja anotado com `@Override`, ele necessariamente precisa estar reescrevendo um método da classe mãe.

9.3 Invocando o método reescrito

Depois de reescrito, não podemos mais chamar o método antigo que fora herdado da classe mãe: realmente alteramos o seu comportamento. Mas podemos invocá-lo no caso de estarmos dentro da classe.

Imagine que para calcular a bonificação de um `Gerente` devemos fazer igual ao cálculo de um `Funcionario` porém adicionando R\$ 1000.

Poderíamos fazer assim:

```
public class Gerente extends Funcionario {
    int senha;
    int numeroDeFuncionariosGerenciados;

    public double getBonificacao() {
        return this.salario * 0.10 + 1000;
    }
    // ...
}
```

Aqui teríamos um problema: o dia que

o `getBonificacao` do `Funcionario` mudar, precisaremos mudar o método do `Gerente` para acompanhar a nova bonificação. Para evitar isso, o `getBonificacao` do `Gerente` pode chamar o `do Funcionario` utilizando a palavra chave `super`.

```
public class Gerente extends Funcionario {
    int senha;
    int numeroDeFuncionariosGerenciados;

    public double getBonificacao() {
        return super.getBonificacao() + 1000;
    }
    // ...
}
```

Essa invocação vai procurar o método com o nome `getBonificacao` de uma super classe de `Gerente`. No caso ele logo vai encontrar esse método em `Funcionario`.

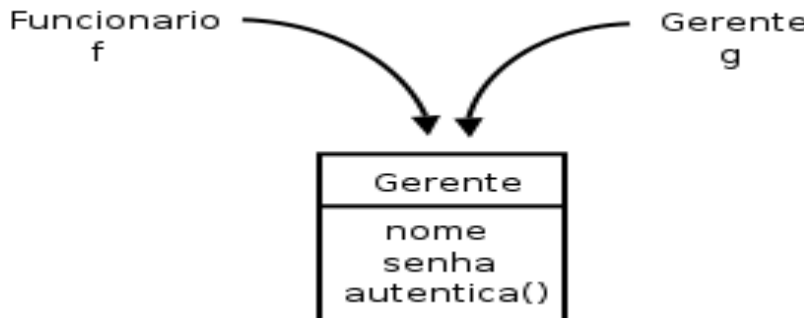
Essa é uma prática comum, pois muitos casos o método reescrito geralmente faz "algo a mais" que o método da classe mãe. Chamar ou não o método de cima é uma decisão sua e depende do seu problema. Algumas vezes não faz sentido invocar o método que reescrevemos.

9.4 Polimorfismo

O que guarda uma variável do tipo `Funcionario`? Uma referência para um `Funcionario`, nunca o objeto em si.

Na herança, vimos que todo `Gerente` é um `Funcionario`, pois é uma extensão deste. Podemos nos referir a um `Gerente` como sendo um `Funcionario`. Se alguém precisa falar com um `Funcionario` do banco, pode falar com um `Gerente`! Porque? Pois `Gerente` é um `Funcionario`. Essa é a semântica da herança.

```
Gerente gerente = new Gerente();  
Funcionario funcionario = gerente;  
funcionario.setSalario(5000.0);
```



```
graph TD
    f[Funcionario f] --> g[Gerente g]
    g --> g_obj[Gerente  
nome  
senha  
autentica()]
```

Polimorfismo é a capacidade de um objeto poder ser referenciado de várias formas. (cuidado, polimorfismo não quer dizer que o objeto fica se transformando, muito pelo contrário, um objeto nasce de um tipo e morre daquele tipo, o que pode mudar é a maneira como nos referimos a ele).

Até aqui tudo bem, mas e se eu tentar:

```
funcionario.getBonificacao();
```

Qual é o retorno desse método? 500 ou 750? No Java, a invocação de método sempre vai ser decidida em tempo de execução. O Java vai

procurar o objeto na memória e, aí sim, decidir qual método deve ser chamado, sempre relacionando com sua classe de verdade, e não com a que estamos usando para referenciá-lo. Apesar de estarmos nos referenciando a esse Gerente como sendo um Funcionario, o método executado é o do Gerente. O retorno é 750.

Parece estranho criar um gerente e referenciá-lo como apenas um funcionário. Por que faríamos isso? Na verdade, a situação que costuma aparecer é a que temos um método que recebe um argumento do tipo Funcionario:

```
class ControleDeBonificacoes {  
    private double totalDeBonificacoes = 0;  
  
    public void registra(Funcionario funcionario) {  
        this.totalDeBonificacoes +=  
funcionario.getBonificacao();  
    }  
  
    public double getTotalDeBonificacoes() {  
        return this.totalDeBonificacoes;  
    }  
}
```

E, em algum lugar da minha aplicação (ou no main, se for apenas para testes):

```
ControleDeBonificacoes controle = new  
ControleDeBonificacoes();  
  
Gerente funcionario1 = new Gerente();  
funcionario1.setSalario(5000.0);  
controle.registra(funcionario1);  
  
Funcionario funcionario2 = new Funcionario();  
funcionario2.setSalario(1000.0);  
controle.registra(funcionario2);
```

```
System.out.println(controle.getTotalDeBonificacoes());  
Executar código
```

Repare que conseguimos passar um Gerente para um método que recebe um Funcionario como argumento. Pense como numa porta na agência

bancária com o seguinte aviso: "Permitida a entrada apenas de Funcionários". Um gerente pode passar nessa porta? Sim, pois `Gerente` é um `Funcionario`.

Qual será o valor resultante? Não importa que dentro do método `registra` do `ControleDeBonificacoes` receba `Funcionario`. Quando ele receber um objeto que realmente é um `Gerente`, o seu método reescrito será invocado. Reafirmando: não importa como nos referenciamos a um objeto, o método que será invocado é sempre o que é dele.

No dia em que criarmos uma classe `Secretaria`, por exemplo, que é filha de `Funcionario`, precisaremos mudar a classe de `ControleDeBonificacoes`? Não. Basta a classe `Secretaria` reescrever os métodos que lhe parecerem necessários. É exatamente esse o poder do polimorfismo, juntamente com a reescrita de método: diminuir o acoplamento entre as classes, para evitar que novos códigos resultem em modificações em inúmeros lugares.

Repare que quem criou `ControleDeBonificacoes` pode nunca ter imaginado a criação da classe `Secretaria` ou `Engenheiro`. Contudo, não será necessário reimplementar esse controle em cada nova classe: reaproveitamos aquele código.

Herança versus acoplamento

Note que o uso de herança aumenta o acoplamento entre as classes, isto é, o quanto uma classe depende de outra. A relação entre classe mãe e filha é muito forte e isso acaba fazendo com que o programador das classes filhas tenha que conhecer a implementação da classe mãe e vice-versa - fica difícil fazer uma mudança pontual no sistema.

Por exemplo, imagine se tivermos que mudar algo na nossa classe `Funcionario`, mas não quiséssemos que todos os funcionários sofressem a mesma mudança. Precisaríamos passar por cada uma das filhas de `Funcionario` verificando se ela se comporta como deveria ou se devemos sobrescrever o tal método modificado.

Esse é um problema da herança, e não do polimorfismo, que resolveremos mais tarde com a ajuda de Interfaces.

Já conhece os cursos online Alura?

alura

A Alura oferece centenas de cursos online em sua plataforma exclusiva de ensino que favorece o aprendizado

com a qualidade reconhecida da Caelum. Você pode escolher um curso

nas áreas de Programação, Front-end, Mobile, Design & UX, Infra e

Business, com um plano que dá acesso a todos os cursos. Ex-aluno da

Caelum tem 15% de desconto neste link!

[Conheça os cursos online Alura.](#)

9.5 Um outro exemplo

Imagine que vamos modelar um sistema para a faculdade que controle as despesas com funcionários e professores. Nosso funcionário fica assim:

```
public class EmpregadoDaFaculdade {
    private String nome;
    private double salario;
    public double getGastos() {
        return this.salario;
    }
    public String getInfo() {
        return "nome: " + this.nome + " com salário " +
this.salario;
    }
    // métodos de get, set e outros
}
```

O gasto que temos com o professor não é apenas seu salário. Temos de somar um bônus de 10 reais por hora/aula. O que fazemos então?

Reescrevemos o método. Assim como o `getGastos` é diferente, o `getInfo` também será, pois temos de mostrar as horas/aula também.

```
public class ProfessorDaFaculdade extends
EmpregadoDaFaculdade {
    private int horasDeAula;
    public double getGastos() {
        return this.getSalario() + this.horasDeAula *
10;
    }
```

```

    public String getInfo() {
        String informacaoBasica = super.getInfo();
        String informacao = informacaoBasica + " horas
de aula: "
                                + this.horasDeAula;

        return informacao;
    }
    // métodos de get, set e outros que forem
necessários
}

```

A novidade, aqui, é a palavra chave `super`. Apesar do método ter sido reescrito, gostaríamos de acessar o método da classe mãe, para não ter de copiar e colocar o conteúdo desse método e depois concatenar com a informação das horas de aula.

Como tiramos proveito do polimorfismo? Imagine que temos uma classe de relatório:

```

public class GeradorDeRelatorio {
    public void adiciona(EmpregadoDaFaculdade f) {
        System.out.println(f.getInfo());
        System.out.println(f.getGastos());
    }
}

```

Podemos passar para nossa classe qualquer `EmpregadoDaFaculdade`!

Vai funcionar tanto para professor, quanto para funcionário comum.

Um certo dia, muito depois de terminar essa classe de relatório, resolvemos aumentar nosso sistema, e colocar uma classe nova, que representa o `Reitor`. Como ele também é um `EmpregadoDaFaculdade`, será que vamos precisar alterar algo na nossa classe de `Relatorio`? Não. Essa é a ideia! Quem programou a classe `GeradorDeRelatorio` nunca imaginou que existiria uma classe `Reitor` e, mesmo assim, o sistema funciona.

```

public class Reitor extends EmpregadoDaFaculdade {
    // informações extras
    public String getInfo() {
        return super.getInfo() + " e ele é um reitor";
    }
    // não sobrescrevemos o getGastos!!!
}

```

}

9.6 Um pouco mais...

- Se não houvesse herança em Java, como você poderia reaproveitar o código de outra classe?
- Uma discussão muito atual é sobre o abuso no uso da herança. Algumas pessoas usam herança apenas para reaproveitar o código, quando poderiam ter feito uma composição. Procure sobre herança versus composição.
- Mesmo depois de reescrever um método da classe mãe, a classe filha ainda pode acessar o método antigo. Isto é feito através da palavra chave `super.método()`. Algo parecido ocorre entre os construtores das classes, o que?

Mais sobre o mau uso da herança

No blog da Caelum existe um artigo interessante abordando esse tópico:

<http://blog.caelum.com.br/2006/10/14/como-nao-aprender-orientacao-a-objetos-heranca/>

James Gosling, um dos criadores do Java, é um crítico do mau uso da herança. Nesta entrevista ele discute a possibilidade de se utilizar apenas interfaces e composição, eliminando a necessidade da herança:

<http://www.artima.com/intv/gosling3P.html>

9.7 Exercícios: Herança e Polimorfismo

1. Vamos ter mais de um tipo de conta no nosso sistema então vamos precisar de uma nova tela para cadastrar os diferentes tipos de conta. Essa tela já está pronta e para utilizá-la só precisamos alterar a classe que estamos chamando no método `main()` no `TestaContas.java`:

```
package br.com.caelum.contas.main;

import br.com.caelum.javafx.api.main.SistemaBancario;

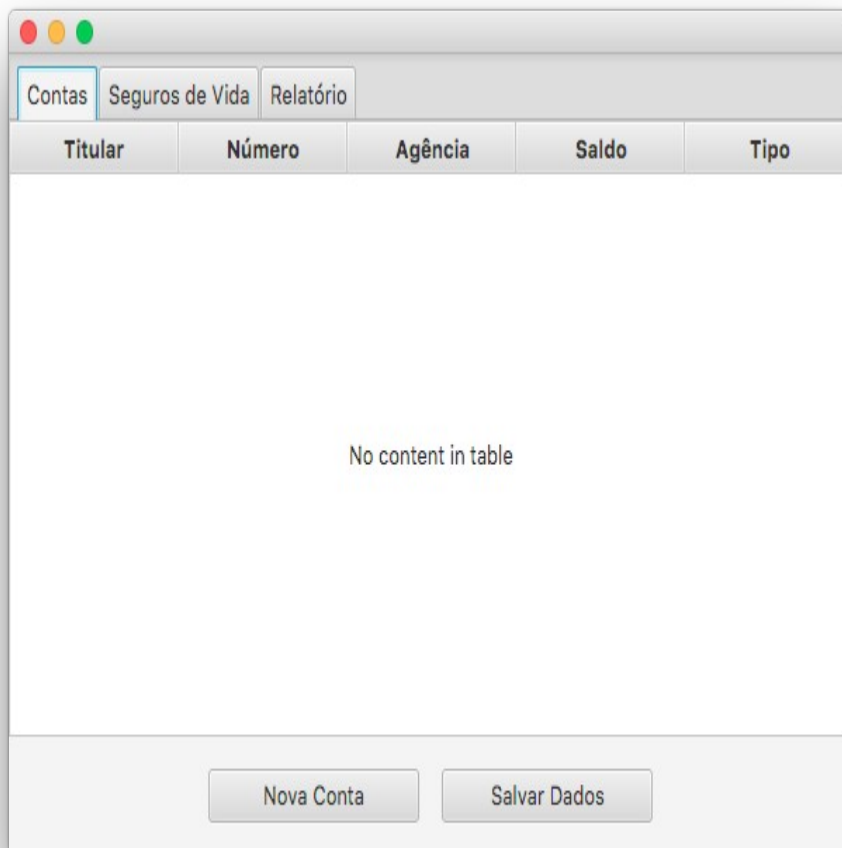
public class TestaContas {

    public static void main(String[] args) {
        SistemaBancario.mostraTela(false);
    }
}
```



```
        // TelaDeContas.main(args);  
    }  
}
```

2. Ao rodar a classe `TestaContas` agora, teremos a tela abaixo:



Vamos entrar na tela de criação de contas para vermos o que precisamos implementar para que o sistema funcione. Para isso, clique no botão Nova Conta. A seguinte tela aparecerá:

Nova Conta

Titular:

Número:

Agência:

Tipo: ☐ Conta Corrente ☐ Conta Poupança

Podemos perceber que além das informações que já tínhamos na conta, temos agora o tipo: se queremos uma conta corrente ou uma conta poupança. Vamos então criar as classes correspondentes.

- Crie a classe `ContaCorrente` no pacote `br.com.caelum.contas.modelo` e faça com que ela seja filha da classe `Conta`
- Crie a classe `ContaPoupanca` no pacote `br.com.caelum.contas.modelo` e faça com que ela seja filha da classe `Conta`

3. Precisamos pegar os dados da tela para conseguirmos criar a conta correspondente. No `ManipuladorDeContas` vamos alterar o método `criaConta`. Atualmente, apenas criamos uma nova conta com os

dados direto no código. Vamos fazer com que agora os dados sejam recuperados da tela para colocarmos na nova conta, faremos isso utilizando o objeto evento:

```
public void criaConta(Evento evento) {
    this.conta = new Conta();
    this.conta.setAgencia(evento.getString("agencia"));
    this.conta.setNumero(evento.getInt("numero"));
    this.conta.setTitular(evento.getString("titular"));
}
```

Mas precisamos dizer qual tipo de conta que queremos criar! Devemos então recuperar o tipo da conta escolhido e criar a conta correspondente. Para isso, ao invés de criar um objeto do tipo 'Conta', vamos usar o método `getSelecionadoNoRadio` do objeto evento para pegar o tipo, fazer um `if` para verificar esse tipo e só depois criar o objeto do tipo correspondente. Após essas mudanças, o método `criaConta` ficará como abaixo:

```
public void criaConta(Evento evento) {
    String tipo = evento.getSelecionadoNoRadio("tipo");
    if (tipo.equals("Conta Corrente")) {
        this.conta = new ContaCorrente();
    } else if (tipo.equals("Conta Poupança")) {
        this.conta = new ContaPoupanca();
    }
    this.conta.setAgencia(evento.getString("agencia"));
    this.conta.setNumero(evento.getInt("numero"));
    this.conta.setTitular(evento.getString("titular"));
}
```

4. Apesar de já conseguirmos criar os dois tipos de contas, nossa lista não consegue exibir o tipo de cada conta na lista da tela inicial. Para resolver isso, podemos criar um método `getTipo` em cada uma de nossas contas fazendo com que a conta corrente devolva a string "Conta Corrente" e a conta poupança devolva a string "Conta Poupança":

```
public class ContaCorrente extends Conta {
    public String getTipo() {
        return "Conta Corrente";
    }
}
```

```
public class ContaPoupanca extends Conta {
    public String getTipo() {
        return "Conta Poupança";
    }
}
```

5. Altere os métodos `saca` e `deposita` para buscarem o campo `valorOperacao` ao invés de apenas `valor` na classe `ManipuladorDeContas`.

6. Vamos mudar o comportamento da operação de saque de acordo com o tipo de conta que estiver sendo utilizada. Na classe `ManipuladorDeContas` vamos alterar o método `saca` para tirar 10 centavos de cada saque em uma conta corrente:

```
public void saca(Evento evento) {
    double valor = evento.getDouble("valorOperacao");
    if (this.conta.getTipo().equals("Conta Corrente")) {
        this.conta.saca(valor + 0.10);
    } else {
        this.conta.saca(valor);
    }
}
```

Ao tentarmos chamar o método `getTipo`, o Eclipse reclamou que esse método não existe na classe `Conta` apesar de existir nas classes filhas. Como estamos tratando todas as contas genericamente, só conseguimos acessar os métodos da classe mãe. Vamos então colocá-lo na classe `Conta`:

```
public class Conta {
    public String getTipo() {
        return "Conta";
    }
}
```

7. Agora o código compila mas temos um outro problema. A lógica do nosso saque vazou para a classe `ManipuladorDeContas`. Se algum dia precisarmos alterar o valor da taxa no saque, teríamos que mudar em todos os lugares onde fazemos uso do método `saca`. Esta lógica deveria estar encapsulada dentro do método `saca` de cada conta. Vamos então sobrescrever o método dentro da classe `ContaCorrente`:

```

public class ContaCorrente extends Conta {
    @Override
    public void saca(double valor) {
        this.saldo -= (valor + 0.10);
    }

    // restante da classe
}

```

Repare que, para acessar o atributo saldo herdado da classe Conta, você vai precisar mudar o modificador de visibilidade de saldo para protected.

Agora que a lógica está encapsulada, podemos corrigir o método saca da classe ManipuladorDeContas:

```

public void saca(Evento evento) {
    double valor = evento.getDouble("valorOperacao");
    this.conta.saca(valor);
}

```

Perceba que agora tratamos a conta de forma genérica!

8. Rode a classe TestaContas, adicione uma conta de cada tipo e veja se o tipo é apresentado corretamente na lista de contas da tela inicial.

Agora, clique na conta corrente apresentada na lista para abrir a tela de detalhes de contas. Teste as operações de saque e depósito e perceba que a conta apresenta o comportamento de uma conta corrente conforme o esperado.

E se tentarmos realizar uma transferência da conta corrente para a conta poupança? O que acontece?

9. Vamos começar implementando o método transfere na classe Conta:

```

public void transfere(double valor, Conta conta) {
    this.saca(valor);
    conta.deposita(valor);
}

```

Também precisamos implementar o método transfere na classe ManipuladorDeContas para fazer o vínculo entre a tela e a classe Conta:

```

public void transfere(Evento evento) {

```

```
Conta destino = (Conta)
evento.getSelectedNoCombo("destino");

conta.transfere(evento.getDouble("valorTransferencia"),
destino);
}
```

Rode de novo a aplicação e teste a operação de transferência.

10. Considere o código abaixo:

```
Conta c = new Conta();
ContaCorrente cc = new ContaCorrente();
ContaPoupanca cp = new ContaPoupanca();
```

Se mudarmos esse código para:

```
Conta c = new Conta();
Conta cc = new ContaCorrente();
Conta cp = new ContaPoupanca();
```

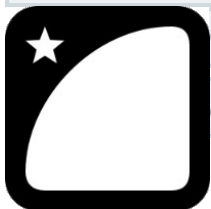
Compila? Roda? O que muda? Qual é a utilidade disso? Realmente, essa não é a maneira mais útil do polimorfismo. Porém existe uma utilidade de declararmos uma variável de um tipo menos específico do que o objeto realmente é, como fazemos na classe `ManipuladorDeContas`.

É extremamente importante perceber que não importa como nos referimos a um objeto, o método que será invocado é sempre o mesmo! A JVM vai descobrir em tempo de execução qual deve ser invocado, dependendo de que tipo é aquele objeto, não importando como nos referimos a ele.

11. (Opcional) A nossa classe `Conta` devolve a palavra "Conta" no método `getTipo`. Use a palavra chave `super` nos métodos `getTipo` reescritos nas classes filhas, para não ter de reescrever a palavra "Conta" ao devolver os tipos "Conta Corrente" e "Conta Poupança".

12. (Opcional) Se você precisasse criar uma classe `ContaInvestimento`, e seu método `saca` fosse complicadíssimo, você precisaria alterar a classe `ManipuladorDeContas`?

Você pode também fazer o curso FJ-11 dessa apostila na Caelum



Querendo aprender ainda mais sobre? Esclarecer dúvidas dos
exercícios? Ouvir explicações detalhadas com um instrutor?

A Caelum oferece o curso FJ-11 presencial nas cidades de São Paulo, Rio de Janeiro e Brasília, além de turmas incompany.

[Consulte as vantagens do curso Java e Orientação a Objetos](#)

9.8 Discussões em aula: Alternativas ao atributo `protected`

Discuta com o instrutor e seus colegas alternativas ao uso do atributo `protected` na herança. Preciso realmente afrouxar o encapsulamento do atributo por causa da herança? Como fazer para o atributo continuar `private` na mãe e as filhas conseguirem de alguma forma trabalhar com ele?

Exceções

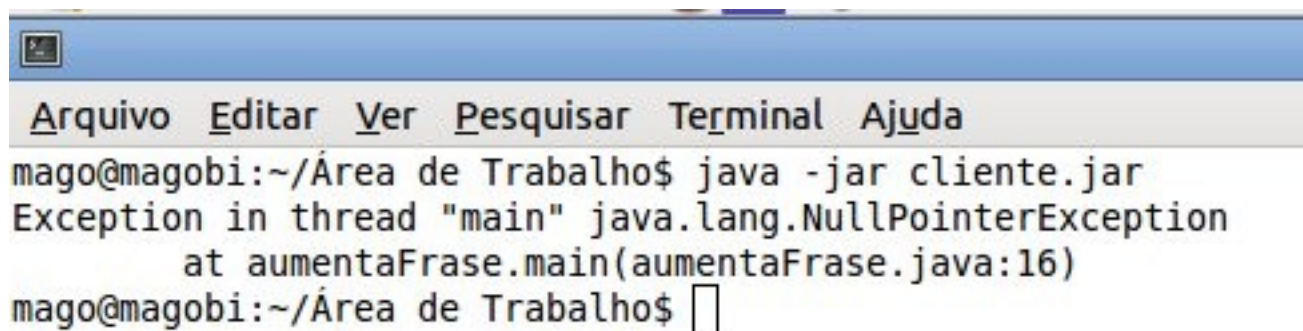
Quando se cria programas de computador em Java, há possibilidade de ocorrer erros imprevistos durante sua execução, esses erros são conhecidos como exceções e podem ser provenientes de erros de lógica ou acesso a dispositivos ou arquivos externos.

Entendendo as exceções

As exceções ocorrem quando algo imprevisto acontece, elas podem ser provenientes de erros de lógica ou acesso a recursos que talvez não estejam disponíveis.

Alguns possíveis motivos externos para ocorrer uma exceção são:

- Tentar abrir um arquivo que não existe.
- Tentar fazer consulta a um banco de dados que não está disponível.
- Tentar escrever algo em um arquivo sobre o qual não se tem permissão de escrita.
- Tentar conectar em servidor inexistente.

A screenshot of a Java IDE's terminal window. The window has a blue title bar with a small icon on the left. Below the title bar is a menu bar with the following items: 'Arquivo', 'Editar', 'Ver', 'Pesquisar', 'Terminal', and 'Ajuda'. The terminal area shows the following text: 'mago@magobi:~/Área de Trabalho\$ java -jar cliente.jar', 'Exception in thread "main" java.lang.NullPointerException', 'at aumentaFrase.main(aumentaFrase.java:16)', and 'mago@magobi:~/Área de Trabalho\$' followed by a cursor icon.

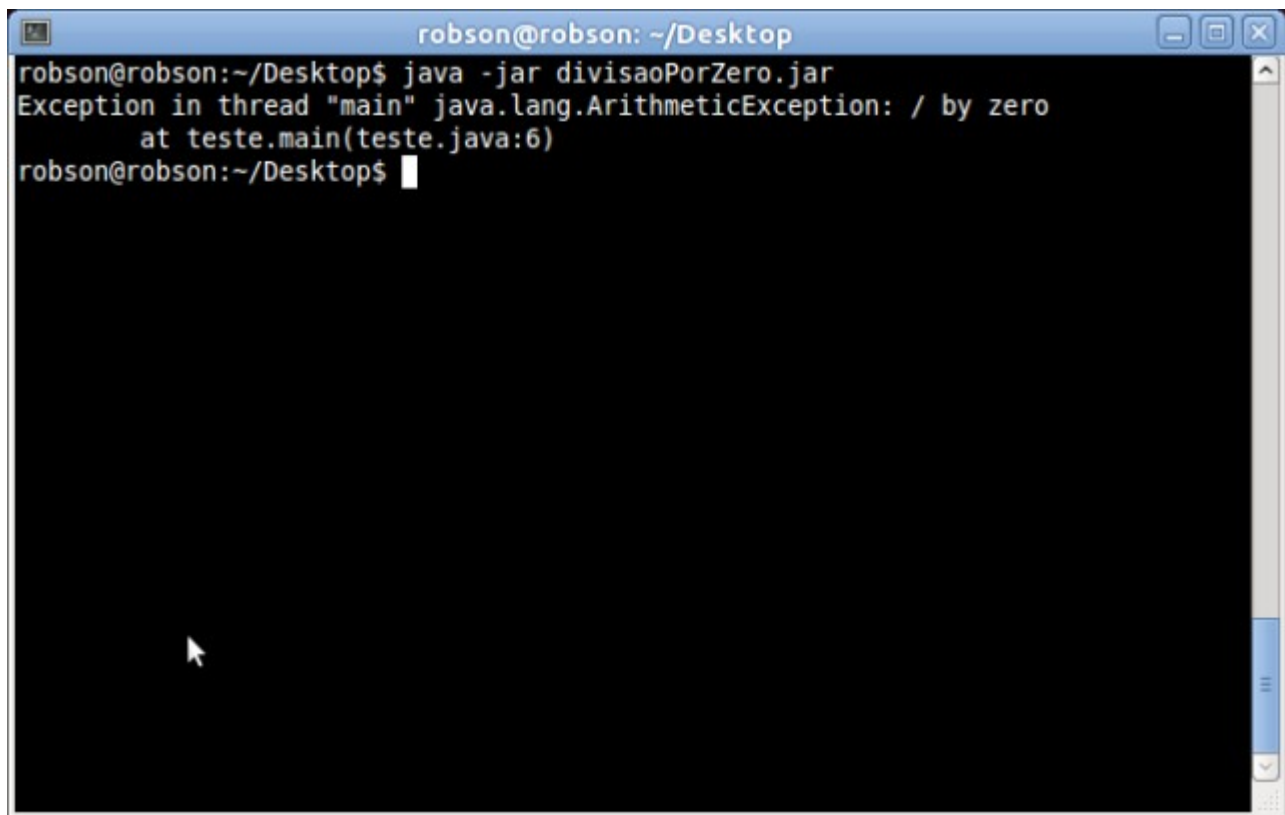
```
Arquivo  Editar  Ver  Pesquisar  Terminal  Ajuda
mago@magobi:~/Área de Trabalho$ java -jar cliente.jar
Exception in thread "main" java.lang.NullPointerException
    at aumentaFrase.main(aumentaFrase.java:16)
mago@magobi:~/Área de Trabalho$
```

F

igura 1. Ocorrência de exceção apontada no console

Alguns possíveis erros de lógica para ocorrer uma exceção são:

- Tentar manipular um objeto que está com o valor nulo.
- Dividir um número por zero.
- Tentar manipular um tipo de dado como se fosse outro.
- Tentar utilizar um método ou classe não existentes.

A terminal window titled 'robson@robson: ~/Desktop' with standard window controls. The command 'java -jar divisaoPorZero.jar' has been executed. The output shows an 'Exception in thread "main" java.lang.ArithmeticException: / by zero' followed by the stack trace 'at teste.main(teste.java:6)'. The prompt 'robson@robson:~/Desktop\$' is visible with a cursor.

```
robson@robson:~/Desktop$ java -jar divisaoPorZero.jar
Exception in thread "main" java.lang.ArithmeticException: / by zero
    at teste.main(teste.java:6)
robson@robson:~/Desktop$
```

Fig

ura 2. Exemplo de exceção por divisão por zero

Tratando Exceções

Uma maneira de tentar contornar esses imprevistos é realizar o tratamento dos locais no código que podem vir a lançar possíveis exceções, como por exemplo, campo de consulta a banco de dados, locais em que há divisões, consulta a arquivos de propriedades ou arquivos dentro do próprio computador.

Para tratar as exceções em Java são utilizados os comandos try e catch.

Sintaxe:

```
try
{
    //trecho de código que pode vir a lançar uma exceção
}
catch(tipo_excecao_1 e)
{
    //ação a ser tomada
}
catch(tipo_excecao_2 e)
{
    //ação a ser tomada
}
catch(tipo_excecao_n e)
{
    //ação a ser tomada
}
```

Listagem 1. Sintaxe de uso d try-catch

Onde:

- `try{ ... }` - Neste bloco são introduzidas todas as linhas de código que podem vir a lançar uma exceção.
- `catch(tipo_excessao e) { ... }` - Neste bloco é descrita a ação que ocorrerá quando a exceção for capturada.

Exemplificando uma exceção

Imagine uma classe que tem um método principal `main` que tem como seu único objetivo alterar todas as letras de um frase para maiúsculas utilizando o método `toUpperCase()` da classe `String`, caso a frase esteja nula e se tente usar o método `toUpperCase()` na mesma será lançada uma exceção de `NullPointerException`.

Primeiro vamos ver como ficaria a tal classe sem a utilização do `try/catch`.

```
public class aumentaFrase {  
    public static void main(String args[])  
    {  
        String frase = null;  
        String novaFrase = null;  
        novaFrase = frase.toUpperCase();  
        System.out.println("Frase antiga: "+frase);  
        System.out.println("Frase nova: "+novaFrase);  
    }  
}
```

Listagem 2. Exemplo de código sem try-catch

Quando este código for executado, o mesmo lançará uma `NullPointerException`, como poder ser visto na saída do console quando executamos tal programa.

```
Exception in thread "main" java.lang.NullPointerException  
at aumentaFrase.main(aumentaFrase.java:15)
```

Listagem 3. Saída gerada pelo programa sem try-cach

Ou seja, o mesmo tentou acessar um atributo de um objeto que estava nulo. Para ajudar a melhorar a situação, deve-se usar o `try/catch`.

```
public static void main(String args[])  
{  
    String frase = null;  
    String novaFrase = null;  
    try
```

```

    {
        novaFrase = frase.toUpperCase();
    }
    catch(NullPointerException e) //CAPTURA DA POSSÍVEL exceção.
    {
        //TRATAMENTO DA exceção
        System.out.println("O frase inicial está nula,
        para solucionar tal o problema, foi lhe atribuido um valor
default.");
        frase = "Frase vazia";
        novaFrase = frase.toUpperCase();
    }
    System.out.println("Frase antiga: "+frase);
    System.out.println("Frase nova: "+novaFrase);
}

```

Listagem 4. Reformulação do código com try-catch

Quando este código for executado, o mesmo lançará uma `NullPointerException`, porém esta exceção será tratada desta vez, sendo a mesma capturada pelo `catch{}` e dentro deste bloco as devidas providências são tomadas. Neste caso é atribuído um valor default à variável `frase`. A saída deste programa seria a seguinte:

array4

Listagem 5. Saída do programa reformulado

Comando finally

Imagine a seguinte situação: foi aberta uma conexão com o banco de dados para realizar determinada ação, e no meio deste processo seja lançada alguma exceção, como por exemplo, `NullPointerException` ao tentar manipular um determinado atributo de um objeto. Neste caso seria necessário que mesmo sendo lançada uma exceção no meio do processo a conexão fosse fechada. Um outro exemplo bom seria a abertura de determinado arquivo para escrita no mesmo, e no meio deste processo é lançada uma exceção por algum motivo, o arquivo não seria fechado, o que resultaria em deixar o arquivo aberto.

Quando uma exceção é lançada e é necessário que determinada ação seja tomada mesmo após a sua captura, utilizamos a palavra reservada `finally`.

Sintaxe:

```

try
{
    //trecho de código que pode vir a lançar uma exceção
}
catch(tipo_excecao_1 e)
{

```

```

        //ação a ser tomada
    }
    catch(tipo_excecao_2 e)
    {
        //ação a ser tomada
    }
    catch(tipo_excecao _n e)
    {
        //ação a ser tomada
    }
    finally
    {
        //ação a ser tomada
    }

```

Listagem 6. Sintaxe de uso do bloco finally

Exemplo:

```

public class aumentaFrase {
    public static void main(String args[])
    {
        String frase = null;
        String novaFrase = null;
        try
        {
            novaFrase = frase.toUpperCase();
        }
        catch(NullPointerException e)
        {
            System.out.println("O frase inicial está nula, para
default.");
            frase = "Frase vazia";
        }
        finally
        {
            novaFrase = frase.toUpperCase();
        }
        System.out.println("Frase antiga: "+frase);
        System.out.println("Frase nova: "+novaFrase);
    }
}

```

Listagem 7. Programa aumetaFrase com bloco finally

Quando este código fosse executado, o mesmo lançaria uma `NullPointerException`, porém esta exceção será tratada desta vez, sendo a mesma capturada pelo `catch{}` e dentro deste bloco as devidas providências são tomadas. Neste caso é atribuído um valor default à variável `frase`. Neste exemplo,

mesmo o código lançando uma exceção durante a sua execução e a mesma sendo capturada pelo catch, uma determinada ação será tomada no bloco finally, neste caso tanto com a exceção ou não será executada a linha “ novaFrase = frase.toUpperCase();, tornando todas letras da frase maiúsculas. A saída deste programa seria a seguinte:

```
array4
```

Listagem 8. Saída do programa com bloco finally

Comandos throw e throws

Imagine uma situação em que não é desejado que uma exceção seja tratada na própria classe ou método, mas sim em outro que venha lhe chamar. Para solucionar tal situação utilizamos o comando throws na assinatura do método com a possível exceção que o mesmo poderá a vir lançar.

Sintaxe:

```
tipo_retorno nome_metodo() throws tipo_exceção_1, tipo_exceção_2,
tipo_exceção_n
{
...
}
```

Listagem 9. Sintaxe de declaração de método com definição de exceções

Onde:

- tipo_retorno – Tipo de retorno do método.
- nome_metodo() - Nome do método que será utilizado.
- tipo_exceção_1 a tipo_exceção_n – Tipo de exceções separadas por virgula que o seu método pode vir a lançar.

Exemplo:

```
public class TesteString {
    private static void aumentarLetras() throws NullPointerException //
lançando excessão
    {
        String frase = null;
        String novaFrase = null;
        novaFrase = frase.toUpperCase();
        System.out.println("Frase antiga: "+frase);
        System.out.println("Frase nova: "+novaFrase);
    }

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

```

        try
        {
            aumentarLetras();
        }
        catch (NullPointerException e)
        {
            System.out.println("Ocorreu um
                                NullPointerException ao executar o método
aumentarLetras() "+e);
        }
    }
}

```

Listagem 10. Definição de exceções que um método pode gerar

Neste exemplo será lançada uma exceção no método `aumetarLetras()`:

```
private static void aumentarLetras() throws NullPointerException
```

Listagem 11. Definição da exceção gerada pelo método `aumentarLetras`

E o mesmo será tratado no método `main()`.

```

...
try
{
    aumentarLetras();
}
catch (NullPointerException e)
{
    System.out.println("Ocorreu um NullPointerException ao
                        executar o método aumentarLetras() "+e);
}
...

```

Listagem 12. Aplicação da exceção definida

Saída:

```
Ocorreu um NullPointerException ao executar o método
aumentarLetras() java.lang.NullPointerException.
```

Listagem 13. Saída do programa atualizado

Agora imagine o caso em que seja necessário lançar uma exceção padrão ao invés de uma específica. Para resolver este problema, utilizamos o comando `throw` dentro do bloco `catch` que desejamos converter a exceção.

Sintaxe:

```

try
{
    //...
}
catch (tipoExcessão_1 e)
{
    throw new novoTipoExcecao (e);
}

```

Listagem 14. Sintaxe de uso do comando throw

Onde:

- tipoExcessão_1 e – Tipo de exceção que pode ser capturada pelo bloco catch.
- NovoTipoExcecao – Tipo de exceção que será lançada.

Exemplo:

```

public class TesteString {
    private static void aumentarLetras() throws Exception //lançando
    exceção
    {
        String frase = null;
        String novaFrase = null;
        try
        {
            novaFrase = frase.toUpperCase();
        }
        catch (NullPointerException e)
        {
            throw new Exception(e);
        }
        System.out.println("Frase antiga: "+frase);
        System.out.println("Frase nova: "+novaFrase);
    }
    public static void main(String args[])
    {
        try
        {
            aumentarLetras();
        }
        catch (Exception e)
        {
            System.out.println("Ocorreu uma exceção ao
            executar o método aumentarLetras() "+e);
        }
    }
}

```

Listagem 15. Exemplo de uso do comando throw

Neste exemplo será lançada uma `NullPointerException` e a mesma será convertida para `Exception` e relançada como `Exception` no método `aumentarLetras()` e, por fim, a mesma é tratada no método `main()`.

Saida:

```
Ocorreu uma exceção ao executar o método aumentarLetras()  
java.lang.Exception: java.lang.NullPointerException
```

Listagem 16. Saída do programa atualizada

Criando exceções

Assim como qualquer objeto, em Java também é possível criar suas próprias exceções. Imagine um cenário em que nenhuma exceção existente faça sentido para ser lançada por você.

Por exemplo, imagine que por algum motivo você precisa que uma exceção seja lançada quando a letra B ou b não existe e determinada frase, como não existe nenhuma exceção específica para este caso será necessário criar uma exceção.

Criando uma exceção para ser lançada toda vez que uma letra B ou B não é encontrada em uma determinada frase.

```
public class SemLetraBException extends Exception {  
    @Override  
    public String getMessage() {  
        return "Não existe letra B em sua frase";  
    }  
}
```

Listagem 17. Exemplo de exceção customizada

Toda exceção criada deve estender `Exception`, neste exemplo foi sobrescrito o método `getMessage()`, que é exibida no prompt toda vez que a exceção é lançada.

Utilizando a exceção

Abaixo segue um exemplo que é utilizada a exceção criada acima.

```
public class TesteExcecao {  
    public static void main(String args[]) throws SemLetraBException  
    {  
        String frase = "Sou um teste!";  
        if(!frase.contains("b") || !frase.contains("B"))  
            throw new SemLetraBException();  
    }  
}
```


Listagem 18. Utilizando a exceção customizada

Quando o programa acima fosse executado, uma exceção do tipo `SemLetraBException()` seria lançada. Abaixo está a saída exibida no prompt:

```
Exception in thread "main" SemLetraBException: Não existe letra B ou  
b em  
sua frase at TesteExcecao.main(TesteExcecao.java:8)
```

Listagem 19. Saída do método com a mensagem da exceção customizada

Conclusão

Como foi possível ver ao longo do artigo, o tratamento de exceções em Java é bem simples e sua manipulação pode ser feita de acordo com o que o programador deseja, desde tratá-la no próprio método ou lançá-la para ser tratada em um método que venha chamar o método que lança a mesma.

Exceções: Boas Práticas, Más Práticas

O problema

O mecanismo de exceções em Java criou o conceito de exceção verificada. Este conceito implica que qualquer método que chame outro que lance uma exceção verificada é obrigado, pelo compilador, a dar um tratamento à exceção. O compilador Java é muito bom mas não chega a ser inteligente ao ponto de saber se o tratamento que você deu à exceção é realmente uma solução ou uma enrolação.

Muitas vezes se torna chato ter que trabalhar a todo o momento com exceções verificadas. É usual ter que trabalhar com `SQLException` ou `IOException` que são muito comuns e pouco específicas. Frente a exceções como estas rapidamente o programador desiste de tratar tantas exceções e aí começa o problema. Por convicção, inexperiência ou omissão, o programador acaba desconsiderando a exceção que aconteceu muitas vezes fazendo a exceção, simplesmente, desaparecer. Obviamente isso é atirar pela janela um dos mecanismos mais importantes e avançados do Java: o tratamento de exceções verificadas.

No [artigo anterior desta série](#) vimos como os conceitos de exceção e tratamento de exceção foram incorporados na linguagem Java. Veremos neste artigo um conjunto de regras para tirar o máximo proveito desse mecanismo, sem que ele se torne onnipresente e chato.

O quê , onde e porquê

Uma exceção é um evento que ocorre quando um problema acontece. Assim, a exceção carrega consigo informações sobre qual é o problema, onde aconteceu o problema, e se possível, porque o problema aconteceu. Quanto melhor a exceção responder a estas três perguntas mais útil ela será.

O quê

O que deu errado é transmitido no nome da própria classe que representa a exceção. Por exemplo, `OutOfMemoryError` significa que não há mais memória disponível; `NumberFormatException` significa que o formato do número é incorreto; `FileNotFoundException` significa que o arquivo não foi encontrado. O nome da exceção é muito importante pois identifica o problema em si mesmo. É comum deixar a palavra `Exception` (ou `Error` se for um erro) no fim para deixar claro que esta classe será tratada de forma especial pelo compilador.

O Porquê

A razão do problema é explicada na mensagem contida na própria exceção e que pode ser obtida com `getMessage()`. Essa explicação é textual e normalmente em inglês. É possível internacionalizar a mensagem, mas isso só é feito em casos em que a mensagem tem que ser apresentada ao usuário final.

O Onde

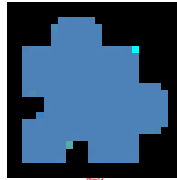
O onde deu errado é respondido pelo *stack trace*, o rastro, que revela em qual linha de qual classe o problema aconteceu. O *stack trace* mostra a hierarquia de todas as exceções que derivaram da exceção original. Caso seja necessário podemos navegar pelo *stack trace* para saber se um certo tipo de exceção aconteceu, ou simplesmente qual foi a exceção que originou o problema. Normalmente a ocorrência da exceção é acompanhada do número da linha e do nome da classe onde ela foi lançada. Isso é uma informação vital, extremamente útil quando queremos saber onde a exceção aconteceu.

Taxionomia de uma Exceção

A classe `Throwable` é, em Java, a classe mãe de todas as exceções. Ela oferece alguns métodos que nos permitem conhecer um pouco mais do problema que aconteceu. Eis os mais relevantes:

```
public class Throwable {  
  
    public Throwable(String message, Throwable cause); // construtor  
    public Throwable getCause(); // outro Throwable que foi a causa deste  
    public String getMessage(); // mensagem relativa à causa  
    public void printStackTrace(PrintStream s);  
    public void printStackTrace(); ... }  
}
```

Código 1: Métodos básicos da classe `Throwable`



REPORT THIS AD

O construtor será importante quando criarmos nossa própria classe de exceção. O método `getCause()` retorna a exceção, se alguma, que deu origem a esta. Este método retorna a exceção passada no construtor. Isto é importante quando encapsulamos uma exceção em outra. O método `getMessage()` retorna a mensagem passada no construtor. Os métodos `printStackTrace()` são importantes para passar a informação do rastro para uma hierarquia legível por um humano. São extremamente úteis para encontrar o problema, mas normalmente usados no lugar errado.

Tratando de Exceções

O tratamento de exceções tem algumas regras impostas pelo Java:

1. Todas as exceções devem ser manipuladas (handled). Manipular, aqui, significa fazer uma de duas coisas: capturar a exceção com try-catch ou declarar a exceção com throws
2. Métodos que sobre-escrevem outros métodos não podem declarar o lançamento de mais exceções que o método que está sendo sobre-escrito (métodos filhos não podem lançar mais exceções que os pais)
3. O compilador Java verificará a conformidade do código com as duas regras acima sempre que a exceção for do tipo verificada

Manipular a exceção não significa necessariamente tratá-la. Para tratarmos uma exceção temos que entender a resposta a cada uma das seguintes perguntas:

- Aconteceu uma exceção?
- O que podemos fazer para resolver o problema?
- Como indicar que não conseguimos resolver o problema?

Como descobrir se aconteceu uma exceção?

Quando uma exceção acontece, ela acontece porque o método atual a lançou ou porque um outro método chamado a lançou. Se o método atual a lançou é porque ele não consegue resolver o problema. Neste caso passamos a exceção ao método superior. Se a exceção veio de um outro método temos que saber, com antecedência, se a podemos tratar.

Para isso é importante consultar a documentação do métodos que o nosso método está chamando. Nessa documentação devemos encontrar a lista de exceções lançadas assim como explicações sobre as condições em que elas acontecem. Analisando a documentação poderemos ter uma idéia se podemos ou não resolver o problema. Algumas vezes a documentação é escassa ou mal construída e não nos informa o suficiente. Em casos poderemos fazer pequenos testes tentando forçar exceções e teriam algumas conclusões simples. Contudo, o normal, é aceitar que não sabemos tratar as exceções vindas do método. Se não podermos resolver o problema faremos o mesmo que antes: passar a exceção ao método anterior.

O QUE PODEMOS FAZER PARA RESOLVER O PROBLEMA?

Cada caso é um caso e não há nada como a experiência para nos guiar. O que podemos fazer é muito diversificado. Se sabemos que podemos resolver a exceção é porque sabemos o que fazer para a resolver. Por exemplo, se um arquivo não for encontrado e obtivermos uma `FileNotFoundException` podemos criar um vazio, buscá-lo em outra localização ou simplesmente abortar todo o processo.

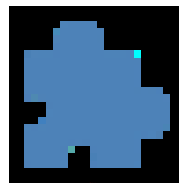
COMO INDICAR QUE NÃO CONSEGUIMOS RESOLVER O PROBLEMA?

Se ao tentar resolver um problema encontramos um outro problema, ou não conseguimos resolver o problema original então temos que indicar isso ao método chamador. A forma de indicar isso é lançando uma exceção. Podemos relançar a exceção original que tentámos – em vão – tratar ou enviar a exceção que apareceu depois. No caso de enviarmos a segunda exceção a primeira deve pertencer ao seu rastro (stack trace). O ideal é retornar a primeira exceção se não conseguirmos resolver a segunda.

Boas práticas de tratamento de exceções

Porque o tratamento de exceções é muitas vezes frustrante o programador tende a se livrar do problema a qualquer custo. Obviamente esta não é a forma certa de lidar com exceções. Eis algumas das diretivas que sempre deve seguir quando estiver diante de uma exceção a ser tratada.

Não capture o que você não pode segurar



[REPORT THIS AD](#)

Quando um método que você está invocando lançar uma exceção, se você não sabe o que fazer com ela, simplesmente não faça nada. Deixe para quem entende. Se todos os métodos aplicarem esta regra, você não precisa ficar enchendo seu código com tratamentos imprestáveis e chatos. Mas atenção, também não seja irresponsável achando que sempre existirá um outro método que cuide de seu problema. O método tem que conhecer o seu lugar no sistema. Se o método não tem a quem passar a batata quente então não lhe resta alternativa senão tratar o problema, ou pelo menos, reportar que o problema aconteceu. A importância desta regra é minimizar o código que trata exceções impossíveis de resolver, centralizando ações necessárias nesses casos, como apresentar uma mensagem ao usuário, reportar o evento para um registro (log) ou educadamente terminar a aplicação.

Cumprir esta regra não é tão simples quanto parece. A forma fácil é simplesmente declarar que o método chamador também lança a exceção do método chamado. No exemplo, deixámos `leArquivo()` lançar `IOException` porque `read()` lança essa exceção e não sabemos como tratá-la. Contudo, se este método fizesse parte de uma interface que não pode lançar `IOException` teríamos que excapsular esse tipo de exceção em outra que a interface

define. Se não define nenhuma teríamos que encapsulá-la numa classe derivada de `RuntimeException`.

```
public void leArquivo(File arquivo) throws IOException{  
  
    read(new FileInputStream(arquivo)); // chama função auxiliar lança uma IOException genérica  
}
```

Código 2: Declare as exceções que não pode tratar com `throws`

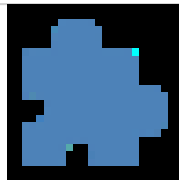
Seja específico

Quando você tiver que lançar uma exceção, seja específico. Não lance exceções genéricas que significam tudo e nada ao mesmo tempo. Não lance diretamente `Exception` ou `RuntimeException`. Use a exceção que melhor detalha o problema. Se nenhuma existir, crie a sua própria classe de exceção que seja específica o bastante. Apenas específica o bastante pois não é bom ser específica demais. Por exemplo, não pense em lançar uma exceção se o problema aconteceu antes do almoço e outra se aconteceu depois do almoço. Esse tipo de informação será inútil se a exceção não for registrada para posterior consulta, mas se for, o próprio mecanismo de registro poderá injetar essa informação. Olhando o nosso exemplo, podemos ser mais específicos:

```
public void leArquivo(File arquivo) throws IllegalArgumentException,  
MyIOException{  
  
    if (arquivo == null){  
        throw new IllegalArgumentException("parametro arquivo não pode ser nulo");  
    } else if (!arquivo.exists()){ // o arquivo não existe  
        throw new MyFileNotFoundException(arquivo); // criará a mensagem a partir dos dados de arquivo  
    } else if (!arquivo.isFile()){ // não é um arquivo  
        throw new MyNotAFileException(arquivo); // criará a mensagem a partir dos dados de arquivo  
    }  
    // chama função auxiliar  
    try {  
        read(new FileInputStream(arquivo)); // este método lança IOException genérica  
    } catch (FileNotFoundException e) {  
        //já verificamos que o arquivo existe no if anterior, então  
        // se esta exceção acontecer significa que não temos privilégios para ler o arquivo  
        throw new NoReadPriviledgeException()  
    } catch (IOException e){  
        // no caso genérico não ha muito a fazer, a não ser, possivelmente, encapsular a exceção  
        throw new MyIOException(e);  
    }  
}
```

```
}  
}  
}
```

Código 3: Como fazer – seja específico



REPORT THIS AD

`MyIOException` é uma `RuntimeException` com a função específica de encapsular `IOException`. Usamos `IllegalArgumentException` porque é uma exceção que já existe (sempre dê preferência a usar uma exceção da API padrão) e que especifica o problema o bastante. Todas as outras exceções são filhas mais específicas de `MyIOException`.

A importância de ser específico é aumentar o detalhe sobre o problema, para que seja mais fácil identificá-lo e resolvê-lo independentemente de outros. Veremos mais tarde como criar sua própria classe de exceção.

Não deixe para os outros o que você pode lançar primeiro

Detalhe exatamente o que o seu método faz e o que o impediria de completar essa tarefa. Se alguma das condições de impedimento está presente lance *imediatamente* uma exceção explicando porque o método não pode fazer o seu trabalho. Não espere até que um outro método auxiliar que você vai usar lance uma exceção incompreensível quando você pode lançar uma muito mais detalhada. No exemplo anterior o código testa primeiro um conjunto de condições e apenas se tudo estiver bem que o código principal é executado.

```
public void leArquivo(File arquivo) throws IOException{  
  
    if (arquivo == null){  
        throw new IllegalArgumentException("parametro arquivo não pode ser nulo");  
    } else if (!arquivo.exists()){ // o arquivo não existe  
        throw new MyFileNotFoundException(arquivo); // criará a mensagem a partir dos dados de arquivo  
    } else if (!arquivo.isFile()){ // não é um arquivo  
        throw new MyNotAFileException(arquivo); // criará a mensagem a partir dos dados de arquivo  
    }  
    ...  
}
```

Código 4: Como fazer – não deixe para os outros o que você pode lançar primeiro

A importância desta regra é deixar o rastro o mais curto possível para que seja mais fácil saber onde, e porquê, o problema aconteceu. Quanto mais cedo você lançar a exceção, mais perto fica o onde, e mais claro fica o porquê. Nem sempre as exceções acontecem porque o método

fez algo errado. Muitas vezes acontecem porque o método recebeu os parâmetros errados. Testar as pré-condições do método é tão (ou mais) importante que o método em si mesmo.

Más práticas

Baseados nas idéias anteriores vamos revisar alguns códigos comuns que infelizmente são más práticas, ou seja, nunca faça isto.

Exceções e loops

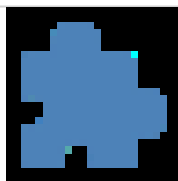
Quando executamos um código que pode lançar exceções em um loop (for, while, do) temos que ter atenção se a exceção invalida todo o loop ou apenas uma iteração. Se soubermos que o problema apenas invalida uma iteração deveremos roedar o código com o `try-catch` e deixar as demais iterações continua (se necessário memorizando as exceções encontradas numa coleção para que possam ser tratadas depois que o loop acabar).

Log e lança

É um erro comum contruir código semelhante a algum destes:

```
catch (SQLException e) {  
  
    LOG.error("Blablabla", e); throw e;  
}  
// ou  
catch (IOException e) {  
    LOG.error("Blablabla", e);  
    throw new MinhaException("Blablabla outra vez", e);  
} // ou  
catch (Exception e) {  
    e.printStackTrace();  
    throw new MinhaException("Blablabla", e);  
}
```

Código 5: Log e lança



[REPORT THIS AD](#)

Provavelmente essa exceção relaçada será enviada para o log (registro) em outro ponto do sistema, então não há razão para fazer o log aqui. Muito menos imprimir o rastro no console. Afinal, o console pode nem ser visível neste ponto.

Lançar Exception

Código com esta assinatura:

```
public void algumMetodo() throws Exception
```

Código 6: Lançando *Exception*

Significa não dar nenhuma informação ao método chamador do tipo de exceções que podem acontecer. Basicamente isto significa: “posso lançar qualquer coisa”. Isso viola a regra de ser específico. Em opção deve-se explicitar qual ou quais as exceções que são lançadas. Mas cuidado com lançar a casa pela janela.

Lançando a casa pela janela

Código com este tipo de assinatura:

```
public void algumMetodo() throws EstaException, AquelaException,  
UmaOutraException, OutraPossivelException, EAindaMaisUmaException {
```

Código 7: Lançando *Exception*

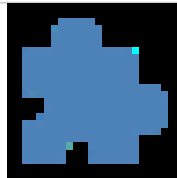
Onde várias exceções podem acontecer e todas são registradas. Isto é específico, mas polui a interface do método além de ser um pesadelo para o método que chamar este. A opção é usar um conjunto menor, e mais específico, de exceções que encapsulem aquelas. Exceções não verificadas podem ser declaradas, mas o ideal é só declará-las na documentação javadoc.

Retenção da Exceção

Reter a exceção significa que após a capturar ela não é relançada, em nenhuma forma. Simplesmente se sequestra a exceção sem nunca a lançar de volta ou dar um tratamento apropriado. Não capture o que não pode tratar, diz a regra. Enviar para o log, não é tratar. Imprimir o *stack trace* não é tratar.

```
try { algumMetodo(); } catch (Exception e) { LOG.error(“metodo falhou”, e); }
```

Código 8: Retenção errada de *Exception*



Apenas existe uma ocasião onde você pode sequestrar a exceção: quando chegou no nível mais próximo ao usuário. Por exemplo, se a aplicação será abortada logo a seguir ou uma

mensagem será mostrada na interface do usuário sem abortar a aplicação. Apenas destes casos você pode sequestrar a exceção.

```
try { algumMetodo(); } catch (Exception e) {  
  
    LOG.error("metodo falhou", e); System.exit(-1); }  
//ou  
try { algumMetodo(); }  
catch (Exception e) {  
    LOG.error("metodo falhou", e);  
    UI.showErrorMessage(e);  
}
```

Código 9: Retenção correta de `Exception`

É importante entender que nestas ocasiões você não está driblando as regras de tratamento. No primeiro caso você está admitindo a derrota e terminando a sua aplicação porque não foi possível tratar o problema em nenhum estágio do sistema. No segundo caso você está informando o usuário e portanto dando-lhe a hipótese de tratar o problemas por outros meios alheios à sua aplicação. Isto é válido. Pense por exemplo numa aplicação que acessa a internet e avisa o usuário quando a conexão não foi possível.

Eliminação do rastro da Exceção

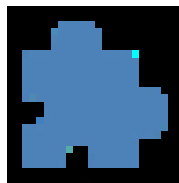
Ao encapsular a exceção em outra, despresas-se o rastro invocando `e.getMessage()` em vez de passar no construtor a causa da exceção que estamos criando. É importante manter o rastro pois é nele que se inclui a informação de onde e porquê o problema aconteceu.

```
try { algumMetodo(); } catch (Exception e) {  
  
    throw new MinhaException("Blablalbla: " + e.getMessage());  
}
```

Código 10: Eliminação do rastro de `Exception`

Contudo existem casos onde é necessário eliminar o rastro. Por exemplo, quando estamos encapsulando uma exceção não serializável numa que é serializável. Nesse caso a única forma de manter informação relativa ao problema é usando a mensagem da exceção original, que se tentarmos colocar a exceção original no rastro causaremos uma outra exceção na hora de serializar a exceção. Este mecanismo é importante principalmente em sistema distribuídos.

Criando suas próprias exceções



REPORT THIS AD

Agora já sabe como tratar exceções. Se esteve com atenção deve ter notado que muitas das vezes a forma de tratar uma exceção é encapsulá-la em uma outra. Normalmente numa criada por si.

Preciso mesmo de criar uma classe de exceção?

A primeira coisa importante a fazer na hora de criar a sua própria exceção é perguntar-se se precisa mesmo criar uma exceção própria. Eis alguns casos em que não é proveitoso criar a sua própria exceção:

- já existe uma exceção nas bibliotecas-padrão que representam o problema que quer tratar.
- A sua exceção não vai adicionar nenhuma informação suplementar é caracterização do problema.

Eis alguns casos onde você terá que criar a sua própria exceção:

- já existe uma exceção nas bibliotecas-padrão que representam o problema que quer tratar, mas é verificada e você precisa de uma exceção não-verificada
- A sua exceção vai adicionar informação suplementar é caracterização do problema para enriquecer possíveis métodos de tratamento, e/ou registros.

Verificada ou não verificada, eis o dilema

Quanto tiver que decidir entre usar uma exceção verificada e uma não verificada não se pergunte: "qual ação o chamador pode tomar para resolver este problema?" e sim "o programador tem *sempre* que saber que este problema existe?". Se a resposta for afirmativa crie uma exceção verificada.

Lembre-se que exceções verificadas são especialmente uteis quando o seu sistema conversa com outros fora do seu ambiente como sistemas de arquivos, recursos remotos, outros sistemas como bancos de dados, ou sistemas embutidos no seu, como plugins. Se você está construindo um framework é provável que precise criar exceções verificadas. Se é apenas um método nas suas classes de negócio, não. Se estiver construindo um framework pense também se a exceção contém informação e com que frequência o programador poderá resolver o problema. Se necessário alterne entre exceções não verificadas e erros.

Camadas e exceções

Uma das maiores dificuldades ao lidar com exceções é decidir como elas se propagarão entre as camadas, andares e nodos do sistema. O bom *design* de camadas resulta num conjunto de interfaces que o cliente da camada usará para invocar as suas funcionalidades. Essas interfaces devem apenas lançar exceções específicas da camada. Um bom exemplo desta arquitectura são as APIs de IO e de JDBC. Elas apenas lançam exceções decorrentes da sua actividade. Como vimos não é bom lançar apenas um tipo de exceção, mas estas APIs

demonstram o princípio de encapsular todas as exceções que acontecem dentro da camada, como exceções próprias da camada. Além disso são API para comunicar com outros sistemas, logo suas exceções devem ser verificadas. O cliente da camada deverá analisar a exceção que recebe, e traduzir essas exceção para um dos tipos de exceção da sua camada. Isto só deverá ser feito se a camada não sabe como resolver o problema apresentado pela camada inferior.

Empacotamento e visibilidade

Ao seguir a regra anterior temos normalmente de criar uma hierarquia de exceções na camada. As classes dessa camada fazem parte de um pacote específico. As exceções da camada devem estar nesse mesmo pacote. As exceções têm que ser públicas para que possam ser capturadas por outras partes do sistema contidas em outros pacotes. Tente, tanto quanto possível, reduzir a visibilidade dos construtores para o nível de pacote. Esta regra é especialmente válida se a exceção carrega informações que podem ser apenas obtidas dentro da camada. Esta regra não se aplica a exceções que fazem parte de uma API genérica ou extensível. A API de IO é de uso genérico e a API JDBC é extensível, por isso elas apresentam exceções públicas com construtores públicos. Mas na sua aplicação as exceções de negócio devem estar fortemente acopladas com as regras de negócio que estão implementadas na camada de negócio. Nesse caso não faz sentido deixar que outras camadas criem e lancem exceções de negócio e como tal seus construtores podem ser de nível de pacote. A visibilidade do construtor não é um detalhe crítico da implementação de exceções, mas pode ser uma ferramenta para simplificar o seu uso, o seu escopo e o seu entendimento. É mais fácil tornar um método público do que torná-lo protegido.

Resumo

Não é difícil ter um bom tratamento de exceções seguindo algumas directivas. Muitas vezes é necessário criar seu próprio pacotes de exceções sobretudo se está construindo frameworks utilitários ou implementando camadas mais baixas. Veremos num próximo artigo um conjunto de padrões e classes que o auxiliarão a fazer o correto tratamento de todas as exceções do seu sistema.

ocê sabe o que é um teste de software e quais são os principais tipos de teste de software? Neste artigo você vai tirar todas as suas dúvidas.

Por mais que se planeje a construção de um software, erros são passíveis de ocorrer. Pode ser um bug num game, uma falha que feche um programa ou um erro que impossibilite você salvar um arquivo.

Quem já passou por esse tipo de situação sabe como é chato quando ficamos na mão por culpa de um programa com falhas. O teste de

software serve justamente para tentar encontrar possíveis erros que um programa recém-desenvolvido possa apresentar, de modo a conseguir corrigi-lo antes que seja lançado no mercado, ficando disponível para uso do público.

O teste de software geralmente é a última etapa na construção de um programa, visando checar o seu nível de qualidade. Os defeitos que um teste busca identificar incluem erro de compatibilidade, de algum algoritmo, de requisitos que não podem ser complementados, limitação de hardware etc. A lista é grande e aumenta com o tamanho do programa.

Quais os tipos de testes de software?

Existem diferentes tipos de testes que podem ser aplicados num software para identificar suas falhas, sendo as principais:

- **Teste da caixa branca** – utiliza o aspecto interno do programa/sistema, o código fonte, para avaliar seus componentes. Ele também é conhecido como teste orientado à lógica ou estrutural. Podem ser analisados itens como: fluxo dos dados, condição, ciclos etc. Na hora de implementá-lo é preciso verificar a criticidade, a complexidade, a estrutura e o nível de

qualidade que se pretende obter do programa, envolvendo confiança e segurança;

– **Teste da caixa preta** – diferente do teste anterior, que prioriza os aspectos internos, o teste da caixa preta verifica aspectos externos. Os requisitos funcionais do sistema são avaliados. Não se observa o modo de funcionamento, sua operação, tendo como foco as funções que deverão ser desempenhadas pelo programa. Desse modo, avalia-se se um grupo de entrada de dados resultou nas saídas pretendidas, levando-se em consideração a especificação do programa. Ou seja, o que se esperava que o software deveria fazer. É conhecido também como técnica funcional;

– **Teste da caixa cinza** – esse tipo de teste une os dois anteriores, por isso o termo “cinza”. Avalia tanto os aspectos internos quanto os externos, de entrada e saída. Pode utilizar-se de engenharia reversa;

– **Teste de regressão** – esse consiste em realizar testes a cada versão de um software, onde se modificam-se funcionalidades. Desse modo, evita-se que erros que foram corrigidos antes no software antes voltem a aparecer na hora de se incrementar algo novo a ele.

– **Teste de unidade** – testa-se unidades menores de um software, de modo isolado, para ver se todas funcionam adequadamente;

– **Teste de integração** – depois das unidades testadas, realiza-se uma verificação se elas funcionam juntas, integradas. Pode ocorrer delas

apresentarem incompatibilidades ao funcionarem em conjunto, mesmo após terem sido aprovadas no teste de unidade;

– **Teste de carga** – esse teste é feito para avaliar os limites de uso do software, o quanto ele suporta em volume de informações, tráfego etc. sem que apresente erros;

– **Teste de usabilidade** – esse teste é feito por um pequeno grupo de usuários para ver se o software satisfaz as suas necessidades. Nesse teste analisa-se como o usuário usa o sistema, verificando onde ele tem mais dificuldade. Ouve-se também suas impressões, porém é preciso confrontá-las com as observações do avaliador;

– **Teste de stress** – aqui leva-se o software ao seu limite de potência e funcionamento, para mais ou para menos, de modo a avaliar em qual ponto ele deixa de funcionar adequadamente. Isso é feito para verificar se suas especificações máximas ou mínimas de uso estão corretas.

O que é um plano de teste de software?

Um plano de teste é feito para colaborar com o desenvolvimento de um software. É por meio desse plano que os componentes técnicos,

funcionais, estruturais etc. serão verificados e validados, de modo a garantir o bom funcionamento do programa junto ao usuário final. Sendo assim, **um plano de teste de software tem como foco garantir a confiabilidade e segurança de um software**, identificando possíveis erros e falhas durante a sua confecção, ou mesmo depois.

Ele deve ser planejado em conjunto com a proposta do software, sendo aplicado em cada etapa do projeto e não somente no final.

Ferramentas de teste de software

Para garantir a qualidade de um programa, as desenvolvedoras realizam testes nele. Isso é necessário para que falhas sejam detectadas antes que o software seja colocado no mercado. Sabe aquele programa que vive travando, não roda direito ou que faz o PC ficar lento? Esse, provavelmente, deve ter passado pelo processo de desenvolvimento com essas imperfeições. Então, para evitar que isso aconteça, as empresas contratam profissionais (os testadores de software ou analistas de testes) para identificarem esses problemas e relatarem para que os desenvolvedores os corrijam. Mas, para fazer isso eles precisam realizar uma bateria de testes diferentes, que envolvem desde análise da estrutura interna do software até a avaliação da interface.

O que são ferramentas de teste de software?

Para que esses testes possam ser realizados de modo mais rápido e com maior abrangência, existem ferramentas que automatizam alguns deles ou auxiliam na execução de outros. Essas são as ferramentas de teste de software.

O que é e o que faz um testador de software?

Com a expansão do mercado de tecnologia da informação (TI), uma nova profissão surgiu e tem crescido nos últimos anos: o testador de software ou analista de testes. Esse profissional é contratado para encontrar erros, falhas, bugs e outros tipos de problemas que não foram detectados durante a confecção de um software.

O mercado para esse tipo de profissão é amplo. Abrange desde a prestação de serviços de testes de softwares para programas gerenciais até aplicativos de smartphones voltados para o público. E a expectativa é de que ele fique cada vez maior, à medida em que clientes de

desenvolvedoras de softwares passam a solicitar a avaliação desse profissional nos programas encomendados.

Automação de testes de software

Num mundo cada vez mais interligado pela tecnologia, os planos de testes de softwares têm um peso importante, pois muitos negócios dependem de que esses estejam funcionando corretamente.

Qualquer falha num programa de gerenciamento financeiro pode acarretar prejuízos grandes em termos monetários. Um erro num software de um equipamento médico pode custar a vida uma pessoa ou dificultar o atendimento a alguém que precisa.

E não é só os casos mais extremos que precisam de testes de software de qualidade, pois uma simples falha de um programa de uso comum que ocorra com frequência pode fazer com que usuário dele abra queixa ou não o recomende para outros usuários, migrando para a concorrência. Quem perde é quem desenvolveu o software.

Processo de Teste de Software

O Processo de Testes de Software representa uma estruturação de etapas, atividades, artefatos, papéis e responsabilidades que buscam a padronização dos trabalhos e ampliar a organização e controle dos projetos de testes.

O Processo de Teste, como qualquer outro processo deve ser revisto continuamente, de forma a ampliar sua atuação e possibilitar aos profissionais uma maior visibilidade e organização dos seus trabalhos, o que resulta numa maior agilidade e controle operacional dos projetos de testes.

Etapas 1: Planejamento dos Testes

Esta etapa caracteriza-se pela definição de uma proposta de testes baseada nas expectativas do Cliente em relação à prazos, custos e qualidade esperada, possibilitando dimensionar a equipe e estabelecer um esforço de acordo com as necessidades apontadas pelo Cliente.

Dinâmica das Macro-Atividades

Este diagrama representa a seqüência das “macro-atividades” a serem executadas na etapa de “Planejamento dos Testes”.

Detalhamento das Macro-Atividades

Esta lista representa o conjunto de atividades que deverão ser executadas para que cada macro-atividade seja considerada finalizada, funcionando como um “check-list” de execução da etapa de “Planejamento dos Testes”.

Estudo do Projeto:

- Estudar as modificações solicitadas pelo Cliente (novos requisitos);
- Estudar as modificações de arquiteturas dos aplicativos;
- Estudar as lições aprendidas dos Projetos Anteriores;
- Avaliar expectativas de custos, prazos e qualidade exigidas pelo Cliente;
- Avaliar os riscos envolvidos nos Projetos e seus impactos neste processo;

Avaliação de Impacto:

- Avaliar se o projeto exige a criação de casos de testes “progressivos”;
- Avaliar se o projeto exige modificações em casos de testes “regressivos”
- Avaliar se o projeto exige adequações na automação dos testes;
- Avaliar se o projeto exige adequação nas atuais ferramentas empregadas;
- Avaliar se o projeto exige a aquisição/construção de novas ferramentas;
- Avaliar se o projeto exige modificações na estruturação do ambiente;
-

Análise Interna de Esforço

- Levantar métricas históricas para auxiliar na elaboração das estimativas de esforço;
- Estimar esforço interno para absorção dos impactos da Arquitetura dos Testes;
- Demonstrar esforço externo para absorção dos impactos da Arquitetura dos Testes;

Análise Externa de Esforço:

- Avaliar disponibilidade de espaço físico e infra-estrutura para os Terceiros;
- Especificar as necessidades de adequações que serão repassadas a Terceiros;
- Especificar métricas de qualidade e produtividades esperadas;
- Especificar SLA's de serviço e multas contratuais;
- Estabelecer concorrência e obter a melhor proposta (opcional);

- Receber Proposta de Trabalho (Cronograma, Prazos e Custos da Terceirização);
- Definição de Cenários Possíveis (Duração, Esforço, Custo e Qualidade):
- Levantar Lista de Projetos em Andamento e a serem Iniciados;
- Avaliar a disponibilidade de recursos internos para alocação no Projeto;
- Identificar Cenários Diversos (Terceirização, Redução de Escopo, Repriorização de Projetos);
- Definir Cronograma-Macro para cada cenário identificado;
- Definir Riscos para cada cenário identificado e Planos de Ação Esperados;
- Estabelecer Propostas e Aguardar aprovação da Diretoria;
- Aprovação do Planejamento:
- Obter o Aceite das Propostas de Cenários Aprovados pela Diretoria;
- Obter o Aceite de uma das Propostas pelo Cliente;
- Divulgar do Cenário Aprovado do Projeto aos colaboradores e terceiros;
- Obter a Assinatura do CONTRATO-MESTE e elaborar os ANEXOS; (no caso de terceirização)
- Alocar Espaço Físico dos Terceiros; (no caso de terceirização)
- Comunicar a Finalização da Etapa de Planejamento dos Testes; (externo)
- Definição das Responsabilidades
- Neste diagrama, está a representação dos papéis e responsabilidades para cada grupo de atividades envolvido na etapa de “Planejamento dos Testes”.

Definição de Cenários Possíveis (Duração, Esforço, Custo e Qualidade):

- Levantar Lista de Projetos em Andamento e a serem Iniciados;
- Avaliar a disponibilidade de recursos internos para alocação no Projeto;
- Identificar Cenários Diversos (Terceirização, Redução de Escopo, Repriorização de Projetos);
- Definir Cronograma-Macro para cada cenário identificado;
- Definir Riscos para cada cenário identificado e Planos de Ação Esperados;
- Estabelecer Propostas e Aguardar aprovação da Diretoria;

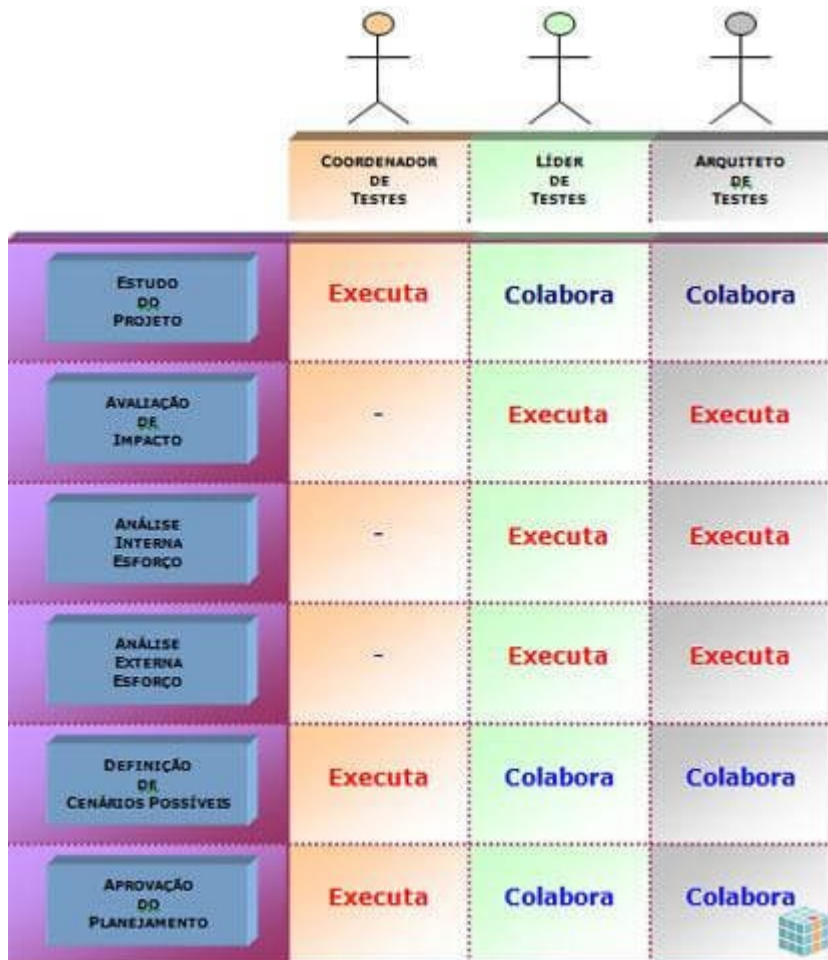
Aprovação do Planejamento:

- Obter o Aceite das Propostas de Cenários Aprovados pela Diretoria;
- Obter o Aceite de uma das Propostas pelo Cliente;
- Divulgar do Cenário Aprovado do Projeto aos colaboradores e terceiros;
- Obter a Assinatura do CONTRATO-MESTE e elaborar os ANEXOS; (no caso de terceirização)
- Alocar Espaço Físico dos Terceiros; (no caso de terceirização)

- Comunicar a Finalização da Etapa de Planejamento dos Testes; (externo)

Definição das Responsabilidades

Neste diagrama, está a representação dos papéis e responsabilidades para cada grupo de atividades envolvido na etapa de “Planejamento dos Testes”.



Mapeamento dos Artefatos

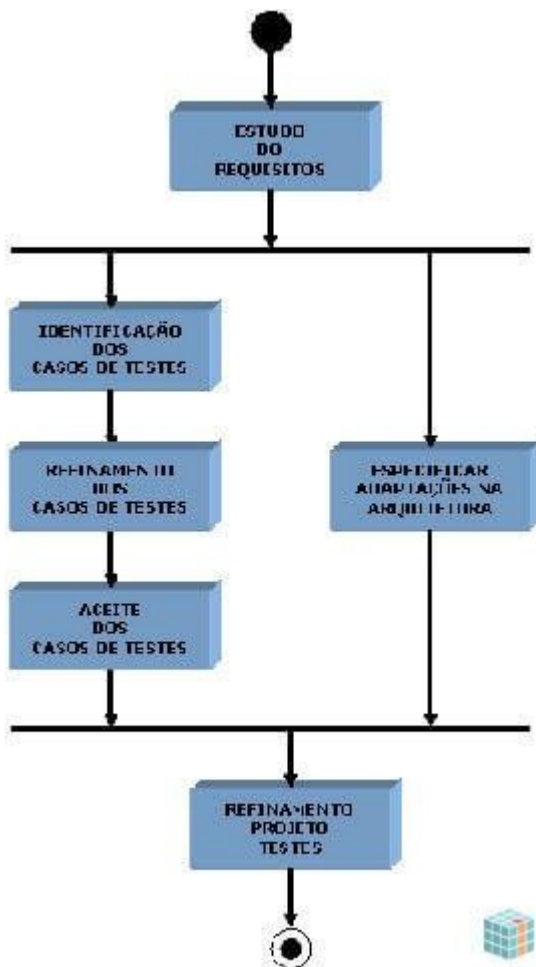
Nesta representação gráfica, estão destacados os “artefatos de entrada” exigidos como premissa para que cada macro-atividade possa ser realizada. Também são destacados os “artefatos de saída” produzidos como resultado da atividade.

Etapa 2: Especificação dos Testes

Esta etapa é caracterizada pela identificação dos casos de testes que deverão ser construídos e modificados em função das mudanças solicitadas pelo Cliente, bem como pelo próprio aperfeiçoamento do processo de testes (ampliação da cobertura).

Dinâmica das Macro-Atividades

Este diagrama representa a seqüência das “macro-atividades” a serem executadas na etapa de “Especificação dos Testes”.



Detalhamento das Macro-Atividades

Esta lista representa o conjunto de atividades que deverão ser executadas para que cada macro-atividade seja considerada finalizada, funcionando como um “check-list” de execução da etapa de “Especificação dos Testes”.

ESTUDO DOS REQUISITOS:

- Estudar os requisitos funcionais e não funcionais solicitadas pelo Cliente (novos requisitos);
- Estudar as modificações de requisitos solicitados pelo Cliente (mudanças de requisitos);
- Revisar os artefatos e identificar “inconsistências” dos requisitos;
- Estabelecer o Aceite dos Documentos fornecidos e “feedback” da qualidade dos mesmos;
- Estudar as lições aprendidas da Etapa “Especificação de Testes”;

ESPECIFICAR AS ADAPTAÇÕES DA ARQUITETURA DOS TESTES:

- Especificar as adequações nas atuais ferramentas empregadas;
- Especificar as novas ferramentas exigidas pelo projeto;
- Especificar as modificações estruturais na organização do ambiente;
- Especificar as adequações na automação da preparação do ambiente (script de teste);
- Especificar as adequações na automação da execução dos testes (script de teste);

- Especificar as adequações na automação da análise dos resultados (script de teste);

IDENTIFICAÇÃO DOS CASOS DE TESTES

- Identificar cada solicitação de mudança requisitada pelo Cliente;
- Identificar todos os Casos de Uso envolvidos em cada solicitação;
- Identificar Casos de Uso não cobertos adequadamente por Casos de Testes; (legado)
- Identificar todos o Fluxos do Caso de Uso (Básico, Alternativo e Exceção);
- Identificar os casos de testes que garantam cada Fluxo do Caso de Uso;

REFINAMENTO DOS CASOS DE TESTES:

- Estabelecer dinâmica com os Analistas de Testes que possuem conhecimento horizontal;
- Apresentação de um quadro-geral do impacto das mudanças nos respectivos aplicativos;
- Cada Analista de Testes apresenta seus casos de testes por aplicativo;
- O grupo de Analistas de Testes criticam e sugerem melhorias nos casos de testes;
- O grupo de Analista de Testes avaliam o nível de cobertura alcançado;
- Novas reuniões serão realizadas até que seja alcançado o patamar ideal de casos de testes;

Aceite dos Casos de Testes:

- Identificar Áreas-Chaves para apresentação dos casos de testes (Clientes Internos e Externos)
- Apresentar os casos de testes “progressivos” que serão aplicados nos testes;
- Apresentar os casos de testes “regressivos” que serão aplicados nos testes;
- Realizar refinamento dos casos de testes apresentados (“regressivos e progressivos”);
- Estabelecer o acordo Mútuo de Responsabilidade sobre o Nível de Qualidade do Software;

REFINAMENTO DO PROJETO DE TESTES:

- Reavaliar as estimativas de esforço e duração do Processo de Teste; (se necessário)
- Estabelecer um Cronograma-Detalhado, baseado no Cronograma-Macro já elaborado;
- Reavaliar riscos do Projeto em função de uma maior detalhamento sobre os requisitos;
- Negociar eventuais modificações em relação à duração, prazo e custo do projeto de testes;
- Comunicar a Finalização da Etapa de “Especificação dos Testes”; (externo)

Definição das Responsabilidades

Neste diagrama, está a representação dos papéis e responsabilidades para cada grupo de atividades envolvido na etapa de “Especificação dos Testes”.

Mapeamento dos Artefatos

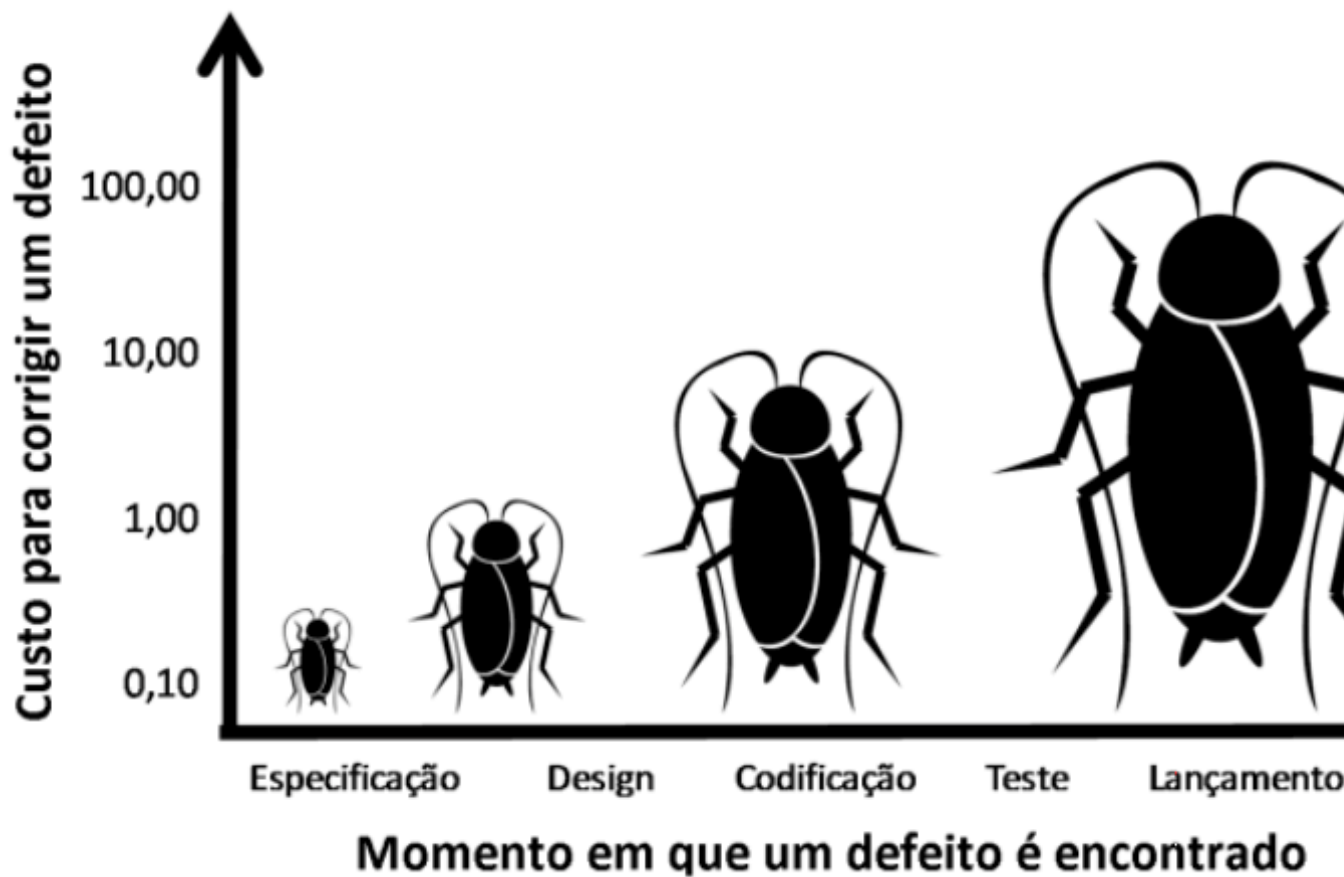
Nesta representação, estão destacados os “artefatos de entrada” exigidos como premissa para que cada macro-atividade possa ser realizada. Também são destacados os “artefatos de saída” produzidos como resultado da atividade.

Calculando o ROI (Retorno de Investimentos) em Testes de Softwares

Convido você agora a se aprofundar e calcular junto comigo os custos dos defeitos para um software e o retorno do investimento na realização de teste de software.

Primeiramente devemos identificar custos, incidências e percentuais de correção de defeitos nas fases do ciclo de desenvolvimento do software, assim conseguiremos realizar um cálculo de valores baseado na realidade.

Em primeiro lugar, é preciso observar que o custo de correção de defeitos encontrados em qualquer pacote de software cresce exponencialmente a cada fase da criação ou existência deste software, o que pode ser representado, segundo o autor Ron Patton, da seguinte forma [8] como mostra a figura abaixo.



Aumento exponencial dos custos com defeitos segundo Patton [8]

O custo de encontrar defeitos e removê-los na fase de especificação é baixíssimo, na ordem de grandeza de dezenas de centavos (de uma moeda qualquer). Na fase de design estes custos já crescem para a ordem de grandeza de unidades de moeda, e assim por diante, até chegarem a custos na casa das centenas quando o software já está em produção. O custo baixo de se encontrar defeitos nas fases de especificação e design

se justifica pelo baixo — ou, em alguns casos, desprezível — retrabalho resultante da descoberta e correção destes defeitos nestas fases iniciais. Seria o equivalente a amassar o guardanapo e começar a rabiscar novamente. Defeitos encontrados na fase de codificação já são mais caros porque muitas vezes exigem testes realizados pelos desenvolvedores (o que toma tempo), correção ou mesmo descarte de código desenvolvido e, nos piores casos, um retrocesso às fases de especificação e design.

Já os defeitos encontrados na fase de produção, além de todos os custos anteriores, implicam em custos de atendimento ao cliente, replicação dos defeitos em laboratório, chegando até mesmo aos temidos recalls. Em suma, o raciocínio de Patton é claro: quanto mais cedo forem encontrados os defeitos, menos custarão, tanto para os desenvolvedores quanto para os clientes.

Quanto à eficiência dos testes na descoberta de defeitos de software, dados obtidos a partir de projetos da própria **One Day Testing** mostram que quando o cliente atribui tarefas de testes para parte de seu time de desenvolvimento, este consegue descobrir no máximo 40% dos defeitos do software em questão.

Calculando o retorno sobre o investimento (ROI)



Tomando por base empresas que seguem o modelo CMMI, temos que aquelas que se enquadram no nível 1 tipicamente entregam software com 7,5 defeitos a cada 1.000 linhas de código, enquanto as empresas com certificação CMMI nível 5 produzem software com melhor qualidade, reduzindo os defeitos para uma média de 1 a cada 1.000 linhas de código [9].

Nesta situação, tomemos por hipótese um pacote de software com 1 milhão de linhas de código — nada incomum em projetos atuais — que

tenha sido codificado com 1.000 (mil) defeitos inadmissíveis, isto é, que terão que ser corrigidos a qualquer custo, seja em que fase da vida do software forem encontrados. Assumindo, de acordo com o raciocínio de Patton, que defeitos encontrados durante a fase de desenvolvimento tenham custo de R\$ 1,00, durante a fase de testes tenham um custo de R\$ 10,00 e que os mesmos defeitos encontrados com o software em produção tenham um custo de R\$ 100,00, podemos construir um cenário interessante.

Análise de 3 situações:

1. O software vai para o mercado sem ter sido formalmente testado;
2. O software passa por um processo de testes realizados por pessoal interno, interinamente designado pela empresa para realizá-los;
3. O software passa por um processo testes realizado por equipe especializada terceirizada;

Premissas da análise

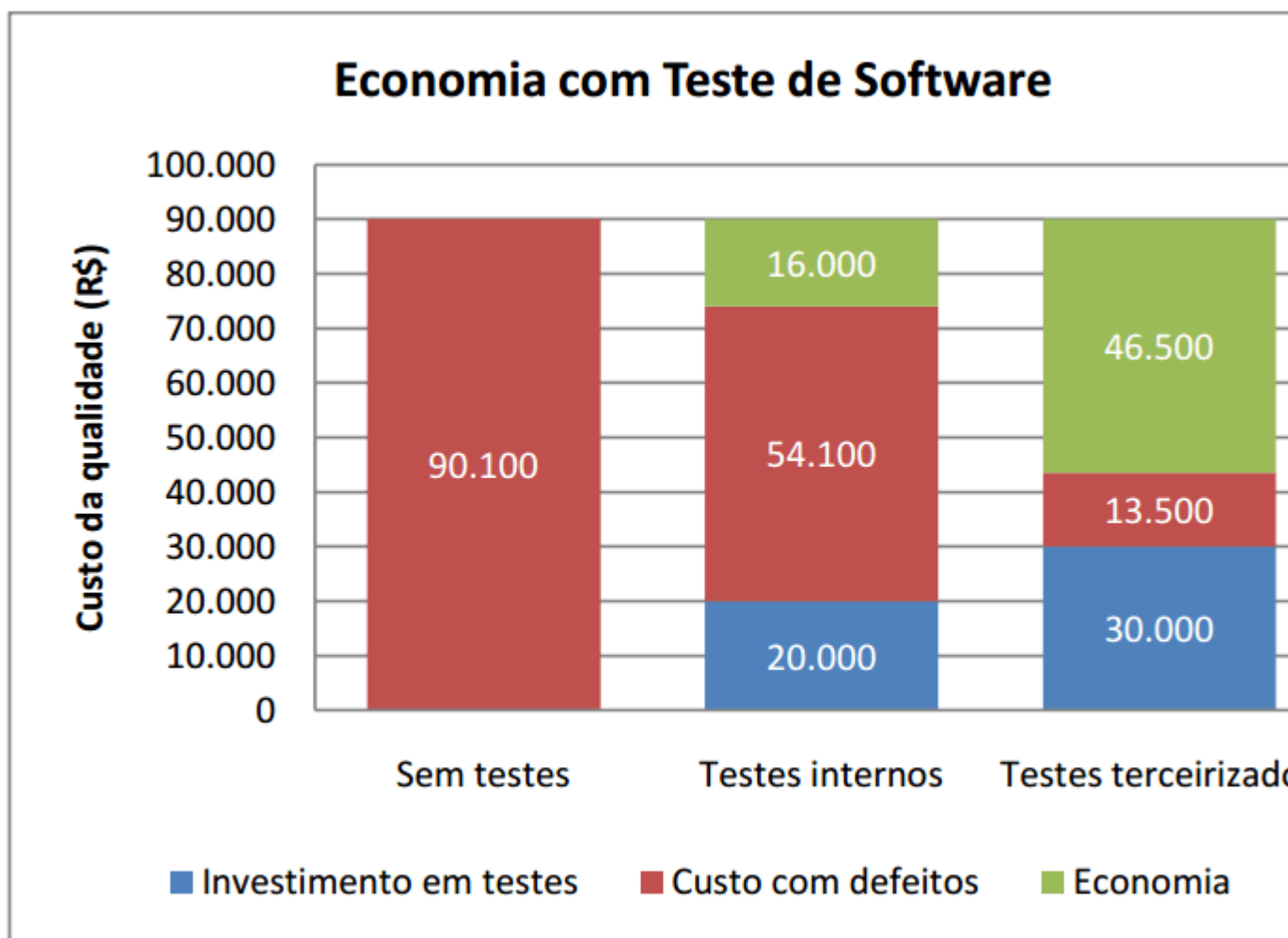
- 10% dos defeitos serão encontrados na fase de codificação, 0%, 40% ou 85% dos defeitos serão encontrados na fase de testes (respectivamente nos cenários 1, 2 e 3) e o restante será encontrado na fase de produção;

- O salário considerado no caso dos testes internos é de R\$3.000,00 mensais, com custo para a empresa (acrescido dos benefícios) de R\$5.000,00 mensais;
- Ainda no caso de testes internos, considera-se a atuação parcial de um gerente de testes com salário de R\$5.000,00 (R\$10.000,00, com benefícios), com 50% de seu tempo designado para o processo de testes;
- Os testes terceirizados são conduzidos por uma equipe que custa R\$10.000,00 por mês, durante três meses;
- Considera-se ROI como sendo a razão entre os valores economizados (em relação à abordagem sem testes) e o valor dos investimentos em cada uma das opções de testes avaliadas neste caso;

Resultados – Tabela ROI

	Tipos de testes		
	Sem testes	Testes internos	Testes terceirizados
Testes			
Pessoal: Salário de 1 analista (R\$)	-	15.000,00	-
Gestão: Salário parcial de 1 gerente de testes (R\$)	-	5.000,00	-
Serviços (R\$)	-	-	30.000,00
Total do investimento (R\$)	-	20.000,00	30.000,00
Desenvolvimento			
Defeitos encontrados	100	100	100
Custo dos defeitos (R\$)	100,00	100,00	100,00
Testes			
Defeitos encontrados	-	400	850
Custo dos defeitos (R\$)	-	4.000,00	8.500,00
Produção			
Defeitos encontrados	900	500	50
Custo dos defeitos (R\$)	90.000,00	50.000,00	5.000,00
Totalização			
Investimentos (R\$)	-	20.000,00	30.000,00
Custos com defeitos (R\$)	90.100,00	54.100,00	13.500,00
Total de custos (R\$)	90.100,00	74.100,00	43.600,00
ROI (em relação ao processo sem testes)	n/a	80%	155%
Economia (em relação ao processo sem testes)	n/a	16.000,00 17,75%	46.500,00 51,60%

De acordo com este raciocínio, podemos compor a tabela apresentada acima. A tabela mostra com clareza que os investimentos em testes de software, sejam estes realizados por pessoal interno ou por equipe especializada terceirizada, trazem vantagens financeiras, constituindo-se em investimentos que geram reduções relevantes dos custos de desenvolvimento do produto.



Os resultados, como pode ser claramente visto na imagem acima, apresentam uma justificativa sólida para a realização dos testes.

Um certo dia, navegando por comunidades de desenvolvimento de software me deparei com a seguinte imagem:

Pirâmide de testes

Confesso que na primeira vez não entendi do que a imagem se tratava , mas tive certeza de que um coelho barato era melhor que uma tartaruga cara.

O tempo passou, eu finalmente aprendi o que são testes unitários, o que esse coelho barato representa e gostaria de te ajudar a entender também.

Mas o que são testes unitários?

Um teste unitário basicamente é o teste da menor parte testável de um programa.

Tá, mas o que isso quer dizer?

Se você programa em uma linguagem que suporte paradigma funcional por exemplo, a menor parte testável do seu código deve ser uma função. Então um teste unitário seria o teste de qualquer função. No caso de orientação a objetos seria o teste de um método de seu objeto.

Bom... como tudo é mais fácil de entender na prática, vamos ao código!

No exemplo abaixo, temos uma função que soma dois números e retorna o valor da soma.

Para testar este código, só o que precisamos fazer é executar a função e verificar se seu valor de saída é o que esperamos.

Pronto! Já temos nosso primeiro teste unitário. Bem fácil não é?

Bom.. Agora que você já entendeu o que são e como são feitos testes unitários, você deve estar se perguntando:

Para que servem testes unitários?

Perguntar para que servem testes unitários ou qualquer outro teste automatizado é uma ótima pergunta, afinal existem diversas formas que aparentemente são mais rápidas de testar se minha função está fazendo o que deveria. Eu poderia simplesmente executar o código para verificar se está funcionando. Então, por que motivo eu vou escrever outro código para testar o meu código? O Que garante que o segundo código funciona? Quem testa o teste?

Testes unitários, assim como qualquer teste automatizados não servem principalmente para verificar se uma função específica está funcionando, mas sim para garantir que sua aplicação continue funcionando após alguma alteração em sua base de código.

Por que escrever testes unitários?

Pode parecer tentador em um primeiro momento não escrever testes para uma função que você acaba de desenvolver, afinal, costuma-se escrever mais código para testar uma função, que o próprio código da função. Mas você deve lembrar que irá passar a maior parte do tempo de desenvolvimento de um sistema trabalhando em sua manutenção.

Sua aplicação em pouco tempo terá algumas centenas de funções sendo executadas, e em muitas vezes executando umas às outras, sua base de código fica enorme e logo torna-se humanamente impossível de ser testada de forma manual após qualquer alteração. Testes unitários na maioria das vezes levam apenas alguns segundos para testar toda sua aplicação.

Por onde começar?

Existem diversas ferramentas de testes unitários para cada linguagem de programação. Você pode começar lendo a documentação destas ferramentas a partir de seus exemplos.

Neste [link](#) você pode encontrar algumas destas ferramentas classificadas por linguagem.

Tá, mas ainda não entendi o lance do Coelho barato e da Tartaruga cara

Não poderia encerrar este post sem antes explicar o que a pirâmide de testes quer dizer.

Ela é um conceito desenvolvido por Mike Cohn que diz que você deve ter muito mais testes de unidade (testes unitários) que testes de GUI que são testes mais a nível de usuário. Como clicar em um link por exemplo.

O Conceito da pirâmide explica o quanto testes de GUI são custosos em relação a testes unitários, pois eles demoram muito mais tempo para serem executados e também são de difícil manutenção, enquanto os testes unitários são muito mais simples, rápidos e baratos.

Bônus!

Implementei uma versão um pouco mais completa do teste de exemplo da função soma em Javascript e em Ruby. Você pode conferir o código neste [link](#)

estes Unitários com a ferramenta JUnit – Execução

March 30, 2015 by [Mauda](#) [2 Comments](#)

Conteúdo do Post:

1. [Histórico do JUnit](#)
2. [Planejando seus testes...](#)
3. [Execução de um arquivo de teste JUnit](#)
4. [Explicando o arquivo de teste...](#)
5. [Alterando o arquivo de teste para gerar uma AssertionError](#)
6. [Alterando o arquivo de teste para gerar uma Exception](#)
7. [Rodando um package de testes](#)
8. [Explicação sobre a View JUnit](#)
9. [Identificação de Erros](#)
10. [finally{](#)

Olá Pessoal, tudo bom? Neste artigo de hoje vamos verificar como funciona a execução de um teste JUnit. Para configurar um projeto com JUnit favor observar o seguinte [artigo](#).

Histórico do JUnit

[JUnit](#) é uma ferramenta de automatização de testes unitários e de testes de regressão para projetos Java. Foi criada por:



Figura 01 -Eric Keck

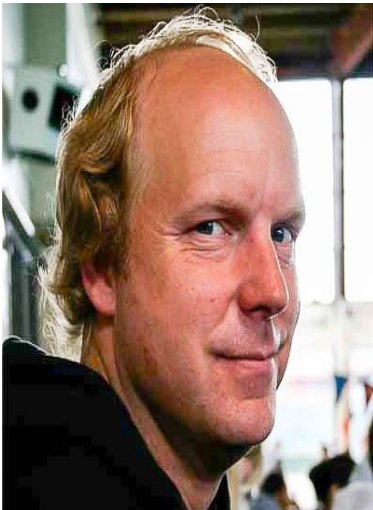


Figura 02 – Kent Bech

A API do JUnit é dividida em duas partes, uma parte que possui métodos e annotations para a construção de *classes* de teste e outra parte que realiza a execução das *classes* de teste criadas pela primeira parte.

A execução dos testes é realizada de forma automática e sem intervenção humana. A apresentação dos resultados dos testes é realizada através de um “semáforo”. Uma cor verde no teste significa que este executou conforme estava descrito no arquivo de testes. Uma execução com cor azul, significa que o teste possui algum erro de Assertion, ou seja, que alguma validação do JUnit não foi realizada corretamente. Uma execução com cor vermelha, significa que houve algum tipo de exception no código Java como `NullPointerException` ou `IndexOutOfBoundsException`.

Planejando seus testes...

Algo muito importante que deve ser realizado é o planejamento para a execução dos testes. Existem algumas coisas tarefas relacionadas ao planejamento que iremos descrever abaixo:

- Defina um pacote próprio para armazenar todas as *classes* de testes
- Cada *classe* de teste deve ser responsável por testar uma *classe* do sistema
- Use o paradigma do Dividir para Conquistar
- Crie vários métodos de teste que realizam pequenos testes cada um
- A cobertura de testes será muito maior
- Caso ocorram erros, a identificação é mais simples

Vamos que você possui um projeto que seja no formato MVC-DAO, dessa forma teoricamente você teria 3 camadas que não possuem interação direta com o usuário do software, Model, Business Controller e DAO. Uma boa forma de planejar testes unitários é realizar um produto cartesiano entre as *classes* da camada Model x 3 camadas que não possuem interação direta com o usuário (Model, BC e DAO). Vamos imaginar que existem 4 *classes* na camada Model, dessa forma teremos 12 testes no total, onde seria 4 testes por camada do software.

Talvez nesse cálculo não será possível testar todos os aspectos de todas as *classes*, mas uma grande cobertura de testes existirá dentro do seu sistema.

Execução de um arquivo de teste JUnit

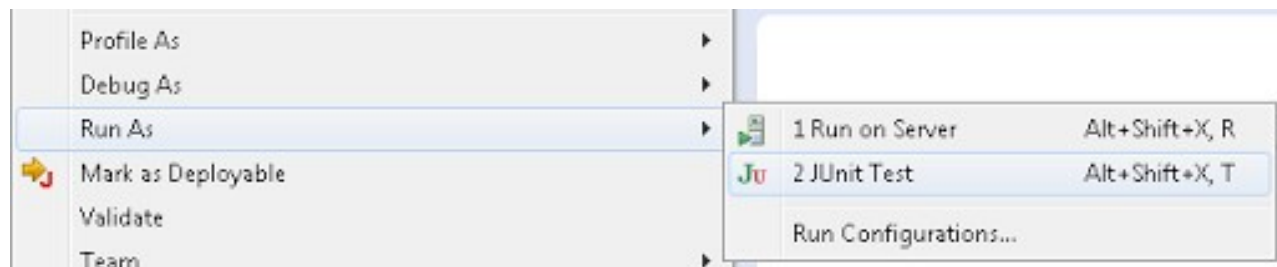
Um arquivo de teste JUnit é aquele que possui ao menos um método com a annotation **@Test**, do pacote **org.junit**. Assim suponha o código de teste abaixo:

```

import static org.junit.Assert.assertTrue;
1 import java.util.Collection;
2
3 import org.junit.Test;
4
5 public class JunitTest{
6     private Collection collection;
7
8     @Test
9     public void testEmptyCollection(){
10        assertTrue(collection == null);
11        System.out.println("@Test -Collection nula");
12    }
13 }

```

Vamos agora executar diretamente o código. Depois iremos retornar e falar sobre alguns trechos deste código. Clique sobre a *classe* de teste e vá na opção **Run As > JUnit Test**, conforme Figura 03.



Fig

ura 03 – Opção para rodar um teste JUnit

O resultado da execução da *classe* de teste acima pode ser conferido na Figura 04. Repare que a parte esquerda da Figura possui uma linha verde indicando que o teste foi executado com sucesso!

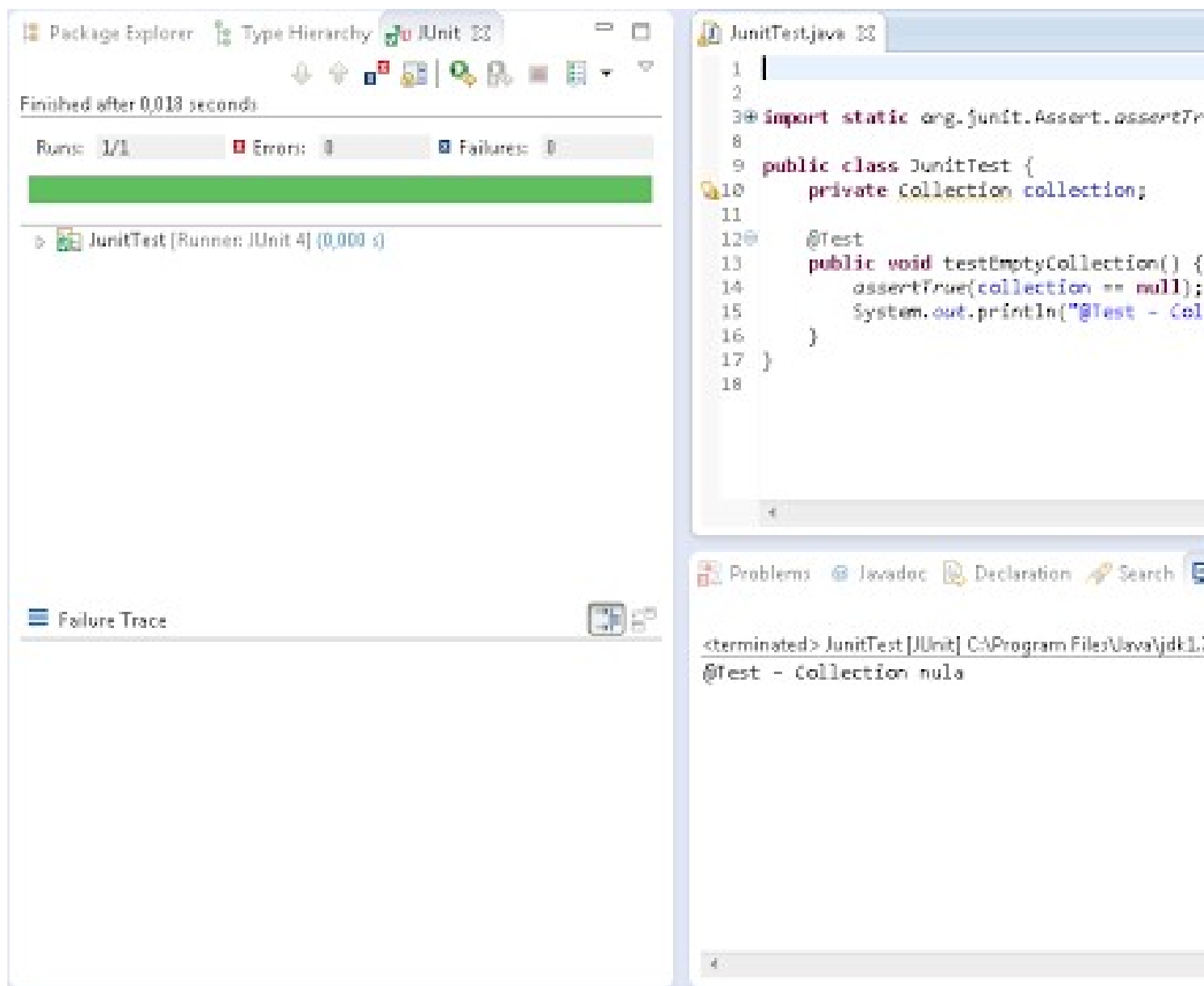
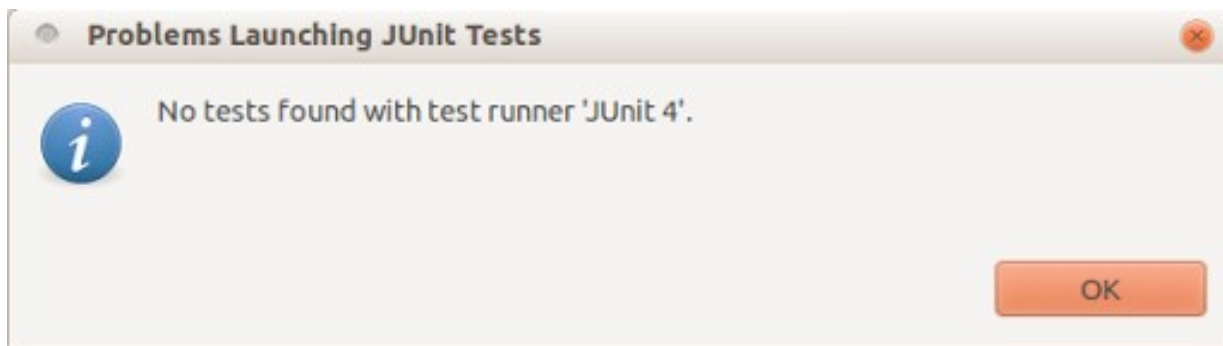


Figura 04 – Indicando que o teste foi realizado com sucesso!

Explicando o arquivo de teste...

Vamos observar com maior detalhes o código executado acima. Repare que esse código possui algumas linhas diferentes do tradicional ao programar *classes* Java. A linha 08 possui uma annotation `@Test`. Essa anotação é o que caracteriza um método ser um teste dentro do JUnit. Caso essa anotação não existisse no método o JUnit ignoraria o método. Caso não existe nenhum método com essa anotação ocorreria a seguinte mensagem de erro ao executar o teste JUnit:



Figura

05 – Não existe nenhum método na *classe* com a annotation `@Test`

A linha 10 possui a invocação de um método static da *classe* `org.junit.Assert`, conforme pode ser conferido pelo `import static` realizado na linha 1. O método da linha 10 está exemplificado abaixo:

```
1 assertTrue(collection == null)
```

Ele indica que a operação deverá ser verdadeira, ou seja, o atributo `collection` deve ser nulo. Caso o que se encontra como parâmetro do método `assertTrue` não seja `True`, ocorrerá uma `AssertionError` e o seu teste será encerrado. Essa exceção é própria do JUnit e serve para realizar verificações de como está indo o teste e em caso de não estar caminhando de acordo com o que você deseja ocorra a exceção.

As outras linhas da *classes* não representam nenhuma linha nova para quem já possui alguma experiência com programação Java.

Alterando o arquivo de teste para gerar uma `AssertionError`

Vamos alterar o arquivo de teste para gerar um `AssertionError`, a fim de demonstrar como é o comportamento do JUnit. O código abaixo apresenta a alteração realizada nas linhas marcadas:

```
import static org.junit.Assert.assertTrue;
1 import java.util.Collection;
2
3 import org.junit.Test;
4
5 public class JunitTest{
6     private Collection collection;
7
8     @Test
9     public void testEmptyCollection(){
10         assertTrue(collection != null);
11         System.out.println("@Test -Collection nula");
12     }
13 }
```

Vamos realizar a execução desta *classe* de teste e o resultado será a Figura 06:

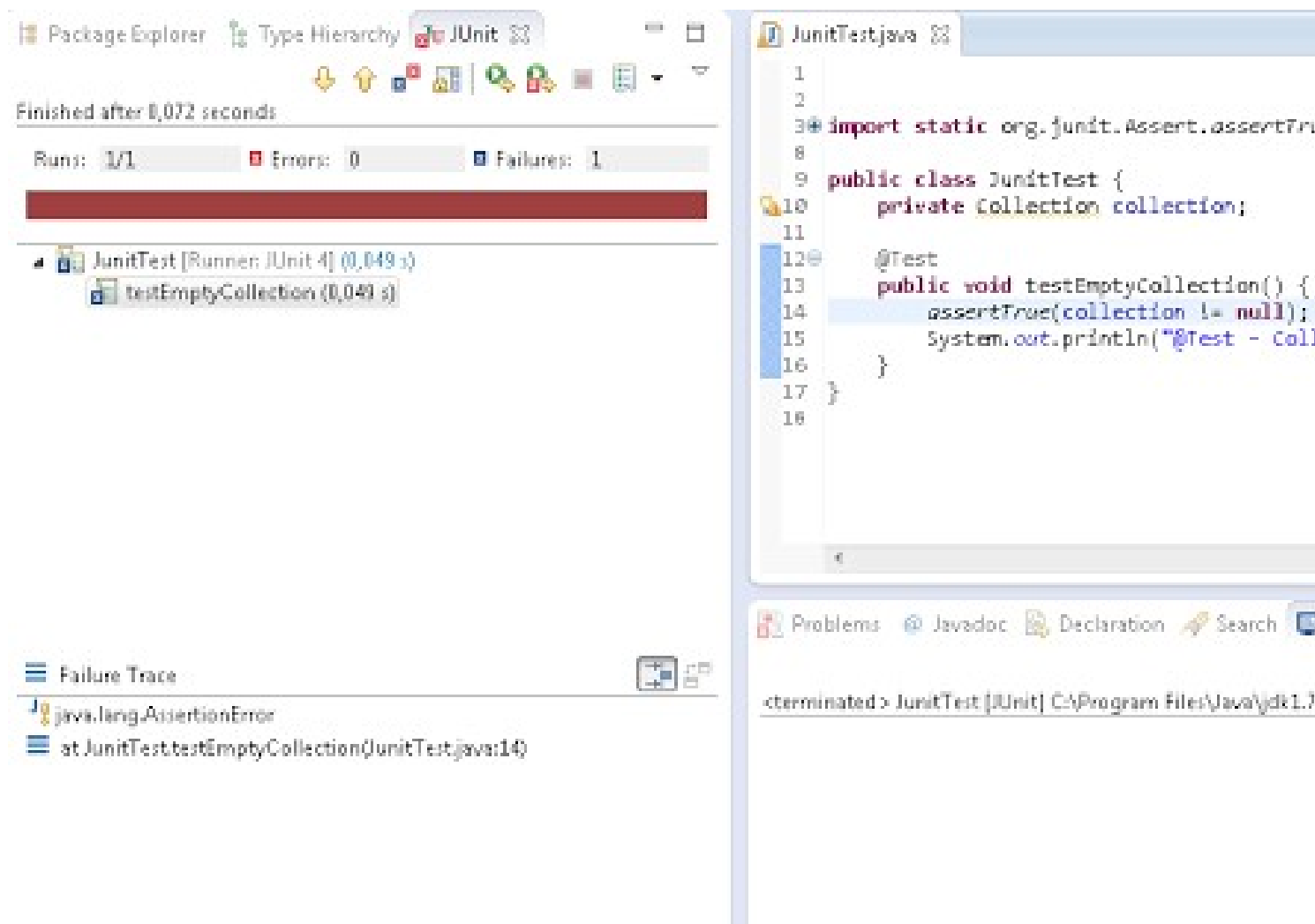


Fig06 – Obtendo um AssertionError via JUnit

A linha 10 foi modificada para que ao invés de o atributo collection seja igual a null, este seja diferente de null. Como em nenhum momento foi realizada a instanciação dessa Collection, logo o atributo será null, o que irá gerar o erro de assertion. Repare que a linha ficou vermelha, mas a execução errada foi marcada em uma parte da tela indicada por Failures, onde ficou marcado o número 1. Sempre que ocorrer um erro que fique azul o ícone do X ao lado da classe de teste, isso indica que houve um AssertionError. Alguma característica que deveria estar em um determinado estado, mas encontra-se em outro.

Alterando o arquivo de teste para gerar uma Exception

Vamos alterar o arquivo de teste para gerar um Exception, a fim de demonstrar como é o comportamento do JUnit. O código abaixo apresenta a alteração realizada nas linhas marcadas:

```

import static org.junit.Assert.assertTrue;
1 import java.util.Collection;
2 import org.junit.Test;
3
4
5 public class JUnitTest{
6     private Collection collection;
7
8     @Test
9     public void testEmptyCollection(){
10 collection.clear();
11 assertTrue(collection == null);
12 System.out.println("@Test -Collection nula");
13 }
14 }

```

Vamos realizar a execução desta *classe* de teste e o resultado será a Figura 07:

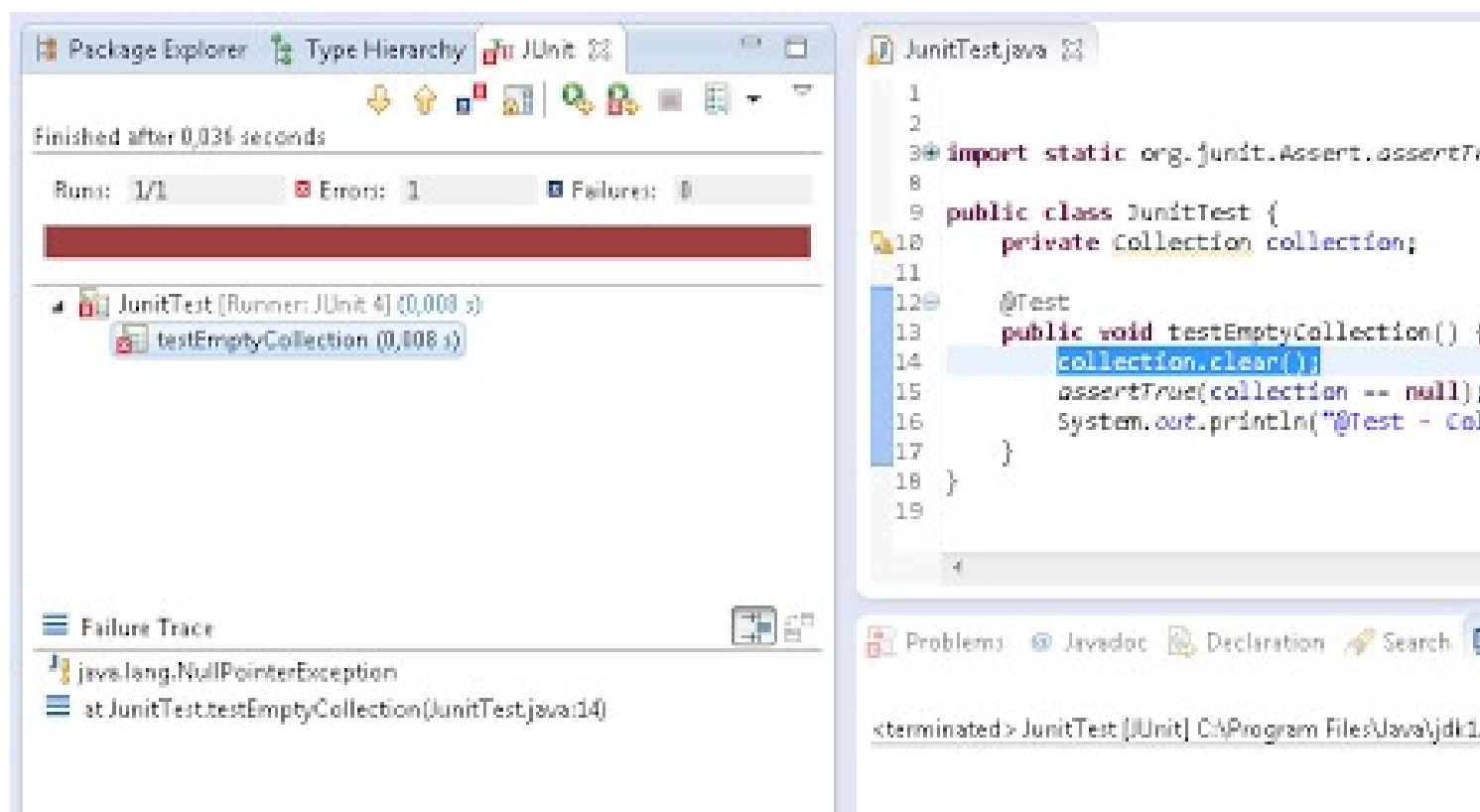


Figura 07 – Obtendo uma Exception via JUnit

A linha 10 foi modificada para chamar o método `clear()` do atributo `collection`. Mas o problema é que o atributo nunca foi inicializado, logo ao chamar o método `clear()` irá estourar uma `NullPointerException`, caracterizando um erro no JUnit por exception. Repare que a linha

ficou vermelha, e a execução errada foi marcada em uma parte da tela indicada por Errors, onde ficou marcado o número 1. Sempre que ocorrer um erro que deixe o ícone X ao lado da *classe* de teste na cor vermelha, isso indica que houve um Exception. Normalmente é um erro Java, NullPointerException, IndexOutOfBoundsException entre outras exceptions Java.

Rodando um package de testes

É possível utilizando o comando Run As > JUnit Test rodar um package de testes. Basta clicar sobre um pacote de testes e realizar o comando. O resultado será algo parecido com a Figura 08:

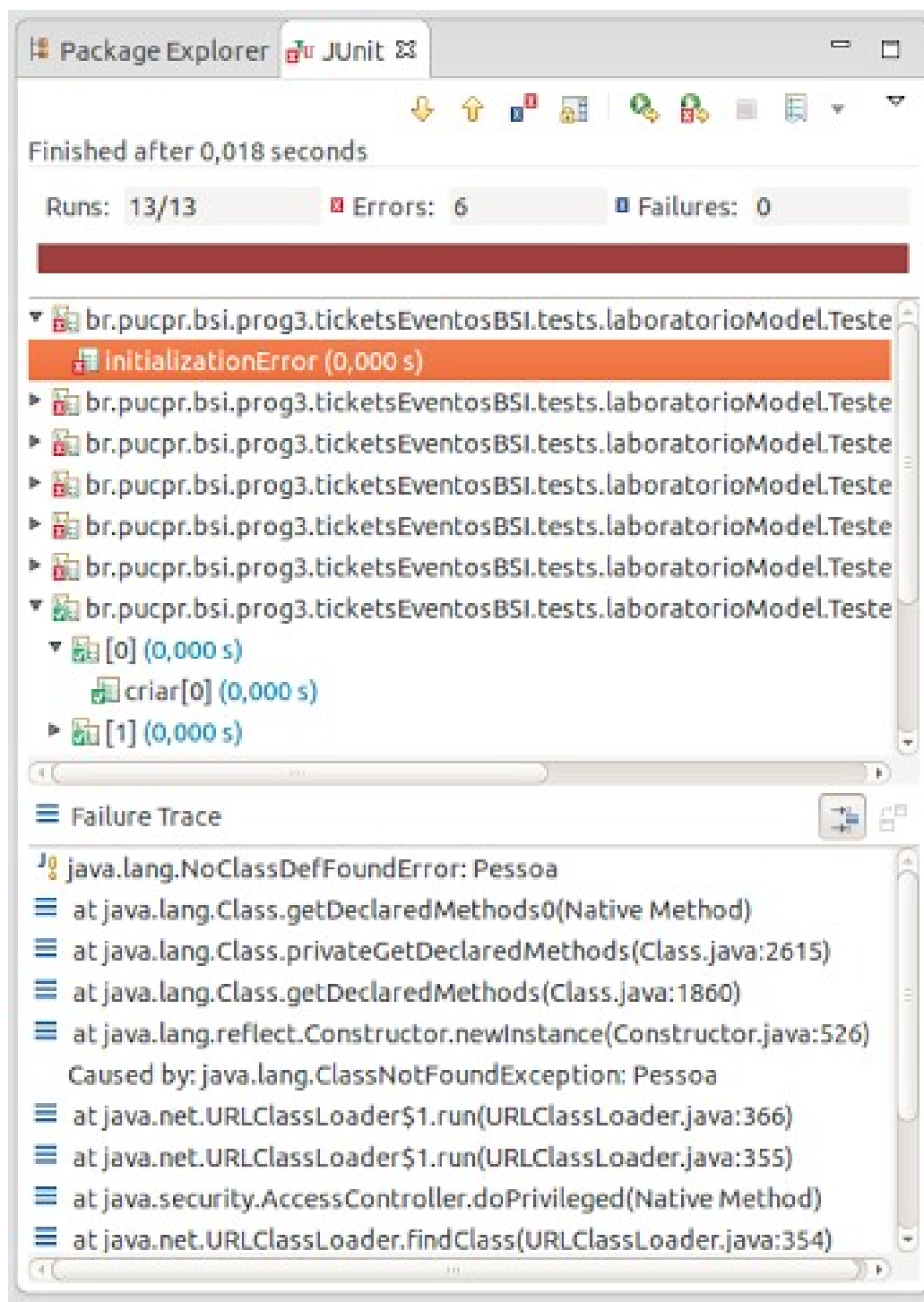


Figura 08 –

Rodando um pacote de testes

É possível perceber pela Figura 08, que ao rodar esse pacote, cada *classe* de teste foi exibida na View JUnit e ao expandir uma das *classes* é possível identificar o que ocorreu para cada teste. Isso facilita na hora de testar vários arquivos de teste com um só comando, otimizando assim o tempo para realizar os comandos de execução dos testes. Da mesma forma que a execução de um único arquivo de teste, o JUnit irá exibir testes de sucesso, AssertionErros e Exceptions.

Explicação sobre a View JUnit

A Figura 09 abaixo mostra a View do JUnit que é aberta ao iniciar a execução de um teste. Vamos explicar abaixo cada um dos pontos, a fim de facilitar o entendimento desta View.

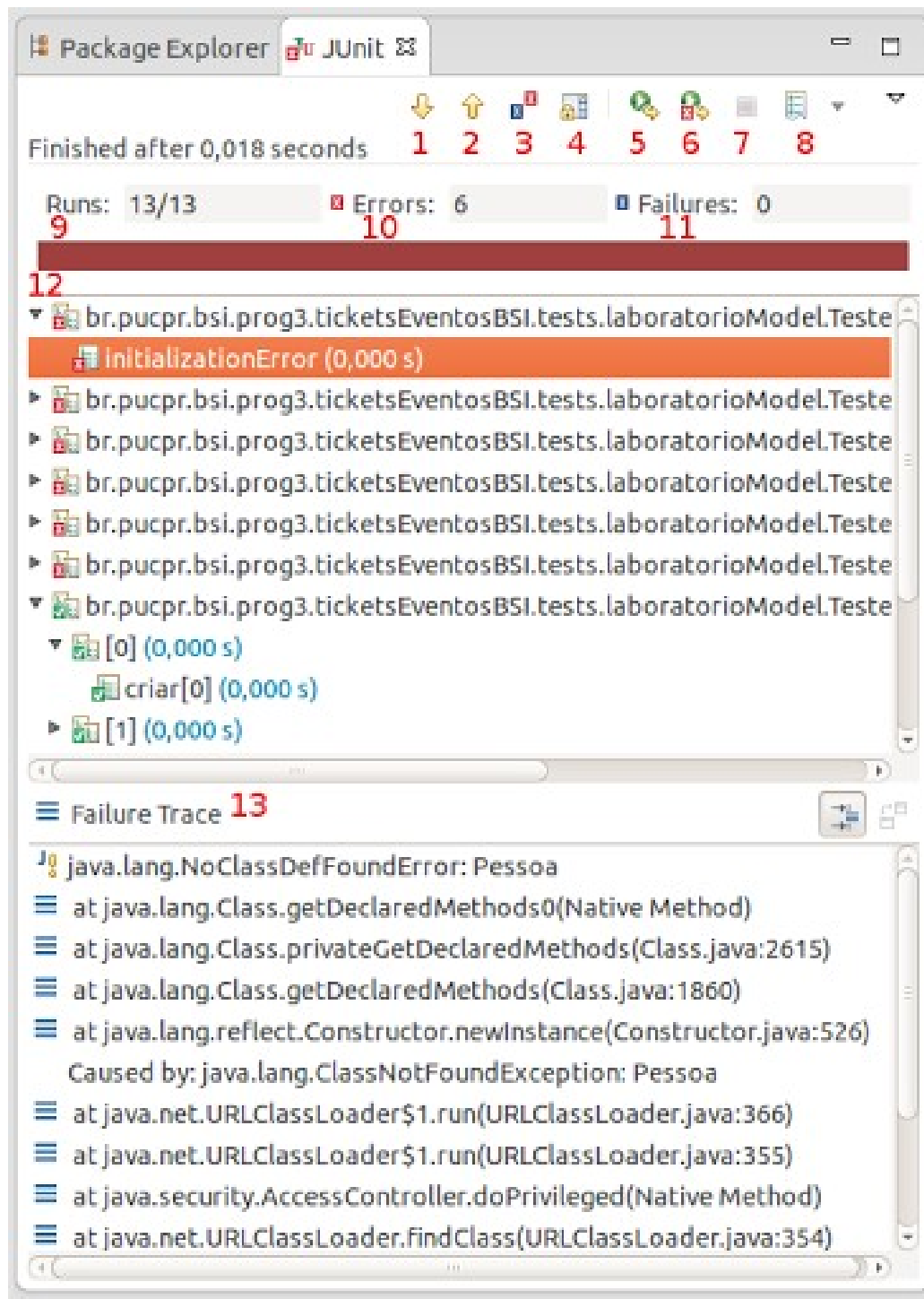


Figura 09 – View

JUnit na IDE Eclipse

Cada número representa um item dentro da view do JUnit na IDE Eclipse, vamos então explicar cada um deles:

1. Botão **Next Failed Test** – Apresenta o próximo teste que falhou na Test's List
2. Botão **Previous Failed Test** – Apresenta o teste anterior que falhou na Test's List
3. Botão **Show Failures Only** – Mostra somente os testes que falharam na Test's List
4. Botão **Scroll Lock** – Durante a execução dos testes impede de a Test's List ir deslocando para baixo a cada execução de teste
5. Botão **Rerun Test** – Realiza novamente a execução de todos os testes
6. Botão **Rerun Test – Failures First** – Realiza novamente a execução de todos os testes, mas com os que falharam por primeiro
7. Botão **Stop JUnit Test Run** – Para a execução dos testes
8. Botão **Test Run History...** – Verifica o histórico dos testes executados
9. Text **Runs** – Mostra a quantidade de testes já rodados / quantidade de testes existentes
10. Text **Errors** – Mostra a quantidade de testes que geraram exceções Java
11. Text **Failures** – Mostra a quantidade de testes que geraram AssertionsErrors
12. **Test's List** – Lista de testes que estão sendo/foram executados
13. **Failure Trace** – Stack trace de onde ocorreu o erro ou falha

Ao lado do item 8, Botão Test Run History... existe um triângulo, o qual há um menu, o qual é mostrado pela Figura 10. A opção mais importante deste menu é a opção Export... onde você poderá exportar um XML com os resultados de todos os testes realizados nessa execução. Dessa forma se você executou o teste de um pacote inteiro, será apresentado cada arquivo de teste e os resultados deste.

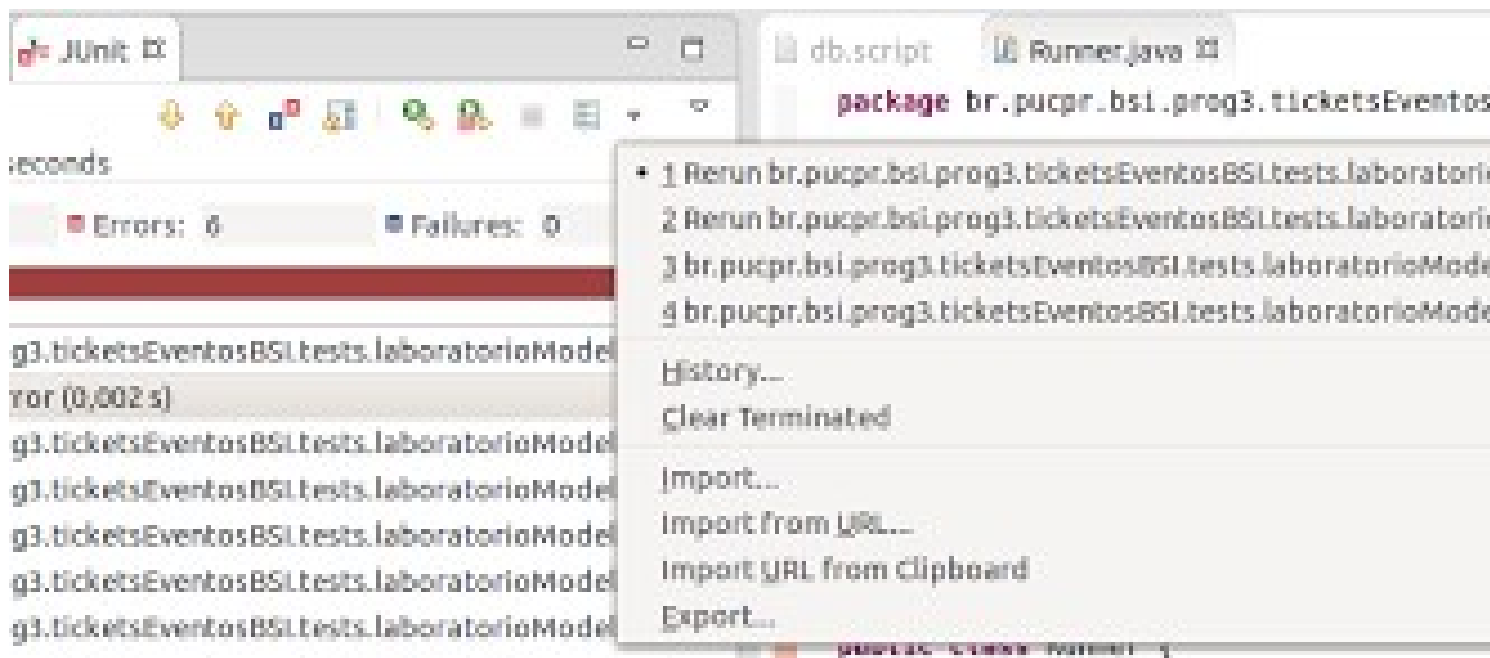


Figura 10 – Menu “Test Run History...”

Por fim é possível clicar em um teste com erro e realizar a execução somente deste, como mostra a Figura 11.

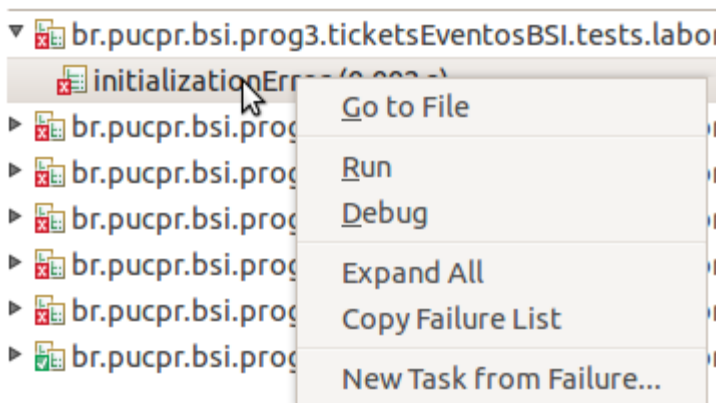


Figura 11 – Menu apresentado sobre um

Teste com falhas

Aqui você pode rodar o teste novamente com a opção **Run**, ou até mesmo Debugar o teste com a opção **Debug** (Não sabe debugar? Sem problemas a partir do dia 06/04/2015 teremos um tutorial nesse [link](#)). As outras opção não são importantes nesse momento.

Identificação de Erros

Existe dentro da view JUnit, uma parte chamada Failure Trace. Essa parte possui uma stack trace para identificar a linha que ocorreu o erro na execução do teste. Os testes apresentados até aqui são simples e estão contido em apenas um arquivo, conforme mostra a Figura 07, ao clicar na linha `at JUnitTest.testEmptyCollection(JUnitTest.java:14)`, o Eclipse

irá mostrar selecionada a linha 14, indicando que o erro ocorreu naquela linha. Caso você esteja observando a Figura 08, ao selecionar um teste cujo resultado foi uma Exception, é possível perceber que houve um erro Java, indicando que ele não possui a definição da *classe* Pessoa.

Quando temos arquivos de testes que necessitam de outras *classes* para que possam ser realizados (por exemplo, os testes dos branches model, business entre outros dos meus projetos públicos no Bitbucket), essa stack trace é de sumária importância, já que pode identificar qual linha das outras *classes* ocorreu o erro.

A stack que se encontra dentro da Failure Trace é apenas mais uma ajuda das stack traces impressas no console do Eclipse, quando o erro JUnit for uma Exception Java. Mas pode ser que seja a única a ser impressa, sendo que no console não há nada indicando o erro. Esse comportamento é comum de ocorrer em AssertionErrors.

finally{

Este é apenas o primeiro post sobre JUnit explicando um pouco mais a fundo essa ferramenta muito importante. Não perca nas próximas semanas posts mais profundos sobre essa ferramenta a fim de melhorar o entendimento e ajudar a você a desenvolver a habilidade de criar testes próprios.

Duvidas ou sugestões? Deixe seu feedback! Isso ajuda a saber a sua opinião sobre os artigos e melhorá-los para o futuro! Isso é muito importante!

Até um próximo post!