

Aprofundamento de herança e polimorfismo em JAVA

Prof. Marlos de Mendonça Corrêa

Descrição

Conceitos avançados de hierarquia e polimorfismo em linguagem Java e aplicação desses conceitos em interfaces e coleções. Os tipos estático e dinâmico de hierarquia e suas repercussões. Métodos importantes de objetos, implementação da igualdade, a conversão para o tipo String e o cálculo do hash, assim como o operador instanceof.

Propósito

Obter o conhecimento da linguagem Java, puramente orientada a objetos (OO) e que está entre as mais utilizadas no mundo, é imprescindível para o profissional de Tecnologia da Informação. A orientação a objetos (OO) é um paradigma consagrado no desenvolvimento de software e empregado independentemente da metodologia de projeto – ágil ou prescritiva.

Preparação

Para melhor absorção do conhecimento, recomenda-se o uso de computador com o Java Development Kit – JDK e um IDE (Integrated Development Environment) instalados e acesso à Internet para realização de pesquisa.

Objetivos

Módulo 1

Hierarquia de herança

Identificar a hierarquia de herança em Java.

[Acessar módulo](#)



Buscar



Baixar conteúdo em PDF



Vídeos



Menu

Métodos de objetos

Empregar os principais métodos de objetos em Java.

[Acessar módulo](#)

Módulo 3

Polimorfismo

Descrever polimorfismo em Java.

[Acessar módulo](#)

Módulo 4

Interfaces

Aplicar a criação e o uso de interfaces em Java.

[Acessar módulo](#)



Introdução

A orientação a objetos (OO) está no cerne da linguagem Java. A própria linguagem foi estruturada com foco no paradigma OO. Assim, Java está ligada de forma indissociável a esse paradigma. Para extrair todos os recursos que Java tem a oferecer, torna-se fundamental conhecer bem OO.

Na verdade, muito do poder da Java vem das capacidades que a OO trouxe para o desenvolvimento de softwares. Exemplos disso são, justamente, a herança e o polimorfismo.

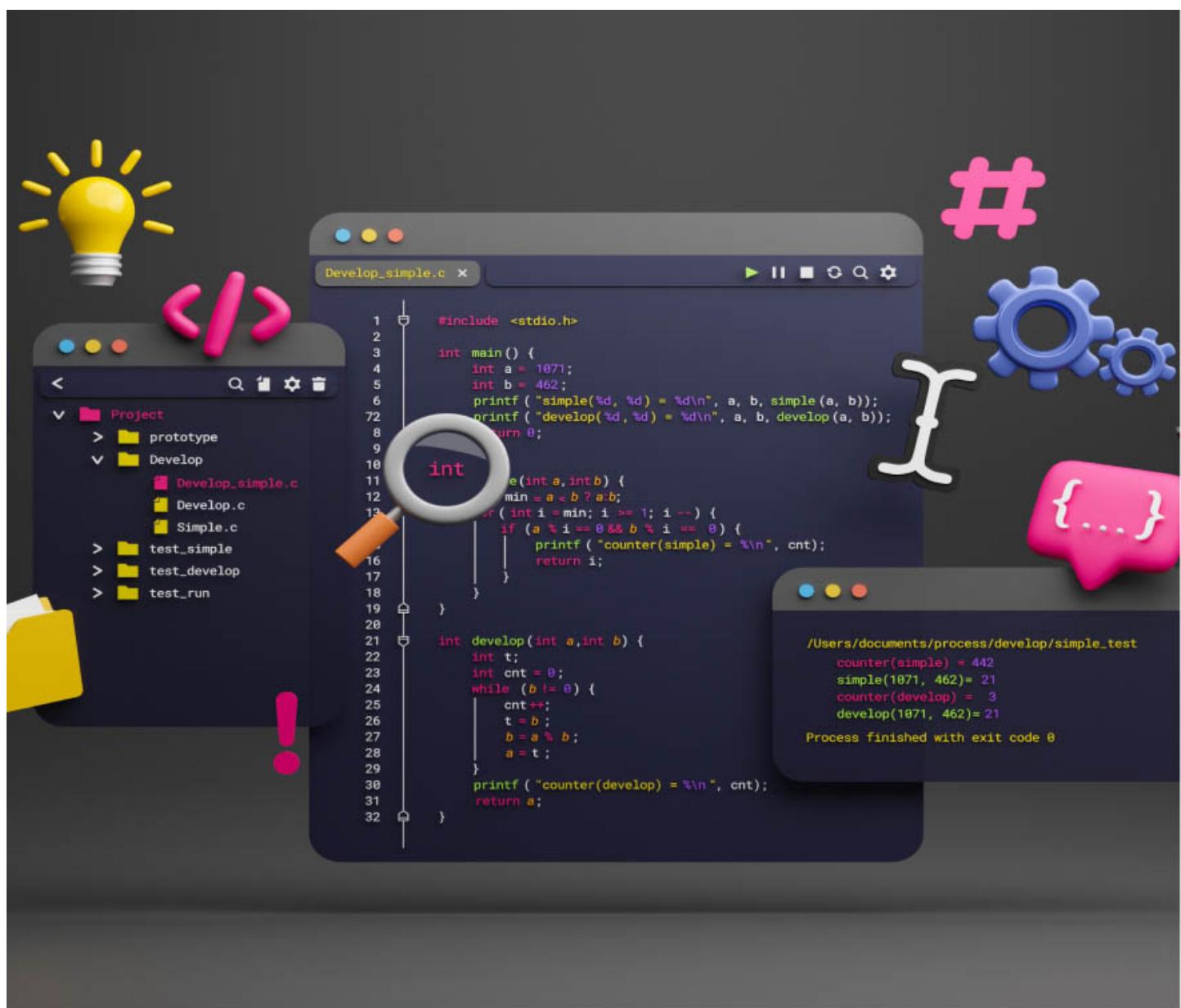
Por esses motivos, vamos nos aprofundar no conhecimento de herança e polimorfismo na linguagem Java. Veremos como a Java emprega um conceito chamado interfaces e estudaremos métodos de especial interesse para a manipulação de objetos.

Isso será feito com o apoio de atividades que reforçarão o aprendizado e fornecerão um feedback. Ao fim, teremos compreendido apenas uma parte do desenvolvimento em Java, mas que é indispensável para qualquer estudo mais avançado.

1

Hierarquia de herança

Ao final deste módulo, você será capaz de identificar a hierarquia de herança em Java.



Herança em Java

No vídeo a seguir, apresentamos o conceito de herança na linguagem de programação Java.

Herança e a instanciação de objeto em Java

A palavra herança não é uma criação do paradigma de programação orientada a objetos (POO). Você deve estar familiarizado com o termo que, na área do Direito, designa a transmissão de bens e valores entre pessoas. Na Biologia, o termo significa a transmissão de características aos descendentes.

Independentemente da aplicação do seu conceito, a palavra herança tem em comum o entendimento de que envolve legar algo a alguém.

Em OO, herança é, também, a transmissão de características aos descendentes. Visto de outra forma, é a capacidade de uma “entidade” legar outra seus métodos e atributos. Por legar, devemos entender que os métodos e atributos estarão presentes na classe derivada. Para melhor compreendermos, vamos entender a criação de um objeto em Java.

Quando definimos uma classe, definimos um mecanismo de criação de objetos.

Uma classe define um tipo de dado, e cada objeto instanciado pertence ao conjunto formado pelos objetos daquela classe.

Nesse conjunto, todos os objetos são do tipo da classe que lhes deu origem.

Uma classe possui métodos e atributos. Os métodos modelam o comportamento do objeto e atuam sobre o seu estado, que é definido pelos atributos. Quando um objeto é instanciado, uma área de memória é reservada para acomodar os métodos e os atributos que o objeto deve possuir.

Todos os objetos do mesmo tipo terão os mesmos métodos e atributos, porém cada objeto terá sua própria cópia destes.

Consideremos, a título de ilustração, a definição de classe mostrada no Código 1.

JAVA



Código 1: declaração de classe.

Qualquer objeto instanciado a partir dessa classe terá os atributos “**nome**”, “**nacionalidade**” e “**naturalidade**”, além de uma cópia dos métodos mostrados. Na verdade, como “String” é um objeto de tamanho desconhecido em tempo de programação, as variáveis serão referências que vão guardar a localização em memória dos objetos do tipo “String”, quando esses forem criados.

Quando objetos se relacionam por associação, por exemplo, essa relação se dá de maneira horizontal. Normalmente são relações do tipo “contém” ou “possui”. A herança introduz um novo tipo de relacionamento, inserindo a ideia de “é tipo” ou “é um”. Esse relacionamento vertical origina o que se

Uma hierarquia de classe é um conjunto de classes que guardam entre si uma relação de herança (verticalizada), na qual as classes acima generalizam as classes abaixo ou, por simetria, as classes abaixo especializam as classes acima.

Vamos analisar a classe derivada mostrada no Código 2.

JAVA



Código 2: declaração de subclasse.

A classe “Aluno” estende a classe “Pessoa”, ou seja, “Aluno” é uma subclasse de “Pessoa”, formando assim uma relação hierárquica (“Aluno” deriva de “Pessoa”). Podemos aplicar a ideia introduzida com o conceito de herança para deixar mais claro o que estamos falando: “Aluno” é um tipo de/é uma “Pessoa”. Um objeto do tipo “Aluno” também é do tipo “Pessoa”.

Nesse sentido, como vimos, a classe “Pessoa” lega seus métodos e atributos à classe “Aluno”.

Verificando o Código 2, notamos que a classe “Aluno” não possui métodos e atributos definidos em sua declaração, exceto pelo seu construtor. Logo, ela terá apenas os componentes legados por “Pessoa”. A instanciação de um objeto do tipo “Aluno”, nesse caso, levará à reserva de um espaço de memória para guardar apenas os atributos “nome”, “nacionalidade” e “naturalidade”, e os métodos de “Pessoa” e o construtor de “Aluno”. No nosso exemplo, o objeto seria virtualmente igual a um objeto do tipo “Pessoa”.

Esse exemplo, porém, tem apenas um fim didático e busca mostrar na prática, ainda que superficialmente, o mecanismo de herança. Como os tipos do exemplo são praticamente idênticos, nenhuma utilidade prática se obtém – “Aluno” não especializa “Pessoa”.

Mas aí também atingimos outro fim: mostramos que a versatilidade da herança está na possibilidade de realizarmos especializações e generalizações no modelo a ser implementado. A especialização ocorre quando a classe derivada particulariza comportamentos. Assim, são casos em que as subclasses alteram métodos da superclasse.

No caso do Código 3, a classe “Aluno” definida possui um novo atributo – “**matrícula**” – que é gerado automaticamente quando o objeto for instanciado. Apesar de esse objeto possuir os métodos e atributos de “Pessoa”, ele agora tem um comportamento particular: na instanciação, além dos atributos definidos em “Pessoa”, ocorre também a definição de “matrícula”.



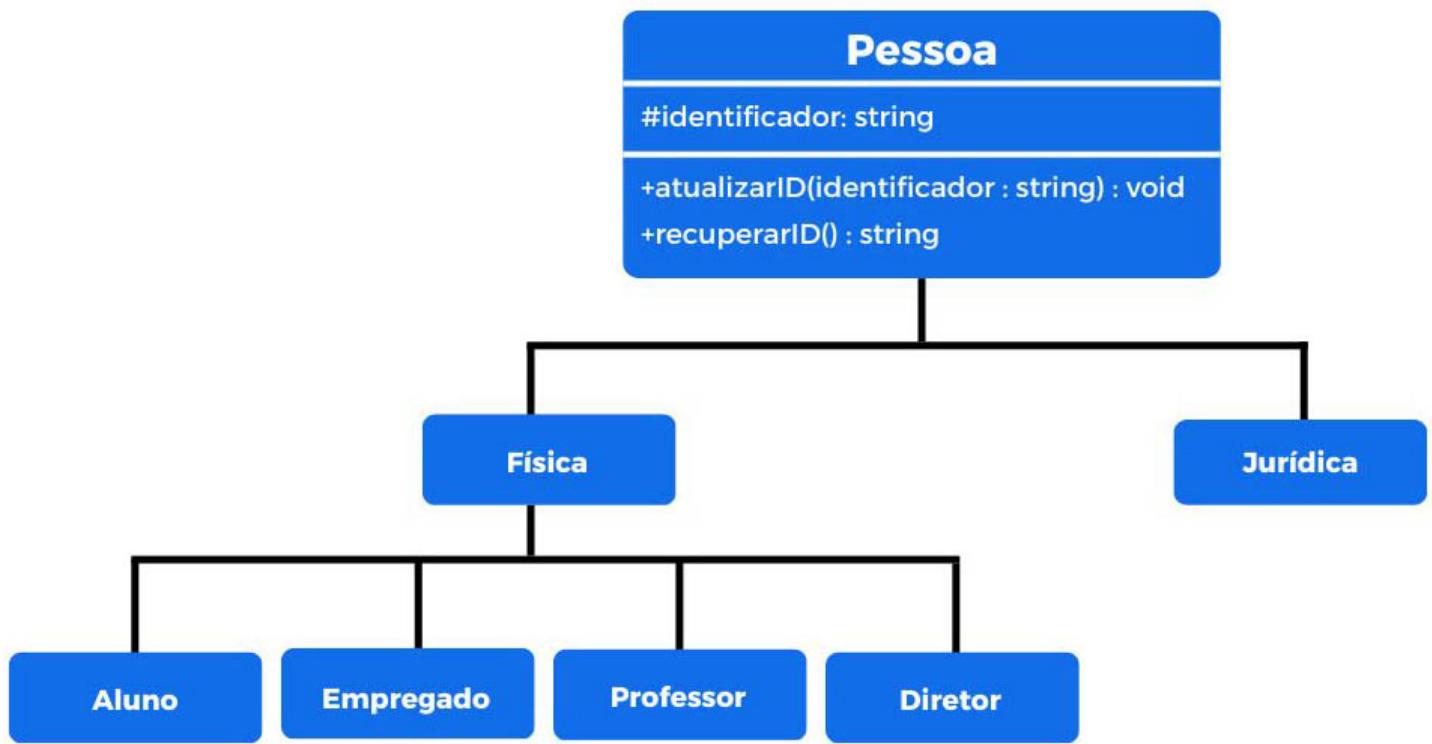
Código 3: exemplo de especialização.

Vejamos a hierarquia de classes mostrada na imagem a seguir.

Já sabemos que o atributo “**identificador**” e os métodos que operam sobre ele (“**atualizarID**” e “**recuperarID**”) são herdados pelas classes filhas.

Também sabemos que um objeto da classe “**Fisica**” é também do tipo “Pessoa”.

Mas vale destacar que a relação de herança se propaga indefinidamente, isto é, um objeto da classe “Aluno” também é um objeto das classes “Fisica” e “Pessoa”.



Especialização de comportamento.

A classe “Pessoa” implementa o comportamento mais genérico de todos. Ela possui um atributo de identificação definido como do tipo “String” (assim ele aceita letras, símbolos e números); e métodos que operam sobre esse atributo (veja o Código 4).

As classes “Física”, “Jurídica” e “Aluno” especializam o método que define o atributo “Identificador”, particularizando-o para suas necessidades específicas, como observamos no Código 5, no Código 6 e no Código 7, que mostram as respectivas implementações.

JAVA



Código 4: código parcial da classe "Pessoa" – métodos genéricos.

JAVA



Código 5: método "atualizarID" da classe "Fisica".

JAVA



Código 6: método "atualizarID" da classe "Juridica".

JAVA



Código 7: método "atualizarID" da classe "Aluno".

Repare que cada código possui um comportamento específico. Apesar de não exibirmos a implementação do método “**validaCPF**” e “**validaCNPJ**”, fica claro que os comportamentos são distintos entre si e entre as demais implementações de “atualizarID”, pois o Cadastro de Pessoas Físicas (CPF) e o Cadastro Nacional de Pessoas Jurídicas (CNPJ) possuem regras de formação diferentes.

Explorando a hierarquia de herança

O exemplo anterior de especialização de comportamento pode ter despertado em você alguns questionamentos. Nele, "Aluno" está especializando o método "atualizarID", que é da superclasse "Pessoa".

Ao mesmo tempo, um objeto "Aluno" também é um tipo de "Fisica", pois esta classe busca modelar o conceito de "pessoa física" (é razoável esperar que um aluno possua CPF além de seu código de matrícula). Será que essa situação não seria melhor modelada por meio de herança múltipla, com "Aluno" derivando diretamente de "Pessoa" e "Fisica"?

Já falamos que Java não permite herança múltipla e que é possível solucionar tais casos modificando a modelagem. Agora, entenderemos o problema da herança múltipla que faz a Java evitá-la. Para isso, vamos modificar o modelo mostrado na figura "Especialização de comportamento", criando a classe "ProfessorJuridico", que modela os professores contratados como pessoa jurídica.

Essa situação é vista na próxima imagem. Nela, as classes "Fisica" e "Juridica" especializam o método "atualizarID" de "Pessoa" e legam sua versão especializada à classe "ProfessorJuridico".

A questão que surge agora é: quando "atualizarID" for invocado em um objeto do tipo "ProfessorJuridico", qual das duas versões especializadas será executada?

Essa é uma situação que não pode ser resolvida automaticamente pelo compilador. Por causa dessa ambiguidade, a próxima imagem é também chamada de "Diamante da morte".

Pessoa

#identificador: string

+atualizarID(identificador : string) : void

+recuperarID() : string

Física

Juridica

Professor
Juridico

Diamante da morte.

Linguagens que admitem herança múltipla deixam para o desenvolvedor a solução desse problema.

Em C++, por exemplo, a desambiguação é feita fazendo-se um cast explícito para o tipo que se quer invocar.

Entretanto, deixar a cargo do programador é potencializar o risco de erro, o que Java tem como meta evitar.

Portanto, em Java, tal situação é proibida, obrigando a se criar uma solução de modelagem distinta.

Herança, subtipos e o princípio da substituição de Liskov

Nas seções anteriores, você compreendeu melhor o mecanismo de herança e como ele afeta os objetos instanciados. Falamos sobre especialização e generalização, mostrando como tais conceitos funcionam dentro de uma hierarquia de classes.



Atenção!

Se você estava atento, deve ter notado que o método “atualizarID” das classes “Pessoa”, “Fisica” e “Juridica” possui a mesma assinatura. Isso está alinhado ao princípio de Projeto por Contrato (Design by Contract).

No Projeto por Contrato, os métodos de uma classe definem o contrato que se estabelece com a classe ao consumir os serviços que esta disponibiliza. Assim, para o caso em questão, o contrato “reza” que, para atualizar o campo “**identificador**” de um objeto, deve-se invocar o método “atualizarID”, passando-lhe um objeto do tipo “String” como parâmetro. E não se deve esperar qualquer retorno do método.

O método “atualizarID” da classe “Aluno” é diferente. Sua assinatura não admite parâmetros, o que altera o contrato estabelecido pela superclasse. Não há erro que seja conceitual ou formal, isto é, trata-se de uma situação perfeitamente válida na linguagem e dentro do paradigma OO, de maneira que “Aluno” é subclasse de “Fisica”. Porém, “Aluno” não define um subtipo de “Pessoa”, apesar de, corriqueiramente, essa distinção não ser considerada.

Rigorosamente falando, **subtipo e subclasse são conceitos distintos**.



Uma subclasse é estabelecida quando uma classe é derivada de outra.



Um subtipo tem uma restrição adicional.

Para que uma subclasse seja um subtipo da superclasse, faz-se necessário que todas as propriedades da superclasse sejam válidas na subclasse.

Isso não ocorre para o caso que acabamos de descrever. Nas classes “Pessoa”, “Fisica” e “Juridica”, a propriedade de definição do identificador é a mesma: o campo “identificador” é definido a partir de um objeto “String” fornecido. Na classe “Aluno”, o campo “identificador” é definido automaticamente.

A consequência imediata da mudança de propriedade feita pela classe “Aluno” é a modificação do contrato estabelecido pela superclasse. Com isso, um programador que siga o contrato da superclasse, ao usar a classe “Aluno”, terá problemas pela alteração no comportamento esperado.

Essa situação nos remete ao princípio da substituição de Liskov. Esse princípio foi primeiramente apresentado por Barbara Liskov, em 1987, e faz parte dos chamados princípios SOLID (Single responsibility, Open-closed, Liskov substitution, Interface segregation and Dependency inversion).

Ele pode ser encontrado com diversos enunciados, sempre afirmando a substitutibilidade de um objeto da classe base pela classe derivada, sem prejuízo para o funcionamento do software.

Podemos enunciá-lo da seguinte forma:

Seja um programa de computador P e os tipos *B* e *D*, tal que *D* deriva de *B*.

Se *D* for subtipo de *B*, então qualquer que seja o objeto *B* do tipo *B*, ele pode ser substituído por um objeto *d* do tipo *D* sem prejuízo para P.

Voltando ao exemplo em análise, as classes “Fisica” e “Juridica” estão em conformidade com o contrato da classe “Pessoa”, pois não o modificam, embora possam adicionar outros métodos.

Assim, em todo ponto do programa no qual um objeto de “Pessoa” for usado, objetos de “Fisica” e “Juridica” podem ser aplicados sem quebrar o programa, mas o mesmo não se dá com objetos do tipo “Aluno”.

Hierarquia de coleção

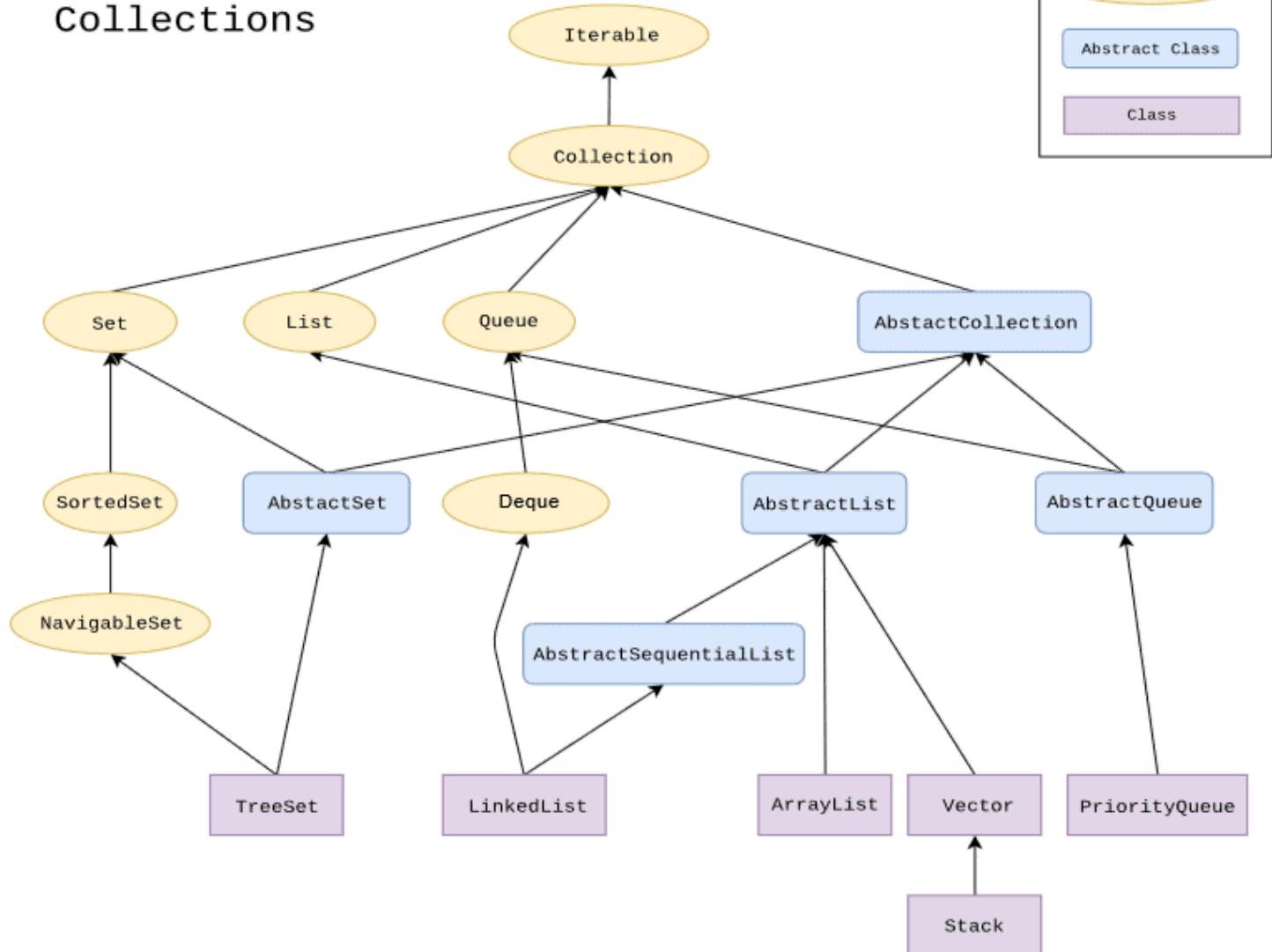
Coleções ou containers são um conjunto de classes e interfaces Java chamados de Java Collections Framework. Essas classes e interfaces implementam estruturas de dados comumente utilizadas para agrupar múltiplos elementos em uma única unidade. Sua finalidade é armazenar, manipular e comunicar dados agregados.

Entre as estruturas de dados implementadas, temos:

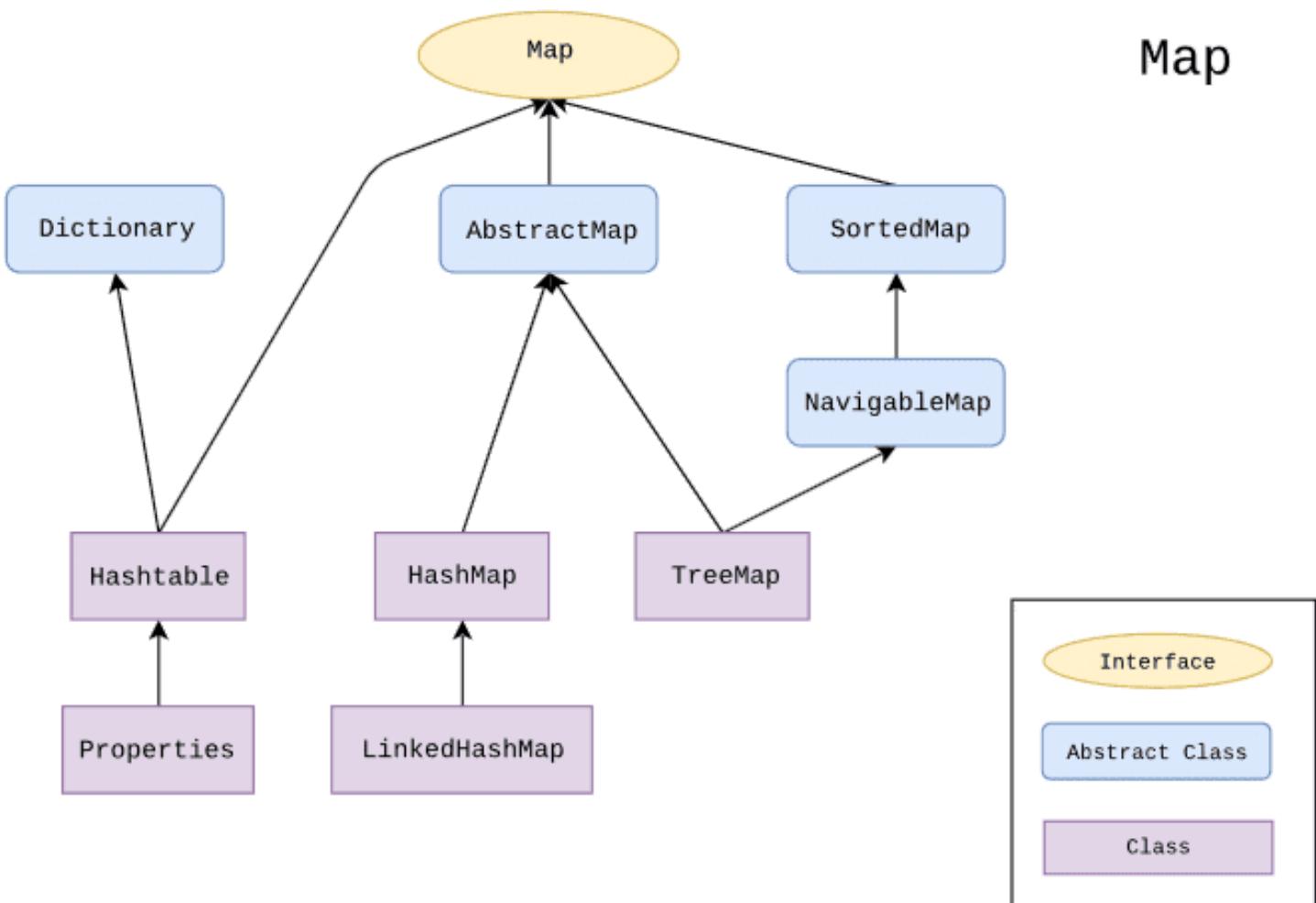
- Set
- List
- Queue
- Deque

Curiosamente, apesar de a estrutura “Map” não ser, de fato, uma coleção, ela também é provida pela Java Collections Framework. Isso significa que o framework é formado por duas árvores de hierarquia, uma correspondente às entidades derivadas de “Collection” e a outra, formada pela “Map”, como pode ser observado nas imagens a seguir.

Collections



Hierarquia de herança – Collections.



Hierarquia de herança – Map.

Como você pode observar pelas imagens que representam ambas as hierarquias de herança, essas estruturas são implementadas aplicando-se os conceitos de herança vistos anteriormente.



Exemplo

“SortedSet” e “SortedMap” são especializações de “Set” e “Map”, respectivamente, que incorporam a ordenação de seus elementos.

A interface “Collection” possui os comportamentos mais genéricos (assim como “Map”, na sua hierarquia). Por exemplo, o mecanismo para iterar pela coleção pode ser encontrado nela. Ela possui, também, métodos que fornecem comportamentos genéricos. É um total de 15 métodos, mostrados a seguir.

Insere um elemento na coleção.

addAll(Collection c)

Adiciona os elementos de uma coleção em outra.

clear()

Limpa ou remove os elementos de uma coleção.

contains(Object c)

Verifica se dado elemento está presente na coleção.

containsAll(Collection c)

Verifica se todos os elementos de uma coleção estão presentes em outra.

equals(Object e)

Verifica se dois objetos são iguais.

hashCode()

Retorna o hash de uma coleção.

isEmpty()

Retorna verdadeiro se a coleção estiver vazia.

Iterator()

Retorna um iterador.

remove(Object e)

Remove determinado elemento da coleção.

removeAll(Collection c)

Remove todos os elementos da coleção.

retainAll(Collection c)

Remove todos os elementos de uma coleção exceto os que correspondem a "c".

size()

Retorna o número de elementos da coleção.

toArray()

Retorna os elementos de uma coleção em um vetor.

Object [] toArray (Object e)

Retorna um vetor que contém todos os elementos da coleção que o invoca.

As interfaces "Set", "List", "Queue" e "Deque" possuem comportamentos especializados, conforme as seguintes particularidades de cada tipo:

SET



[LIST](#)[QUEUE](#)[DEQUE](#)

Nas hierarquias de coleções mostradas, as interfaces e as classes abstratas definem o contrato que deve ser seguido pelas classes que lhes são derivadas. Essa abordagem atende ao **Princípio de Substituição de Liskov**, garantindo que as derivações constituam subtipos. As classes que implementam os comportamentos desejados são vistas em roxo nas imagens.

Tipos estáticos e dinâmicos

Esse é um assunto propenso à confusão. Basta uma rápida busca pela Internet para encontrar o emprego equivocado desses conceitos. Muitos associam a vinculação dinâmica na invocação de métodos dentro de hierarquias de classe à tipagem dinâmica. Não é. Trata-se, nesse caso, de vinculação dinâmica (dynamic binding).

Nesta seção, vamos elucidar essa questão.

Tipagem é a atribuição a uma variável de um tipo.

Um tipo de dado é um conjunto fechado de elementos e propriedades que afeta os elementos. Por exemplo, quando definimos uma variável como sendo do tipo inteiro, o compilador gera comandos para alocação de memória suficiente para guardar o maior valor possível para esse tipo.

Você se lembra de que o tipo inteiro em Java (e em qualquer linguagem) é uma abstração limitada do conjunto dos números inteiros, que é infinito. Além disso, o compilador é capaz de operar com os elementos do tipo "int". Ou seja, adições, subtrações e outras operações são realizadas sem a necessidade de o comportamento ser implementado. O mesmo vale para os demais tipos primitivos, como "String", "float" ou outros.

Há duas formas de se tipar uma variável: estática ou dinâmica.



A tipagem estática ocorre quando o tipo é determinado em tempo de compilação.



A tipagem dinâmica é determinada em tempo de execução.

Observe que não importa se o programador determinou o tipo. Se o compilador for capaz de inferir esse tipo durante a compilação, então, trata-se de tipagem estática.



Atenção!

A tipagem dinâmica ocorre quando o tipo só pode ser identificado durante a execução do programa.

Um exemplo de linguagem que emprega tipagem dinâmica é a JavaScript. O Código 8 no emulador a seguir, mostra um trecho de programa em JavaScript que exemplifica a tipagem dinâmica.

Input Console



Podemos ver que a mesma variável ("a") assumiu, ao longo da execução, diferentes tipos. Observando a linha 1 do Código 8, podemos ver que, na declaração, o compilador não tinha qualquer informação que lhe permitisse inferir o tipo de dado de "a". Esse tipo só pode ser determinado na execução, nas linhas 3, 5, 7 e 9. Todas essas linhas alteram sucessivamente o tipo atribuído a "a".

Você deve ter claro em sua mente que a **linguagem de programação Java é estaticamente tipada**, o que significa que todas as variáveis devem ser primeiramente declaradas (tipo e nome) e depois utilizadas [2]. Uma das possíveis fontes de confusão é a introdução da instrução "var" a partir da versão 10 da Java [3]. Esse comando, aplicável apenas às variáveis locais, instrui o compilador a inferir o tipo de variável. Todavia, o tipo é determinado ainda em tempo de compilação e, uma vez determinado, o mesmo não pode ser modificado. A Java exige que, no uso de "var", a variável seja inicializada, o que permite ao compilador determinar o seu tipo. O Código 9 mostra o emprego correto de "var", enquanto o Código 10 gera erro de compilação. Ou seja, "var", em Java, não possui a mesma semântica que "var" em JavaScript. São situações distintas e, em Java, trata-se de tipagem estática.

JAVA



Código 9: uso correto de "var" em Java – tipagem estática.

JAVA



Código 10: uso errado de "var" em Java.

Outra possível fonte de confusão diz respeito ao dynamic binding ou vinculação dinâmica de método. Para ilustrar essa situação, vamos adicionar o método “retornaTipo” às classes “Pessoa”, “Fisica” e “Juridica”, cujas respectivas implementações são vistas no Código 11, no Código 12 e no Código 13.

JAVA



Código 11: “retornaTipo” de “Pessoa”.

JAVA



Código 12: “retornaTipo” de “Fisica”.

JAVA



Código 13: “retornaTipo” de “Juridica”.

JAVA



Código 14: classe Principal – exemplo de vinculação dinâmica.

A execução desse código produz como saída:

TERMINAL



O motivo disso é a vinculação dinâmica de método. O vetor “grupo []” é um arranjo de referências para objetos do tipo “Pessoa”.

Como “Fisica” e “Juridica” são subclasses de “Pessoa”, os objetos dessas classes podem ser referenciados por uma variável que guarda referência para a superclasse. Isso é o que ocorre nas linhas 5 e 6 do Código 14.

A vinculação dinâmica ocorre na execução da linha 9. O compilador não sabe, a priori, qual objeto será referenciado nas posições do vetor, mas, durante a execução, ele identifica o tipo do objeto e vincula o método apropriado. Essa situação, entretanto, não se configura como tipagem dinâmica, pois o tipo do objeto é usado para se determinar o método a ser invocado.

Nesse caso, devemos lembrar, também, que Java oculta o mecanismo de ponteiros, e uma classe definida pelo programador não é um dos tipos primitivos. Então “grupo []” é, como dito, um vetor de referências para objetos do tipo “Pessoa”, e isso se mantém mesmo nas linhas 5 e 6 (as subclasses são um tipo da superclasse). Essa é uma diferença sutil, mas significativa, com relação ao mostrado no Código 8.

Exploraremos esse mecanismo em outra oportunidade. No entanto, é preciso ficar claro que essa situação não é tipagem dinâmica, pois o vetor “grupo []” não alterou seu tipo, qual seja, referência para objetos do tipo “Pessoa”.

Falta pouco para atingir seus objetivos.

Vamos praticar alguns conceitos?

Questão 1

Considere duas classes chamadas “Base” e “Derivada”, de tal forma que a primeira é superclasse da última. A classe “derivada” atende ao princípio da substituição de Liskov.

Analise as afirmações a seguir e marque a opção em que todas são corretas.

- I) Derivada não pode sobrescrever os métodos protegidos de “Base”.
- II) Todos os métodos públicos e protegidos de “Derivada” devem ter a mesma assinatura que os de “Base”.
- III) Todos os métodos públicos e protegidos de “Base” que forem redefinidos em “Derivada” devem ter a mesma assinatura.

A I

B II

C III

D I e III

E I e II

[Responder](#)

Questão 2

Estruturas de dados são mecanismos fundamentais para a manipulação de dados e possuem impacto na performance de um software. Java fornece implementações de algumas estruturas por meio do Java Collections Framework. Sobre o assunto, marque a única opção correta.

A Uma desvantagem das coleções é que os iteradores não são providos, precisando ser desenvolvidos pelo programador.

B Uma desvantagem das coleções é que não podem ser usadas com tipos de dados criados pelo programador (classes), já que sua estrutura é desconhecida.

C Nenhuma das estruturas implementa o controle de duplicidade de objetos inseridos.

D Mais de um tipo de coleção permite a implementação de fila (FIFO).

E A coleção “List” implementa uma lista de prioridades.

[Responder](#)

2

Métodos de objetos

Ao final deste módulo, você será capaz de empregar os principais métodos de objetos em Java.



Vamos começar!



Principais Métodos em Java

Os principais métodos em Java serão apresentados no vídeo a seguir.

Métodos

O método “`toString`”

Já vimos que em Java todas as classes descendem direta ou indiretamente da classe “Object”. Essa classe provê um conjunto de métodos que, pela herança, estão disponíveis para as subclasses. Tais métodos oferecem ao programador a facilidade de contar com comportamentos comuns fornecidos pela própria biblioteca da linguagem, além de poder especializar esses métodos para atenderem às necessidades particulares de sua implementação.

Neste módulo, abordaremos alguns dos métodos da classe “Object” – “`toString`”, “`equals`” e “`hashCode`” – e aproveitaremos para discutir as nuances do acesso protegido e do operador “`instanceof`”.

O método “`toString`” permite identificar o objeto retornando uma representação “String” do próprio. Ou seja, trata-se de uma representação textual que assume a forma:

< nome completamente qualificado da classe à qual o objeto pertence >@ < código hash do objeto >.

O < nome completamente qualificado da classe à qual o objeto pertence > é a qualificação completa da classe, incluindo o pacote ao qual ela

Veja um exemplo de saída do método “toString”:

TERMINAL



Nessa saída, notamos o pacote (com.mycompany.GereEscola), a classe (Jurídica) e o hash do objeto (72ea2f77) formando o texto representativo daquela instância.

A implementação do método “toString” é mostrada no Código 15. Nela, identificamos o uso de outros métodos da classe “Object”, como “getClass” e “hashCode”. Não vamos nos aprofundar no primeiro, mas convém dizer que ele retorna um objeto do tipo “Class” que é a representação da classe em tempo de execução do objeto. O segundo será visto mais tarde neste módulo.

JAVA



Código 15: método “toString”.

Essa, porém, é a implementação padrão fornecida pela biblioteca Java, que pode ser prontamente usada pelo programador. Todavia, prover uma especialização é algo interessante, pois permite dar um formato mais útil à saída do método.

Nesse caso, os mecanismos de polimorfismo e herança permitirão que o método mais especializado seja empregado. Remetendo à saída anteriormente mostrada, vemos que há pouca informação, prejudicando sua utilidade.

Além de saber o nome completamente qualificado da classe, temos somente o seu código hash. Assim, vamos modificar o método “toString” para que ele apresente outras informações, conforme a linha 18 da implementação da classe “Pessoa” parcialmente mostrada no Código 16.

A classe “Pessoa” tem “Fisica” como subclasse. A implementação de “Fisica” é mostrada no Código 17.

JAVA



Código 16: classe "Pessoa" – código parcial com método "toString" especializado.

JAVA.



Código 17: classe "Fisica" (derivada de "Pessoa").

JAVA



O Código 18 faz a invocação do método "toString" para dois objetos criados, um do tipo "Pessoa" e outro do tipo "Fisica".

Código 18: invocação da versão especializada de "toString".

A execução do Código 18 tem como saída:

TERMINAL



Veja que o método “toString” agora apresenta outra composição de saída, trazendo informações como o nome e o identificador. Além disso, formatamos a saída para apresentar as informações em múltiplas linhas identadas.

Se olharmos novamente o Código 18, veremos que, na linha 5, o objeto criado é do tipo “Fisica”. Apesar de “Fisica” não reescrever o método “toString”, ela herda a versão especializada de sua superclasse, que é invocada no lugar da versão padrão. Olhando também para a saída, percebemos que o objeto foi corretamente identificado como sendo da classe “Fisica”.

Ao modificar a implementação do método, tornamos seu retorno mais útil. Por esse motivo, ele é frequentemente sobreescrito nas diversas classes providas pela biblioteca Java.

Os métodos “EQUALS” e “HASHCODE”

Outros dois métodos herdados da classe “Object” são “equals” e “hashCode”. Da mesma maneira que o método “toString”, eles têm uma implementação provida por “Object”. Mas também são métodos que, usualmente, iremos especializar, a fim de que tenham semântica útil para as classes criadas.

O método “equals” é utilizado para avaliar se outro objeto é igual ao objeto que invoca o método. Se forem iguais, ele retorna “true”; caso contrário, ele retorna “false”. A sua assinatura é “boolean equals (Object obj)”, e sua implementação é mostrada no Código 19. Já que ele recebe como parâmetro uma referência para um objeto da classe “Object”, da qual todas as classes descendem em Java, ele aceita referência para qualquer objeto das classes derivadas.

JAVA



A implementação de “equals” busca ser o mais precisa possível em termos de comparação, retornando verdadeiro (“true”), e somente se os dois objetos comparados são o mesmo objeto. Ou seja, “equals” implementa uma relação de equivalência, referências não nulas de objetos tal que as seguintes propriedades são válidas:

Reflexividade



Simetria

Transitividade

Consistência

Um caso excepcional do equals é a seguinte propriedade: qualquer que seja uma referência não nula R, R.equals(null) retorna sempre "false".

Dissemos que usualmente a implementação de "equals" é sobreescrita na classe derivada. Quando isso ocorre, é dever do programador garantir que essa relação de equivalência seja mantida. Obviamente, nada impede que ela seja quebrada, dando uma nova semântica ao método, mas se isso ocorrer, então deve ser o comportamento desejado, e não por consequência de erro. Garantir a equivalência pode ser relativamente trivial para tipos de dados simples, mas se torna elaborado quando se trata de classes.

A seguir, veja uma aplicação simples de "equals". Para isso, vamos trabalhar com 9 objetos, dos quais 3 são do tipo "int" e 3, do tipo "String" – ambos tipos primitivos; e 3 são referências para objetos das classes "Pessoa" e "Fisica", conforme consta no Código 20.

Nas linhas 21, 22 e 23 do Código 20, o que fazemos é comparar os diversos tipos de objeto utilizando o método "equals".

JAVA



Código 20: exemplo de uso de "equals".

Nas linhas 21, 22 e 23 do Código 20, o que fazemos é comparar os diversos tipos de objeto utilizando o método "equals". Com essa especialização, a saída agora é:

TERMINAL



Podemos ver que o objeto “l1” foi considerado igual ao “l3”, assim como “S1” e “S3”. Todavia, “p1” e “p2” foram considerados diferentes, embora sejam instâncias da mesma classe (“Fisica”), e seus atributos, rigorosamente iguais. Por quê?

A resposta dessa pergunta mobiliza vários conceitos, a começar pelo entendimento do que ocorre nas linhas 5 e 6. As variáveis “p1”, “p2” e “p3” são referências para objetos das classes “Fisica” e “Pessoa”.

Em contrapartida, as variáveis “l1”, “l2”, “l3”, “S1”, “S2” e “S3” são todas de tipos primitivos. O operador de comparação “==” atua verificando o conteúdo das variáveis que, para os tipos primitivos, são os valores neles acumulados. Mesmo o tipo “String”, que é um objeto, tem seu valor acumulado avaliado. O mesmo ocorre para os tipos de dados de usuário (classes), só que nesse caso “p1”, “p2” e “p3” armazenam o endereço de memória (referência) onde os objetos estão. Como esses objetos ocupam diferentes endereços de memória, a comparação entre “p1” e “p2” retorna “false” (são objetos iguais, mas não são o mesmo objeto). Uma comparação “p1.equals(p1)” retornaria “true” obviamente, pois, nesse caso, trata-se do mesmo objeto.

A resposta anterior também explica por que precisamos sobrescrever “equals” se quisermos comparar objetos. No Código 21, mostramos a reimplementação desse método para a classe “Pessoa”.

JAVA



Com essa especialização, a saída agora é:

Terminal



Contudo, da forma que implementamos “equals”, “p1” e “p2” serão considerados iguais mesmo que os demais atributos sejam diferentes. Esse caso mostra a complexidade que mencionamos anteriormente, em se estabelecer a relação de equivalência entre objetos complexos. Cabe ao programador determinar quais características devem ser consideradas na comparação.

A reimplementação de “equals” impacta indiretamente o método “hash”. A própria documentação de “equals” aponta isso quando diz “que geralmente é necessário substituir o método “hashCode” sempre que esse método (“equals”) for substituído, de modo a manter o contrato geral para o método “hashCode”, que afirma que objetos iguais devem ter códigos “hash” iguais”.

Isso faz todo sentido, já que um código hash é uma impressão digital de uma entidade. Pense no caso de arquivos. Quando baixamos um arquivo da Internet, como uma imagem de um sistema operacional, usualmente calculamos o hash para verificar se houve erro no download. Isto é, mesmo sendo duas cópias distintas, esperamos que, se forem iguais, os arquivos tenham o mesmo hash. Esse princípio é o mesmo por trás da necessidade de se reimplementar “hashCode”.

Nós já vimos o uso desse método quando estudamos “toString”, e a saída do Código 18 mostra o resultado do seu emprego. A sua assinatura é “int hashCode()”, e ele tem como retorno o código hash do objeto.

Esse método possui as seguintes propriedades:

Invocações sucessivas sobre o mesmo objeto devem consistentemente retornar o mesmo valor, dado que nenhuma informação do objeto usada pela comparação em “equals” tenha mudado.

Se dois objetos são iguais segundo “equals”, então a invocação de “hashCode” deve retornar o mesmo valor para ambos.

Caso dois objetos sejam desiguais segundo “equals”, não é obrigatório que a invocação de “hashCode” em cada um dos dois objetos produza resultados distintos. Mas produzir resultados distintos para objetos desiguais pode melhorar o desempenho das tabelas hash.

A invocação de “hashCode” nos objetos “p1” e “p2” do Código 20 nos dá a seguinte saída:

TERMINAL



Esse resultado contraria as propriedades de “hashCode”, uma vez que nossa nova implementação de “equals” estabeleceu a igualdade entre “p1” e “p2” instanciados no Código 18. Para restaurar o comportamento correto, fornecemos a especialização de “hashCode” vista no Código 22.

JAVA



Código 22: reimplementação de “hashCode”.

Veja que agora nossa implementação está consistente com a de “equals”. Na linha 2 do Código 22, asseguramo-nos de que se trata de um objeto da classe “Pessoa”. Se não for, chamamos a implementação padrão de “hashCode”. Caso seja, retornamos o valor de hash da “String” que forma o atributo “nome”, o mesmo usado em “equals” para comparação. A saída é:

TERMINAL



Agora, conforme esperado, “p1” e “p2” possuem os mesmos valores.



Comentário

Em ambos os casos, oferecemos uma reimplementação simples para corrigir o comportamento. Mas você deve estar preparado para necessidades bem mais complexas. O programador precisa, no fundo, compreender o comportamento modelado e as propriedades desses métodos, para que sua modificação seja consistente.

O operador “INSTANCEOF”

O operador “instanceof” é utilizado para comparar um objeto com um tipo específico. É o único operador de comparação de tipo fornecido pela linguagem.

tipo "op2". Então, devemos esperar que se "op1" for uma subclasse de "op2", o retorno será "true". Vejamos se isso é válido executando o Código 23.

JAVA



Código 23: operador "instanceof"

Nesse código, fazemos quatro comparações, cujos resultados são:

Terminal



Assim, o código valida nossa expectativa: "instanceof" retorna verdadeiro para uma instância da subclasse quando comparada ao tipo da superclasse. Tal resultado independe se a variável é uma referência para a superclasse (linha 9) ou a própria subclasse (linha 10).

Entendendo o acesso protegido

Já abordamos o acesso protegido em outras partes. De maneira geral, ele restringe o acesso aos atributos de uma classe ao pacote dessa classe e às suas subclasses. Portanto, uma subclasse acessará todos os métodos e atributos públicos ou protegidos da superclasse, mesmo se

Agora vamos explorar algumas situações particulares. Para isso, vamos criar o pacote "Matemática" e, doravante, as classes que temos utilizado ("Principal", "Pessoa", "Fisica", "Juridica", "Aluno" etc.) estarão inseridas no pacote "GereEscola".

No pacote "Matemática", vamos criar a classe "Nota", que implementa alguns métodos de cálculos relativos às notas dos alunos. Essa classe pode ser parcialmente vista no Código 24.

JAVA



Código 24: código parcial da classe "Nota".

No pacote "GereEscola", vamos criar a classe "Desempenho", conforme mostrado, parcialmente, no Código 25.

JAVA



Código 25: classe "Desempenho".

Observando a linha 11, vemos que ela também possui um objeto do tipo “Nota”, instanciado na linha 15.

Sendo assim, será que poderíamos comentar as linhas 15 e 16, substituindo-as pelas linhas 18 e 19? A resposta é **não**.

Se descomentarmos a linha 18, não haverá problema, mas a linha 19 causará erro de compilação.

Isso ocorre porque “Desempenho” tem acesso ao método protegido “calcularCoeficienteRendimento” por meio da herança. Por esse motivo, ele pode ser invocado diretamente na classe (linha 17). Mas a invocação feita na linha 19 se dá por meio de uma mensagem enviada para o objeto “nota”, violando a restrição de acesso imposta por “protected” para objetos de pacotes distintos.

Outra violação de acesso ocorrerá se instanciarmos a classe “Desempenho” em outra parte do pacote “GereEscola” e tentarmos invocar o método “calcularCoeficienteRendimento”, conforme o trecho mostrado no Código 26.

JAVA



Código 26: exemplo de invocação ilegal de “calcularCoeficienteRendimento”.

Na linha 9, a tentativa de invocar “calcularCoeficienteRendimento” gerará erro de compilação, pois apesar de a classe “Desempenho” ser do mesmo pacote que “NovaClasse”, o método foi recebido por “Desempenho” por meio de herança de superclasse de outro pacote. Logo, como impõe a restrição de acesso protegida, ele não é acessível por classe de outro pacote que não seja uma descendente da superclasse.

Falta pouco para atingir seus objetivos.

Vamos praticar alguns conceitos?

Questão 1

Sobre os métodos “equals” e “hashCode” da classe “Objects”, podemos afirmar que

- A mudanças em “equals” podem ensejar mudanças em “hashCode”, mas o inverso nunca ocorre.
- B como “equals” é herdado por todos os objetos, não necessita ser modificado para estabelecer a igualdade de objetos.
- C o método “equals” sempre depende de “hashCode” para estabelecer a igualdade.
- D a modificação de um sempre imporá a modificação de outro.
- E objetos iguais terem código hash distintos viola o contrato geral de “hashCode”.

Responder

Um dos métodos da classe "Objects" é o "toString", que retorna uma representação textual do objeto. Sobre esse método, são feitas as seguintes afirmações:

- I) São informações obrigatórias, mesmo se ele for redefinido, o nome completamente qualificado do objeto e seu código hash.
- II) Ele pode ser invocado em um objeto do tipo String.
- III) O código hash informado por "toString" é um identificador único do objeto.

Marque a opção que contém a afirmativa correta:

A I

B II

C III

D I e II

E II e III

Responder

3

Polimorfismo

Ao final deste módulo você será capaz de descrever polimorfismo em Java.



Classes e Polimorfismo em Java

O especialista abordará os conceitos de polimorfismo no vídeo a seguir.

Neste módulo, vamos avançar no estudo de polimorfismo. Introduziremos o conceito de classes abstratas e veremos como a abstração de classes funciona em Java; como esse conceito afeta o polimorfismo e como ele é aplicado numa hierarquia de herança.

O módulo contempla, também, o estudo do modificador “final” e seu impacto, assim como as interações com as variáveis da super e subclasse.

No nosso estudo de hierarquia de herança e seus desdobramentos, constantemente voltamos aos conceitos de especialização e generalização. Quanto mais subimos na hierarquia, tanto mais esperamos encontrar modelos genéricos.

A generalização de modelos corresponde à generalização dos comportamentos implementados pelas superclasses. Vimos isso ocorrer no modelo mostrado na figura “Especialização de comportamento”, no qual a classe “Pessoa” representa a generalização de mais alto nível dessa hierarquia.

Se você revir os métodos da classe “Pessoa”, notará que sempre buscamos dar um comportamento genérico. Veja, como exemplo, o caso do método “atualizarID”, mostrado no Código 4. Para a classe “Pessoa” nenhuma crítica é feita ao se atualizar o atributo “identificador”, ao contrário de suas subclasses “Fisica” e “Juridica”, que aplicam teste para verificar se o identificador é válido.

Apesar de “atualizarID” ser uma generalização de comportamento, se você refletir bem, notará que a implementação é provavelmente inútil. Qualquer caso de instanciação num caso real deverá fazer uso das versões especializadas.

Mesmo no caso de um aluno que ainda não possua CPF, é mais razoável prever isso no comportamento modelado pela versão especializada de “atualizarID” da subclasse “Pessoa”, pois esse aluno é “pessoa física”. Podemos esperar que, em algum momento, ele venha a ter um CPF e, se usar



Atenção!

Pensando em termos de flexibilidade, o que esperamos de “Pessoa” é que ela defina o contrato comum a todas as classes derivadas e nos forneça o comportamento comum que não necessite de especialização.

Vejamos o exemplo a seguir:

O comportamento de “atualizarNome” (vide Código 1)



Classes e Métodos Abstratos

O propósito de uma classe abstrata é, justamente, o que mencionamos antes: fornecer uma interface e comportamentos (implementações) comuns

Em Java, uma classe é declarada abstrata pela aplicação do modificador “abstract” na declaração. Isto é, podemos declarar a classe “Pessoa” como abstrata simplesmente fazendo “public abstract class Pessoa {...}”.

Já se a instrução “abstract” for aplicada a um método, este passa a ser abstrato. Isso quer dizer que ele não pode possuir implementação, pois faz parte do contrato (estrutura) fornecido para as subclasses.

Desse modo, a classe deverá, obrigatoriamente, ser declarada abstrata também. No exemplo em questão, transformamos “atualizarID” em abstrato, declarando-o “protected abstract void atualizarID (String identificador)”.

Observe que, agora, o método não pode possuir corpo.

Uma nova versão da classe “Pessoa” é mostrada no Código 27. Nessa versão, o método “atualizarID” é declarado abstrato, forçando a declaração da classe como abstrata. Entretanto, o método “recuperarID” é concreto, uma vez que não há necessidade para especializar o comportamento que ele implementa.

JAVA



Código 27: classe “Pessoa” declarada como abstrata.

Em Java, o efeito de declarar uma classe como abstrata é impedir sua instanciação. Quando um método é declarado abstrato, sua implementação é postergada. Esse método permanecerá sendo herdado como abstrato até que alguma subclasse realize sua implementação. Isso quer dizer que a abstração de um método se propaga pela hierarquia de classes. Por extensão, uma subclasse de uma classe abstrata será também abstrata a menos que implemente o método abstrato da superclasse.

Apesar de mencionarmos que uma classe abstrata fornece, também, o comportamento comum, isso não é uma obrigação. Nada impede que uma classe abstrata apresente apenas a interface ou apenas a implementação. Aliás, uma classe abstrata pode ter dados de instância e construtores.

Vejamos o uso da classe abstrata “Pessoa” no seguinte trecho de código:

JAVA



Código 28: polimorfismo com classe abstrata.

A linha 7 do Código 28 cria um vetor de referências para objetos do tipo “Pessoa”. Nas linhas 14 e 16, são instanciados objetos do tipo “Fisica” e “Juridica”, respectivamente. Esses objetos são referenciados pelo vetor “ref”. Nas linhas 17 e 18, é invocado o método “atualizarID”, que é abstrato na superclasse, mas concreto nas subclasses.

Assim, o sistema determinará em tempo de execução o objeto, procedendo à vinculação dinâmica do método adequado. Esse é tipicamente um comportamento polimórfico. Na linha 17, o método invocado pertence à classe “Fisica” e na linha 18, à classe “Juridica”.

Métodos e classes “Final”

Vamos analisar outra hipótese agora. Considere o método “recuperarID” mostrado no Código 4. Esse método retorna o identificador na forma de uma “String”. Esse comportamento serve tanto para o caso de “Fisica” quanto de “Juridica”. Em verdade, é muito pouco provável que seja necessário especializá-lo, mesmo se, futuramente, uma nova subclasse for adicionada.

Esse é um dos comportamentos comuns que mencionamos antes e, por isso, ele está na superclasse. Mas se não há motivo para que esse método possa ser rescrito por uma subclasse, é desejável que possamos impedir que isso ocorra inadvertidamente. Felizmente, a linguagem Java fornece um mecanismo para isso. Trata-se do modificador “final”.

Esse modificador pode ser aplicado à classe e aos membros da classe. Diferentemente de “abstract”, declarar um método “final” não obriga que a classe seja declarada “final”. Porém, se uma classe for declarada “final”, todos os seus métodos são, implicitamente, “final” (isso não se aplica aos seus atributos).

Métodos “final” não podem ser redefinidos nas subclasses. Dessa forma, se tornarmos “recuperarID” “final”, impediremos que ele seja modificado, mesmo por futuras inclusões de subclasses. Esse método permanecerá imutável ao longo de toda a hierarquia de classes. Métodos “static” e “private” são, implicitamente, “final”, pois não poderiam ser redefinidos de qualquer forma.



Comentário

O uso de “final” também permite ao compilador realizar otimizações no código em prol do desempenho. Como os métodos desse tipo nunca podem ser alterados, o compilador pode substituir as invocações pela cópia do código do método, evitando desvios na execução do programa.

Vimos que, quando aplicados à classe, todos os seus métodos se tornam “final”. Isso quer dizer que nenhum deles poderá ser redefinido. Logo, não faz sentido permitir que essa classe seja estendida, e a linguagem Java proíbe que uma classe “final” possua subclasses. Contudo, ela pode possuir uma superclasse.

Quando aplicada a uma variável, “final” irá impedir que essa variável seja modificada e exigirá sua inicialização. Esta pode ser feita junto da declaração ou no construtor da classe. Quando inicializada, qualquer tentativa de modificar seu valor gerará erro de compilação.

Como podemos ver no Código 29, uma classe abstrata pode ter membros “final”. A variável “dias letivos” foi declarada “final” (linha 9) e, por isso, precisa ser inicializada, o que é feito na linha 14, no construtor da classe. A não inicialização dessa variável assim como a tentativa de alterar seu valor gerarão erro em tempo de compilação.



Código 29: uso de "final".

O método “calcularFrequencia” (linha 13) foi declarado “final”, não podendo ser redefinido. Mas, como dito, a classe não é “final” e pode, então, ter classes derivadas.

Atribuições permitidas entre variáveis de superclasse e subclasse

Já que estamos falando de como modificadores afetam a herança, é útil, também, entender como as variáveis da superclasse e da subclasse se relacionam para evitar erros conceituais, difíceis de identificar e que levam o software a se comportar de maneira imprevisível.

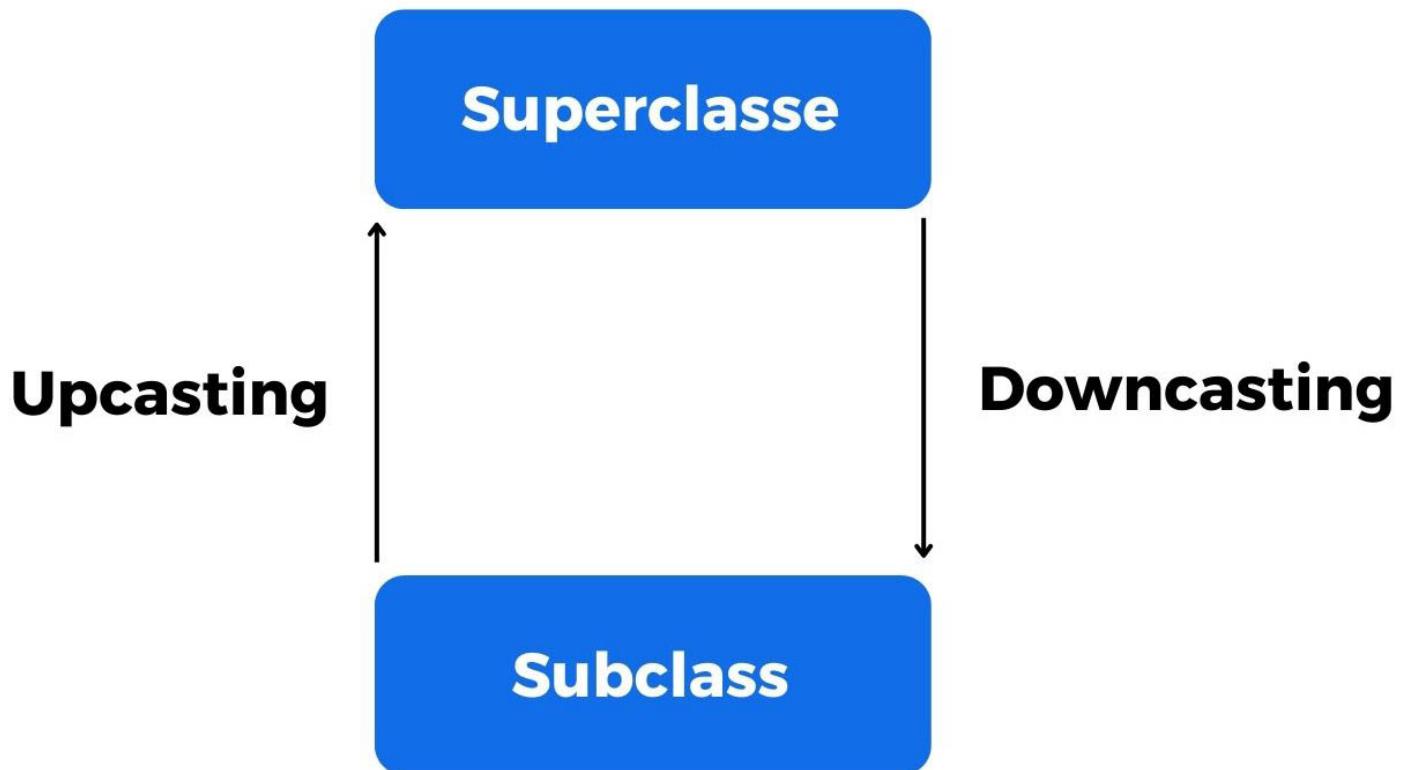
Vamos começar lembrando que, quando uma variável é declarada “private”, ela se torna diretamente inacessível para as classes derivadas. Como vimos, nesse caso, elas são implicitamente “final”.

Elas ainda podem ter seu valor alterado, mas isso só pode ocorrer por métodos públicos ou privados providos pela superclasse. Pode soar contraditório, mas não é. As atualizações, feitas pelos métodos da superclasse, ocorrem no contexto desta, no qual a variável não foi declarada “final” e nem é, implicitamente, tomada como tal.

Podemos acessar membros públicos ou protegidos da classe derivada a partir da superclasse ou de outra parte do programa, fazendo downcasting da referência.

O downcasting leva o compilador a reinterpretar a referência empregada como sendo do tipo da subclasse. É evidente que isso é feito quando a variável de referência é do tipo da superclasse. Simetricamente, o upcasting causa a reinterpretation de uma variável de referência da subclasse como se fosse do tipo da superclasse. A imagem a seguir ilustra o que acabamos de dizer.

O caso de variáveis privadas é óbvio. Vejamos o que ocorre com as variáveis públicas e protegidas. Como, no escopo da herança, as duas se comportam da mesma maneira, quer dizer, como ambas são plenamente acessíveis entre as classes pai e filha, trataremos, apenas, de variáveis protegidas. O mesmo raciocínio se aplicará às públicas.



Reinterpretação de referência.

Para esclarecer o comportamento, usaremos as classes mostradas no Código 30 e no Código 31. O Código 32 implementa o programa principal.

JAVA



Código 30: classe Base.

JAVA



Código 31: classe Derivada.

JAVA



Código 32: variáveis protegidas na hierarquia de classes.

Esse código tem como saída:

TERMINAL



Em primeiro lugar, o código mostra que “var_base” representa uma variável compartilhada por ambas as classes da hierarquia. Então, a mudança por meio de uma referência para a subclasse afeta o valor dessa variável na superclasse. Isso é esperado, pois é o mesmo espaço de memória.

A linha 6 do Código 32 deixa isso claro. Ao instanciar o objeto do tipo “Derivada”, seu construtor é chamado. A primeira ação é imprimir o valor da variável “var_base” da superclasse. Como a classe pai é instanciada primeiro, “var_base” assume valor -1. Em seguida, é sobreescrita na classe derivada com o valor -2, e uma nova impressão do seu conteúdo confirma isso.

A linha 10 do Código 32 chama o método “atualizarVarSub”, que atualiza o valor de “var_der” da subclasse, fazendo o downcasting do atributo (linha 13 do Código 30). As impressões de “var_der” confirmam que seu valor foi modificado.

Falta pouco para atingir seus objetivos.

Vamos praticar alguns conceitos?

Questão 1

Uma classe abstrata em Java é declarada pelo uso do modificador abstract. Analise as opções a seguir e marque a correta sobre o assunto.

A Se declararmos uma classe como abstrata, seus métodos também deverão ser declarados abstratos.

B Uma classe abstrata pode estender uma classe concreta.

C Uma classe abstrata admite herança múltipla se as superclasses também forem abstratas.

- D Uma classe concreta não pode possuir referência para uma classe abstrata.
- E Um método abstrato não pode ser herdado.

Responder

Questão 2

Considere o trecho de código a seguir:

Java



O método “atualizarAtributoNome” atualiza o atributo “nome” de tipo “String” da classe “Escola”. É correto afirmar que

- A a linha 7 gerará erro de compilação, pois “ref” é final.
- B a linha 8 criará uma instância que não é final.
- C a linha 8 gerará erro de compilação, pois “ref” é final.
- D a linha 3 gerará erro de compilação, pois está instanciando uma variável final.

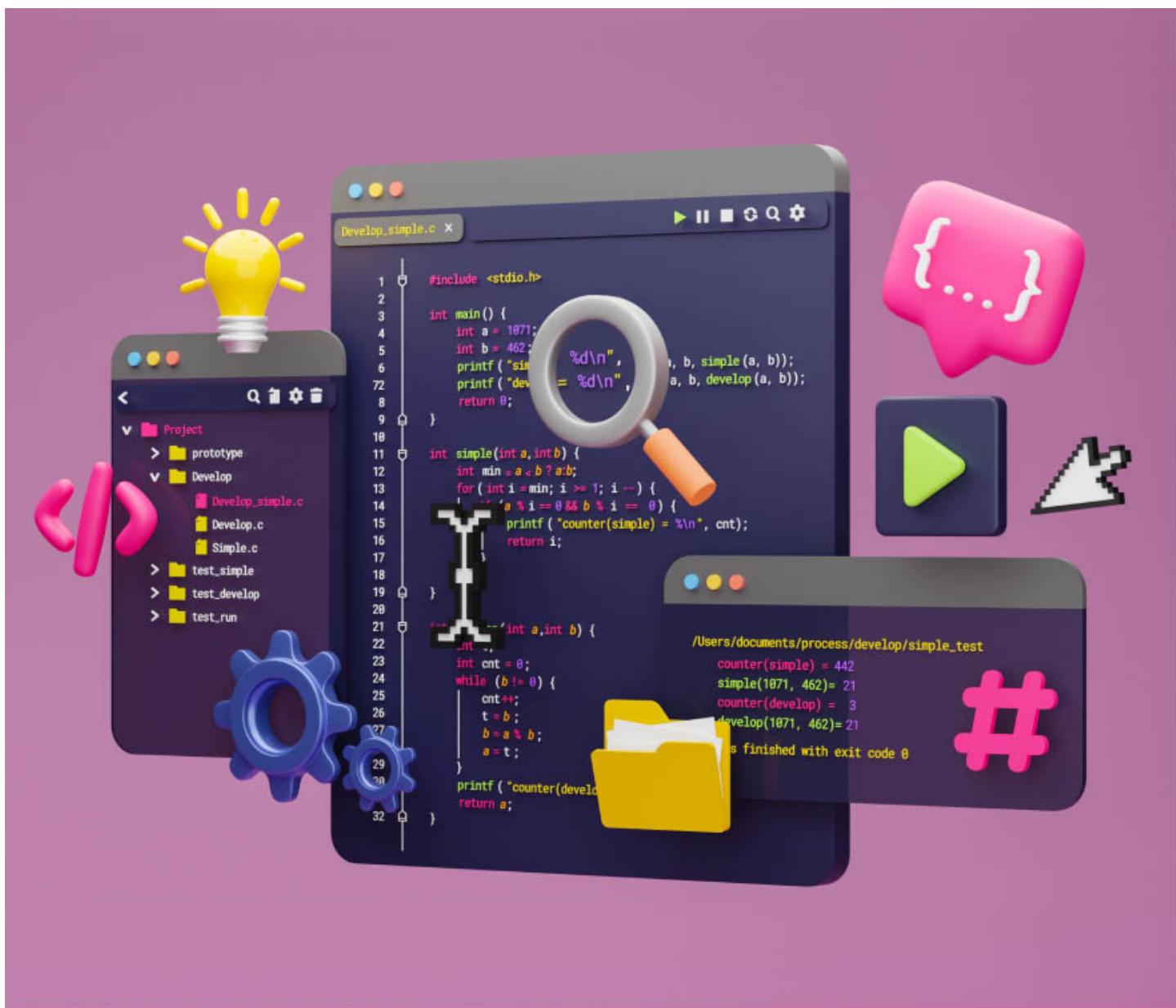
E As linhas 7 e 8 precisam ser invertidas para o programa compilar.

Responder

4

Interfaces

Ao final deste módulo, você será capaz de aplicar a criação e o uso de interfaces em Java.



A entidade “Interface”



Interfaces

No vídeo a seguir, apresentaremos os conceitos de interfaces e exemplos do emprego de herança em Java.

Entendendo a entidade “Interface”

Neste módulo, estudaremos um artefato da linguagem Java chamado de Interfaces. Esse é outro conceito de OO suportado pela linguagem. Primeiramente, vamos entender o conceito e a entidade “Interface” da linguagem Java. Prosseguiremos estudando, com maior detalhamento, seus aspectos e terminaremos explorando seu uso.

Uma interface é um elemento que permite a conexão entre dois sistemas de natureza distintos, que não se conectam diretamente.



Exemplo

Um teclado fornece uma interface para conexão homem-máquina, bem como as telas gráficas de um programa, chamadas Graphic User Interface (GUI) ou Interface Gráfica de Usuário, que permitem a ligação entre o usuário e o software.

O paradigma de programação orientada a objetos busca modelar o mundo real por meio de entidades virtuais. Já discutimos as classes e os objetos como entidades que fazem tal coisa. A interface tem o mesmo propósito.



Exemplo

Podemos movê-lo no plano, e o sensor de movimento transformará esse movimento em deslocamentos do ponteiro na tela. Podemos pressionar cada um dos três botões individualmente ou de maneira combinada; e podemos girar a roda para frente e para trás. Todas essas ações físicas são transformadas pelo mouse em comandos e enviados ao computador.

Esta é toda a possibilidade de interação física com o mouse. A interface de interação com o mouse é exatamente o que acabamos de descrever. Ela é uma especificação das interações possíveis com o dispositivo, que, no entanto, não define o comportamento. Sabemos que podemos pressionar o botão direito para interagir, mas o comportamento que isso desencadeará dependerá do programa em uso.

Trazendo nosso exemplo para o software, podemos criar um mouse virtual com o intuito de fazer simulações. Nesse caso, para que nosso mouse represente o modelo físico, devemos impor que sua interface de interação com o usuário seja a mesma.

Dessa forma, independentemente do tipo de mouse modelado (ótico, laser, mecânico), todos terão de oferecer a mesma interface e versão ao usuário: detecção de movimento, detecção de pressionamento dos botões 1, 2 e 3 e detecção de movimento da roda. Ainda que cada mouse simule sua própria versão do mecanismo.

Podemos então dizer que uma interface no paradigma OO é uma estrutura que permite garantir certas propriedades de um objeto. Ela permite definir um contrato padrão para a interação, que todos deverão seguir, isolando do mundo exterior os detalhes de implementação.

Uma definição de interface pode ser dada pelo conjunto de métodos a seguir. Qualquer programador que deseje estender nosso mouse deverá obedecer a esse contrato.



Particularidades da “Interface”

Em C++, o conceito de interface é suportado por meio de classes virtuais puras, também chamadas **classes abstratas**. A linguagem Java possui uma entidade específica chamada “Interface”. A sintaxe para se declarar uma interface em Java é muito parecida com a declaração de classe: “public interface Nome { ... }”. No entanto, uma interface não admite atributos e não pode ser instanciada diretamente.

Ela é um tipo de referência e somente pode conter constantes, assinaturas de métodos, tipos aninhados, métodos estáticos e default. Apenas os métodos default e estático podem possuir implementação. Tipicamente, uma interface declara um ou mais métodos abstratos que são implementados pelas classes.

Interfaces também permitem herança, mas a hierarquia estabelecida se restringe às interfaces. Uma interface não pode estender uma classe e vice-versa. Contudo, diferentemente das classes, aquelas admitem herança múltipla, o que significa que uma interface pode derivar de duas superinterfaces (interfaces pai). Nesse caso, a derivada herdará todos os métodos, as constantes e os outros tipos membros da superinterface, exceto os que ela sobrescreva ou oculte.

Uma classe que implemente uma interface deve declará-la explicitamente pelo uso do modificador “implements”. É possível a uma classe implementar mais de uma interface. Nesse caso, as implementadas devem ser separadas por vírgula.

Veja, a seguir, um exemplo de implementação de interfaces por uma classe:



Código 33: interface "Identificador".

JAVA



Código 34: interface "iPessoa".

JAVA



Código 35: implementação das interfaces "Identificador" e "iPessoa" pela classe "Pessoa".

Veja que na linha 1 do Código 35 foi declarado que a classe "Pessoa" implementará as interfaces "iPessoa" e "Identificador"

Quando uma classe implementa uma ou mais interfaces, ela deve implementar todos os métodos abstratos das interfaces.

E de fato é o que ocorre. A classe, contudo, não pode alterar a visibilidade dos métodos da interface. Assim, métodos públicos não podem ser tornados protegidos por ocasião da implementação.

Lembre-se de que uma interface define um contrato. Sua finalidade é proporcionar a interação com o mundo exterior. Por isso, não faz sentido a restrição de acesso aos métodos da interface.

Ou seja, numa interface, todos os métodos são públicos, mesmo se omitirmos o modificador “public”, como no Código 33 e Código 34. Mais ainda, em uma interface, todas as declarações de membro são implicitamente estáticas e públicas.

A declaração de um membro em uma interface oculta toda e qualquer declaração desse membro nas superinterfaces.



Exemplo

Se uma interface derivada de “Identificador” declarar um método com o nome “validarID”, a correspondente declaração desse método no Código 33 tornar-se-á oculta. Assim, uma interface herda de sua superinterface imediata todos os membros não privados que não sejam escondidos.

Podemos fazer declarações aninhadas, isto é, declarar uma interface no corpo de outra. Nesse caso, temos uma interface aninhada. Quando uma interface não é declarada no corpo de outra, ela é uma interface do nível mais alto (top level interface). Uma interface também pode conter uma classe declarada no corpo.

Observando o Código 33, vemos, na linha 2, que a interface possui um atributo estático (“tamanho_max”). Esse é um caso de atributo permitido em uma interface. Também não é possível declarar os métodos com o corpo vazio em uma interface. Imediatamente após a assinatura, a cláusula deve ser fechada com “;”. Java também exige que sejam utilizados identificadores nos parâmetros dos métodos, não sendo suficiente informar apenas o tipo.

Além do tipo normal de interface, existe um especial, o Annotation, que permite a criação de tipos de anotações pelo programador. Ele é declarado precedendo-se o identificador “interface” com o símbolo “@”, por exemplo: “@interface Preambulo {...}”. Uma vez definido, esse novo tipo de anotação torna-se disponível para uso juntamente com os tipos built-in.

Fica claro, pelo que estudamos até aqui, que as interfaces oferecem um mecanismo para ocultar as implementações. São, portanto, mecanismos que nos permitem construir as API (Application Programming Interface) de softwares e bibliotecas.



Curiosidade

As API possibilitam que o código desenvolvido seja disponibilizado para uso por terceiros, mantendo-se oculta a implementação, tendo muita utilidade quando não se deseja compartilhar o código que realmente implementa as funcionalidades, seja por motivos comerciais, de segurança, de proteção de conhecimento ou outros.

A pergunta que você deve estar se fazendo é:

Qual a diferença entre classes abstratas e interfaces?

Essa é uma ótima pergunta. A resposta está no tópico seguinte.

Diferença entre Classe Abstrata e Interface

Uma classe abstrata define um tipo abstrato de dado. Define-se um padrão de comportamento segundo o qual todas as classes que se valham dos métodos da classe abstrata herdarão da classe que os implementar.



Relembrando

Uma classe abstrata pode possuir estado (atributos) e membros privados, protegidos e públicos. Isso é consistente com o que acabamos de dizer e muito mais do que uma interface pode possuir, significa que as classes abstratas podem representar ou realizar coisas que as interfaces não podem.

É claro que uma classe puramente abstrata e sem atributos terminará assumindo o comportamento de uma interface. Mas elas não serão equivalentes, já que uma interface admite herança múltipla, e as classes, não.

Uma interface é uma maneira de se definir um contrato. Se uma classe abstrata define um tipo (especifica o que um objeto é), uma interface especifica uma ou mais capacidades. Veja o caso do Código 33. Essa interface define as capacidades necessárias para se manipular um identificador. Não se trata propriamente de um tipo abstrato, pois tudo que ela oferece é o contrato.

Não há estado nem mesmo comportamentos ocultos. Então, uma classe que implementar essa interface proverá o comportamento especificado pela interface.



Atenção!

Cuidado para não confundir a linha 2 com um estado da instância, pois não é. Além de essa variável ser declarada como final, não há métodos que atuam sobre ela para modificar seu estado.

Esclarecidas as diferenças entre classes abstratas e interfaces, resta saber quando usá-las. Essa discussão é aberta, e não há uma regra. Isso dependerá muito de cada caso, da experiência do programador e da familiaridade com o uso de ambas. Mas, com base nas diferenças apontadas, podemos estabelecer uma regra geral.

Quando seu objetivo for **especificar** as capacidades que devem ser disponibilizadas, a interface é a escolha mais indicada.



Quando estiver buscando **generalização** de comportamento e compartilhamento de código e atributos comuns (um tipo abstrato de dado), classes abstratas surgem como a opção mais adequada.

Veja a Figura Hierarquia de herança – Collections e a Figura: Hierarquia de herança – Map. Ambas mesclam o uso de interfaces e classes abstratas. Na Figura Hierarquia de herança – Collections, as interfaces “Set”, “List” e “Queue”, para citar apenas alguns, definem as capacidades que estão ligadas ao contrato que deverá ser estabelecido para a estrutura de dados com que cada uma se relaciona.

As classes abstratas, como “AbstractList”, vão oferecer implementações compartilhadas. No caso de “AbstractList”, podemos ver que pelo menos duas classes, “ArrayList” e “Vector”, fazem uso dela. Essa figura, que mostra o modelo das coleções da API Java, é um ótimo exemplo de que os conceitos de classes abstratas e de interfaces são complementares e, portanto, não se trata de escolher entre uma e outra.

Uso de interfaces

Vimos a declaração das interfaces “Identificador” e “iPessoa” e da classe “Pessoa”, que as implementa, respectivamente no Código 33, no Código 34 e no Código 35. Nota-se que todos os métodos abstratos de ambas as interfaces foram implementados em “Pessoa”. O método “main” do programa, mostrado no Código 36, permite explorar seu uso.

JAVA



Código 36: trabalhando com interfaces.

Nas linhas 3 e 4 do Código 36, declaramos duas variáveis, uma que referencia “Identificador” e outra “iPessoa”. Recorde-se de que dissemos que uma interface não admite instânciação direta, mas nenhuma restrição fizemos quanto a se ter uma variável declarada como referência para a interface. E, de fato, na linha 8, não estamos instanciando a interface, mas a classe “Pessoa”. O objeto instanciado, contudo, será referenciado pela variável “refId”.



Podemos usar uma variável que guarda referências do tipo da interface para referenciar objetos da classe que a implementa. Entretanto, não é suficiente que a classe forneça o comportamento dos métodos abstratos, ela precisa explicitamente declarar implementar a interface.

Uma vez instanciado o objeto, podemos invocar o comportamento dos métodos especificados por "Identificador". Mas apenas esses métodos e os que a interface "Identificador" herda estarão disponíveis nesse caso.

Nenhum dos métodos especificados em "iPessoa", apesar de implementados em "Pessoa", ou mesmo outros métodos implementados nela, estarão acessíveis por meio dessa referência ("refIdt"). Por isso, "descomentar" a linha 10 gerará erro de compilação.

Não é possível fazermos a atribuição de "refIdt" para "refiPessoa", pois os tipos não são compatíveis. Todavia, se fizermos um typecasting, como o mostrado na linha 12, a atribuição se tornará possível. Agora, podemos usar a variável "refiPessoa" e acessar os métodos dessa interface que foram implementados em "Pessoa".

É claro que, se tivéssemos usado uma variável do tipo "Pessoa" para referenciar o objeto instanciado, todos os métodos estariam prontamente acessíveis.

A saída do Código 36 é:

TERMINAL



Falta pouco para atingir seus objetivos.

Vamos praticar alguns conceitos?

Questão 1

Sobre interfaces em Java, é correto afirmar que

- A a omissão do modificador “final” da declaração de um atributo gerará erro de compilação.
- B um atributo deve sempre ser declarado privado.
- C na implementação de uma interface, uma classe precisa implementar apenas os métodos abstratos que usará.
- D em uma interface, não é necessário declarar os métodos como abstratos.
- E nenhum membro de uma interface pode conter corpo.

[Responder](#)

Questão 2

Suponha que um programa em Java possua a interface “iContabil”, que é superinterface de “iBalanco”. “iBalanco” é implementada pela classe concreta “Balanco”. Julgue as afirmativas:

- I) Podemos usar uma variável do tipo “iContabil” para referenciar um objeto do tipo “Pessoa” e, nesse caso, teremos acesso aos métodos de “iContabil” e “iBalanco”.
- II) Não é possível usar uma variável do tipo da superinterface para referenciar um objeto da classe que implementa “iBalanco”.
- III) “Balanco” deverá implementar todos os métodos abstratos de “iBalanco” e os de “iContabil” que não forem ocultados por “iBalanco”.

A afirmativa correta é

A somente a I.

B somente a II.

C somente a III.

D I e II.

E II e III.

Responder

Considerações finais

Neste conteúdo, tivemos a oportunidade de aprofundar o estudo da linguagem Java. Fizemos isso investigando, em maiores detalhes, a hierarquia de herança. Compreendemos como ocorre a subtipagem e sua diferença para a subclasse, assim como vimos a vinculação dinâmica de métodos. Esses tópicos são essenciais para a melhor compreensão do polimorfismo e da programação orientada a objetos.

Nossa exploração nos mostrou os desdobramentos da herança e as potencialidades do polimorfismo. Vimos isso tanto por exemplos, como observando modelos da própria API Java. Também aprendemos sobre coleções, um tipo de interface da API que oferece importantes ferramentas de estrutura de dados para o programador.

Concluímos conhecendo a entidade “Interface” e sua relação com os conceitos vistos anteriormente. Tudo isso nos trouxe um importante entendimento sobre as potencialidades da linguagem Java e da programação OO.

Esses conceitos não são apenas uma base importante, eles são indispensáveis para qualquer programador que deseje tirar proveito de tudo o que a linguagem Java tem a oferecer.



00:00

12:56



Explore +

Os princípios Solid formam um importante conjunto que todo bom programador deve dominar. Nesse módulo vimos apenas um, o princípio da substituição. Sugerimos que você procure e domine, também, os demais. Para isso, uma busca como os termos princípios Solid ou com cada um deles deve lhe trazer bons resultados.

As coleções, em Java, são outras ferramentas que merecem ser dominadas por quem quer melhorar suas habilidades. Uma boa fonte de pesquisa sobre cada tipo de coleção é a documentação Java disponibilizada pela Oracle.

Por fim, estude os códigos da API das coleções para aumentar o entendimento de como interfaces funcionam e se integram com as classes concretas e abstratas. A documentação de Java da Oracle é um bom ponto de partida!

Referências

ORACLE AMERICA INC. **Java 10 Local Variable Type Inference**. Oracle Developer Resource Center, 2020.

ORACLE AMERICA INC. **Lesson: Introduction to Collections** (The JavaTM Tutorials > Collections). Java Documentation, 2020.

ORACLE AMERICA INC. **Object (Java SE 15 & JDK 15)**. Java Documentation, 2020.

ORACLE AMERICA INC. **Primitive Data Types (The JavaTM Tutorials > Learning the Java Language > Language Basics)**. The Java Tutorial, 2020.

Material para download

Clique no botão abaixo para fazer o download do conteúdo completo em formato PDF.

Download material

O que você achou do conteúdo?



Relatar problema



FASES DO DESENVOLVIMENTO DE SOFTWARE

DESCRIÇÃO

O processo de desenvolvimento de software e suas atividades genéricas.

PROpósito

Compreender as etapas do processo de desenvolvimento de software, que abrange atividades de levantamento de requisitos, análise, projeto, implementação, teste, implantação e manutenção.

OBJETIVOS

MÓDULO 1

Descrever as atividades da Engenharia de Requisitos do processo de desenvolvimento de software

MÓDULO 2

Reconhecer as atividades do projeto de software do processo de desenvolvimento de software

MÓDULO 3

Reconhecer as etapas de implementação e testes do processo de desenvolvimento de software

MÓDULO 4

Descrever as etapas de implantação e manutenção do processo de desenvolvimento de software

INTRODUÇÃO

Nos dias atuais, o software tem destaque cada vez maior. Uma tendência que ratifica essa importância está no uso intensivo dos *smartphones*.

O sucesso dessa tecnologia, não descartando a importância do hardware, tem um forte respaldo na competência do software, ou seja, interfaces com *designs* atraentes, responsivas, usabilidade intuitiva, entre outras características.

Por trás dessa competência, temos vários especialistas, tais como engenheiros de software, *web designers*, administradores de banco de dados, arquitetos de software e outros que

trabalham arduamente para manutenção desse sucesso exponencial.

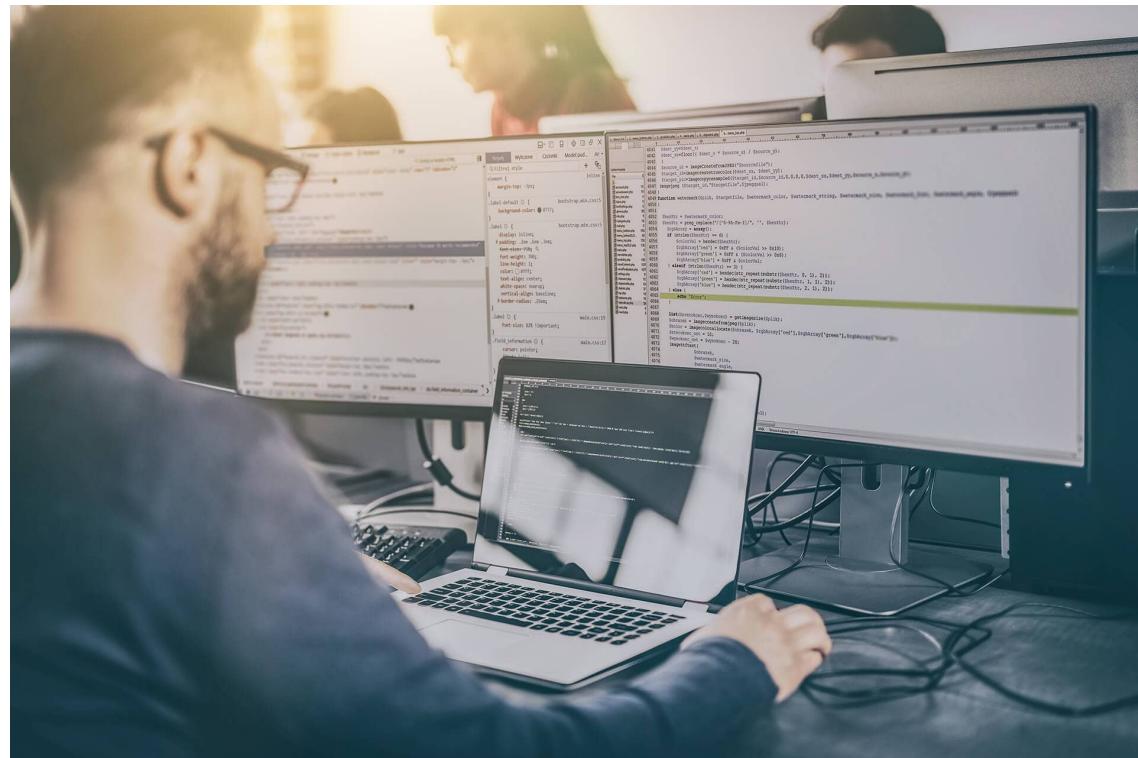
A referida equipe tem uma composição multidisciplinar, refletindo uma característica do software: a complexidade . O trato dessa complexidade requer a aplicação da Engenharia de Software, que, por sua vez, tem como base a camada de processos. Podemos afirmar que não existe engenharia sem processo.

Nesse contexto, é importante compreender as principais atividades desenvolvidas nas etapas genéricas de um Processo de Desenvolvimento de Software, também conhecido pela sigla PDS.

MÓDULO 1

-
- Descrever as atividades da Engenharia de Requisitos do processo de desenvolvimento de software

ENGENHARIA DE SOFTWARE



Fonte: REDPIXEL.PL / Shutterstock

Vamos iniciar nosso estudo sobre o processo de desenvolvimento de software, no contexto da Engenharia de Software, que tem como principal produto o **software**. Aproveito para destacar que a bibliografia Pressman (2016) é uma referência mundial nesta área.

Imaginando um software embarcado em uma aeronave com centenas de pessoas ou controlando o tráfego aéreo de uma grande cidade, podemos destacar uma característica comum: a **complexidade**. A melhor tratativa para a complexidade é a aplicação de metodologia que permita a decomposição do problema em problemas menores de forma sistemática, cabendo à **Engenharia de Software** essa sistematização.

Entretanto, diferentemente de produtos de outras engenharias em produção, o software possui alta volatilidade, em função de constantes evoluções na tecnologia e nos requisitos, agregando a ele uma complexidade adicional.

A Engenharia de Software é uma tecnologia em camadas. Vejamos as descrições das referidas camadas na tabela a seguir:

Camada	Descrição
Camada qualidade	Garante que os requisitos atendam às expectativas dos usuários.
Camada de processo	Define as etapas de desenvolvimento do software.
Camada de métodos	Determina as técnicas e os artefatos de software.
Camada ferramentas	Estimula a utilização de ferramentas CASE.

 **Atenção!** Para visualização completa da tabela utilize a rolagem horizontal

Importante destacar que a base da Engenharia de Software é a camada de processo que trata das etapas de desenvolvimento.

PROCESSO DE DESENVOLVIMENTO DE SOFTWARE

VOCÊ SABE POR QUE SE APLICA DE FORMA INTENSA O CONCEITO DE ABSTRAÇÃO NO DESENVOLVIMENTO DE SOFTWARE?

Porque esse processo é iniciado com especificações e modelos com alto nível de abstração e, à medida que o desenvolvimento de software se aproxima da codificação, o nível de abstração diminui, de modo que o código representa o nível mais baixo da abstração ou de maior detalhamento na especificação do software.

Os diferentes modelos de processos de desenvolvimento de software possuem as seguintes atividades típicas:



Fonte: Kraphix / Shutterstock

LEVANTAMENTO DE REQUISITOS



Fonte: Kraphix / Shutterstock

ANÁLISE



Fonte: Kraphix / Shutterstock

PROJETO



Fonte: Kraphix / Shutterstock

IMPLEMENTAÇÃO



Fonte: Kraphix / Shutterstock

TESTES



Fonte: Kraphix / Shutterstock

IMPLEMENTAÇÃO

Vamos, agora, descrever cada uma das atividades comumente previstas em um processo de desenvolvimento de software.

ENGENHARIA DE REQUISITOS

As etapas de levantamento de requisitos e análise, no processo de desenvolvimento de software, compõem a Engenharia de Requisitos, de modo que essa engenharia está no contexto da Engenharia de Software.

Neste momento, precisamos apresentar a conceituação de requisito:

“OS REQUISITOS DE UM SISTEMA SÃO DESCRIÇÕES DOS SERVIÇOS FORNECIDOS PELO SISTEMA E AS

SUAS RESTRIÇÕES OPERACIONAIS. ESSES REQUISITOS REFLETEM AS NECESSIDADES DOS CLIENTES DE UM SISTEMA QUE AJUDA A RESOLVER ALGUM PROBLEMA”.

SOMMERVILLE, 2007

A Engenharia de Requisitos inclui as atividades de descobrir, analisar, documentar e verificar os serviços fornecidos pelo sistema e suas restrições operacionais, possuindo um processo próprio, tal qual ilustrado na Figura 1. Destacamos a existência de outras propostas de processos.

CONCEPÇÃO



LEVANTAMENTO



ELABORAÇÃO



NEGOCIAÇÃO



ESPECIFICAÇÃO



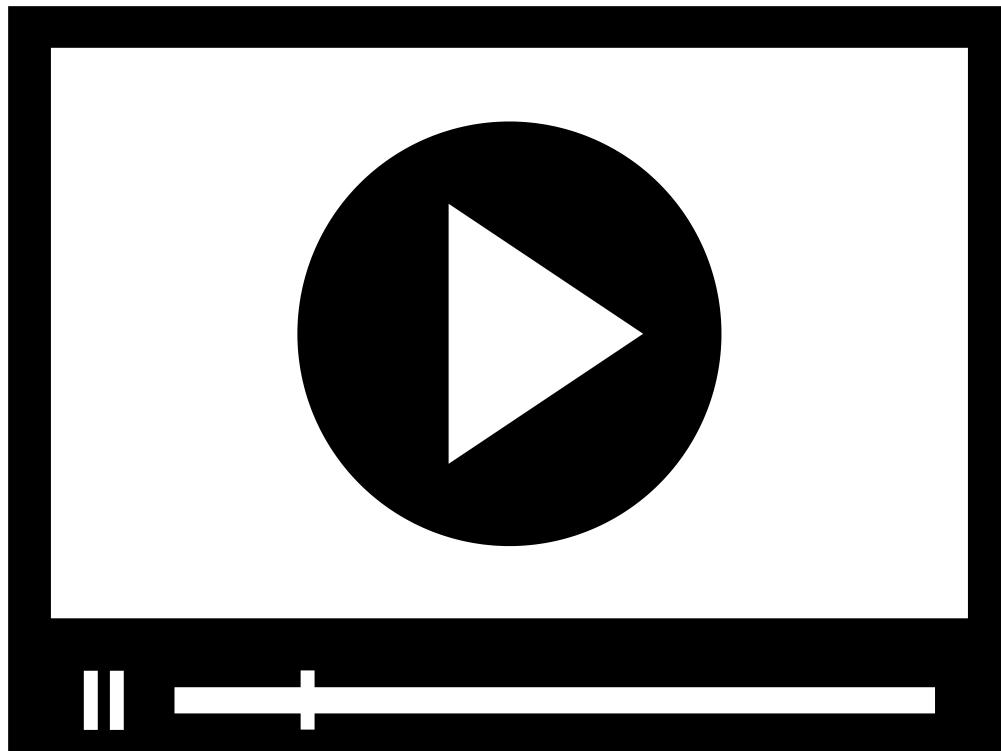
VALIDAÇÃO



GESTÃO

Figura 1 – Tarefas da Engenharia de Requisitos.

Você poderia perguntar: “Outro processo?”. Sim! Lembre-se de que a engenharia tem como base a camada de processos. Vamos entender cada etapa desse processo ilustrado na Figura 1



Neste vídeo, você conhecerá um pouco sobre as atividades da Engenharia de Requisitos.

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.



CONCEPÇÃO

Essa etapa exige do engenheiro de software o estabelecimento do entendimento inicial do problema, a identificação das partes interessadas que serão atendidas pelo software, a natureza da solução desejada e a eficácia da comunicação e colaboração preliminares entre clientes e usuários com a equipe de projeto.

Cabe destacar que um software costuma ter vários tipos de usuários, como, por exemplo, as partes interessadas em diferentes níveis gerenciais de uma empresa.

LEVANTAMENTO

Atividade que permite definir o escopo do projeto, ou “tamanho do problema”, além de possibilitar que usuários e desenvolvedores tenham a mesma visão do problema a ser resolvido. Isso é um desafio! Concorda?

Nesta etapa, é gerada uma especificação de requisitos que serve como um contrato entre clientes e equipe de projeto, esclarecendo aos clientes o que será entregue como produto do trabalho da equipe de desenvolvimento.

Ainda sobre a etapa de levantamento, esses clientes devem ser capazes de compreender os requisitos e fornecer *feedback* sobre eventuais falhas na especificação, para que estas sejam corrigidas de imediato, antes que o trabalho errado se propague pelo projeto.

A referida especificação comumente tipifica os requisitos em três categorias: requisitos funcionais, não funcionais e de domínio.

Os requisitos funcionais estão relacionados aos serviços fornecidos pelo sistema, ou seja, as funcionalidades que estarão disponíveis no software, tal como a geração de um histórico escolar em um sistema de gestão acadêmica.

Os requisitos não funcionais incluem as restrições operacionais impostas ao software, tais como o sistema gerenciador de banco de dados, a linguagem de programação, legislação

pertinente a *compliance*, entre outros, bem como os requisitos de qualidade, e.g., confiabilidade, manutenibilidade, usabilidade etc.

Os requisitos de domínio também são conhecidos como “regras de negócio”, que normalmente apresentam-se como restrições ao requisito funcional. Como exemplo, temos o cálculo da média para aprovação em determinada disciplina, a contagem de pontuação de multas para cômputo da perda de uma carteira de motorista ou o cálculo dos impostos quando da geração de uma nota fiscal. O não cumprimento de um requisito de domínio pode comprometer o uso do sistema.

AGORA, VEJAMOS AS TÉCNICAS MAIS UTILIZADAS PARA LEVANTAR REQUISITOS.

A observação, ou etnografia, permite ao engenheiro de software imergir no ambiente de trabalho onde a solução será usada, observando o trabalho rotineiro e tomando notas das tarefas em execução nas quais as partes interessadas estão envolvidas.

A entrevista é uma forma de diálogo, formal ou informal, onde o entrevistador busca respostas para um conjunto de questões previamente definidas e os entrevistados se apresentam como fontes de informação.

A pesquisa consiste na aplicação de um questionário às partes interessadas e posterior análise das respostas. Essa técnica permite a rápida obtenção de informações quantitativas e qualitativas de um público-alvo numeroso, particularmente quando não estão em um único local físico.

O JAD, *Joint Application Design*, é um método de projeto interativo que substitui as entrevistas individuais por reuniões de grupo, onde participam representantes dos usuários e dos desenvolvedores.

A técnica *brainstorming* inclui reuniões na qual participam todos os envolvidos na idealização do produto, como os engenheiros de software, clientes e usuários finais. Todos os envolvidos devem expor suas ideias e necessidades em relação ao produto.

VOCÊ PODERIA IMAGINAR UMA ENTREGA NESTA ETAPA?

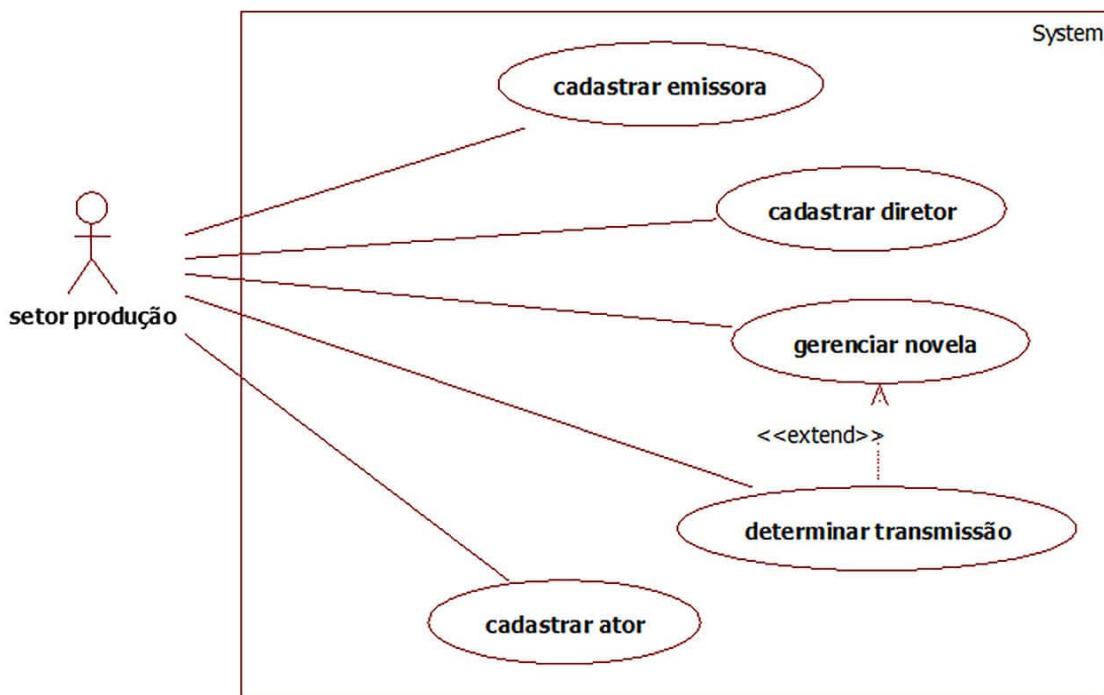
Temos uma entrega denominada “documento de requisitos”, cujo objetivo é documentar de forma fiel e completa todas as necessidades dos clientes e obter um aceite sobre o que se está propondo entregar em termos de produto.

A partir desse documento, inicia-se a rastreabilidade dos requisitos, garantindo que as especificações geradas até a codificação estejam de acordo com a documentação de requisitos.

ELABORAÇÃO

Nesta etapa, os engenheiros de software realizam um estudo detalhado dos requisitos levantados e, a partir desse estudo, são construídos modelos para representar o sistema a ser construído.

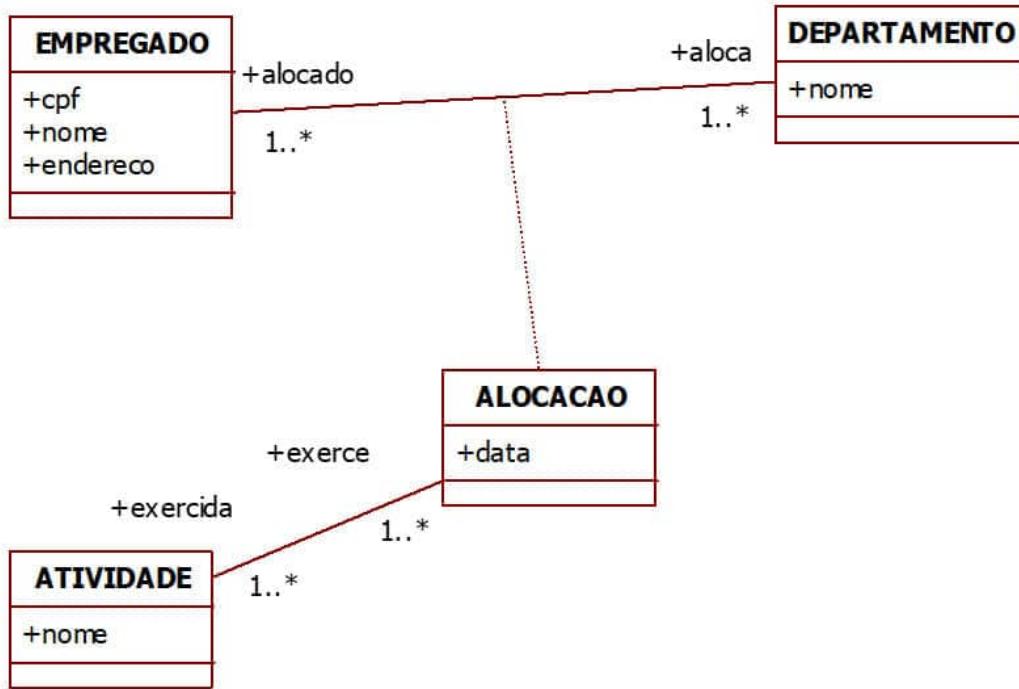
Na etapa de elaboração, a modelagem é guiada pela criação e pelo refinamento de cenários, identificados a partir dos requisitos funcionais, que descrevem como os usuários interagem com o sistema. A modelagem de casos de uso da UML (*Unified Modeling Language*) representa os referidos cenários, incluindo diagramas de casos de uso, artefatos gráficos, e descrições de casos de uso, artefatos textuais. A Figura 2 ilustra um diagrama de caso de uso.



Fonte: Autor

Figura 2 – Exemplo de diagrama de casos de uso.

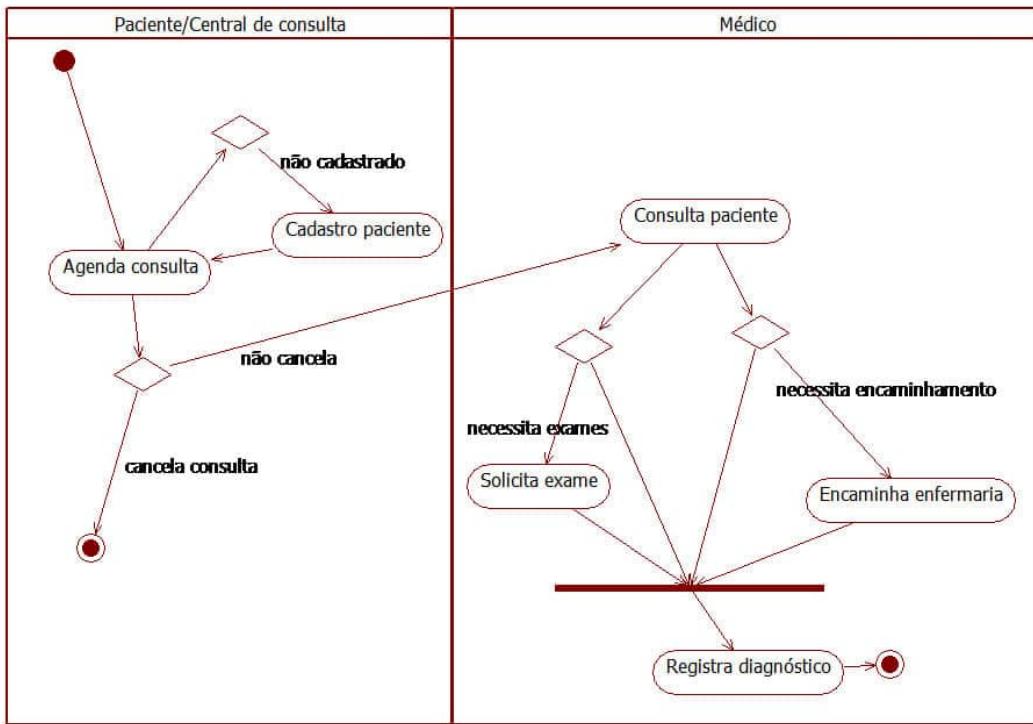
A partir dos casos de uso, podemos identificar as classes de análise que representam os objetos do negócio ou do domínio do problema. A Figura 3 ilustra um diagrama de classes da UML.



Fonte: Autor

Figura 3 – Exemplo de diagrama de classes.

Na modelagem de análise, podemos considerar a construção do modelo de atividades e do modelo de estados. O modelo de atividades, ilustrado na Figura 4, possui diagramas que podem ser utilizados no mapeamento de processos de negócios, na descrição gráfica de um caso de uso ou na definição do algoritmo de um método. O modelo de estados permite representar mudanças de estados significativas e respectivos eventos que causam a referida mudança.



Fonte: Autor

▣ Figura 4 – Exemplo de diagrama de atividades.



NEGOCIAÇÃO

Nesta etapa, ocorre a priorização e a resolução de conflitos entre os requisitos definidos nas etapas anteriores. Todos os envolvidos, equipe de projeto e usuários, participam da avaliação de custos, riscos e conflitos, a fim de eliminar, combinar e/ou modificar os referidos requisitos e, logicamente, priorizá-los.

ESPECIFICAÇÃO

O engenheiro de software deverá gerar um documento de especificação incluindo todos os requisitos e modelos gerados nas etapas anteriores.



VALIDAÇÃO

A validação permite ao engenheiro de software evidenciar que os modelos refletem as necessidades dos usuários com relação ao sistema a ser desenvolvido. Um defeito não considerado gera a construção de um sistema que não corresponderá às expectativas do usuário.

GESTÃO

Finalizando o processo da engenharia de requisitos, a etapa de gestão permite controlar as mudanças dos requisitos à medida que o projeto evoluí.



Podemos destacar que o engenheiro de software deverá considerar que o gerenciamento do escopo do projeto inclui um documento denominado matriz de rastreabilidade dos requisitos, sendo essa uma tabela que liga os requisitos dos produtos desde as suas origens até as entregas que lhes satisfazem. A manutenção desse documento permite monitorar a estabilidade dos requisitos.

RESUMINDO

Neste módulo, podemos destacar a relevância da Engenharia de Requisitos no contexto do processo de desenvolvimento de software. Importante lembrar que a Engenharia de Requisitos inclui as etapas de levantamento de requisitos e análise do referido processo.

Os requisitos são comumente categorizados em requisitos funcionais, não funcionais e de domínio. Os requisitos funcionais estão associados aos serviços ou às funcionalidades disponibilizadas pelo sistema. Os requisitos não funcionais estão relacionados com as restrições operacionais, tais como linguagem de programação, padrão de arquitetura etc., além de requisitos de qualidade, e.g., manutenibilidade, usabilidade e outros. Os requisitos de domínio detalham as regras de negócio identificadas no domínio do problema.

A Engenharia de Requisitos inclui um processo com tarefas que, de forma simplificada, permitem a identificação dos requisitos, a geração de modelos de análise, a validação dos requisitos por parte dos usuários e a gestão dos requisitos, possibilitando a rastreabilidade durante as etapas seguintes do projeto de software.

Agora, é com você! Vamos verificar se atingimos o objetivo deste módulo? Você está convidado a realizar as atividades a seguir.

VERIFICANDO O APRENDIZADO

1. (UFG - 2010 - UFG - ANALISTA DE TI - DESENVOLVIMENTO DE SISTEMAS) REQUISITOS NÃO FUNCIONAIS SÃO RESTRIÇÕES AOS SERVIÇOS DE UM SISTEMA DE SOFTWARE E AO PROCESSO DE DESENVOLVIMENTO DO SISTEMA. A EQUIPE DE DESENVOLVIMENTO DE UM SISTEMA DE CONTROLE DE TRÁFEGO AÉREO DEVE CONSIDERAR OS REQUISITOS NÃO FUNCIONAIS DE:

- A)** Cadastro e monitoramento de aeronaves.
- B)** Alta disponibilidade e baixo tempo de resposta de usuário por evento.
- C)** Uso conjunto de método ágil de sistemas e linguagem de programação orientada a objetos.

D) Alto desempenho e baixo tempo médio entre falhas.

2. (FCC - 2019 - SEMEF MANAUS - AM - TÉCNICO DE TECNOLOGIA DA INFORMAÇÃO DA FAZENDA MUNICIPAL) CONSIDERANDO A ANÁLISE DE REQUISITOS, AS INFORMAÇÕES DE RASTREABILIDADE DESEMPENHAM UM PAPEL DE GRANDE IMPORTÂNCIA. ASSIM, A EQUIPE RESPONSÁVEL DA FAZENDA MUNICIPAL DEVE ESTAR CIENTE DE QUE A RASTREABILIDADE DE PROJETO SIGNIFICA:

- A)** Listar os compiladores utilizados no desenvolvimento de cada módulo de software.
- B)** Determinar o mapeamento entre os requisitos de projeto e os locais onde o sistema será utilizado.
- C)** Determinar o desempenho de cada um dos requisitos do sistema.
- D)** Possuir o mapeamento entre os requisitos e os módulos de projeto que implementam os requisitos.

GABARITO

1. (UFG - 2010 - UFG - Analista de TI - Desenvolvimento de Sistemas) Requisitos não funcionais são restrições aos serviços de um sistema de software e ao processo de desenvolvimento do sistema. A equipe de desenvolvimento de um sistema de controle de tráfego aéreo deve considerar os requisitos não funcionais de:

A alternativa "**B**" está correta.

Quando ocorre uma falha no software, o requisito não funcional de disponibilidade contabiliza o tempo disponível para uso e o tempo necessário para a solução de um problema, de modo que alta disponibilidade significa que o sistema está em condições de uso a maior parte do tempo. O requisito não funcional tempo de resposta especifica o tempo que o sistema responderá à determinada solicitação de serviço. Ambos os requisitos são fundamentais para um sistema de controle de tráfego aéreo.

2. (FCC - 2019 - SEMEF Manaus - AM - Técnico de Tecnologia da Informação da Fazenda Municipal) Considerando a análise de requisitos, as informações de rastreabilidade

desempenham um papel de grande importância. Assim, a equipe responsável da Fazenda Municipal deve estar ciente de que a rastreabilidade de projeto significa:

A alternativa "D" está correta.

A rastreabilidade, iniciada com o levantamento de requisitos, permite gerenciar volatilidade dos requisitos durante o processo de desenvolvimento de software.

MÓDULO 2

◎ Reconhecer as atividades do projeto de Software do processo de desenvolvimento de software

PROJETO DE SOFTWARE

Podemos observar que os principais modelos da etapa de análise do processo de desenvolvimento de software são: o modelo de casos de uso, o modelo de classes de análise, o modelo de atividades e o modelo de estados.

ATENÇÃO

Muitos autores e profissionais de Engenharia de Software preferem usar o termo original do inglês *design* no lugar de projeto, para não confundir com o homógrafo projeto, no sentido de *project*, cujo significado é diferente de *design*. Neste texto, o termo **projeto** é usado indistintamente no sentido de *project* ou de *design*, dependendo do contexto.

Você acha possível implementar um software com os referidos modelos? Provavelmente, faltam mais detalhes. Vamos, agora, tratar desses detalhes.

O conceito de abstração é fundamental no processo de desenvolvimento de software, pois ele é iniciado com especificações e modelos com alto nível de abstração.

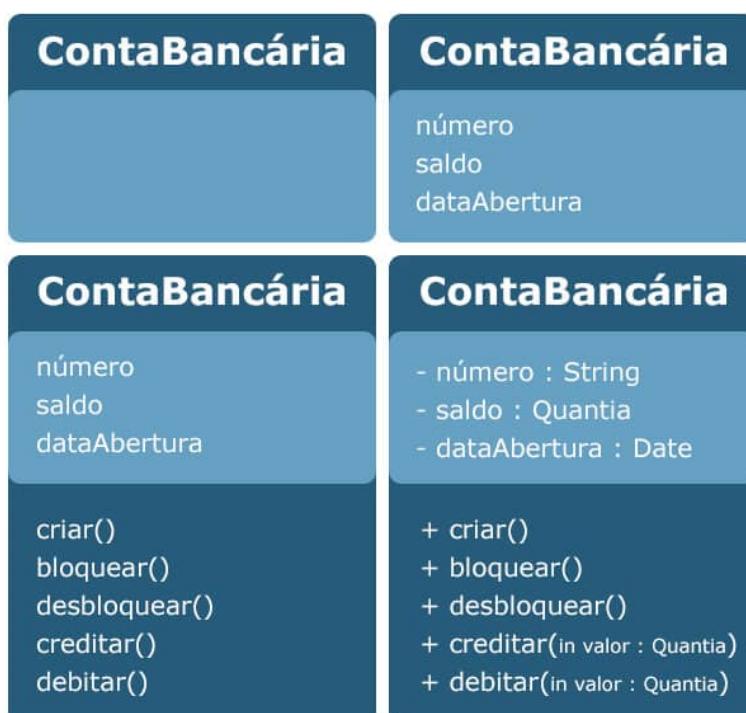
A etapa projeto de software diminui o nível de abstração dos diagramas de análise por meio de refinamentos sucessivos, como também exige a criação de novos modelos.

As principais atividades realizadas na fase de projeto serão descritas a seguir.

REFINAMENTO DOS ASPECTOS ESTÁTICOS E ESTRUTURAIS DO SISTEMA

O modelo de classes representa os aspectos estáticos e estruturais do sistema. A partir do modelo de classes gerado na análise, vamos inserir novos elementos que permitirão a implementação das classes, ou seja, aplicaremos refinamentos que possibilitam reduzir o grau de abstração. A Figura 5 exemplifica as abstrações possíveis de uma classe durante o processo de desenvolvimento de software. Ao final, temos uma especificação de classe em condições de ser codificada.

★ EXEMPLO



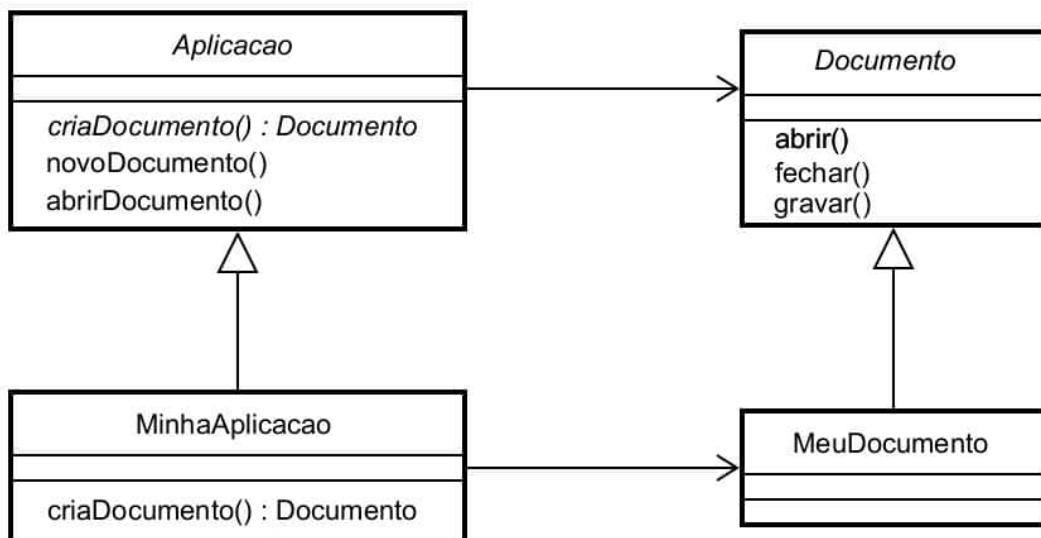
Fonte: Adapatado de Bezerra (2014)

Figura 5 - Exemplo de refinamentos em uma classe.

Vamos analisar os refinamentos: a primeira estrutura indica uma classe e o respectivo nome; a segunda inclui os atributos; a terceira adiciona os métodos ou funções das classes, e a quarta detalha os atributos e métodos. Na análise, não é importante definirmos os tipos de dados, sendo fundamental no projeto, pois a especificação de projeto tem de garantir a implementação, ou seja, a codificação da classe.

Além desse refinamento, novos elementos são inseridos em função das associações entre as classes, das heranças existentes, ou mesmo em virtude da criação de novas classes.

Um aspecto importante é a possibilidade de aplicação de padrões de projeto, ou design *patterns*, no modelo de classes de projeto. Esses padrões representam *templates* de melhores práticas para a solução de determinados problemas comuns, que serão resolvidos mais facilmente quando implementados pelo programador. A Figura 6 exemplifica o padrão *Factory Method*.



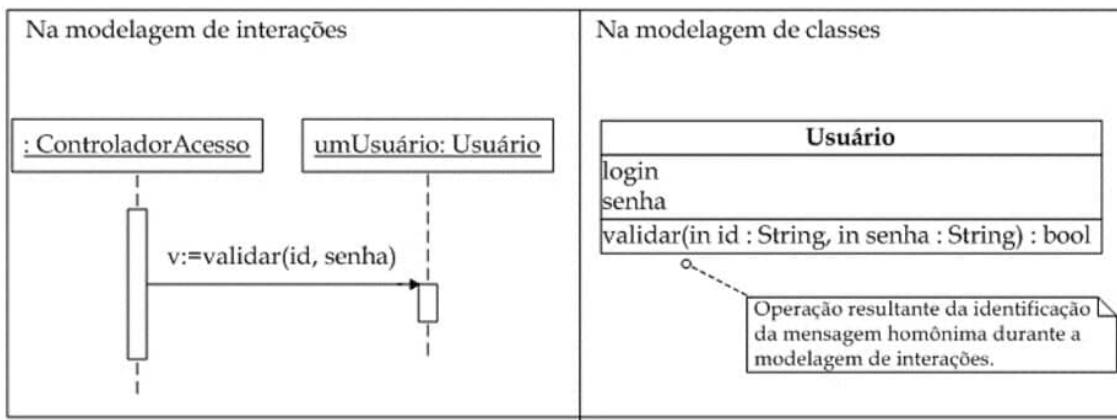
Fonte: Luís Felipe Braga / Wikipédia

Figura 6 - Exemplo do padrão *Factory Method*.

DETALHAMENTO DOS ASPECTOS DINÂMICOS DO SISTEMA

O que seriam aspectos dinâmicos? Lembre-se: o paradigma dominante na indústria de software é a orientação a objetos, que, de forma simplificada, é a representação do mundo real por meio de objetos que se comunicam por meio de mensagens.

A referida comunicação representa a dinâmica existente entre os objetos e, para a sua representação, temos o modelo de interação, que inclui dois diagramas: diagrama de comunicação e diagrama de sequência. Ambos os diagramas têm a mesma abstração dinâmica com representações distintas e recíprocas.



Fonte: Bezerra (2014)

Figura 7 - Diagrama sequência *versus* classes de projeto.

QUAL A RELAÇÃO EXISTENTE ENTRE O MODELO DE CLASSES E O MODELO DE INTERAÇÃO?

O modelo de interação é desenvolvido usualmente em paralelo com o modelo de classes de projetos, de modo que os métodos implementados nas classes são identificados a partir das mensagens definidas no modelo de interação. A Figura 7 exemplifica, à esquerda, parte de um diagrama de sequência, onde a mensagem “validar” estabelece uma comunicação entre os objetos “controlador acesso” e “usuário”, sendo a mensagem implementada no objeto receptor “usuário”, ou seja, a classe “usuário” inclui o método “validar” com os respectivos parâmetros.

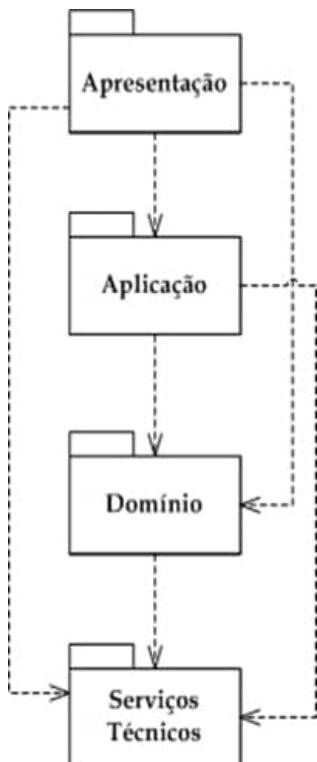
DETALHAMENTO DA ARQUITETURA DO SISTEMA

Uma atividade que aplicamos de forma intensa na Engenharia de Software é a fatoração, ou seja, a decomposição da solução do problema em partes menores, facilitando principalmente o trato da complexidade.

Imaginemos um sistema de software complexo; intuitivamente, temos de realizar a decomposição em subsistemas. O processo do projeto que permite identificar os referidos subsistemas, estabelecendo um *framework* para controle e comunicação entre eles, é denominado projeto de arquitetura.

A arquitetura é definida por meio de duas abstrações: lógica e física.

A Figura 8 ilustra uma divisão lógica comumente aplicada no projeto de arquitetura, sendo composta das seguintes camadas: apresentação, aplicação, domínio e serviços técnicos.



Fonte: Bezerra (2014)

Figura 8 - Camadas de software.

Importante destacar que as camadas mais altas devem depender das camadas mais baixas – essa fatoração permite melhor trato da complexidade, ou seja, decomposição do sistema em partes menores. Outra vantagem é o reúso, pois as camadas inferiores são independentes das camadas superiores. A arquitetura em camadas permite baixo acoplamento, ou seja, mínimo de dependências entre os subsistemas, e alta coesão, isto é, deve haver mais associações entre as classes internamente do que externamente.

A camada de apresentação da Figura 8 inclui as denominadas classes de fronteira, que permitem a instanciação dos objetos de fronteira que interagem com os usuários.

A camada de aplicação inclui as classes de controle, que permitem a instanciação dos objetos de controle que atuam como intermediários entre os objetos da camada de apresentação e os objetos do domínio do problema.

A camada de domínio é composta pelas classes de domínio oriundas do diagrama de classes, que permitem a instanciação dos objetos do domínio do problema – esses objetos podem conter métodos que alteram o próprio estado.

Finalmente, a camada de serviços técnicos pode incluir serviços genéricos, cabendo destaque à camada de persistência de dados.

Assim como existem padrões de projeto aplicados ao projeto de classes (Figura 6), temos também padrões aplicados à arquitetura. Como exemplo, podemos destacar o padrão de arquitetura em camadas denominado MVC (*Model-View-Controller*), formulado na década de 1970 e focado no reúso de código e na separação de conceitos em três camadas interconectadas, tal como ilustra a Figura 9.

O controlador (*controller*) tem funcionalidade para atualizar o estado do modelo, como, por exemplo, uma nota fiscal, ou alterar a apresentação da visão do modelo.

Um modelo (*model*) mantém os dados, notificando visões e controladores quando da ocorrência de mudança em seu estado.

A visão (*view*) permite a exibição dos dados.

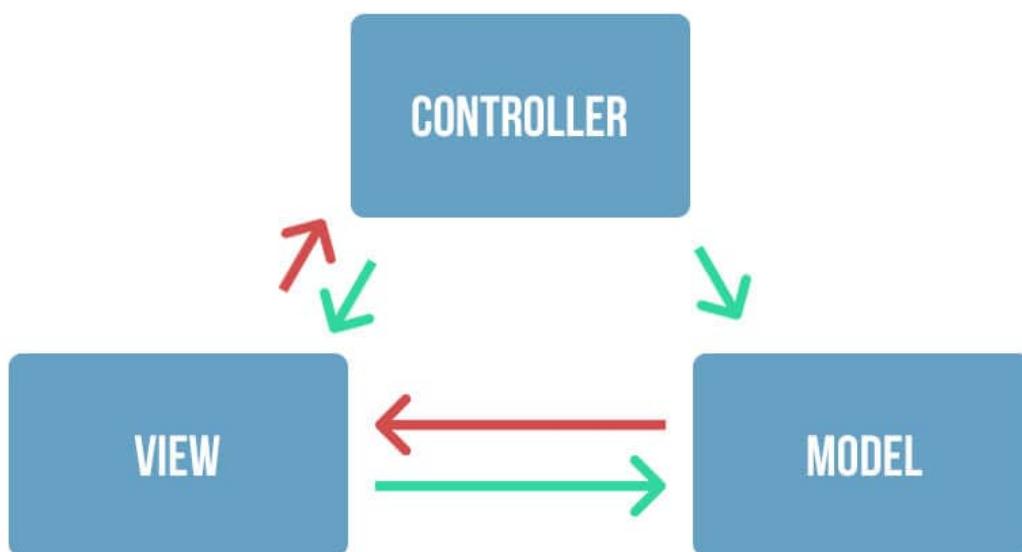


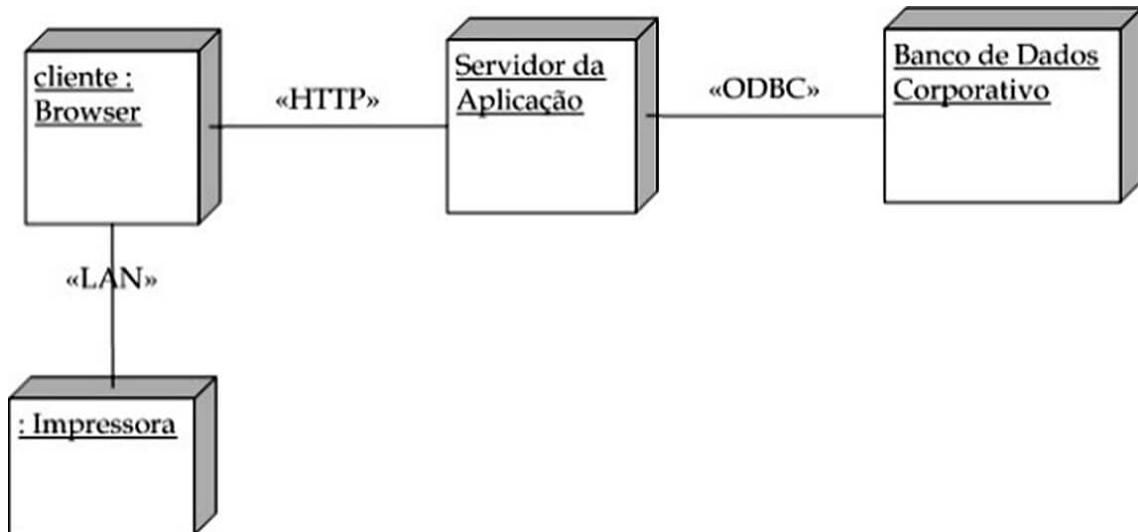
Figura 9 - Arquitetura MVC.

Após a definição lógica da arquitetura, o engenheiro de software precisará definir a arquitetura física por meio do modelo de implantação, que projeta a infraestrutura de hardware necessária

para o software projetado.

VOCÊ PODERIA QUESTIONAR: “O ENGENHEIRO DE SOFTWARE ESPECIFICA HARDWARE?”.

Nesse caso, não, mas o projeto tem de definir os nós computacionais necessários para a implantação do software, devendo a especificação do hardware ficar por conta da equipe de infraestrutura. A Figura 10 exemplifica um diagrama de implantação com quatro nós computacionais e as respectivas conexões.



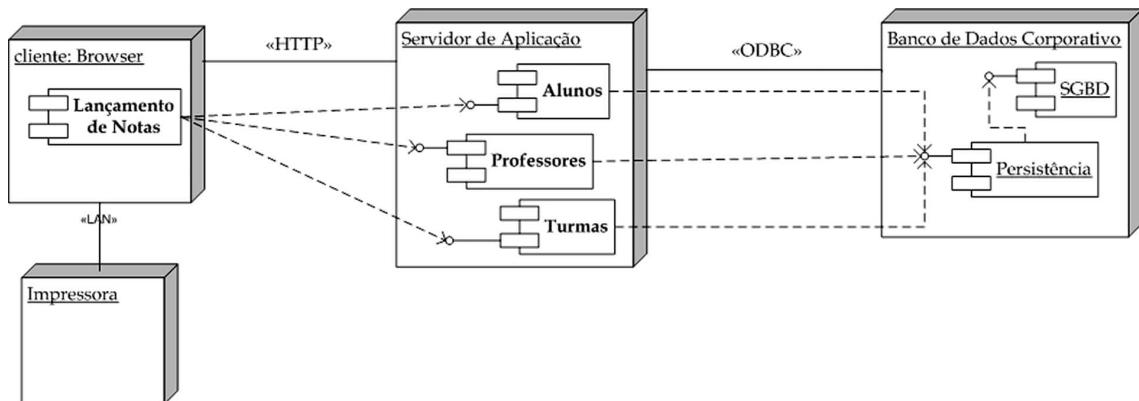
Fonte: Bezerra (2014)

Figura 10 - Diagrama de implantação.

VOCÊ LEMBRA QUAL A COMPOSIÇÃO DAS CAMADAS APRESENTADAS?

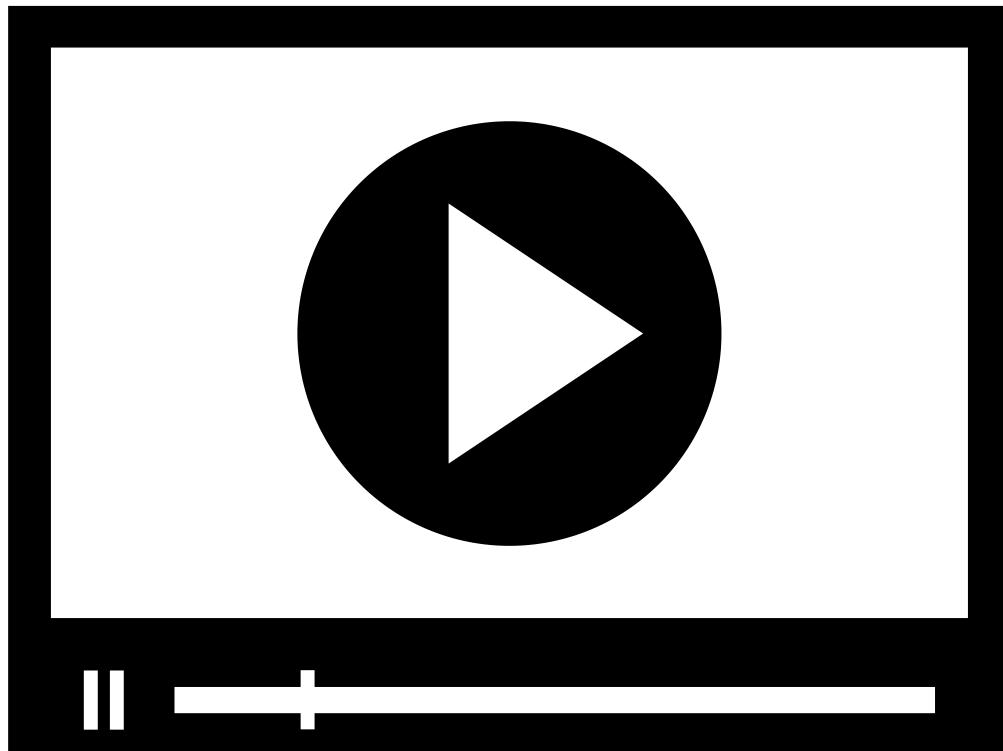
Cada camada é composta por um conjunto de classes, formando componentes diversos. Os componentes podem ser compostos por uma ou mais classes que encapsulam funcionalidades de forma independente – um conjunto de componentes pode compor um sistema complexo. A forma de representar essa “componentização” do software é definida no modelo de implementação. A Figura 11 ilustra o diagrama de componentes embutido no diagrama de

implantação da Figura 10, de modo que podemos verificar a alocação por nó computacional dos referidos componentes.



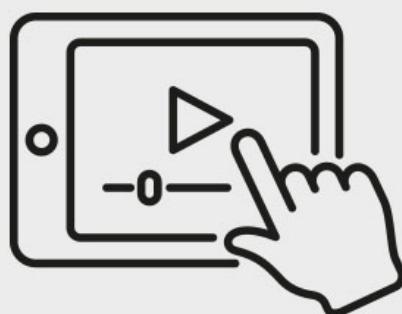
Fonte: Bezerra (2014)

Figura 11 - Diagrama de componentes embutido.



Neste vídeo, você conhecerá um pouco sobre a arquitetura de sistemas de software.

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.



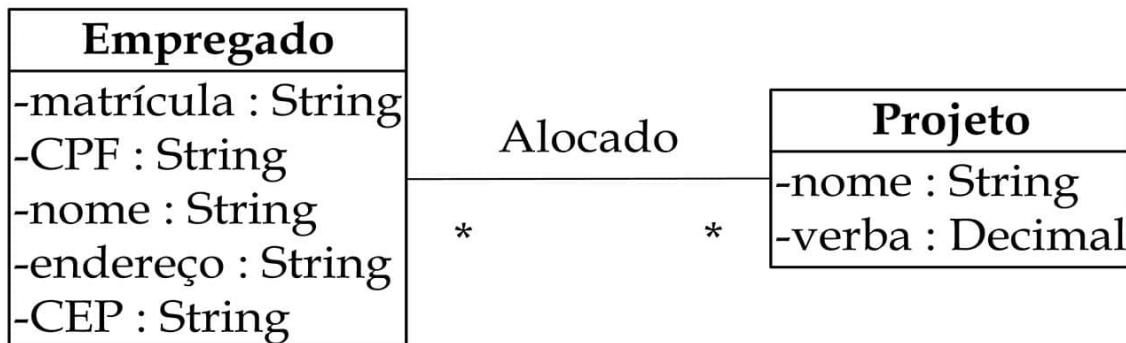
MAPEAMENTO OBJETO-RELACIONAL

O padrão da indústria de software atualmente é o paradigma orientado a objetos, que podem assumir estruturas de dados com elevado grau de complexidade, sendo que a maioria deles, em algum momento, necessita de persistência.

Por outro lado, o padrão das tecnologias de banco de dados ainda é o modelo relacional, ou seja, temos uma estrutura de armazenamento bidimensional denominada tabela.

O QUE FAZER PARA INTEGRAR OS DOIS PADRÕES?

Vamos utilizar o diagrama de classes como modelo conceitual do banco de dados e, a partir de então, gerar o modelo lógico de banco de dados, criando as tabelas necessárias às persistências exigidas pelos objetos. Esse processo é denominado mapeamento objeto-relacional. A Figura 12 ilustra um exemplo de associação muitos-para-muitos em um diagrama de classes, cujo mapeamento gera as três tabelas descritas na Figura 13.



Fonte: Bezerra (2014)

Figura 12 - Exemplo de associação muitos-para-muitos.

Empregado(id,matrícula,CPF,nome,endereço,CEP,idDepartamento)

Alocação(id, idProjeto, idEmpregado)

Projeto(id, nome, verba)

Figura 13 - Resultado do mapeamento muitos-para-muitos.

REALIZAÇÃO DO PROJETO DA INTERFACE GRÁFICA COM O USUÁRIO



Fonte: Rawpixel.com / Shutterstock

O diagrama de casos de uso define as funcionalidades do sistema, sendo que cada caso de uso representa uma interação completa entre o sistema e um agente externo denominado ator. Nessa interação, precisamos definir uma interface como meio de comunicação homem-máquina. Nesse ponto, o requisito-chave não funcional é usabilidade, portanto é fundamental a participação do usuário no processo de definição das interfaces, ou seja, “deixe o usuário no comando”.

Vamos apresentar possíveis questões relativas à interface que podem surgir: tempo de resposta – em um sistema de venda online com alto tempo de resposta, talvez ocorra uma desistência e uma ida para o concorrente; recursos de ajuda, disponíveis sem que haja necessidade de abandonar a interface; tratamento de erros, por meio de uma linguagem inteligível para o usuário; acessibilidade da aplicação, com foco no atendimento aos usuários com necessidades especiais.

Neste módulo, pudemos avaliar a importância da etapa de projeto no processo de desenvolvimento de software.

Cabe ao engenheiro de software implementar as seguintes atividades que genericamente compõem a etapa de projeto:



Fonte: Kraphix / Shutterstock

REFINAMENTO DOS ASPECTOS ESTÁTICOS E ESTRUTURAIS DO SISTEMA.



Fonte: Kraphix / Shutterstock

DETALHAMENTO DOS ASPECTOS DINÂMICOS DO SISTEMA.



Fonte: Kraphix / Shutterstock

DETALHAMENTO DA ARQUITETURA DO SISTEMA.



Fonte: Kraphix / Shutterstock

MAPEAMENTO OBJETO-RELACIONAL.



Fonte: Kraphix / Shutterstock

REALIZAÇÃO DO PROJETO DA INTERFACE GRÁFICA COM O USUÁRIO.

Ao final desta etapa, podemos iniciar a implementação do software de acordo com as especificações contidas nos modelos de projeto gerados.

VERIFICANDO O APRENDIZADO

1. SOBRE AS CAMADAS DO MODELO DE ARQUITETURA MVC (*MODEL - VIEW-CONTROLLER*) USADO NO DESENVOLVIMENTO WEB, É CORRETO AFIRMAR:

- A)** Todos os dados e a lógica do negócio para processá-los devem ser representados na camada *controller*.

- B)** A camada *model* pode interagir com a camada *view* para converter as ações do cliente em ações que são compreendidas e executadas na camada *controller*.
- C)** A camada *view* é a camada responsável por exibir os dados ao usuário. Em todos os casos, essa camada somente pode acessar a camada *model* por meio da camada *controller*.
- D)** A camada *controller*, geralmente, possui um componente controlador padrão, criado para atender a todas as requisições do cliente.

2. O ENGENHEIRO DE SOFTWARE ESTÁ ENCERRANDO A ETAPA DE ANÁLISE E INICIANDO A ETAPA DE PROJETO. ASSINALE A AFIRMATIVA QUE NÃO É UMA ATIVIDADE DE PROJETO:

- A)** Aumentar o grau de abstração do modelo de classes.
- B)** Identificar os métodos das classes a partir de modelos dinâmicos.
- C)** Definir o modelo lógico de banco de dados.
- D)** Utilizar padrões de projeto no diagrama de classes.

GABARITO

1. Sobre as camadas do modelo de arquitetura MVC (*Model - View-Controller*) usado no desenvolvimento web, é correto afirmar:

A alternativa "D" está correta.

O padrão de arquitetura de projeto MVC torna fácil a manutenção do software, permitindo a implantação modular de forma rápida, cabendo à camada *view* gerar um evento a partir de uma requisição do cliente. O referido evento é atendido por um *controller*.

2. O engenheiro de software está encerrando a etapa de análise e iniciando a etapa de projeto. Assinale a afirmativa que NÃO é uma atividade de projeto:

A alternativa "A" está correta.

A etapa de projeto permite o refinamento de modelos de análise, tal como o modelo de classes, de forma a diminuir o nível de abstração. Exemplo: o modelo de classes é refinado aumentando o nível de detalhamento.

MÓDULO 3

◎ Reconhecer as etapas de implementação e testes do processo de desenvolvimento de software

IMPLEMENTAÇÃO

Quais são as entregas da etapa “projeto” do processo de desenvolvimento de software?

Modelos, ou seja, diagramas e especificações textuais que permitem a implementação, ou codificação do software. Realizando um paralelo com a Engenharia Civil, temos plantas baixas, projeto estrutural, hidráulico-sanitário, elétrico etc.; agora, podemos ir para o terreno construir a casa.

A etapa “implementação” do processo de desenvolvimento de software permite realizar a tradução dos modelos de projeto em código executável por meio do uso de uma ou mais linguagens de programação.

Temos aqui um desafio! Ainda não conhecemos todos os detalhes dos algoritmos que efetivamente resolvem o problema; para solucioná-lo, entram em cena os programadores.

Podemos destacar o impacto de muitos requisitos não funcionais nesta etapa, tais como linguagens de programação, frameworks de persistência, e.g., framework para o mapeamento objeto-relacional escrito na linguagem Java denominado *hibernate*, sistema de gerenciamento de banco de dados, requisitos de qualidade, e.g., confiabilidade, usabilidade etc., reutilização de componentes, entre outros.

Nesse contexto, podemos descrever a denominada “Crise do software” sintetizada pela afirmativa de Edsger Dijkstra, em 1972:

A MAIOR CAUSA DA CRISE DO SOFTWARE É QUE AS MÁQUINAS TORNARAM-SE VÁRIAS ORDENS DE MAGNITUDE MAIS POTENTES! EM TERMOS DIRETOS, ENQUANTO NÃO HAVIA MÁQUINAS, PROGRAMAR NÃO ERA UM PROBLEMA; QUANDO TIVEMOS COMPUTADORES FRACOS, ISSO SE TORNOU UM PROBLEMA PEQUENO, E, AGORA QUE TEMOS COMPUTADORES GIGANTESCOS, PROGRAMAR TORNOU-SE UM PROBLEMA GIGANTESCO.

PRESSMAN, 2016.

Importante destacar que o desenvolvimento tecnológico do hardware nos últimos anos permitiu o desenvolvimento de softwares cada vez mais complexos, tendo forte impacto na indústria de software. Como exemplo, podemos apresentar a substituição do paradigma estruturado pelo paradigma orientado a objetos, baseado na programação orientada a objetos, que permite o reúso intensivo de especificação, bem como melhor manutenibilidade e, como consequência, o desenvolvimento de softwares mais complexos.

QUALIDADE DE SOFTWARE

Um dos fatores que exerce influência negativa sobre um projeto é a complexidade, estando associada a uma característica bastante simples: o tamanho das especificações. Em programas de computador, o problema de complexidade é ainda mais grave, em razão das interações entre os diversos componentes do sistema.

A discussão sobre problemas de software *versus* complexidade não estará completa sem abordar o assunto qualidade. De forma simplificada, a qualidade está em conformidade com os

requisitos, tendo como objetivo satisfazer o cliente por meio da aplicação da Engenharia de Software, nosso contexto.



Fonte: TierneyMJ / Shutterstock

A norma ISO 9126 identifica seis atributos fundamentais de qualidade para o software:

FUNCIONALIDADE

Satisfação dos requisitos funcionais.

CONFIABILIDADE

Tempo de disponibilidade do software.

USABILIDADE

Facilidade de uso.

EFICIÊNCIA

Otimização dos recursos do sistema.

FACILIDADE DE MANUTENÇÃO

Realização de correção no software.

PORTABILIDADE

Adequação a diferentes ambientes.

A qualidade possui duas dimensões fundamentais aplicáveis ao software: as dimensões da qualidade do processo e da qualidade do produto.

Considerando que estamos descrevendo a etapa de implementação do processo de software, vamos apresentar considerações relativas à qualidade do produto software, ou seja, aos testes que garantem a qualidade do produto software.

TESTE DE SOFTWARE

A qualidade do produto software pode ser garantida através de sistemáticas aplicações de testes nos vários estágios do desenvolvimento da aplicação. Esses testes permitem a validação da estrutura interna do software e sua aderência aos respectivos requisitos. A integração dos subsistemas também é avaliada, com destaque para as interfaces de comunicação existentes entre os referidos componentes de software.

Podemos, então, definir teste como um processo sistemático e planejado que tem por finalidade única a identificação de erros. No caso de softwares complexos, sabe-se que o teste será capaz de descobrir a presença de erros, mas não a sua ausência!

Softwares mal testados podem gerar prejuízos às empresas. Um defeito de projeto poderá encadear requisições de compras inadequadas, gerar resultados financeiros incorretos, entre outros. Até mesmo as tomadas de decisões gerenciais da empresa podem ser negativamente impactadas, pois muitos profissionais se apoiam nas informações com o objetivo de tornar a empresa mais eficiente. A qualidade das decisões está intimamente ligada à qualidade das informações disponibilizadas aos diversos níveis gerenciais.

Os procedimentos de testes aplicados diretamente em softwares são também conhecidos como testes de software ou testes dinâmicos, podendo, em sua maioria, sofrer alto nível de automação, o que possibilita a criação de complexos ambientes de testes que simulam diversos cenários de utilização.

ATENÇÃO

Lembre-se: quanto mais cenários simulados, maior o nível de validação que obtemos do produto, caracterizando maior nível de qualidade do software desenvolvido.

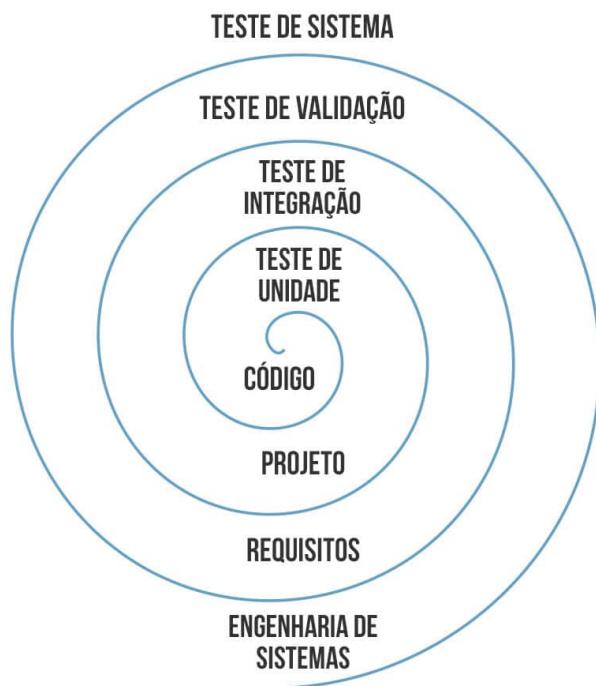
Os testes de software têm por objetivo avaliar a qualidade desse produto nas mais variadas dimensões possíveis, ou seja, em relação à sua estrutura interna, na aderência às regras de

negócios estabelecidas, nos parâmetros de performance esperados pelo produto etc.

Durante a codificação do software, podemos adotar a estratégia representada na Figura 14.

Essa espiral é percorrida a partir do interior, aumentando o nível de abstração a cada volta.

Vamos descrever como cada etapa dessa estratégia funciona.



Fonte: Adaptado de Pressman (2016)

Figura 14 – Estratégia de Teste.



Neste vídeo, você conhecerá um pouco sobre os tipos de teste de software.

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.



TESTE DE UNIDADE

A validação da unidade de software é a primeira etapa da estratégia, tendo por objetivo testar componentes individuais de uma aplicação.

Nesta estratégia de testes, o objetivo é executar o software a fim de exercitar adequadamente toda a estrutura interna de um componente, como os desvios condicionais, os laços de processamento etc.

TESTE DE INTEGRAÇÃO

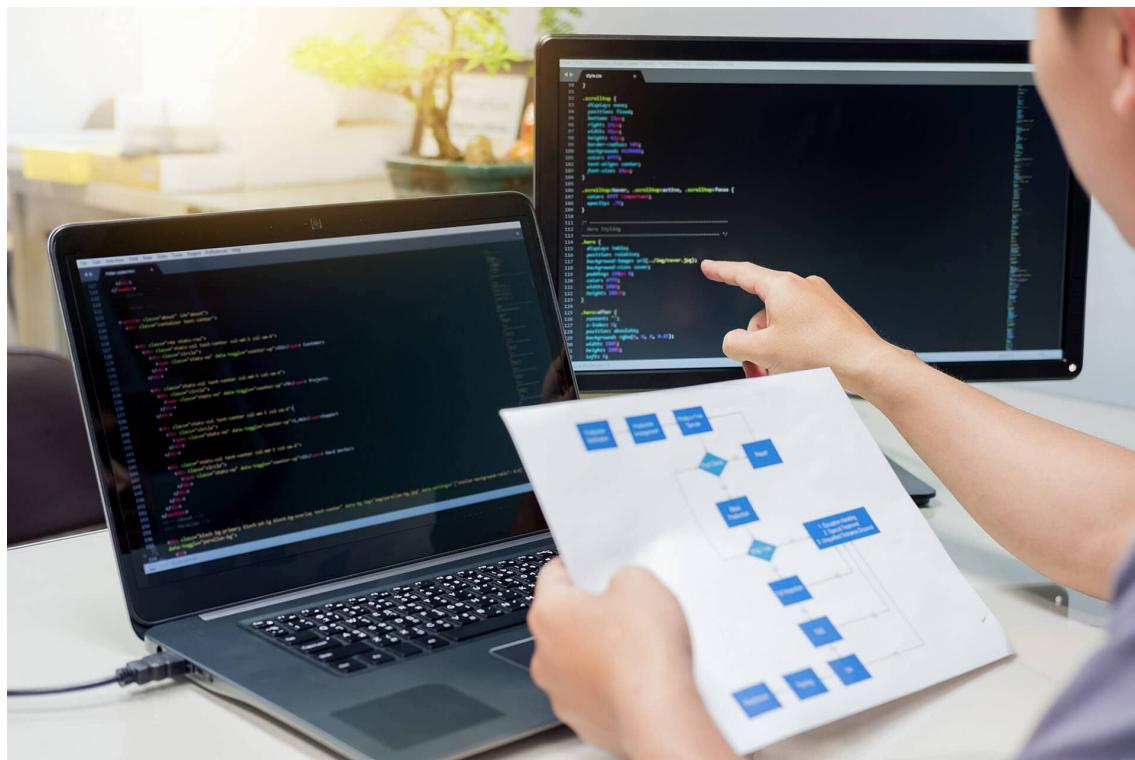
Objetiva a manutenção de compatibilidade das novas unidades construídas, ou modificadas, com os componentes previamente existentes.

A compatibilidade de um componente é quebrada sempre que existe alteração ou exclusão de uma rotina ou propriedade pública de um componente. Quando isso ocorre, todos os outros componentes que empregam essas rotinas ou propriedades estão automaticamente incompatíveis com o componente modificado, gerando erros durante a execução do software. São exatamente esses erros que devem ser identificados nesse estágio dos testes de software.

TESTE DE VALIDAÇÃO

Quando atingimos esse estágio dos testes, a maior parte das falhas de funcionalidade deve ter sido detectada nos testes unitários e nos testes de integração, sendo o objetivo desta etapa realizar a validação da solução como um todo. Os testes de validação contam com uma infraestrutura mais complexa de hardware, visando estar o mais próximo do ambiente real de produção.

Nesta etapa, são realizados os testes funcionais, que visam verificar se a versão corrente do sistema permite executar processos ou casos de uso completos do ponto de vista do usuário, de modo a obter os resultados esperados; e os testes não funcionais, que visam testar o software em relação às características não funcionais, como desempenho, segurança, confidencialidade, recuperação a falhas etc.



Fonte: por_suwat / Shutterstock

Devemos planejar os testes de software a partir dos cenários contidos nas descrições de casos de uso e identificados na etapa de análise do processo de software; normalmente, o engenheiro de software define os cenários com alto nível de abstração lógica, denominados também de casos de teste. Neste caso, o analista de teste da equipe de qualidade deverá diminuir o nível de abstração dos referidos cenários, detalhando os mesmos por meio dos casos de testes. Como exemplo, um caso de uso “Realizar saque de conta”, onde podemos imaginar um cenário em que o saldo supera o valor do saque e a transação ocorre sem problema, ou cenários alternativos ou de exceção, tais como bloqueio do saque por saldo insuficiente; liberação do saque utilizando o limite disponível pelo banco com cobrança de juros; liberação do saque com retirada de fundo de aplicação, entre outros.

Nesta etapa, ocorrem os procedimentos de aceite, que deverão ser realizados a cada ciclo de implementação da solução, permitindo correções antecipadas de determinados pontos que não foram adequadamente atendidos pelo software.

Como essa é a última oportunidade efetiva de detectar falhas no software, poderemos empregar o aceite como uma estratégia para reduzir riscos de uma implantação onde um erro vital não detectado pode comprometer a imagem da solução como um todo. Dessa forma, podemos dividir o aceite em três momentos distintos: o Teste Alpha, o Teste Beta e o Aceite Formal.



ACEITE TESTE ALPHA

Os usuários finais são convidados a operar o software dentro de um ambiente controlado, sendo realizado na instalação do desenvolvedor.

ACEITE TESTE BETA

Os usuários finais são convidados a operar o produto utilizando suas próprias instalações físicas, ou seja, o software é testado em um ambiente não controlado pelo desenvolvedor.



ACEITE FORMAL

Trata-se de uma variação do Teste Beta, cabendo aos próprios clientes e usuários determinarem o que deverá ser testado e validar se os requisitos foram adequadamente implementados.

A validação tem sucesso quando o software atende os requisitos, ou seja, funciona da forma esperada pelo usuário.

TESTE DE SISTEMA

A última etapa na espiral, apresentada na Figura 14, é o teste de sistema, que extrapola os limites da engenharia de software, ou seja, considera o contexto mais amplo da engenharia de sistemas de computadores. Após ser validado, o software deve ser combinado com outros elementos do sistema, tais como hardware, base de dados etc.

Vejamos alguns testes sugeridos nesta etapa.



TESTE DE RECUPERAÇÃO

Este teste tem por objetivo avaliar o comportamento do software após a ocorrência de um erro ou de determinadas condições anormais.

Os testes de recuperação devem também contemplar os procedimentos de recuperação do estado inicial da transação interrompida, impedindo que determinados processamentos sejam realizados pela metade e sejam futuramente interpretados como completos. Como exemplo, simular saque com defeito no caixa eletrônico ou simular saque com queda de energia.

TESTE DE SEGURANÇA

Este cujo objetivo é detectar as falhas de segurança que podem comprometer o sigilo e a fidelidade das informações, bem como provocar perdas de dados ou interrupções de processamento. Esses ataques à segurança do software podem ter origens internas ou externas.

A ideia é avaliar o nível de segurança que toda a infraestrutura oferece, simulando situações que provocam a quebra de protocolos de segurança. Como exemplo, avaliar se a senha do cartão está sendo requisitada antes e depois da transação ou avaliar se a senha adicional e randômica está sendo requisitada no início da operação.



TESTE POR ESFORÇO

Este com o objetivo de simular condições atípicas de utilização do software, provocando aumentos e reduções sucessivas de transações que superem os volumes máximos previstos para o software, gerando contínuas situações de pico e avaliando como o software e toda a infraestrutura estão se comportando. Como exemplo, simular 10.000 saques simultâneos.

TESTE DE DESEMPENHO

Este cujo objetivo é determinar se o desempenho, nas situações previstas de pico máximo de acesso e concorrência, está consistente com os requisitos definidos.

Devemos especificar os tempos de resposta considerados factíveis à realização de cada cenário. Como exemplo, garantir que manipulação com dispositivos físicos no saque não ultrapassem 10 segundos da operação.



TESTE DE DISPONIBILIZAÇÃO

Também chamado de teste de configuração, tem por objetivo executar o software sobre diversas configurações de softwares e hardwares. A ideia é garantir que a solução tecnológica seja executada adequadamente sobre os mais variados ambientes de produção previstos nas fases de levantamento dos requisitos.

O referido teste também examina todos os procedimentos de instalação e instaladores a serem utilizados pelos clientes, bem como a documentação que será usada pelos usuários finais. Como exemplo, simular saque com impressora dos fornecedores X, Y e Z.

RESUMINDO

Neste módulo, pudemos avaliar a importância das etapas de implementação e testes do processo de desenvolvimento de software.

À medida que o programador codifica a solução a partir dos modelos gerados na etapa de projeto, os testes são aplicados por meio de uma estratégia de testes em forma de espiral, incluindo quatro categorias de testes.

Inicialmente, o teste unitário permite validar a estrutura interna de cada componente de software.

O teste de integração possibilita validar a integração de novos componentes, ou componentes que passaram por manutenção, com os componentes previamente validados.

O teste de validação inclui a realização de testes de alto nível realizados pelos usuários finais do sistema, inicialmente em ambientes controlados pelo desenvolvedor e, posteriormente, em ambientes de produção não controlados pelo desenvolvedor, a fim de que ocorra a verificação do atendimento aos requisitos. Ao final, podemos ter a aceitação do sistema pelo usuário.

Finalizando, o teste de sistema, que extrapola os limites da Engenharia de Software, permite combinar o software com outros elementos do sistema, tais como hardware, base de dados etc.

Agora, é com você! Vamos verificar se atingimos o objetivo deste módulo? Você está convidado a realizar as atividades a seguir.

VERIFICANDO O APRENDIZADO

1. (SECRETARIA DA FAZENDA DO ESTADO DA BAHIA - AUDITOR FISCAL - TECNOLOGIA DA INFORMAÇÃO - FCC - 2019) SUPONHA QUE UMA AUDITORA FISCAL DA ÁREA DE TI ATUE NA ETAPA DE TESTES E AVALIAÇÃO DA QUALIDADE DE UM SOFTWARE EM DESENVOLVIMENTO. COMO O SOFTWARE SOFRIA ALTERAÇÕES A CADA NOVA FUNCIONALIDADE A ELE INCORPORADA, A AUDITORA PROPÔS QUE A EQUIPE DE TESTES ADOTASSE COMO PADRÃO UM TIPO DE TESTE QUE GARANTISSE QUE AS MUDANÇAS RECENTES NO CÓDIGO DEIXASSEM O RESTO DO CÓDIGO INTACTO, VISANDO IMPEDIR A INTRODUÇÃO DE ERROS. A EQUIPE DECIDIU REALIZAR UM TIPO DE TESTE PARA AVALIAR A PARTE MODIFICADA E AS ÁREAS ADJACENTES QUE PODEM TER SIDO AFETADAS, DENTRO DE UMA ABORDAGEM BASEADA EM RISCO. ASSIM, OS TESTADORES DESTACARIAM AS ÁREAS DE APLICAÇÃO QUE PODERIAM SER AFETADAS PELAS RECENTES ALTERAÇÕES DE CÓDIGO E SELECIONARIAM OS CASOS DE

TESTES RELEVANTES PARA O CONJUNTO DE TESTES. PROCEDENDO DESTA FORMA, SERIAM REALIZADOS TESTES:

A) De revisão de funcionalidade.

B) Gama.

C) De aceite operacional.

D) De regressão.

2. UMA EQUIPE DE DESENVOLVEDORES DO SOFTWARE ESTÁ NA FASE FINAL DOS TESTES EM AMBIENTE CONTROLADO E DECIDIU INICIAR OS TESTES DE RECUPERAÇÃO E SEGURANÇA IMEDIATAMENTE. ASSINALE A OPÇÃO CORRETA RELATIVA AO INÍCIO DOS REFERIDOS TESTES:

A) A equipe está realizando incorretamente os testes de sistema antes de realizar, por completo, os testes de validação.

B) A equipe está desenvolvendo corretamente os últimos testes antes de disponibilizar o software aos usuários finais.

C) A equipe deveria estar iniciando os testes de integração.

D) A equipe deveria estar iniciando os testes de validação do tipo Aceite Formal.

GABARITO

1. (Secretaria da Fazenda do Estado da Bahia - Auditor Fiscal - Tecnologia da Informação - FCC - 2019) Suponha que uma auditora fiscal da área de TI atue na etapa de testes e avaliação da qualidade de um software em desenvolvimento. Como o software sofria alterações a cada nova funcionalidade a ele incorporada, a auditora propôs que a equipe de testes adotasse como padrão um tipo de teste que garantisse que as mudanças recentes no código deixassem o resto do código intacto, visando impedir a introdução de erros. A equipe decidiu realizar um tipo de teste para avaliar a parte modificada e as áreas adjacentes que podem ter sido afetadas, dentro de uma abordagem baseada em risco. Assim, os testadores destacariam as áreas de aplicação que poderiam ser

afetadas pelas recentes alterações de código e selecionariam os casos de testes relevantes para o conjunto de testes. Procedendo desta forma, seriam realizados testes:

A alternativa "D" está correta.

Teste de regressão permite a reexecução de um subconjunto de testes previamente executados, assegurando que as alterações ou inserções de novas funcionalidades não impactaram outras partes do software.

2. Uma equipe de desenvolvedores do software está na fase final dos testes em ambiente controlado e decidiu iniciar os testes de recuperação e segurança imediatamente. Assinale a opção correta relativa ao início dos referidos testes:

A alternativa "A" está correta.

Os testes de validação do tipo Aceite Alpha ocorrem em ambiente controlado; na sequência, são realizados os testes de Aceite Beta e Formal, encerrando os testes de validação. Após esses testes, são realizados os testes de sistema, que incluem, entre outros, os testes de recuperação e segurança.

MÓDULO 4

● Descrever as etapas de implantação e manutenção do processo de desenvolvimento de software

IMPLANTAÇÃO

Considerando as etapas do processo de desenvolvimento de software apresentadas, o que temos até o momento?

O produto software testado em condições de migrar para o ambiente denominado “produção”, ou seja, mundo real, onde os usuários irão utilizá-lo nas suas rotinas diárias, seja na realização de um controle administrativo, seja no contexto de um processo de tomada de decisão.

A implantação é a etapa do processo de desenvolvimento de software relacionada à transferência do sistema da comunidade de desenvolvimento para a comunidade de usuários, ou seja, o sistema entra em produção no ambiente real.

O objetivo da implantação é produzir com sucesso a entrega do software para seus usuários finais, podendo cobrir vasta gama de atividades, como descritas a seguir:



A PRODUÇÃO DE RELEASES EXTERNOS DO SOFTWARE

Inclui o lançamento de uma nova versão do software, ou resultado de uma manutenção no mesmo, sendo exigidas decisões, por parte de desenvolvedores e usuários, sobre como realizar a distribuição do produto.

A EMBALAGEM DO SOFTWARE

Quando o software será comercializado por meio de determinada mídia, como, por exemplo, DVD.



DISTRIBUIÇÃO DO SOFTWARE

É o processo de entrega de software para o usuário final, podendo ocorrer de forma manual, sem auxílio de ferramentas, ou automática, com auxílio de ferramentas.

INSTALAÇÃO DO SOFTWARE

É um processo que inclui a instalação de todos os arquivos necessários à execução do software projetado.



PRESTAÇÃO DE AJUDA E ASSISTÊNCIA AOS USUÁRIOS

Quando a complexidade do software exige um serviço de atendimento aos clientes que procuram esclarecimentos sobre dúvidas relacionadas às funcionalidades ou sobre problemas técnicos relacionados ao software, e.g., serviço de *help desk*.

GESTÃO DE CONFIGURAÇÕES VERSUS

IMPLEMENTAÇÃO

Vamos imaginar que estamos desenvolvendo um software utilizando o fluxo de processo iterativo e incremental, ou evolucionário, sendo esse fluxo comumente adotado atualmente. Após os testes, geramos a primeira versão e a disponibilizamos no ambiente de produção. Neste ponto, realizamos a implantação dessa versão.

O que faz a equipe desenvolvedora? Inicia o desenvolvimento da segunda versão, ou seja, mais um conjunto de requisitos é analisado em um novo ciclo iterativo. Em algum momento, soa um alerta dos usuários informando que “temos um defeito” no software em produção.

O QUE VOCÊ FARIA COMO ENGENHEIRO DE SOFTWARE?

CENÁRIO 1

Terminar a versão 2 com o defeito corrigido e liberá-la para produção.

CENÁRIO 2

Realizar a manutenção, eliminando o defeito na versão 1, liberá-la para produção e realizar a devida alteração na versão 2 em desenvolvimento.

Tecnicamente, o cenário 2 é a melhor solução, pois o usuário teria de aguardar a liberação da versão 2, convivendo por um período com o impacto negativo do defeito do software no ambiente de produção.

Neste pequeno estudo de caso, podemos destacar dois problemas para o engenheiro de software: o controle de alterações e o controle de versões. Nesta etapa, temos de considerar alguns aspectos que relacionam a gestão de configurações com a implantação de um sistema.

Gestão de configuração, de forma simplificada, é um conjunto de tarefas que visam gerenciar as alterações durante o desenvolvimento do software, sendo essa gestão aplicada em todas as etapas do processo de desenvolvimento de software. Uma das tarefas do processo de gestão de configuração é denominada “controle de alterações”, permitindo solucionar o primeiro problema apresentado anteriormente.

A Figura 15 ilustra um processo de controle de alterações, que, a partir de uma solicitação de alteração, requer a avaliação do mérito técnico, dos efeitos colaterais em potencial, do impacto global em termos de configuração e da funcionalidade e do custo da alteração. Na referida figura, o acrônimo ECO (*Engineering Change Order*) representa um pedido de alteração de engenharia.

QUAL A IMPORTÂNCIA DESTE PROCESSO PARA A ETAPA DE IMPLANTAÇÃO?

O processo tem de estar bem definido e ajustado à complexidade do software quando implantado, pois os defeitos serão identificados em produção, e os procedimentos de alteração estarão definidos. Lembre-se de que, em projetos complexos, a falta de um controle de alterações pode gerar o caos. Afinal, sabe-se que a fase de testes não é capaz de zerar a existência de erros.

Vamos, agora, ao segundo problema do nosso pequeno exemplo apresentado, ou seja, “controle de versões”. Neste caso, ainda no contexto do gerenciamento de configuração, podemos destacar o gerenciamento de releases.

Um release de sistema é uma versão do mesmo distribuída aos clientes. A gestão de releases determina quando um release será liberado para o cliente, gerencia o processo de criação e de meios de distribuição, bem como documenta o release.

Um release pode incluir:



Fonte: Kraphix / Shutterstock

ARQUIVOS DE CONFIGURAÇÃO



Fonte: Kraphix / Shutterstock

ARQUIVOS DE DADOS



Fonte: Kraphix / Shutterstock

UM PROGRAMA DE INSTALAÇÃO



Fonte: Kraphix / Shutterstock

DOCUMENTAÇÃO ELETRÔNICA OU EM PAPEL



Fonte: Kraphix / Shutterstock

EMPACOTAMENTO E PUBLICIDADE ASSOCIADA

SOLICITAÇÃO DE ALTERAÇÃO



GERAÇÃO DE RELATÓRIO DE AVALIAÇÃO



APROVAÇÃO



SIM



GERA A ECO E REALIZA A ALTERAÇÃO

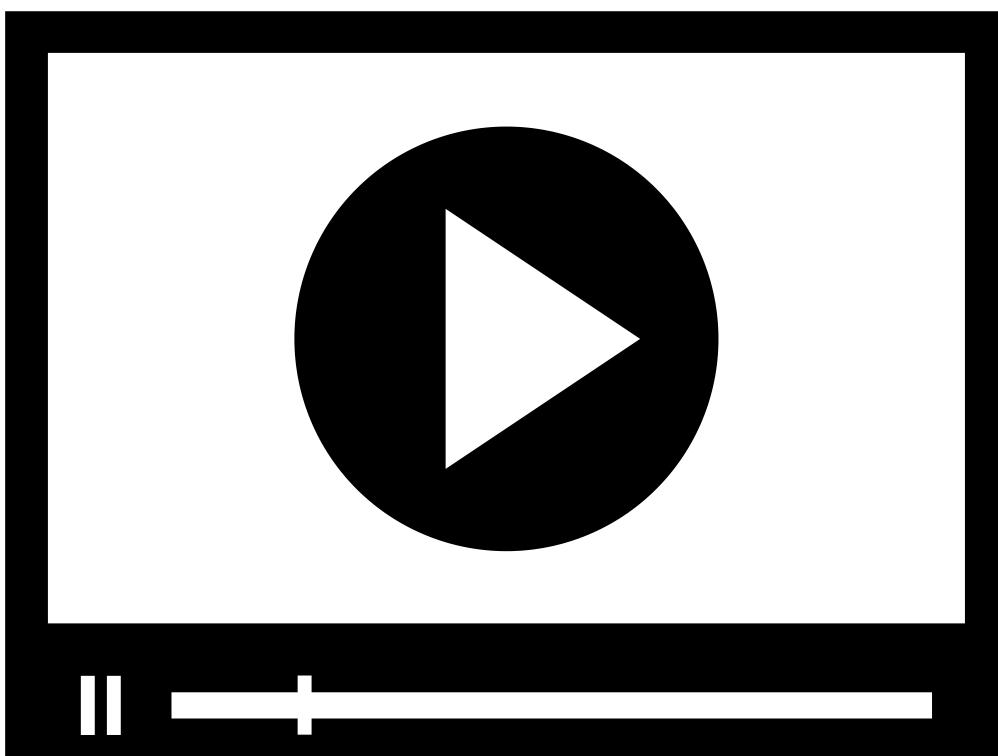


REALIZA GARANTIA DA QUALIDADE



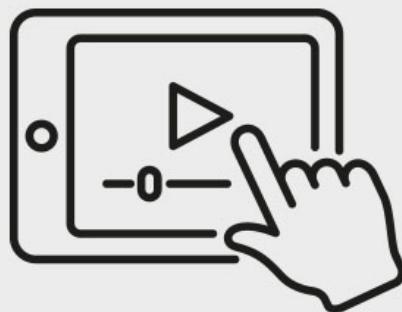
DISTRIBUI A NOVA REVISÃO

Figura 15 – O processo de controle de alterações.



Neste vídeo, você conhecerá um pouco sobre o processo de gerenciamento de riscos na etapa de planejamento.

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.



MANUTENÇÃO

Realizada a implantação do software com sucesso, iniciamos imediatamente a próxima etapa: a manutenção.

A etapa mais longa do ciclo de vida do software, a manutenção inclui a correção de defeitos não identificados nas etapas anteriores do processo de desenvolvimento de software, no aprimoramento da implementação dos subsistemas e na disponibilização de novas funcionalidades a partir da identificação de novos requisitos.

Pouco tempo depois de iniciada a etapa de manutenção, começam a chegar aos engenheiros de software os relatos de erros, bem como solicitações de adaptações e melhorias. Estas devem ser planejadas, programadas e executadas. Reiteramos aqui a importância do processo de gestão de alterações descrito anteriormente e ilustrado na Figura 15.

Outro problema associado à manutenção é a mobilidade dos engenheiros de software, pois, com o passar do tempo, a equipe que trabalhou no desenvolvimento do software pode ter sido substituída ou mesmo dissolvida, ou seja, pode ocorrer a inexistência de um engenheiro de software que conheça o sistema.



Fonte: Pixel-Shot / Shutterstock

Neste ponto, podemos destacar a importância do correto cumprimento de todas as etapas do processo de desenvolvimento de software, de modo a gerar uma solução bem estruturada para o projeto, incluindo os diversos modelos que representam decisões tomadas ao longo do projeto, o que permite o entendimento da solução por parte de um engenheiro de software que não trabalhou no desenvolvimento do produto.

Na verdade, toda solução que aplica as melhores práticas da engenharia de software busca o cumprimento do requisito não funcional denominado manutenibilidade, sendo esse um indicativo qualitativo da facilidade de corrigir, adaptar ou melhorar o software.

Importante! Um dos grandes objetivos da engenharia de software é criar sistemas que apresentem alta manutenibilidade.

MANUTENÇÃO VERSUS REENGENHARIA DE SOFTWARE

Vamos considerar determinado produto de software implantado. Ao longo do tempo, a equipe de engenheiros de software realiza as manutenções necessárias em função de erros

detectados ou de alterações geradas a partir da volatilidade dos requisitos. Paralelamente, a tecnologia vai se tornando obsoleta, e a manutenção, cada vez mais difícil.

O QUE VOCÊ FARIA? QUE TAL RECONSTRUÍRMOS O PRODUTO COM MAIS FUNCIONALIDADES, MELHOR DESEMPENHO, CONFIABILIDADE E MANUTENIBILIDADE? ESSE PROCESSO É DENOMINADO REENGENHARIA.

A reengenharia possui dois níveis: reengenharia de processos de negócio e reengenharia de software.

No primeiro nível, também denominado estratégico, a reengenharia de processos de negócio visa – por meio de alterações em regras de negócio, ou seja, das tarefas que permitem a obtenção de um resultado de negócio – o aumento da eficiência e competitividade de uma empresa.

Destacamos que um software tem como importante objetivo agregar valor a uma organização por meio da automação de processos; portanto, a referida alteração no negócio deve ser absorvida pela engenharia de software, gerando dois cenários:

CENÁRIO 1

Que exige a construção de um novo software.

CENÁRIO 2

Que permite a modificação de um software existente.

No segundo cenário, aplicamos a reengenharia de software, o que extrapola os objetivos da etapa manutenção.

RESUMINDO

Neste módulo, destacamos a importância das etapas de implantação e manutenção do processo de desenvolvimento de software.

A implantação inclui a migração do software de um ambiente de desenvolvimento para um ambiente de produção, permitindo a utilização do software pelos diversos usuários envolvidos.

Após a etapa de implantação, temos a manutenção. A existência de erros descobertos em produção é um fato; para tal, a equipe de engenheiros de software terá de aplicar as tarefas definidas no processo de gerenciamento de alterações. Outro motivo comum de solicitação de manutenção ocorre em função da volatilidade dos requisitos.

Com o tempo, as solicitações vão sendo intensificadas em função de alterações de regras de negócios ou pela obsolescência da tecnologia, podendo-se, neste caso, aplicar a reengenharia de software, o que extrapola a etapa de manutenção.

Agora, é com você! Vamos verificar se atingimos o objetivo deste módulo? Você está convidado a realizar as atividades a seguir.

VERIFICANDO O APRENDIZADO

1. O GERENTE DE DETERMINADO PROJETO DE SOFTWARE POSSUI UMA LONGA LISTA DE REQUISITOS FUNCIONAIS E NÃO FUNCIONAIS, EM FUNÇÃO DA SUA COMPLEXIDADE. AS EQUIPES DE PROGRAMADORES E DE QUALIDADE ESTÃO ENCERRANDO AS ETAPAS DE IMPLEMENTAÇÃO E TESTES, POSSIBILITANDO A IMPLANTAÇÃO DO SOFTWARE. QUAL PROCESSO TEM DE ESTAR BEM DEFINIDO E AJUSTADO À COMPLEXIDADE DO SOFTWARE QUANDO DA EXECUÇÃO DA ETAPA IMPLANTAÇÃO, EM FUNÇÃO DOS DEFEITOS QUE DEVERÃO SER IDENTIFICADOS EM PRODUÇÃO?

- A) Processo de reengenharia.**
- B) Processo de controle de alterações.**

C) Processo de controle de releases.

D) Processo de software.

2. (CESGRANRIO - 2013 - BNDES - PROFISSIONAL BÁSICO - ANÁLISE DE SISTEMAS – DESENVOLVIMENTO) DE MODO GERAL, O PROCESSO DE DESENVOLVIMENTO DE UM SOFTWARE PODE SER ORGANIZADO PARTINDO DE TRÊS FASES IMPORTANTES, QUE SÃO AS DE DEFINIÇÃO, DE DESENVOLVIMENTO E DE MANUTENÇÃO. NA FASE DE MANUTENÇÃO, DENTRE OUTRAS ATIVIDADES, SÃO:

A) Levantados os requisitos dos usuários para a programação das diversas fases do projeto, inclusive as operacionais e as preditivas.

B) Efetuados os testes de funcionalidade do software, revistos os objetivos para os quais ele foi desenvolvido e redefinidas as funções em desacordo com esses objetivos.

C) Incluídas novas funções requeridas pelo cliente e feitas adaptações por modificações de hardware.

D) Reavaliadas as bases operacionais, nas quais o software está sendo executado, e prototipados os novos requisitos de hardware.

GABARITO

1. O gerente de determinado projeto de software possui uma longa lista de requisitos funcionais e não funcionais, em função da sua complexidade. As equipes de programadores e de qualidade estão encerrando as etapas de implementação e testes, possibilitando a implantação do software. Qual processo tem de estar bem definido e ajustado à complexidade do software quando da execução da etapa implantação, em função dos defeitos que deverão ser identificados em produção?

A alternativa "**B**" está correta.

Com certeza, ocorrerão erros em produção, gerando pedidos de manutenção do software. A definição prévia de um processo de controle de alterações permitirá a realização de manutenções sistemáticas e de forma planejada.

2. (CESGRANRIO - 2013 - BNDES - Profissional Básico - Análise de Sistemas – Desenvolvimento) De modo geral, o processo de desenvolvimento de um software pode ser organizado partindo de três fases importantes, que são as de definição, de desenvolvimento e de manutenção. Na fase de manutenção, dentre outras atividades, são:

A alternativa "C" está correta.

As alterações solicitadas durante a etapa de manutenção do software comumente incluem erros gerados em etapas anteriores do processo de desenvolvimento de software. Devido ao uso do software pelos usuários, poderão também surgir solicitações de novas funcionalidades, bem como problemas decorrentes de atualização de hardware.

CONCLUSÃO

CONSIDERAÇÕES FINAIS

Apresentamos as etapas genéricas, ou típicas, de um processo de desenvolvimento de software, que são: levantamento de requisitos, análise, projeto, implementação, testes, implantação e manutenção. Lembramos que a Engenharia de Requisitos inclui as etapas de levantamento de requisitos e análise desse processo.

O levantamento de requisitos tem como principal entrega o documento de requisitos, comumente categorizados em requisitos funcionais, não funcionais e de domínio, ou regras de negócio. Essa etapa define o contexto do projeto.

Na análise, o engenheiro de software cria os modelos a partir dos requisitos. Como exemplos, destacamos os modelos de casos de uso, de classes e atividades.

No projeto, ocorrem os refinamentos dos modelos de análise e a criação de novos modelos, que permitirão a implementação da solução. Como exemplo, podemos destacar os modelos de interação, implementação e implantação.

Na etapa de implementação, ocorre a codificação da solução a partir dos modelos gerados na etapa de projeto. Na sequência, temos a etapa de testes, que permite a validação inicial de unidades de código, terminando na validação do sistema como um todo em ambiente próximo ao de produção.

A etapa de implantação permite a migração para o ambiente de produção, onde estão os usuários finais do sistema. Em seguida, temos a etapa de manutenção, que visa o atendimento às solicitações de alterações no software em virtude de defeitos encontrados ou alterações decorrentes da volatilidade de alguns requisitos.

Para ouvir um *podcast* sobre o assunto, acesse a versão online deste conteúdo.



REFERÊNCIAS

BEZERRA, E. **Princípios de Análise e Projeto de Sistemas com UML**. 3. ed. Rio de Janeiro: Elsevier, 2014.

DIJKSTRA, E. W. **The Humble Programmer**: Turing Award Lecture, Comm. ACM, v. 15, n. 10, p. 859-866, out. 1972. Consultado em meio eletrônico em: 18 ago. 2020.

PRESSMAN, R. S.; MAXIM, B. R. **Engenharia de Software**. 8. ed. Porto Alegre: AMGH Editora, 2016.

SOMMERVILLE, I. **Engenharia de Software**. 8. ed. São Paulo: Pearson Prentice Hall, 2007.

EXPLORE+

Leia o capítulo 2 de *Princípios de Análise e Projeto de Sistemas com UML*, de Eduardo Bezerra.

Leia os capítulos 3, 7, 12 e 22 de *Engenharia de Software*, de Roger Pressman.

Leia os capítulos 4, 6, 11 e 23 de *Engenharia de Software*, de Ian Sommerville.

CONTEUDISTA

Alberto Tavares da Silva

 CURRÍCULO LATTES



DESCRIÇÃO

O processo de desenvolvimento de software e sua aplicação no Gerenciamento de Projeto.

PROpósito

Compreender a importância do processo de desenvolvimento de software e do Gerenciamento de Projetos no contexto da Engenharia de Software.

OBJETIVOS

MÓDULO 1

MÓDULO 2

Descrever as etapas essenciais de um processo de desenvolvimento de software

MÓDULO 3

Relacionar as etapas de um processo de desenvolvimento de software com as etapas de Gerenciamento de Projeto

MÓDULO 4

Descrever a importância do Gerenciamento de Risco no projeto de software

INTRODUÇÃO

Nos dias atuais, a importância do software é facilmente perceptível em função dos inúmeros serviços digitais disponíveis na nossa Sociedade da Informação. Em outros cenários, o software também está presente em sistemas de controle de veículos, aviões, refinarias entre outros.

Podemos afirmar que o “produto” software tem que ser projetado aplicando-se as melhores práticas da engenharia, pois quais seriam as consequências de um defeito de software em uma aeronave com 500 pessoas ou na falha de um sistema de controle de tráfego aéreo?

Realmente, o software necessita das melhores práticas da engenharia no seu projeto.

Neste contexto, destaca-se a disciplina Engenharia de Software, que trata dos aspectos técnicos que permitem a geração do “produto” software. De uma forma geral, a engenharia,

qualquer que seja, necessita de processos. Em função dessa premissa, podemos afirmar que não existe engenharia sem processo.

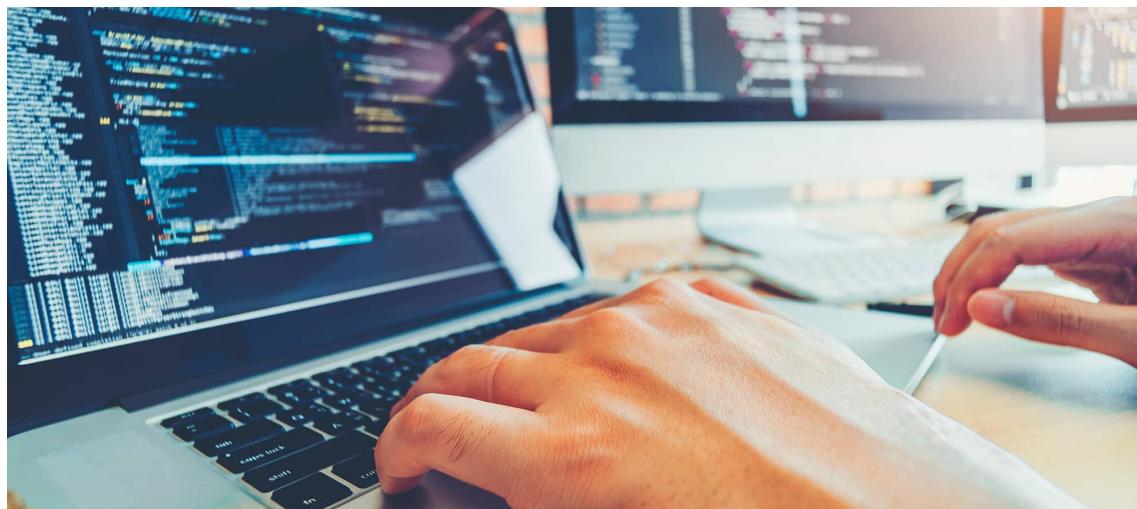
O processo de desenvolvimento de software inclui algumas etapas básicas, cujo momento de aplicação depende do modelo adotado pelo engenheiro de software.

Uma área de conhecimento importante, aplicada em qualquer engenharia, é o Gerenciamento de Projeto. Ele permite elaborar um plano de projeto onde são otimizados os recursos, que incluem pessoas, materiais e equipamentos. A metodologia de gerenciamento de projeto permite, a partir de requisitos identificados nas primeiras etapas do processo de qualquer engenharia, a geração do produto objeto do projeto de acordo com o planejado e de forma controlada.

A Engenharia de Software está alinhada com a área de conhecimento Gerenciamento de Projetos, sendo esta fundamental ao engenheiro de software. Cabe um destaque especial nesse contexto ao Gerenciamento de Riscos, principalmente, na seleção de portfólio de projetos de software.

MÓDULO 1

- Reconhecer os conceitos básicos relacionados com o desenvolvimento de software



Fonte: Andrey Suslov/Shutterstock

SOFTWARE

Seria possível a sua vida hoje sem o smartphone?

Acredito que a maioria em uma pesquisa iria responder um enfático “não”!

O que você acha que torna o telefone móvel tão atrativo? O software?

É provável, pois, não descartando a importância do hardware, o software permite uma relação de usabilidade interativa e intuitiva, gerando um uso intensivo. Essa competência é resultado de trabalho multidisciplinar árduo de engenheiros de softwares, administradores de banco de dados, web designers, entre outros profissionais.

Neste contexto, vamos iniciar nosso estudo sobre a extraordinária área de conhecimento Engenharia de Software, que tem como principal produto o **software**.

► ATENÇÃO

Destacamos que a bibliografia Pressman (2016) é uma referência mundial nessa área.

O conceito de software a seguir define esse importante produto.

SOFTWARE CONSISTE EM:

**(1) INSTRUÇÕES (PROGRAMA DE COMPUTADOR)
QUE, QUANDO EXECUTADAS, FORNECEM
CARACTERÍSTICAS, FUNÇÕES E DESEMPENHO
DESEJADOS; (2) ESTRUTURAS DE DADOS QUE
POSSIBILITAM AOS PROGRAMAS MANIPULAR
INFORMAÇÕES ADEQUADAMENTE; E (3)**

INFORMAÇÃO DESCRIPTIVA, TANTO NA FORMA IMPRESSA QUANTO NA VIRTUAL, DESCREVENDO A OPERAÇÃO E O USO DOS PROGRAMAS.

(PRESSMAN, 2016).

Implicitamente, o software relaciona-se com o hardware. Essa relação gerou a denominada “Crise do Software”, sintetizada pela afirmativa a seguir de Edsger.

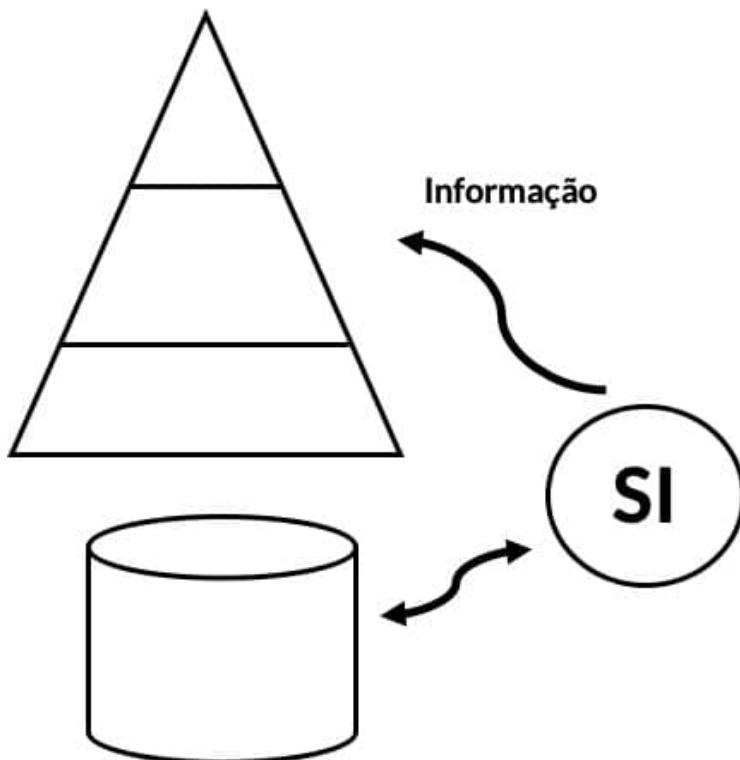
A MAIOR CAUSA DA CRISE DO SOFTWARE É QUE AS MÁQUINAS SE TORNARAM VÁRIAS ORDENS DE MAGNITUDE MAIS POTENTES! EM TERMOS DIRETOS, ENQUANTO NÃO HAVIA MÁQUINAS, PROGRAMAR NÃO ERA UM PROBLEMA; QUANDO TIVEMOS COMPUTADORES FRACOS, ISSO SE TORNOU UM PROBLEMA PEQUENO; E AGORA QUE TEMOS COMPUTADORES GIGANTESCOS, PROGRAMAR TORNOU-SE UM PROBLEMA GIGANTESCO.

(DIJKSTRA, 1972).

É importante destacar que o desenvolvimento tecnológico do hardware nos últimos anos permitiu o desenvolvimento de softwares cada vez mais complexos, tendo um forte impacto na indústria de software. Como exemplo, podemos apresentar a substituição do paradigma estruturado pelo paradigma orientado a objetos, baseado na programação orientada a objetos que permite o reuso intensivo de especificação, bem como uma melhor manutenibilidade e, como consequência, o desenvolvimento de softwares mais complexos.

Vamos enfatizar uma das principais finalidades do software: a geração de informação. A figura 1 ilustra de forma implícita os principais componentes tecnológicos de um Sistema de

Informação, i.e., hardware, software, sistema gerenciador de banco de dados, redes de comunicação e serviços, sendo os referidos componentes interdependentes. O ambiente empresarial está representado pela respectiva pirâmide funcional, cabendo ao componente software, o importante papel de agregar valor aos dados quando da geração de informações aos diferentes níveis de gestão – operacional, gerencial e estratégico, pois essa pode ser utilizada em um processo de tomada de decisão ou no controle de funções empresariais, tais como financeiro, recursos humanos e outras.



Fonte: O autor.

Figura 1 – Sistema de Informação x Software.

Um segundo exemplo, vamos considerar a complexidade do desenvolvimento de um software embarcado em uma aeronave com 500 pessoas e que realiza o seu controle total, em que um defeito pode ter um impacto altamente negativo.

Vamos imaginar o que seria do piloto sem as informações do voo no painel de controle.

Em ambos exemplos, observamos que o software apresenta dois papéis distintos, o primeiro como um produto a ser utilizado pelos usuários, o segundo como veículo que distribui o produto, pois a comunicação entre os diversos componentes de um sistema de informação ocorre por meio de sistemas operacionais, software de comunicação entre outros. O software distribui o produto mais importante da nossa era – a **Informação** (PRESSMAN, 2016).

Você consegue imaginar quais os desafios atuais de um engenheiro de software?

Destacamos que os desafios incluem sete grandes categorias:

SOFTWARE DE SISTEMA

Camadas de software que atendem a outros softwares, tais como, sistemas operacionais, drivers e outros.

SOFTWARE DE APLICAÇÃO

Inclui software com escopo específico, tais como, sistemas de gestão empresarial (ERP).

SOFTWARE DE ENGENHARIA/CIENTÍFICO

Inclui software aplicado às áreas de engenharia e científica, tal como, software para cálculo estrutural na área de engenharia civil ou processamento de imagem.

SOFTWARE EMBARCADO

Instalado em produtos com funções específicas, tal como, o controle de um veículo com informações disponíveis no painel digital.

SOFTWARE PARA LINHA DE PRODUTOS

Projetado com determinado conjunto de funcionalidades e utilizado por diferentes clientes, por exemplo, sistema emissor de nota fiscal.

APLICAÇÕES WEB/APLICATIVOS MÓVEIS

Software específico para dispositivo móvel.

SOFTWARE DE INTELIGÊNCIA ARTIFICIAL

Utilizam técnicas de inteligência artificial, tais como, sistema especialistas, redes neurais, aprendizado de máquinas e outros.

ENGENHARIA DE SOFTWARE

Vimos nos exemplos do software embarcado em uma aeronave ou controlando o tráfego aéreo uma característica comum: a complexidade. A melhor tratativa para a **complexidade** é a aplicação de metodologia que permita a decomposição do problema em problemas menores de forma sistemática, cabendo à engenharia essa sistematização.

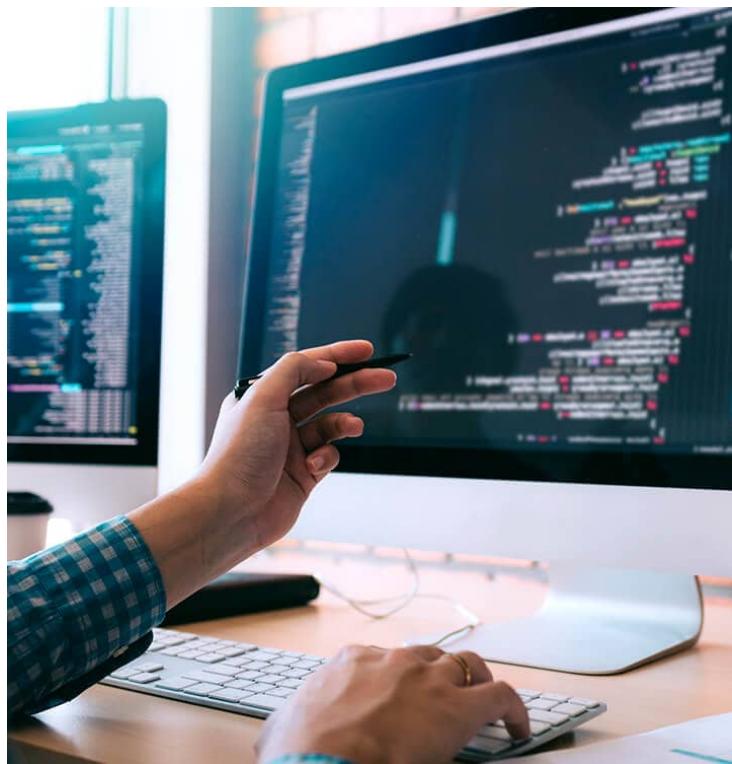
Uma premissa básica é que a engenharia permite a solução de problemas e, quanto mais complexo um produto a ser gerado, mais a engenharia faz-se necessária.



Fonte: KorArkaR/Shutterstock

Vamos ver no caso da construção de uma simples casa, talvez um engenheiro resolva o problema; e no caso de um edifício inteligente com vários andares? Neste caso, o problema tornou-se multidisciplinar, ou seja, serão necessários vários profissionais de várias áreas, e.g., arquiteto, engenheiros civis, mecânicos, elétricos, de software etc. A construção do prédio torna-se inviável caso não haja uma tratativa sistemática por meio da aplicação das melhores práticas da engenharia.

A mesma correlação pode ser aplicada quando o produto a ser gerado é o software. No caso do software, aplica-se a Engenharia de Software.



Fonte: KorArkaR/Shutterstock

O desenvolvimento de software deve submeter-se aos mesmos princípios aplicados nas engenharias tradicionais, sendo um produto distinto em função da sua intangibilidade que o associa, muitas vezes, somente aos códigos dos programas de computador. Entretanto, diferentemente de outros produtos de outras engenharias em produção, possui alta volatilidade em função de constantes evoluções na tecnologia e nos requisitos, agregando ao software uma complexidade adicional.

A primeira referência ao termo “Engenharia de Software” foi em 1968, em uma conferência sobre o tema, sob responsabilidade do Comitê de Ciência da NATO (North Atlantic Treaty Organization). Nesse contexto, o Guide to the Software Engineering Body of Knowledge Version 3.0 (SWEBOk Guide V3.0) apresenta a seguinte definição para a Engenharia de Software:

A APLICAÇÃO DE UMA ABORDAGEM SISTEMÁTICA, DISCIPLINADA E QUANTIFICÁVEL PARA O DESENVOLVIMENTO, OPERAÇÃO E MANUTENÇÃO DE

SOFTWARE, QUE É A APLICAÇÃO DE ENGENHARIA AO SOFTWARE.

TECNOLOGIA EM CAMADAS

A Engenharia de Software é uma tecnologia em camadas, como ilustra a figura 2. Vejamos as descrições das referidas camadas:

CAMADA DE QUALIDADE

Garante que os requisitos que atendem às expectativas do usuário serão cumpridos.

CAMADA DE PROCESSO

Determina as etapas de desenvolvimento do software.

CAMADA DE MÉTODOS

Define, por exemplo, quais as técnicas de elicitação de requisitos, os artefatos gerados em função da técnica de modelagem adotada, tal como, modelo de casos de uso ou de classes.

CAMADA DE FERRAMENTAS

Estimula a utilização de ferramentas “CASE” (Computer-Aided Software Engineering) no desenho dos diversos artefatos ou mesmo na geração automática de código, entre outras aplicações; a tecnologia CASE está disponível para uso em todas as etapas do processo de desenvolvimento de software.



Fonte: O autor

Figura 2 - Camadas da Engenharia de Software.

Considerando um produto tangível para todos nós, você poderia imaginar que um dia construirá sua casa própria? Vamos imaginar que sim. O que você precisaria para levar em frente esta construção? Um processo?

Certamente, pois terá que organizar as etapas da construção: fundação, estrutura, cobertura, alvenaria, instalações etc. Agora a construção da casa fica mais fácil.

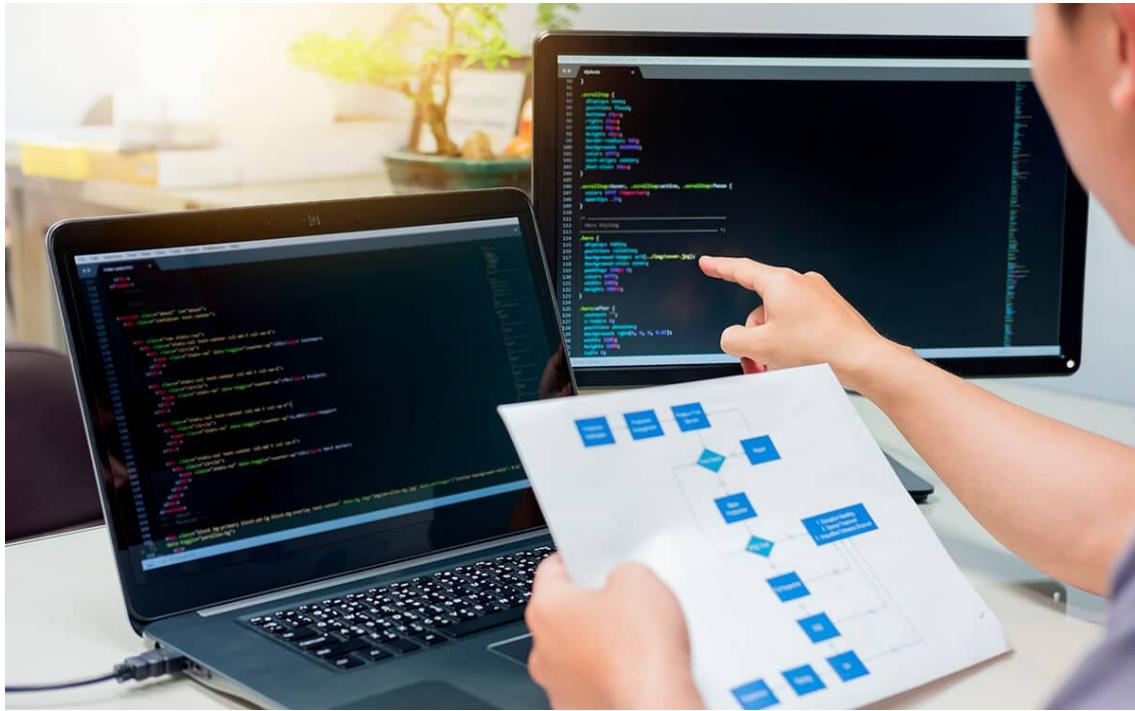
Da mesma maneira, a construção de um software necessita de processo; esse deverá ocorrer de acordo com as melhores práticas estabelecidas pela Engenharia de Software.

É importante destacar que a base da Engenharia de Software é a **camada de processo**.

PROCESSO DE SOFTWARE

Processo é uma sequência de etapas que permitem a geração de um produto, no nosso caso, o software. O processo permite uma melhor tratativa em relação à complexidade de obtenção de um determinado produto, pois, na maioria das vezes, é um trabalho multidisciplinar realizado por analistas, programadores, gerentes de projeto, gerentes de teste e outros profissionais.

Assim como em todas as engenharias, a Engenharia de Software possui uma diversidade de modelos para processos de desenvolvimento de software que atendem a diferentes problemas.



Fonte: por_suwat/Shutterstock

Um requisito importante na seleção de um processo de software é a complexidade.

Basicamente, quanto maior a complexidade de um sistema, mais formal deve ser o processo adotado. Importante: a qualidade é a camada base que sustenta a camada processo.

Uma metodologia de processo genérica compreende cinco atividades (figura 3).



Fonte: O autor

Figura 3 - Metodologia do Processo.

COMUNICAÇÃO

As primeiras atividades de um processo de software requerem uma comunicação intensiva com os usuários, buscando o entendimento do problema, a definição de objetivos para o projeto, bem como a identificação de requisitos.

PLANEJAMENTO

Destaca-se nesta atividade a área de conhecimento Gerenciamento de Projeto, que permitirá a elaboração de um Plano de Gerenciamento do Projeto de forma sistemática, tendo como entrega importante o cronograma que inclui as atividades a serem desenvolvidas no referido projeto, contemplando diferentes áreas de conhecimento.

O processo de desenvolvimento de software disponibiliza as principais atividades que irão compor o Plano de Gerenciamento do Projeto, sendo esse plano executado e monitorado.

MODELAGEM

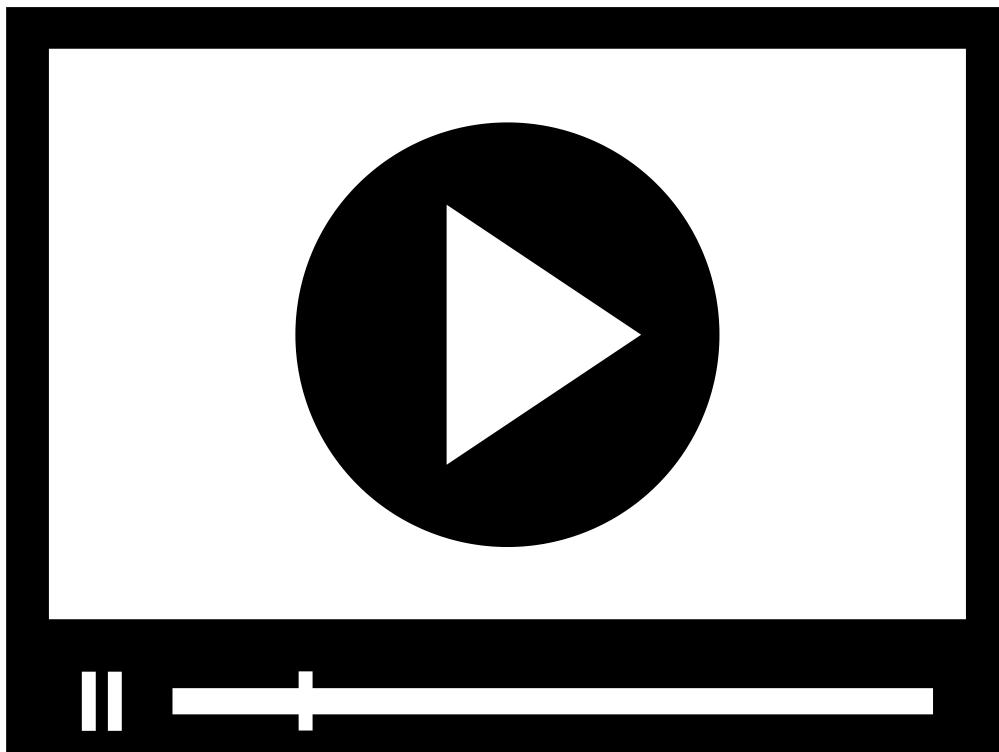
A engenharia tem como melhor prática a geração de modelos, tal como a planta baixa de uma casa. A maioria desses modelos gráficos, denominados de diagramas na Engenharia de Software, podem ser complementados por descrições textuais. Como exemplo, apresentamos o modelo de casos de uso que inclui diagramas de casos de uso, artefatos gráficos e descrições de cenários dos casos de uso, artefatos textuais.

CONSTRUÇÃO

A partir dos modelos gerados, é realizada a construção ou implementação do software, portanto, os modelos determinam o comportamento do software. Essa atividade inclui a codificação e os testes de software de acordo com o planejado.

ENTREGA

Ao final, ocorre o objetivo de um plano de projeto de software, i.e., a entrega do produto em produção de acordo com o planejado.



Assista agora ao vídeo **Metodologia do processo de desenvolvimento**.

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.



As atividades básicas apresentadas são complementadas pelas denominadas “atividades de apoio”, que incluem:

CONTROLE E ACOMPANHAMENTO DE PROJETO

ADMINISTRAÇÃO DE RISCOS

GARANTIA DA QUALIDADE DE SOFTWARE

REVISÕES TÉCNICAS

MEDIÇÃO

GERENCIAMENTO DA CONFIGURAÇÃO DE SOFTWARE

GERENCIAMENTO DA CAPACIDADE DE REUTILIZAÇÃO

PREPARO E PRODUÇÃO DE ARTEFATOS DE SOFTWARE

CONTROLE E ACOMPANHAMENTO DE PROJETO

Dentro do princípio de que “não se controla o que não se mede”, a etapa do Gerenciamento de Projeto Monitoramento e Controle permite verificar se a execução está de acordo com o planejado, pois, caso seja identificada alguma não conformidade, ações corretivas devem ser implementadas.

ADMINISTRAÇÃO DE RISCOS

Ênfase à área de Gerenciamento de Risco, de modo que qualquer evento, positivo ou negativo que possa impactar o desenvolvimento do projeto, seja tratado.

GARANTIA DA QUALIDADE DE SOFTWARE

Ênfase à área de Gerenciamento da Qualidade, a fim de garantir que os requisitos do projeto sejam atendidos.

REVISÕES TÉCNICAS

Atividade intrínseca da qualidade, pois todos os artefatos devem ser testados, inclusive, processos, modelos, código do software e outros.

MEDIÇÃO

Outra atividade da qualidade que permite a definição de métricas para avaliar as várias atividades durante o desenvolvimento do software.

GERENCIAMENTO DA CONFIGURAÇÃO DE SOFTWARE

Inclui os efeitos das mudanças, por exemplo o gerenciamento de versionamento do software em um processo iterativo e incremental, onde um defeito em produção pode surgir em uma versão anterior à desenvolvida no momento, sendo a manutenção realizada na versão anterior e propagada para a versão atual. Este procedimento, muitas vezes, exige uma ferramenta que automatize a solução deste problema, que pode ser bem complexo.

GERENCIAMENTO DA CAPACIDADE DE REUTILIZAÇÃO

O reuso de software deve ser um objetivo persistido por quem desenvolve o software. No paradigma estruturado, existiam as bibliotecas de funções que permitiam a reutilização de código em várias aplicações com possibilidades bem limitadas e com o advento do paradigma orientado a objetos, o reuso ficou mais sofisticado em função de mecanismos, e.g. herança, que possibilitam uma melhor componentização do software.

PREPARO E PRODUÇÃO DE ARTEFATOS DE SOFTWARE

Um processo de software encadeia uma série de atividades, sendo que estas atividades possuem métodos próprios para a geração de artefatos que necessitam ser documentados, e.g., modelo de casos de uso.

Cabe ao engenheiro de software documentar cada atividade que será aplicada no processo de desenvolvimento de software em função da natureza do problema (complexidade), das características das pessoas que realizarão o trabalho e dos usuários envolvidos.

RESUMINDO

Neste módulo, podemos destacar a importância do software atualmente, bem como da complexidade no seu desenvolvimento. A aplicação da Engenharia de Software permite lidar com a referida complexidade, pois melhores práticas podem ser aplicadas de forma a gerar um produto “software” que atenda às necessidades para as quais foi projetado.

Destacamos que a Engenharia de Software é uma tecnologia em camadas, ou seja, com foco na qualidade, processo, métodos e ferramentas. Cabe enfatizar que a base da Engenharia de Software é a camada de processo, por isso foram descritas as principais atividades genéricas que devem compor um processo de software: comunicação, planejamento, modelagem, construção e entrega.

VERIFICANDO O APRENDIZADO

1. FCC – 2010 (ADAPTADO) - SOBRE A ENGENHARIA DE SOFTWARE, CONSIDERE:

- I. ATUALMENTE, TODOS OS PROBLEMAS NA CONSTRUÇÃO DE SOFTWARE DE ALTA QUALIDADE NO PRAZO E DENTRO DO ORÇAMENTO FORAM SOLUCIONADOS.**
- II. AO LONGO DOS ÚLTIMOS 50 ANOS, O SOFTWARE EVOLUIU DE UM PRODUTO DE INDÚSTRIA PARA UM FERRAMENTAL ESPECIALIZADO EM SOLUÇÃO DE PROBLEMAS E ANÁLISE DE INFORMAÇÕES ESPECÍFICAS.**
- III. TODO PROJETO DE SOFTWARE É INICIADO POR ALGUMA NECESSIDADE DO NEGÓCIO.**
- IV. O INTUITO DA ENGENHARIA DE SOFTWARE É FORNECER UMA ESTRUTURA PARA A CONSTRUÇÃO DE SOFTWARE COM ALTA QUALIDADE.**

ESTÁ CORRETO O QUE CONSTA EM:

- A) III e IV, somente.**
- B) II e III, somente.**
- C) I, II e IV, somente.**
- D) II, III e IV, somente.**

2. PREFEITURA MUNICIPAL DE MANAUS – 2010 - A ENGENHARIA DE SOFTWARE COMPREENDE UM CONJUNTO DE ETAPAS COMUMENTE CITADAS COMO PARADIGMAS DE ENGENHARIA DE SOFTWARE. NO TOCANTE A ESSAS ETAPAS, ASSINALE A OPÇÃO CORRETA.

A) Os procedimentos da Engenharia de Software constituem o elo que mantém juntos os métodos e as ferramentas.

B) Os métodos de Engenharia de Software proporcionam os detalhes de “o que fazer” para construir o software.

C) As ferramentas de Engenharia de Software proporcionam apoio totalmente automatizado aos métodos.

D) Os procedimentos da Engenharia de Software garantem o desenvolvimento dentro do prazo.

GABARITO

1. FCC – 2010 (adaptado) - Sobre a engenharia de software, considere:

I. Atualmente, todos os problemas na construção de software de alta qualidade no prazo e dentro do orçamento foram solucionados.

II. Ao longo dos últimos 50 anos, o software evoluiu de um produto de indústria para um ferramental especializado em solução de problemas e análise de informações específicas.

III. Todo projeto de software é iniciado por alguma necessidade do negócio.

IV. O intuito da engenharia de software é fornecer uma estrutura para a construção de software com alta qualidade.

Está correto o que consta em:

A alternativa "A" está correta.

O software tem como um dos principais objetivos agregar valor ao negócio, permitindo a automação de rotinas comumente associadas ao controle administrativo ou em apoio ao processo de decisão. A Engenharia de Software permite o fornecimento dessa estrutura disponibilizando processos, métodos e ferramentas.

2. Prefeitura Municipal de Manaus – 2010 - A Engenharia de Software compreende um conjunto de etapas comumente citadas como paradigmas de Engenharia de Software. No tocante a essas etapas, assinale a opção correta.

A alternativa "A" está correta.

A Engenharia de Software possui quatro camadas: qualidade, processo, métodos e ferramentas. Cabe à camada de processo, a determinação das etapas de desenvolvimento do software. Definido o processo, o engenheiro de software especifica os métodos e as ferramentas que serão utilizadas.

MÓDULO 2

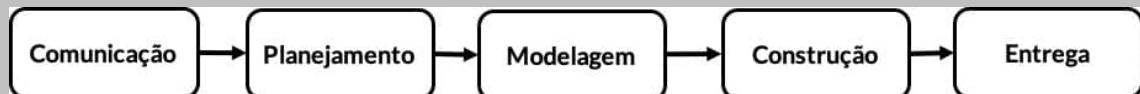
● Descrever as etapas essenciais de um processo de desenvolvimento de software

Você sabe por que se aplica de forma intensa o conceito de abstração no desenvolvimento de software?

Porque o processo de software é iniciado com especificações e modelos com alto nível de abstração e, à medida que o desenvolvimento de software se aproxima da codificação, o nível de abstração diminui, de modo que o código representa o nível mais baixo da abstração ou de maior detalhamento na especificação do software.

Vamos agora diminuir o nível de abstração do Modelo Genérico apresentado no módulo 1 (**figura 3**), ou seja, aumentar os detalhes das atividades do processo de software.

FIGURA 3



Fonte: O autor

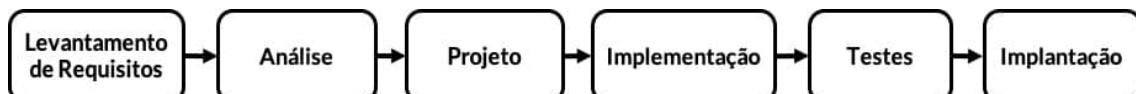
🕒 Figura 3 - Metodologia do Processo.

PROCESSO DE DESENVOLVIMENTO DE SOFTWARE

O engenheiro de software deverá definir qual o processo de desenvolvimento a ser aplicado em um determinado projeto de software como especificação de requisito não funcional.

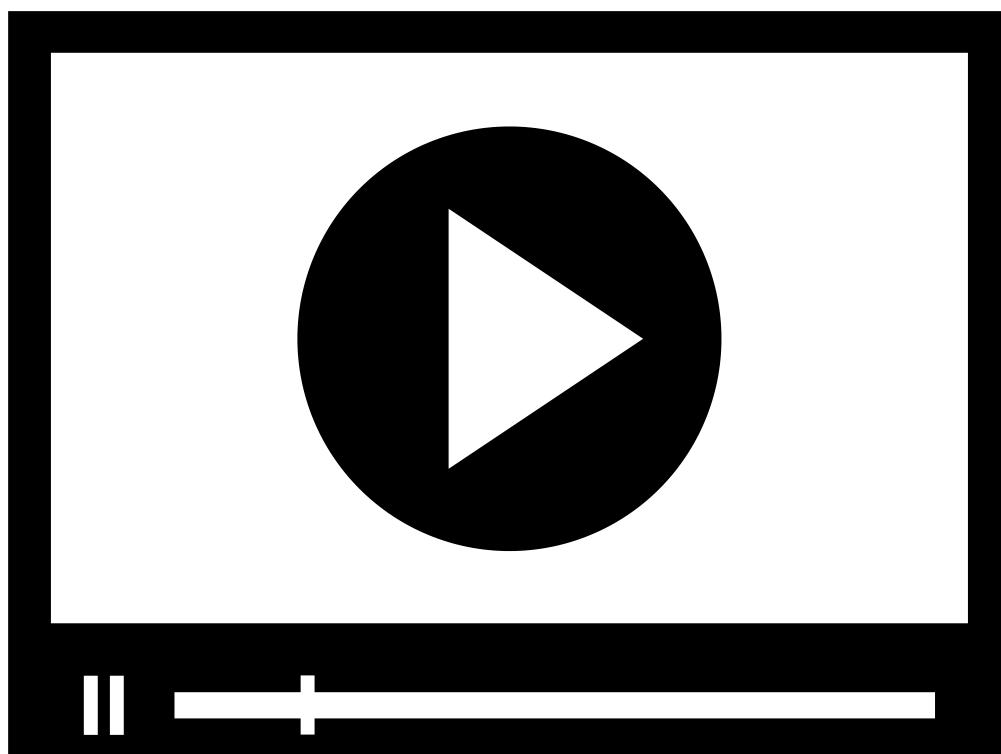
Inicialmente, terá que identificar quais as atividades que irão compor o desejado processo e, em seguida, definir o sequenciamento das referidas atividades, ou seja, o fluxo do processo.

As atividades típicas que compõem o processo de desenvolvimento de software estão ilustradas na figura 4. O objetivo é ilustrar as atividades mais comuns que compõem os processos de desenvolvimento de software, ou seja, qualquer processo deverá possuir as referidas atividades.



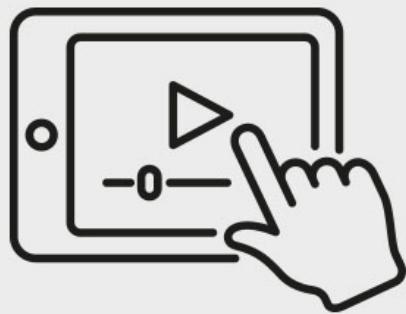
Fonte: O autor

Figura 4 - Atividades típicas de um processo de desenvolvimento de software.



Assista agora ao vídeo sobre **Atividades típicas de um Processo de Desenvolvimento de Software**.

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.

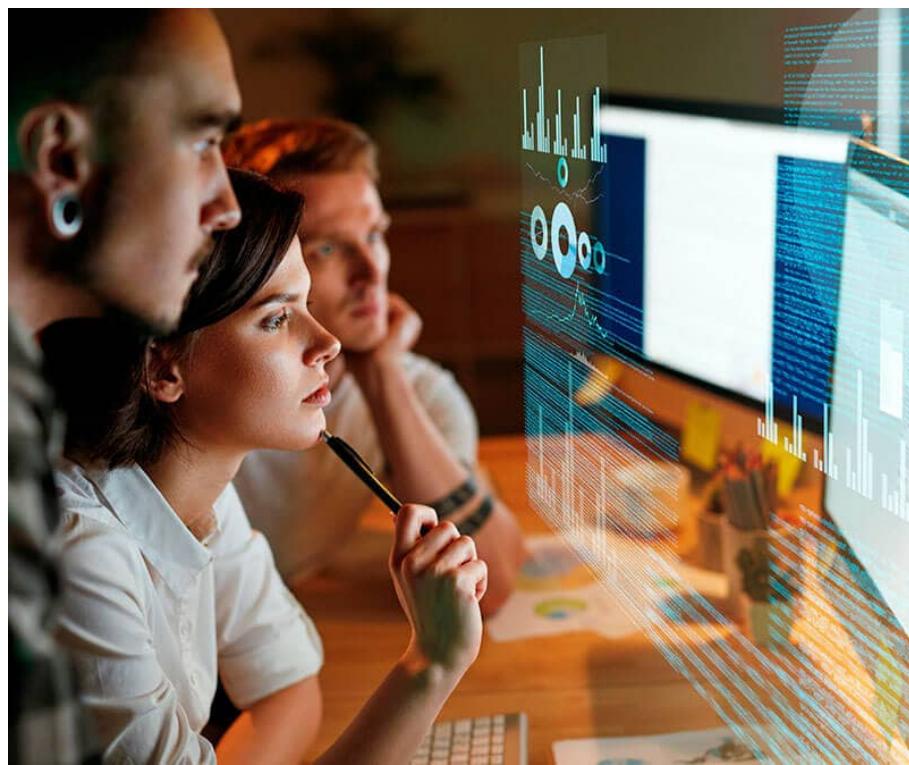


Vamos agora descrever cada uma das atividades comumente previstas em um processo de desenvolvimento de software.

ETAPA DE LEVANTAMENTO DE REQUISITOS

A primeira etapa, também denominada de Elicitação de Requisitos, inclui o primeiro desafio do engenheiro de software, entender o problema!

Imagine você como engenheiro de software, projetando um software para o mercado financeiro! O desafio impressiona? Talvez você concorde comigo – “sim”!



Fonte: puhhha/Shutterstock

Realmente, o desafio é grande, pois o referido engenheiro, normalmente, não entende do ambiente de negócio que o software será implementado. Portanto, terá que se comunicar com diferentes usuários que, muitas vezes, entendem de somente parte do problema; para tal, deverá utilizar diferentes técnicas de levantamento de requisitos, tais como, entrevistas, questionários, leituras de documentos, etnografia e outros.

Nesta etapa, são identificados os requisitos que serão implementados no software projetado. O grande desafio é que o engenheiro de software tenha o mesmo entendimento do negócio que os usuários.

Neste momento, precisamos apresentar a conceituação de requisito:

OS REQUISITOS DE UM SISTEMA SÃO DESCRIÇÕES DOS SERVIÇOS FORNECIDOS PELO SISTEMA E AS SUAS RESTRIÇÕES OPERACIONAIS. ESSES REQUISITOS REFLETEM AS NECESSIDADES DOS CLIENTES DE UM SISTEMA QUE AJUDA A RESOLVER ALGUM PROBLEMA.

SOMMERVILLE, 2007.

A partir dessa conceituação, identificamos a tipificação de requisitos comumente utilizada: requisitos funcionais, não funcionais e de domínio.

REQUISITOS FUNCIONAIS

Estão relacionados com os serviços fornecidos pelo sistema, ou seja, as funcionalidades que estarão disponíveis no software, tal como, a geração de um histórico escolar em um sistema de gestão acadêmico ou geração de uma nota fiscal em um sistema de vendas.

REQUISITOS NÃO FUNCIONAIS

Incluem as restrições operacionais impostas ao software, tais como, o sistema gerenciador de banco de dados, a linguagem de programação, legislação pertinente à *compliance*, entre

outros, bem como os requisitos de qualidade, tais como: confiabilidade, manutenibilidade, usabilidade e outros.

REQUISITOS DE DOMÍNIO

Também são conhecidos como “regras de negócio”, que, normalmente, apresentam-se como restrições ao requisito funcional. Como exemplo, temos o cálculo da média para aprovação em uma determinada disciplina, a contagem de pontuação de multas para computo da perda de uma carteira de motorista ou o cálculo dos impostos quando da geração de uma nota fiscal. O não cumprimento de um requisito de domínio pode comprometer o uso do sistema.

Cabe destacar que Sommerville (2007) apresenta uma classificação relativa aos níveis de abstração aplicados na descrição dos requisitos, utilizando o termo **requisitos de usuários**, para especificação com alto nível de abstração, e **requisitos de sistema**, para especificação com descrições detalhadas, ou seja, com baixo nível de abstração. Realizada essa primeira classificação, os requisitos de sistema são tipificados em funcionais, não funcionais e de domínio, como anteriormente apresentados.

Muitos estudos destacam a importância desta etapa, pois o mau entendimento de um requisito é propagado nas próximas etapas no processo, ilustrado na figura 4, refletindo de forma negativa no produto software.

Quando da especificação de requisitos, o engenheiro de software firma um **contrato** com os usuários, de modo a definir qual **produto** de software será entregue. A participação dos usuários envolvidos, ou clientes, deve permitir feedbacks sobre defeitos na especificação, evitando a propagação dos referidos defeitos.

Nesta etapa do PDS, normalmente, é realizada uma entrega, ou uma especificação, denominada de Documento de Requisitos, sendo determinado o Escopo do projeto.

A partir desse documento, inicia-se a rastreabilidade dos requisitos, ou seja, a relação com os produtos gerados, e.g. modelos, a partir dos mesmos.

★ EXEMPLO

Na etapa de Análise, é gerado o modelo de casos de uso e o modelo de classes; na etapa de Projeto, o modelo de interação; e na etapa de Implementação, a codificação. A rastreabilidade de requisitos garante que as especificações geradas até a codificação estejam de acordo com a documentação de requisitos.

ETAPA DE ANÁLISE

Ainda no contexto da construção de uma casa, quais seriam os requisitos iniciais para que essa construção ocorresse?

Poderíamos imaginar uma descrição especificando que você quer uma casa com uma sala, três quartos, uma suíte, cozinha, piscina, churrasqueira e com aproximadamente 120 m². Textualmente, temos um documento de requisitos.



Fonte: Kanghophoto/Shutterstock

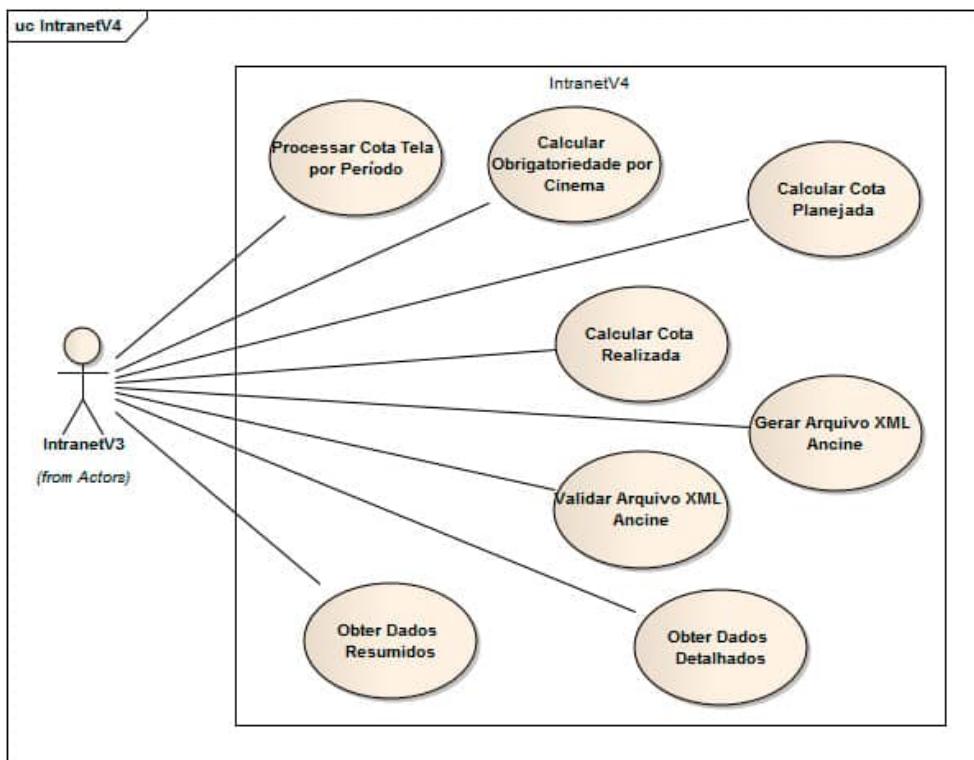
E AGORA, O QUE FAZER COM ESSE DOCUMENTO?

Podemos contratar um arquiteto para desenhar uma planta baixa que atenda aos requisitos descritos. Essa planta é o que chamamos de **modelo**, que representa parte da solução do problema “construir a sua casa”.

A engenharia tem como boa prática a construção de modelos que permitem uma melhor tratativa da complexidade em função de abstrações aplicadas, e.g., o diagrama de casos de uso enfatiza a abstração funcional e o diagrama de sequência, os aspectos dinâmicos do sistema, ou seja, a comunicação entre objetos na realização de um caso de uso. O modelo

permite, também, a comunicação entre as partes interessadas do projeto, pois podemos considerar que um diagrama de caso de uso permite uma comunicação, por exemplo, entre analistas e usuários ou entre analistas e gerentes de projeto. Os modelos permitem, ainda, uma economicidade no projeto, pois uma correção em um modelo de classes na etapa de análise por um erro no entendimento de um determinado requisito evita a propagação desse defeito no código em produção, manutenção essa que seria bem mais custosa. Uma última colocação: os modelos determinam a forma da solução do problema, ou seja, se o diagrama de componentes estabelece três camadas de software, o software implantado deverá refletir a referida especificação.

Na etapa de Análise, as especificações contidas no Documento de Requisitos são convertidas em modelos de análise que incluem artefatos gráficos e textuais. Como exemplo, os requisitos funcionais evoluem para uma especificação gráfica denominada Modelo de Casos de Uso, sendo este composto por diagramas de caso de uso, artefatos gráficos, e descrições de caso de uso, artefatos textuais. Importante destacar que uma descrição de caso de uso permite identificar os diferentes cenários de utilização do referido caso. A figura 5 ilustra um diagrama de casos de uso.



Fonte: O autor

Figura 5 - Exemplo de Diagrama de Casos de Uso.

Nesta etapa, o engenheiro de software aplica um alto nível de abstração de modo a definir “O QUE” o sistema deverá implementar, ou seja, nenhum tipo de restrição tecnológica é

considerado, e.g., como ocorre a comunicação entre os objetos que permitirão implementar o caso de uso “Agendar consulta”.

A entrega da referida etapa inclui os modelos denominados “modelos de análise” com alto nível de abstração.

O entendimento do problema modelado está correto?

Em função desse questionamento, o engenheiro de software necessita VALIDAR seus modelos com os usuários, pois, afinal, são estes que entendem do problema. A validação por parte do usuário garante ao engenheiro que o entendimento da solução do problema está correto e, de acordo com as suas expectativas, i.e., de acordo com os requisitos registrados no documento de requisitos.

Os modelos estão corretos e balanceados?

A verificação é uma atividade técnica do engenheiro de software que permite a verificação da correção de cada modelo e o balanceamento entre esses.

A figura 6 ilustra um diagrama de classes, sendo este um artefato do Modelo de Classes que permite identificar os objetos do domínio do problema que serão utilizados na implementação dos casos de uso. Cabe ao engenheiro de software, verificar a correção do modelo e a consistência com o Modelo de Casos de Uso.



Fonte: Wikipedia

Figura 6 - Exemplo de Diagrama de Classes.

ETAPA DE PROJETO

A fase de Projeto permite os refinamentos dos modelos gerados na análise, bem como a construção de novos modelos gerando especificações com menor nível de abstração e que

permitam definir “COMO” implementar a solução especificada, ou seja, ao final desta etapa, o nível de detalhamento da especificação permite a implementação da solução.

Dentre as principais atividades, destacamos:

Refinamento do modelo de classes iniciado na análise.

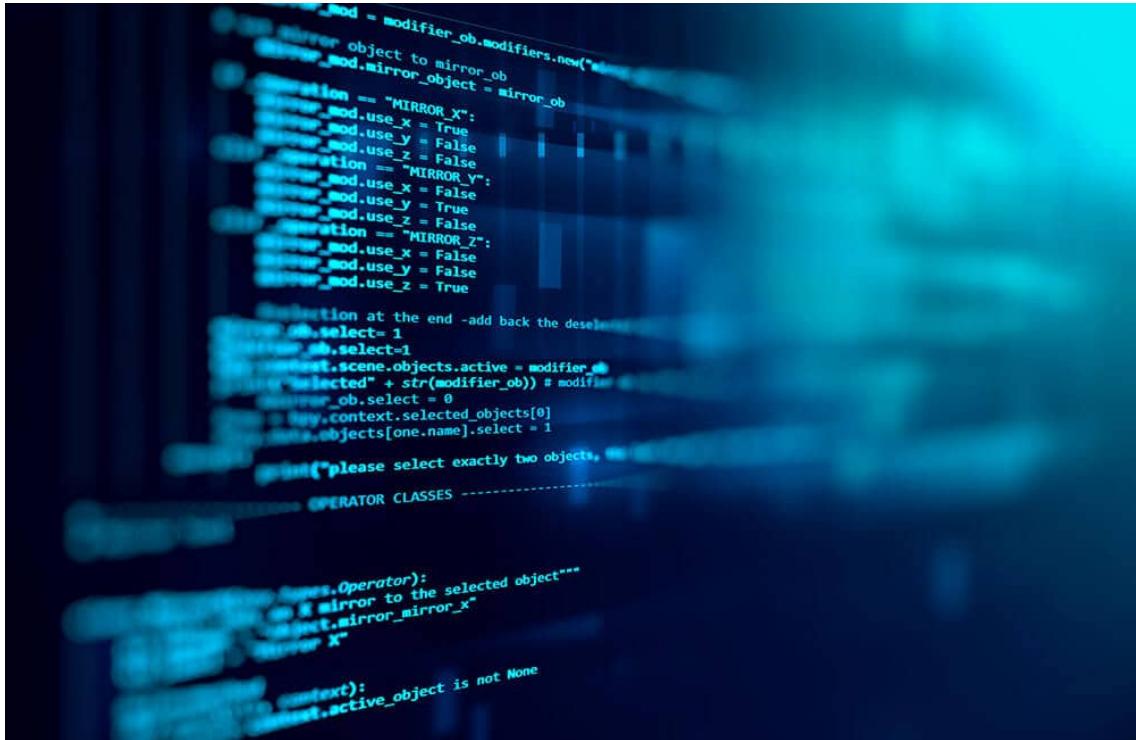
Construção do modelo de interação, que permite identificar a comunicação entre objetos que irão permitir a implementação das funcionalidades especificadas pelos casos de uso.

Mapeamento objeto-relacional, permitindo a geração do modelo lógico do banco de dados, ou seja, identificação das tabelas necessárias ao sistema considerando um requisito não funcional que determine uma tecnologia relacional.

Desenho dos componentes do sistema e dos nós computacionais necessários para a sua implantação, gerando modelos que determinam a arquitetura do sistema.

ETAPA DE IMPLEMENTAÇÃO

Nesta etapa, ocorre a codificação do software de acordo com os requisitos definidos na etapa de Projeto. Os padrões de projeto devem ser considerados por representarem melhores práticas de implementação do software.



Fonte: whiteMocca/Shutterstock

ETAPA DE TESTES

Nesta etapa, são aplicados os denominados testes de software ou testes de validação que permitem verificar se o produto certo está sendo construído.

Os testes validam os componentes individualmente bem como a integração entre eles. Em geral, esta etapa inicia com os testes unitários, em seguida, os testes de integração e os testes de aceitação ou homologação. Ao final dessa sequência, ocorre a migração do sistema para o ambiente de produção.



Fonte: Gorodenkoff/Shutterstock

💡 RECOMENDAÇÃO

Nesta etapa, é fundamental a geração de um Plano de Teste a partir dos casos de testes, que, por sua vez, estão vinculados aos cenários descritos na descrição de cada caso de uso.

Outro aspecto importante é a automação dos testes em função da provável repetição destes em um ciclo iterativo e incremental, onde o software é implementado em versões e, a cada nova versão, os testes anteriores necessitam ser refeitos.

ETAPA DE IMPLANTAÇÃO

Nesta etapa, o software é migrado para o ambiente de produção, de acordo com o aval da equipe de qualidade.

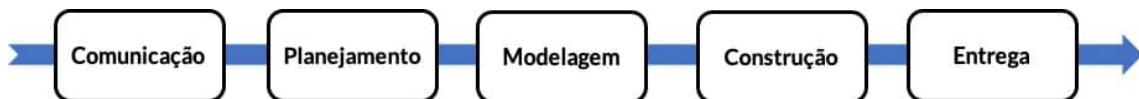
Esta etapa pode exigir a geração de manuais de utilização, a migração de dados do ambiente de homologação para o de produção, treinamento de usuários, ou até mesmo a implantação de um *call center* em caso de sistemas complexos assim o exigirem.

FLUXO DE PROCESSO

A especificação de um processo de desenvolvimento de software, lembrando ser este um requisito não funcional, requer a definição de quais atividades irão compor o respectivo processo e como as referidas atividades serão encadeadas, também denominada de Fluxo de Processo ou Ciclo de Vida.

FLUXO DE PROCESSO LINEAR

A figura 7 ilustra o denominado Fluxo de Processo Linear, onde as atividades são executadas em sequência, de modo que cada atividade é realizada por completo uma única vez.

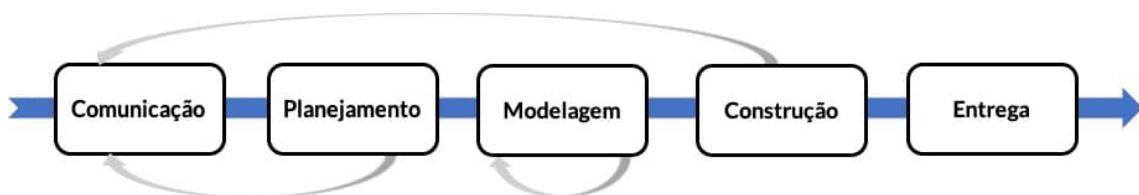


Fonte: O autor

Figura 7 - Fluxo de Processo Linear.

FLUXO DE PROCESSO ITERATIVO

A figura 8 ilustra o denominado Fluxo de Processo Iterativo, onde uma atividade ou um conjunto de atividades podem ser repetidas antes de prosseguir para a seguinte.



Fonte: O autor

Figura 8 - Fluxo de Processo Iterativo.

FLUXO DE PROCESSO EVOLUCIONÁRIO

A figura 9 ilustra o denominado Fluxo de Processo Evolucionário, onde o sequenciamento de cada fluxo inclui todas as atividades, sendo que cada iteração completa gera uma nova versão do software, ou seja, o software agrega valor às suas funcionalidades a cada ciclo completo.

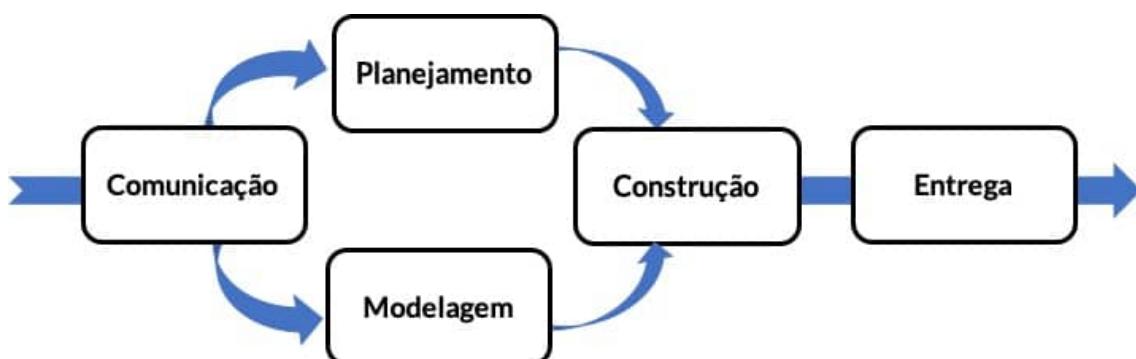


Fonte: O autor

Figura 9 - Fluxo de Processo Evolucionário.

FLUXO DE PROCESSO PARALELO

A figura 10 ilustra o denominado Fluxo de Processo Paralelo, que permite a execução de uma ou mais atividades em paralelo.

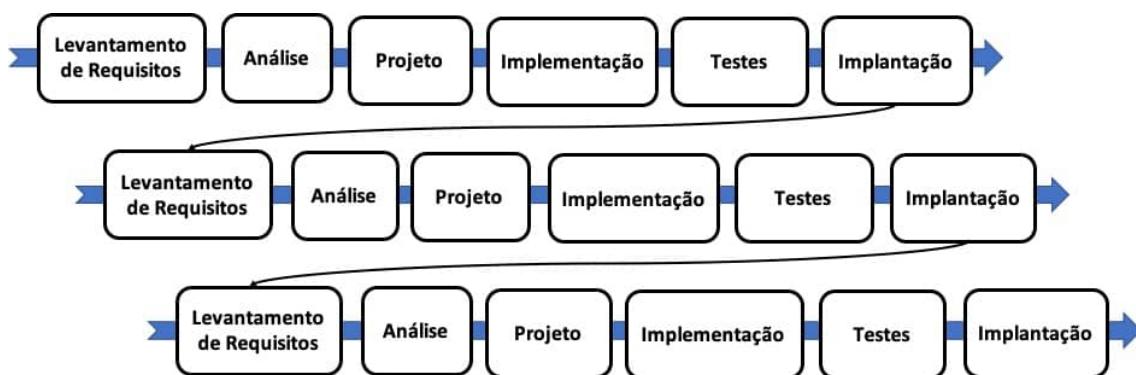


Fonte: O autor

Figura 10 - Fluxo de Processo Paralelo.

MODELO DE CICLO DE VIDA ITERATIVO E INCREMENTAL

A figura 11 ilustra o Modelo de Ciclo de Vida Iterativo e Incremental, onde cada iteração inclui todas as atividades e ocorre o versionamento do produto software gerado. Este modelo corresponde ao Fluxo de Processo Evolucionário. A cada novo ciclo, um subconjunto de requisitos é considerado.



Fonte: O autor

Figura 11 - Modelo de Ciclo de Vida Iterativo e Incremental.

RESUMINDO

Neste módulo, podemos avaliar a importância do processo de desenvolvimento de software. Cabe destacar que processo é a principal camada da Engenharia de Software.

Foram descritas as principais atividades que, genericamente, compõem um processo de desenvolvimento de software, que inclui: levantamento de requisitos, análise, projeto, implementação, testes e implantação.

Cabe ao engenheiro de software determinar as atividades que comporão a especificação de um processo de desenvolvimento de software, sendo o sequenciamento determinado pelo Fluxo de Processo, podendo ser Fluxo de Processo Linear, Fluxo de Processo Iterativo, Fluxo de Processo Evolucionário ou Fluxo de Processo Paralelo.

Lembre-se, não existe engenharia sem processo.

VERIFICANDO O APRENDIZADO

1. O PROCESSO DA ENGENHARIA DE REQUISITOS INCLUI O LEVANTAMENTO DE REQUISITOS, QUE CORRESPONDE À ETAPA DE COMPREENSÃO DO PROBLEMA APLICADA AO DESENVOLVIMENTO DE SOFTWARE, E TEM COMO PRINCIPAL OBJETIVO FAZER COM QUE USUÁRIOS E DESENVOLVEDORES TENHAM A MESMA VISÃO DO PROBLEMA A SER RESOLVIDO.

NESTE CONTEXTO, (1) NA ETAPA LEVANTAMENTO DE REQUISITOS, OS DESENVOLVEDORES, JUNTAMENTE COM OS CLIENTES, TENTAM LEVANTAR E DEFINIR AS NECESSIDADES DOS FUTUROS USUÁRIOS DO SISTEMA A SER DESENVOLVIDO, PROSEGUIMENTO NO PROCESSO, TEMOS (2) A ETAPA DE VALIDAÇÃO DOS REQUISITOS ONDE OS ANALISTAS APRESENTAM OS MODELOS CRIADOS PARA REPRESENTAR O SISTEMA AOS FUTUROS USUÁRIOS PARA QUE ESSES MODELOS SEJAM VALIDADOS.

CONSIDERANDO AS AFIRMATIVAS (1) E (2), ESCOLHA A OPÇÃO CORRETA:

- A)** Ambas as afirmativas são falsas.
- B)** A afirmativa 1 é verdadeira e a afirmativa 2 é falsa.
- C)** Ambas as afirmativas são verdadeiras, mas a (2) não é uma sequência correta de (1).
- D)** A afirmativa 1 é falsa e a afirmativa 2 é verdadeira.

2. UM ENGENHEIRO DE SOFTWARE ESTÁ IDENTIFICANDO OS REQUISITOS NÃO FUNCIONAIS PARA UM NOVO PROJETO DE SOFTWARE COM ELEVADO GRAU DE COMPLEXIDADE EM FUNÇÃO DOS REQUISITOS FUNCIONAIS LEVANTADOS ATÉ O MOMENTO. NESTE PONTO, O REFERIDO ENGENHEIRO IDENTIFICOU AS TAREFAS DO PROCESSO DE SOFTWARE ADOTADO E NECESSITA DEFINIR O ENCADEAMENTO DAS TAREFAS, OU SEJA, O FLUXO DE PROCESSO. NESSE CASO, ASSINALE QUAL A OPÇÃO MAIS ADEQUADA:

- A)** Fluxo de Processo Linear.
- B)** Fluxo de Processo Paralelo.
- C)** Fluxo de Processo Iterativo.
- D)** Fluxo de Processo Evolucionário.

GABARITO

1. O processo da Engenharia de Requisitos inclui o levantamento de requisitos, que corresponde à etapa de compreensão do problema aplicada ao desenvolvimento de software, e tem como principal objetivo fazer com que usuários e desenvolvedores tenham a mesma visão do problema a ser resolvido.

Neste contexto, (1) na etapa levantamento de requisitos, os desenvolvedores, juntamente com os clientes, tentam levantar e definir as necessidades dos futuros usuários do sistema a ser desenvolvido, prosseguimento no processo, temos (2) a etapa

de validação dos requisitos onde os analistas apresentam os modelos criados para representar o sistema aos futuros usuários para que esses modelos sejam validados.

Considerando as afirmativas (1) e (2), escolha a opção correta:

A alternativa "C" está correta.

Após a realização da etapa de levantamento dos requisitos, temos a etapa de análise, onde o engenheiro de software gera os modelos, tal como, o modelo de casos de uso. Após essa etapa, ocorre a validação dos requisitos pelos usuários do sistema.

2. Um engenheiro de software está identificando os requisitos não funcionais para um novo projeto de software com elevado grau de complexidade em função dos requisitos funcionais levantados até o momento. Neste ponto, o referido engenheiro identificou as tarefas do processo de software adotado e necessita definir o encadeamento das tarefas, ou seja, o fluxo de processo. Nesse caso, assinale qual a opção mais adequada:

A alternativa "D" está correta.

O Fluxo de Processo Evolucionário permite o versionamento do software e o melhor trato da complexidade. A cada nova iteração, todas as tarefas são executadas, permitindo a solução de um subconjunto de requisitos, e uma nova versão do software é gerada.

MÓDULO 3

○ Relacionar as etapas de um processo de desenvolvimento de software com as etapas de Gerenciamento de Projeto

GERENCIAMENTO DE PROJETO

Agora que você tem uma boa compreensão das atividades genéricas que compõem um processo e as possibilidades de fluxos de processos, ou seja, os encadeamentos possíveis das

referidas atividades, temos agora que compreender como aplicar de forma sistemática uma metodologia que lhe permita, a partir de uma necessidade de um determinado usuário, gerar o produto software desejado.

A sigla **PMI** lhe diz algo? O PMI, Project Management Institute, é uma organização que visa disseminar as melhores práticas de Gerenciamento de Projetos em todo o mundo e que tem como premissa o fato de ser esse gerenciamento uma **profissão**.

DICA

A certificação de Profissional de Gerenciamento de Projetos (PMP)® do PMI é a certificação para gerentes de projeto reconhecida como uma das mais importantes para a indústria, incluindo a de software. No Brasil, podemos afirmar que é a mais importante.

Qual seria a importância de uma certificação para você?

A certificação complementa a formação acadêmica tornando o profissional de TI mais atualizado e com maior empregabilidade, tanto na busca de um novo posto de trabalho como de uma promoção.

Retornando ao nosso tema PMI, este publica um conjunto de melhores práticas denominado de Guia de Conhecimento em Gerenciamento de Projetos – PMBOK, ou *Project Management Body of Knowledge*, em inglês. O Guia PMBOK é uma base sobre a qual podemos criar uma metodologia para obtenção do nosso produto, i.e., o software.

Nesse contexto, temos que apresentar um primeiro conceito:

**UM PROJETO É UM ESFORÇO TEMPORÁRIO
EMPREENDIDO PARA CRIAR UM PRODUTO, SERVIÇO
OU RESULTADO ÚNICO.**

(PMI, 2017)

Podemos inferir do conceito que todo projeto tem início, meio e fim, por isso é temporário, sendo o mesmo fator impulsionador de mudança nas organizações.

Agora, podemos apresentar um segundo conceito que cabe destaque:



Fonte: Wikipedia

Lembra que a Engenharia de Software está baseada em processos?

A atividade de gerenciamento de projetos tem a mesma base.

Vamos entender a seguir como funciona.

CICLO DE VIDA DO PROJETO X GRUPOS DE PROCESSOS

O ciclo de vida do projeto inclui as fases genéricas ilustradas na figura 12 e que fazem parte de todo projeto.

Início do Projeto	Organização e Preparação	Execução do trabalho	Término do Projeto
-------------------	--------------------------	----------------------	--------------------

Fonte: O autor.

Figura 12 – Ciclo de vida do projeto.

Qual a relação do ciclo de vida do projeto com o fluxo de processo do software?

Os fluxos de processos ou ciclos de vida que apresentamos anteriormente incluem os tipos: linear, iterativo, evolucionário e paralelo. A seleção de um determinado Fluxo de Processo determina as fases pelas quais um projeto irá passar, do início ao término.

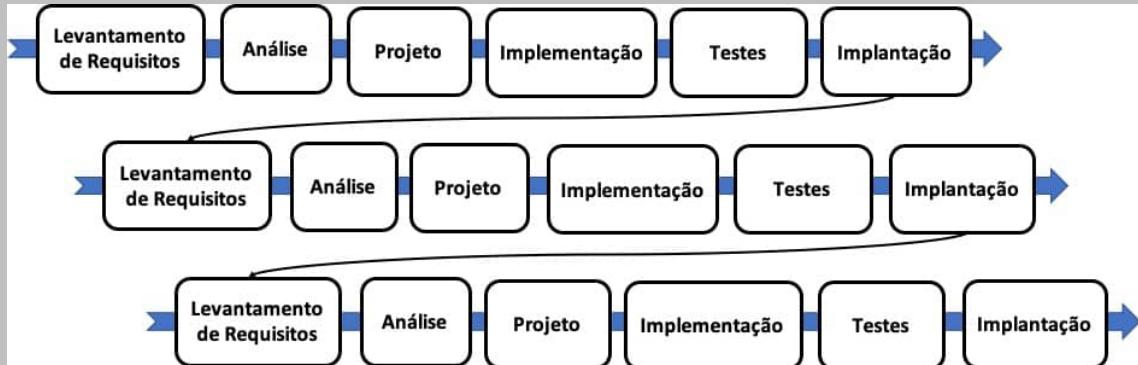
Exemplificando: o engenheiro de software seleciona o Fluxo de Processo Evolucionário, ou Iterativo e Incremental, representado na **figura 11**. A partir dessa especificação, o ciclo de vida do projeto está estabelecido e as entregas do projeto ocorrerão por meio de uma série de iterações.

Como esse ciclo de vida do projeto estabelecido é gerenciado?

Por meio da execução de uma série de atividades de gerenciamento de projeto, conhecidas como processos de gerenciamento de projetos.

Cada processo de gerenciamento de projetos transforma entradas em saídas através da aplicação de técnicas e ferramentas de gerenciamento de projetos apropriadas (figura 13). As saídas podem ser uma entrega ou um resultado.

FIGURA 11



Fonte: O autor

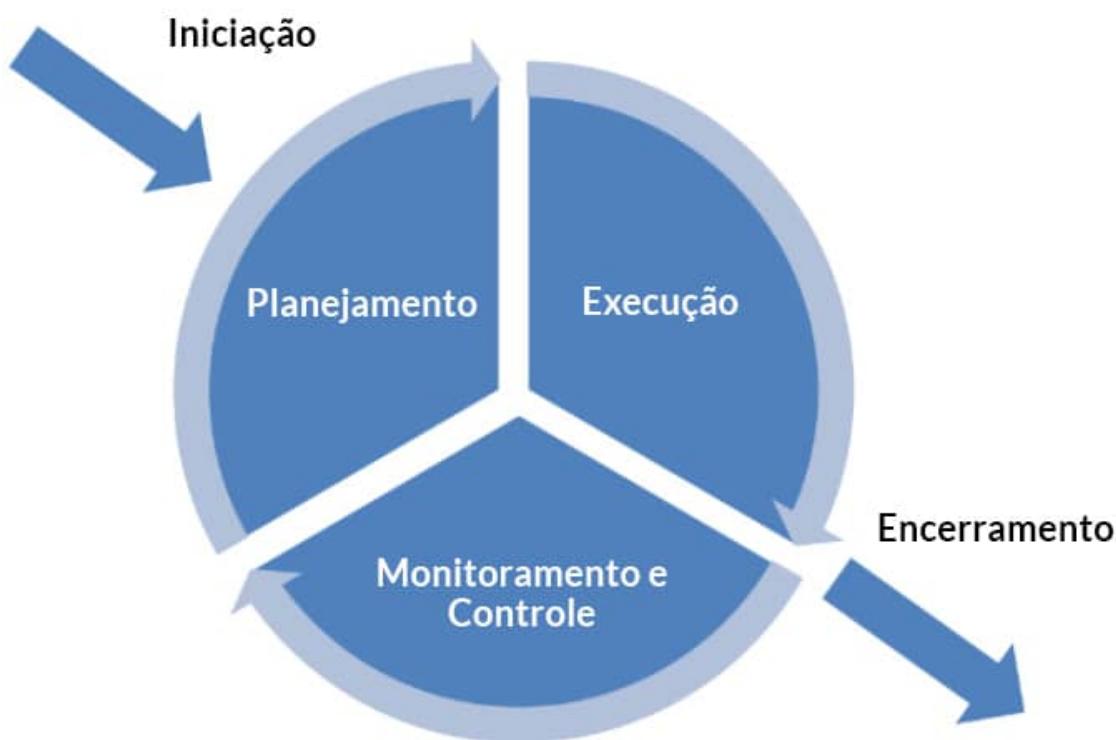
Figura 11 - Modelo de Ciclo de Vida Iterativo e Incremental.



Fonte: O autor

▣ Figura 13 – Processo de Gerenciamento de Projeto.

A figura 14 ilustra as cinco etapas que permitem a “GESTÃO” de um projeto: Iniciação; Planejamento; Execução; Monitoramento e Controle; e Encerramento.



Fonte: Shutterstock

▣ Figura 14 – Grupos de Processos.

Temos que considerar que cada etapa possui um conjunto de processos, conjunto este denominado de **Grupo de Processos de Gerenciamento de Projetos**.

De uma forma simplificada, o gerente de projetos tem que considerar as etapas de gestão, por exemplo, para a etapa INICIAÇÃO:

Seleciona os processos que serão utilizados, considerando a complexidade do projeto.

Aloca responsáveis por cada processo de acordo com a área de conhecimento.

Os responsáveis realizam os processos de acordo com o que preconiza o PMBOK.

Os responsáveis geram seus respectivos resultados para cada processo.

Em seguida, aplica os passos de 1 a 4 anteriores para cada etapa do ciclo de vida do projeto: Planejamento, Execução, Monitoramento e Controle e Encerramento. Em projetos complexos, as etapas podem ocorrer de forma iterativa.

Os cinco grupos de processos ilustrados na figura 14 serão descritos a seguir:

GRUPO DE PROCESSOS DE INICIAÇÃO

Grupo que inclui os processos realizados no início de um projeto, cabendo destaque ao Termo de Abertura do Projeto, termo que autoriza a alocação de recursos ao projeto.

GRUPO DE PROCESSOS DE PLANEJAMENTO

Grupo que inclui os processos realizados para planejar a execução de um novo projeto. A primeira atividade de planejamento é o gerenciamento do escopo do projeto.

A principal entrega do planejamento é o cronograma. O cronograma físico inclui as datas de início e término de todas as atividades do projeto.

GRUPO DE PROCESSOS DE EXECUÇÃO

Aqui está um dos grandes desafios do gerente de projeto, que é executar o projeto de acordo com o planejado. Lembre-se de que o gerente de projetos não está sozinho, sendo um integrador de pessoas e distintos conhecimentos.

GRUPO DE PROCESSOS DE MONITORAMENTO E CONTROLE

Nesse grupo, podemos aplicar a máxima de que “não se controla o que não se mede”.

Grupo que inclui os processos que permitem monitorar e controlar o progresso e desempenho do projeto. Caso alguma atividade não ocorra de acordo com o planejado, terá que ser avaliada uma ação corretiva a fim de que essa não conformidade seja tratada por meio de um gerenciamento de mudanças.

GRUPO DE PROCESSOS DE ENCERRAMENTO

Lembre-se, todo projeto é temporário, portanto, em algum momento, terá que ser, formalmente, encerrado.

Grupo que inclui os processos que permitirão a conclusão ou encerramento do projeto, tais como, a entrega e aceitação do produto, a dispensa da equipe de projeto ou avaliação geral dos resultados.

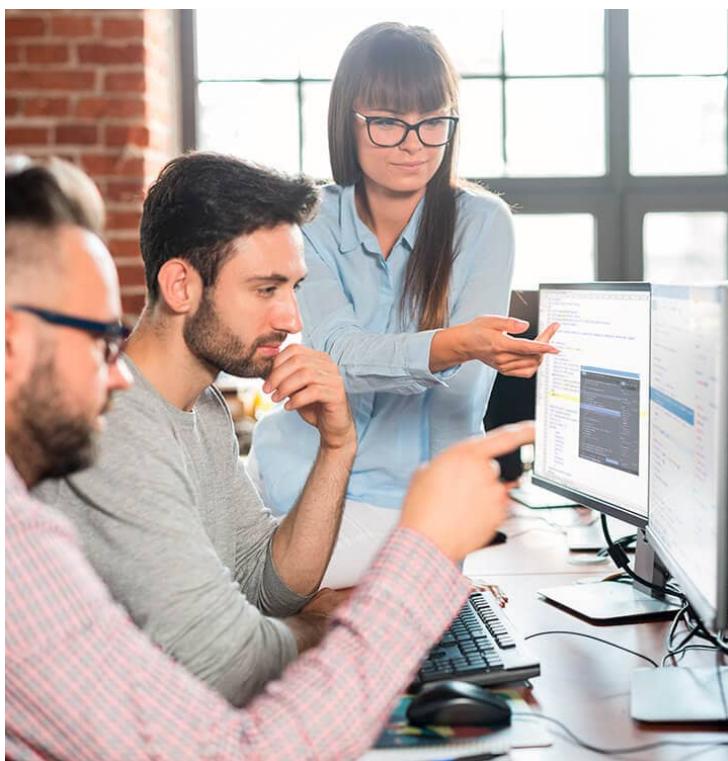
ÁREAS DE CONHECIMENTO EM GERENCIAMENTO DE PROJETOS

Assim como a complexidade é um fator determinante na definição de um processo de software, também é inerente ao Gerenciamento de Projeto, de modo que, quanto maior o problema, mais multidisciplinar este se torna.

Em função dessa complexidade, qual seria a sua equipe técnica para desenvolver um projeto de software?

Provavelmente, você pensaria em engenheiro de software, analistas, administrador de banco de dados, programadores, arquitetos de software, entre outros.

Perfeito, essa seria uma boa equipe!



Fonte: REDPIXEL.PL/Shutterstock

Mas essa equipe seria capaz de realizar a “GESTÃO” de um projeto de software? Podemos considerar que uma equipe de Gerenciamento de Projeto deve ser composta por especialistas

em custos que realizem orçamentos realistas; especialistas em aquisições que tenham capacitações na realização de contratações ou licitações; especialistas em recursos humanos, a fim de gerenciar os direitos trabalhistas da equipe de projeto; e outros.

É possível que essa equipe técnica resolva parte do problema em função da complexidade na área de gestão que um projeto de software possa requerer.

Considerando que o gerenciamento de projetos é uma atividade multidisciplinar, o PMBOK realiza a fatoração dessa complexidade em 10 (dez) áreas de conhecimento, onde cada área é composta por um conjunto de processos, como ilustra a **figura 15**.

Você poderia perguntar como gerente de projeto: qual a minha equipe então?

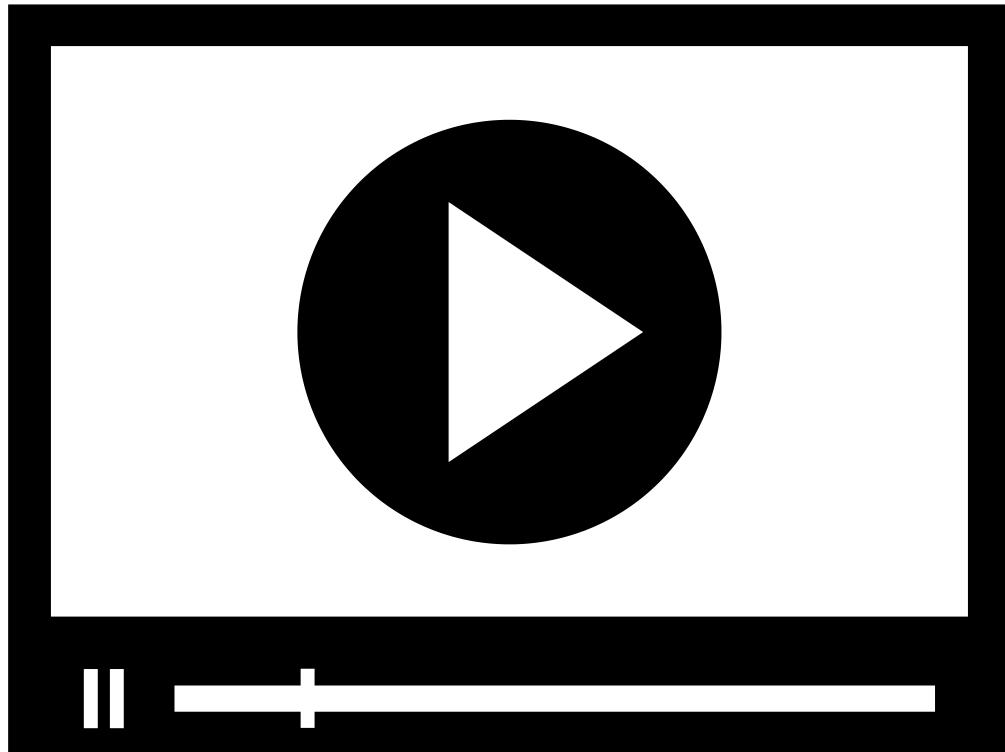
Respondendo ao questionamento, o perfil de sua equipe deverá estar alinhado às áreas de conhecimento que serão descritas a seguir.

FIGURA 15

Áreas de Conhecimento	Iniciação	Planejamento	Execução	Monitoramento e controle	Encerramento
Integração	1. Desenvolver o termo de abertura do projeto	2. Desenvolver o plano de gerenciamento do projeto	3. Orientar e gerenciar a execução do projeto	4. Monitorar e controlar o trabalho do projeto 5. Realizar o controle integrado de mudanças	6. Encerrar o projeto ou fase1
Escopo		1. Coletar os requisitos 2. Definir o escopo 3. Criar a EAP		4. Verificar o escopo 5. Controlar o escopo	
Tempo		1. Definir as atividades 2. Sequenciar as atividades 3. Estimar os recursos das atividades 4. Estimar as durações das atividades 5. Desenvolver o cronograma		6. Controlar o cronograma	
Custos		1. Estimar os custos 2. Determinar o orçamento		3. Controlar os custos	
Qualidade		1. Planejar a qualidade	2. Realizar a garantia de qualidade	3. Realizar o controle da qualidade	
Recursos Humanos		1. Desenvolver o plano de recursos humanos	2. Mobilizar a equipe do projeto 3. Desenvolver a equipe de projeto 4. Gerenciar a equipe do projeto		
Comunicação	1. Identificar as partes interessadas	2. Planejar as comunicações	3. Distribuir as informações 4. Gerenciar as expectativas das partes interessadas	5. Reportar o desempenho	
Riscos		1. Planejar o gerenciamento dos riscos 2. Identificar os riscos 3. Realizar a análise qualitativa dos riscos 4. Realizar a análise quantitativa dos riscos 5. Planejar as respostas aos riscos		6. Monitorar e controlar os riscos	
Aquisição		1. Planejar as aquisições	2. Conduzir as aquisições	3. Administrar as aquisições	4. Encerrar as aquisições

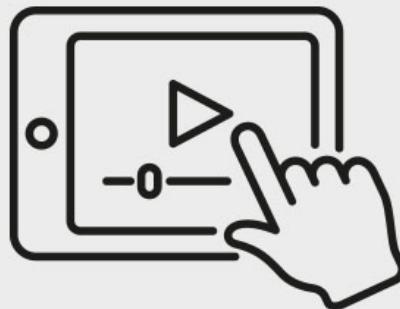
Fonte: O autor

📷 Figura 15 – Áreas de conhecimento x Grupos de processos.



Conheça no vídeo a seguir as **Áreas de Conhecimento e Grupos de Processos do PMBOK aplicados ao Desenvolvimento de Software**.

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.



A figura 15 lista as áreas de conhecimento e o seu relacionamento com os grupos de processos.

Áreas de Conhecimento	Iniciação	Planejamento	Execução	Monitoramento e controle	Encerramento
Integração	1. Desenvolver o termo de abertura do projeto	2. Desenvolver o plano de gerenciamento do projeto	3. Orientar e gerenciar a execução do projeto	4. Monitorar e controlar o trabalho do projeto 5. Realizar o controle integrado de mudanças	6. Encerrar o projeto ou fase1
Escopo		1. Coletar os requisitos 2. Definir o escopo 3. Criar a EAP		4. Verificar o escopo 5. Controlar o escopo	
Tempo		1. Definir as atividades 2. Sequenciar as atividades 3. Estimar os recursos das atividades 4. Estimar as durações das atividades 5. Desenvolver o cronograma		6. Controlar o cronograma	
Custos		1. Estimar os custos 2. Determinar o orçamento		3. Controlar os custos	
Qualidade		1. Planejar a qualidade	2. Realizar a garantia de qualidade	3. Realizar o controle da qualidade	
Recursos Humanos		1. Desenvolver o plano de recursos humanos	2. Mobilizar a equipe do projeto 3. Desenvolver a equipe do projeto 4. Gerenciar a equipe do projeto		
Comunicação	1. Identificar as partes interessadas	2. Planejar as comunicações	3. Distribuir as informações 4. Gerenciar as expectativas das partes interessadas	5. Reportar o desempenho	
Riscos		1. Planejar o gerenciamento dos riscos 2. Identificar os riscos 3. Realizar a análise qualitativa dos riscos 4. Realizar a análise quantitativa dos riscos 5. Planejar as respostas aos riscos		6. Monitorar e controlar os riscos	
Aquisição		1. Planejar as aquisições	2. Conduzir as aquisições	3. Administrar as aquisições	4. Encerrar as aquisições

Fonte: Wikipedia

Figura 15 – Áreas de conhecimento x Grupos de processos.

GERENCIAMENTO DA INTEGRAÇÃO DO PROJETO

Área que inclui os processos que permitem a integração das diferentes áreas de conhecimento, estando essa área sob controle direto do gerente de projetos.

Podemos dizer que o gerente de projeto atua como um maestro, pois não precisa ter conhecimento específico de cada área de conhecimento. Este combina os resultados em todas as outras áreas de conhecimento e tem a visão geral do projeto, pois é o único responsável pelo projeto como um todo.

Observe na **figura 15** que a única área com processos em todos os grupos de processos é a Integração. Isso significa que o gerente de projetos está presente em todas as etapas de gestão.

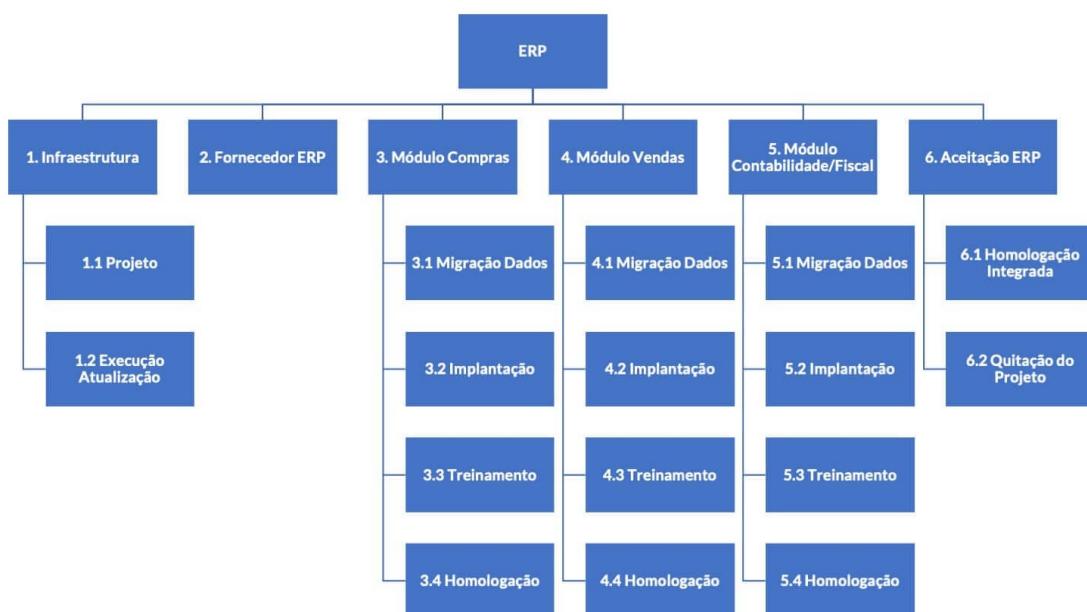
Cabe destacar, mais uma vez, que o Termo de Abertura do Projeto é o documento que autoriza a alocação de recursos ao projeto.

GERENCIAMENTO DO ESCOPO DO PROJETO

Área que inclui os processos necessários para assegurar que o projeto conteplete todo e somente o trabalho necessário para que este termine com sucesso. Nesta área, são especificados e detalhados os requisitos de software.

A principal técnica para a definição do escopo é a confecção da Estrutura Analítica do Projeto (EAP) ou *Work Breakdown Structure* (WBS).

A figura 16 ilustra um exemplo de EAP, onde cada elemento constitui uma entrega e as entregas que não são decompostas são chamadas de pacotes de trabalho ou *workpackages*. Os pacotes de trabalho são conjuntos de tarefas ou ações que, efetivamente, serão executadas no projeto.



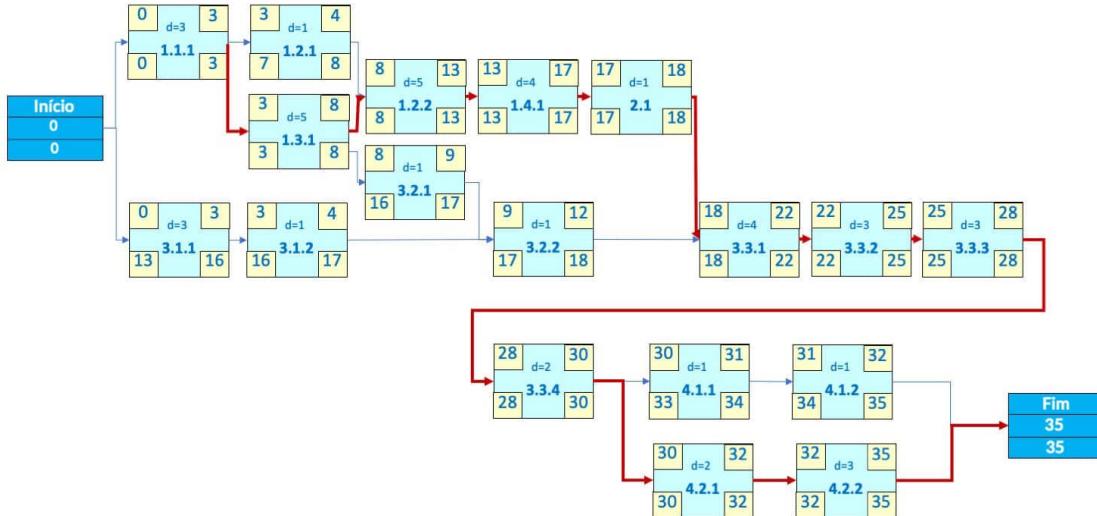
Fonte: O autor

Figura 16 – Exemplo de EAP.

GERENCIAMENTO DO CRONOGRAMA DO PROJETO

Área que inclui os processos necessários para gerenciar o término pontual do projeto, tendo como principal entrega no planejamento o cronograma (físico) do projeto.

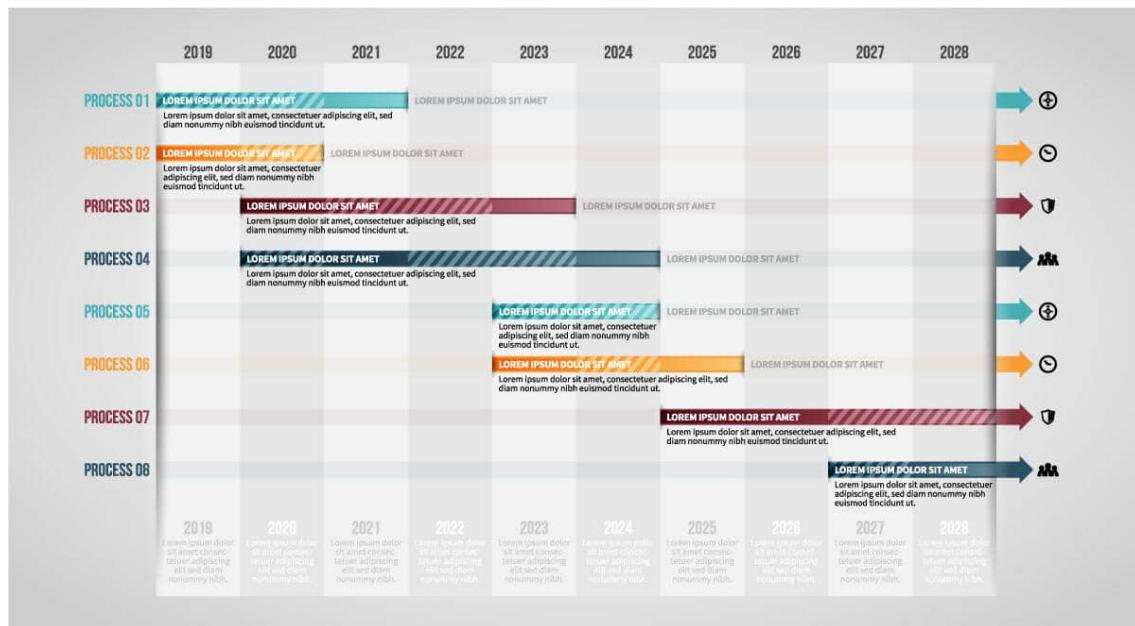
O cronograma do projeto é gerado na etapa de planejamento do projeto. Inicialmente, são identificadas as atividades, a partir dos pacotes de trabalho da EAP e, para cada atividade, devemos especificar a duração e os respectivos insumos. Em seguida, é elaborado o diagrama de rede do cronograma do projeto, que determina as interdependências entre as atividades, tal como ilustrado na figura 17.



Fonte: O autor

Figura 17 – Diagrama de Rede do Cronograma do Projeto.

Finalmente, podemos gerar o cronograma do projeto, exemplificado na figura 18.



Fonte: Hermin/Shutterstock

Figura 18 – Exemplo de Cronograma Detalhado do Projeto.

GERENCIAMENTO DOS CUSTOS DO PROJETO

Inclui os processos envolvidos na gestão dos orçamentos do projeto, a fim de permitir que o projeto possa ser encerrado dentro do orçamento aprovado.

GERENCIAMENTO DA QUALIDADE DO PROJETO

Área que inclui os processos que garantem o atendimento aos requisitos do projeto e, como consequência, as expectativas dos financiadores do projeto.

GERENCIAMENTO DOS RECURSOS DO PROJETO

Área que inclui os processos para identificar, adquirir e gerenciar os recursos necessários para a conclusão bem-sucedida do projeto.

GERENCIAMENTO DAS COMUNICAÇÕES DO PROJETO

Área que inclui os processos necessários para assegurar que as informações do projeto tenham uma gestão oportuna e adequada.

GERENCIAMENTO DOS RISCOS DO PROJETO

Área que inclui os processos de gestão dos riscos em um projeto.

GERENCIAMENTO DAS AQUISIÇÕES DO PROJETO

Área que inclui os processos necessários para comprar ou adquirir produtos, serviços ou resultados externos à equipe do projeto.

GERENCIAMENTO DAS PARTES INTERESSADAS DO PROJETO

Área que inclui os processos exigidos para identificar partes interessadas que possam gerar impacto no projeto, bem como desenvolver estratégias para o seu engajamento eficaz nas decisões e execução do projeto.

RESUMINDO

Neste módulo, podemos avaliar a importância do relacionamento entre o fluxo de processo de software com o ciclo de vida de Gerenciamento do Projeto.

Ambos os ciclos têm como base o conceito **processo**, pois, como vimos, processo é a principal camada da Engenharia de Software e o gerenciamento de projeto é orientado a projeto em função dos grupos de processos. Importante destacar que um processo no Gerenciamento de Projetos tem duas dimensões: a primeira, como parte de um grupo de processos; e a segunda, como parte de uma das 10 (dez) áreas de conhecimento.

As áreas de conhecimento permitem a tratativa da complexidade de projetos multidisciplinares, pois a equipe de gerenciamento de projetos deverá ser composta por especialistas nas diversas áreas de conhecimento.

VERIFICANDO O APRENDIZADO

1. O GERENTE DE PROJETO DE UM DETERMINADO PROJETO DE SOFTWARE DEFINIU O PROCESSO DE DESENVOLVIMENTO COM AS ATIVIDADES COMUMENTE UTILIZADAS, TAIS COMO, LEVANTAMENTO DE REQUISITOS, ANÁLISE, PROJETO ETC. O FLUXO DE PROCESSOS ADOTADO FOI O EVOLUCIONÁRIO POR PERMITIR O VERSIONAMENTO DO SOFTWARE. A EQUIPE DO PROJETO, NO MOMENTO, ESTÁ DEFININDO A DURAÇÃO DE CADA ATIVIDADE DO PROCESSO DE DESENVOLVIMENTO DE SOFTWARE E AS RESPECTIVAS DEPENDÊNCIAS. QUAL O GRUPO DE PROCESSOS DO PMBOK E A ÁREA DE CONHECIMENTO DO PROJETO ENCONTRA-SE A EQUIPE DE PROJETO?

- A)** Grupo de processos planejamento e área de conhecimento gerenciamento do cronograma.
- B)** Grupo de processos execução e área de conhecimento gerenciamento da integração.
- C)** Grupo de processos execução e área de conhecimento gerenciamento do cronograma.
- D)** Grupo de processos planejamento e área de conhecimento gerenciamento do escopo.

2. O ENGENHEIRO DE SOFTWARE NECESSITA DEFINIR O ESCOPO DO PROJETO DE UM DETERMINADO SOFTWARE E DECIDIU UTILIZAR O

PROCESSO QUE PERMITE A CRIAÇÃO DA ESTRUTURA ANALÍTICA DO PROJETO (EAP). ASSINALE A AFIRMATIVA CORRETA RELATIVA À EAP:

- A)** As entregas que sofrem decomposição na EAP são chamadas de pacotes de trabalho.
- B)** A EAP é elaborada no grupo de processo iniciação.
- C)** A área de conhecimento do processo “Criar a EAP” é gerenciamento do cronograma.
- D)** Após a criação da EAP, o engenheiro de software poderá iniciar os processos que permitem o estabelecimento do cronograma do projeto.

GABARITO

1. O gerente de projeto de um determinado projeto de software definiu o processo de desenvolvimento com as atividades comumente utilizadas, tais como, levantamento de requisitos, análise, projeto etc. O fluxo de processos adotado foi o evolucionário por permitir o versionamento do software. A equipe do projeto, no momento, está definindo a duração de cada atividade do processo de desenvolvimento de software e as respectivas dependências. Qual o grupo de processos do PMBOK e a área de conhecimento do projeto encontra-se a equipe de projeto?

A alternativa "A" está correta.

A área de gerenciamento do escopo permite criar a EAP. Na sequência, ainda no grupo de processos planejamento, podemos iniciar os processos da área de conhecimento gerenciamento do cronograma que inclui a identificação das atividades e suas dependências.

2. O engenheiro de software necessita definir o escopo do projeto de um determinado software e decidiu utilizar o processo que permite a criação da Estrutura Analítica do Projeto (EAP). Assinale a afirmativa correta relativa à EAP:

A alternativa "D" está correta.

A Estrutura Analítica de Projeto, ou simplesmente EAP, permite a representação gráfica do escopo do projeto, sendo confeccionada no grupo de processos planejamento e pertence à área de conhecimento gerenciamento de escopo. A criação da EAP permite, em continuidade à

etapa de planejamento, a execução de processo que conduzam ao desenvolvimento do cronograma.

MÓDULO 4

● Descrever a importância do Gerenciamento de Risco no projeto de software

RISCO

As melhores práticas desenvolvidas neste módulo têm por fundamento o PMBOK, cujos processos estão definidos na **figura 15**, vista no módulo 3.



Fonte: kitzcorner/Shutterstock

Imagine você realizando um planejamento de uma viagem ao exterior nas próximas férias. Considerando que o roteiro está definido, bem como as reservas dos hotéis realizadas, as passagens reservadas e uma quantidade de dólares adquiridos, podemos, intuitivamente, aplicar uma abstração denominada **risco**, ou seja, o que pode ocorrer de errado na minha viagem?

Vamos, então, gerar uma pequena lista de possíveis eventos negativos para entendermos os conceitos:

Ocorrência de um problema de saúde, por causa de um acidente ou ter contraído alguma doença.

Cancelamento da passagem pela companhia aérea.

A quantidade de dólar adquirida não ser suficiente.

Observe que a avaliação de risco gera uma preocupação que o conduz ao futuro, inserindo um grau de incerteza – ele pode ou não ocorrer. Ao mesmo tempo, espera-se que fique clara a importância de sua avaliação, pois, caso ocorra algum dos eventos listados durante a sua viagem, essa pode estar seriamente comprometida.

**UM RISCO É UM EVENTO OU CONDIÇÃO INCERTA
QUE, SE OCORRER, PROVOCARÁ UM EFEITO
POSITIVO OU NEGATIVO EM UM OU MAIS OBJETIVOS
DO PROJETO.**

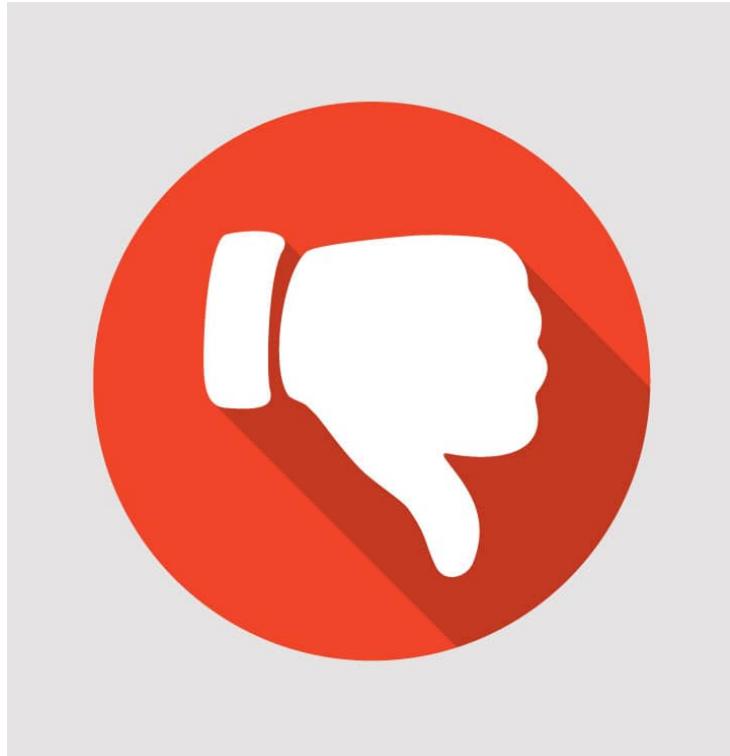
(PMI, 2017)

Você pode agora questionar: efeito positivo?

O risco não trata de efeitos negativos ou coisas que podem dar errado? Realmente, o mais comum é associar risco a algo negativo, mas, conceitualmente, pode ser algo positivo. Vamos exemplificar.

Imagine uma empresa exportadora de minério que elabora o seu plano de investimento para o próximo ano e que identificou o seguinte risco: variação cambial do dólar, sendo essa a moeda padrão da comercialização de commodities. O planejamento inclui aplicar os recursos de investimentos em novos projetos de prospecção de minas.

Neste momento, faz-se necessário realizar uma projeção do valor do câmbio para o próximo ano, pois, como apresentado, o faturamento da empresa está atrelado à referida moeda – vamos imaginar que o valor projetado para o dólar foi de R\$ 5,50 (reais). Temos então dois cenários possíveis:



Fonte: Porcupen/Shutterstock

CENÁRIO NEGATIVO

O dólar chega a valer R\$ 4,00, ou seja, a empresa terá que reduzir o seu portfólio de projetos em função da redução do capital de investimento, pois a empresa irá faturar menos com a mesma quantidade de produtos exportados. Esse cenário é um **obstáculo** para a consecução dos resultados esperados.



Fonte: Porcupen/Shutterstock

CENÁRIO POSITIVO

O dólar chega a valer R\$ 7,00, ou seja, a empresa poderá incrementar o seu portfólio de projetos em função do aumento de capital de investimento, pois a empresa irá faturar mais com mesma quantidade de produtos exportados. Esse cenário é uma **oportunidade** para a consecução dos resultados esperados.

► ATENÇÃO

Lembrando que nosso contexto é a Engenharia de Software, a melhor prática é focar nos cenários negativos.

Como citado anteriormente, o risco permite planejar o futuro e evitar que empresas tenham “sustos” nos seus projetos em função de eventos inesperados e, a partir destes, iniciar as devidas tratativas. Em síntese, a área de Gerenciamento de Risco permite a sistematização dessas tratativas em forma de plano.

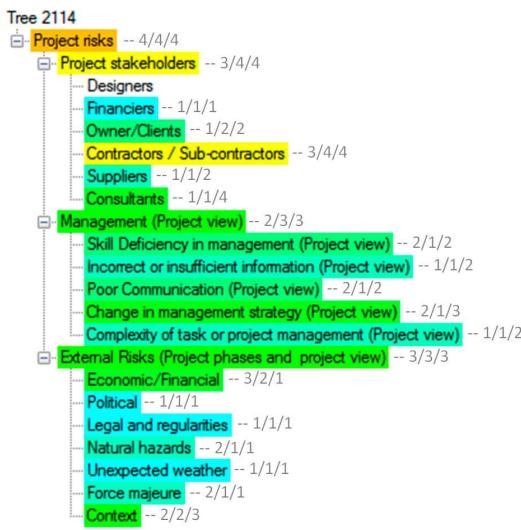
Veja, na **figura 15**, os processos que fazem parte da área de conhecimento Gerenciamento de Riscos. Os referidos processos permitem a sistematização da identificação, da análise e das respostas aos riscos de um projeto.

IDENTIFICAÇÃO DE RISCOS

Alinhado às descrições apresentadas, podemos determinar que um risco possui três componentes: um evento; a probabilidade de ocorrência do evento, associada ao grau de incerteza; e o impacto decorrente do evento, caso aconteça, associado ao grau de perda.

Como parte do Plano de Gerenciamento de Riscos, a elaboração de uma Estrutura Analítica do Risco (EAR) auxilia na definição de potenciais fontes de riscos para o projeto, permitindo a identificação de riscos de forma categorizada, exemplificada na figura 19.

O processo de Identificação de Riscos requer uma clara compreensão da missão, escopo e objetivos do projeto.



RBS#2114
Objective= Cost & delay & quality
N_{convenience}=0.4171
N_{phase}=0.6173
N_{stakeholder}=0.9565
N_{objective}=0.8434
N_{contrast}=0.1471
N_{global}=0.5963
Project global risk note= 4



Fonte: Wikipedia

Figura 19 - Extrato de um exemplo de EAR.

Segundo Pressman (2016), uma proposta para identificação de categorias genéricas de risco relacionadas com o software inclui:

TAMANHO DO PRODUTO

Riscos associados ao tamanho do software.

IMPACTO NO NEGÓCIO

Riscos oriundos de restrições impostas pelo cliente ou pelo mercado.

CARACTERÍSTICAS DO ENVOLVIDO

Riscos relacionados com a comunicação oportuna, por exemplo, entre engenheiro de software e um determinado cliente.

DEFINIÇÃO DO PROCESSO

Riscos relacionados com a auditoria de processos por parte da equipe de qualidade.

AMBIENTE DE DESENVOLVIMENTO

Riscos associados às ferramentas disponíveis para criar o software, tais como, ferramentas case disponíveis.

TECNOLOGIA A SER CRIADA

Risco relacionado com as inovações tecnológicas aplicadas no desenvolvimento do software, tal como, um novo framework de persistência de dados.

QUANTIDADE DE PESSOAS E EXPERIÊNCIA

Riscos que associados aos perfis dos engenheiros de software, administradores de banco de dados, web designers, enfim, da equipe que realizará o trabalho.



Fonte: G-Stock Studio/Shutterstock

O processo de identificação de riscos é iterativo, pois novos riscos podem aparecer durante o projeto. Uma técnica muito comum a ser utilizada na identificação de riscos é a técnica denominada de **Brainstorming**. Esta inclui uma reunião com um conjunto multidisciplinar de especialistas com a finalidade de obter uma lista abrangente de cada risco de projeto e as fontes do risco geral do projeto, sendo as ideias geradas sob a orientação de um facilitador.



Fonte: Boophuket/Shutterstock

Uma segunda técnica é conhecida como Lista de Verificação ou *Check List*, ou seja, uma lista de riscos baseada em informações históricas e conhecimentos acumulados de projetos semelhantes e outras fontes de informações. Essas listas ajudam a identificar pontos fracos no projeto atual a partir de experiências passadas.



Fonte: djile/Shutterstock

Uma terceira técnica proposta pelo PMBOK é a entrevista.

A seguir, vamos explorar um exemplo da aplicação dessa técnica com experientes gerentes de projeto de software, que levantaram as seguintes questões e foram apresentadas por Pressman (2016).

Exemplo da Técnica de Entrevista

A alta gerência e o cliente estão formalmente comprometidos em apoiar o projeto?

Os usuários estão bastante comprometidos com o projeto e o sistema/produto a ser criado?

Os requisitos são amplamente entendidos pela equipe de engenharia de software e clientes?

Os clientes foram totalmente envolvidos na definição dos requisitos?

Os usuários têm expectativas realistas?

O escopo do projeto é estável?

A equipe de Engenharia de Software tem a combinação de aptidões adequadas?

Os requisitos de projetos são estáveis?

A equipe de projeto tem experiência com a tecnologia a ser implementada?

O número de pessoas na equipe de projeto é adequado para o trabalho?

Todos os clientes e usuários concordam com a importância do projeto e com os requisitos do sistema/produto a ser criado?

Caso alguma questão seja respondida negativamente, insira o risco, ou riscos, na sua lista de riscos.

ANÁLISE QUALITATIVA DOS RISCOS

Até o momento, temos uma lista com os prováveis riscos do projeto.

Vamos imaginar que tenhamos uma longa lista. Isso, intuitivamente, nos leva a considerar a necessidade de determinarmos a importância relativa entre os referidos riscos, a fim de que possamos saber quais os riscos prioritários.

É interessante destacar que a tratativa de riscos poderá gerar alocação de recursos financeiros, impactando o orçamento do projeto, ou seja, não é possível tratar os riscos listados com a mesma prioridade, devido às limitações de tempo e recursos.

O objetivo da Análise Qualitativa dos Riscos é priorizar os riscos, combinando a probabilidade de ocorrência e, caso o risco ocorra, o seu impacto. Essa combinação permite determinar o potencial de cada risco em influenciar os resultados do projeto. A probabilidade determina a dimensão da incerteza e o impacto, o efeito sobre os objetivos do projeto.

No início deste módulo, definimos três eventos, a título de exemplo, no projeto de uma viagem ao exterior. Vamos agora aplicar a Análise Qualitativa de Riscos nesses eventos.

PROBABILIDADES

Temos que definir as probabilidades que serão adotadas:

Grande chance de ocorrer	0,95
Provavelmente ocorrerá	0,75
Igual chance de ocorrer ou não	0,50
Baixa chance de ocorrer	0,25
Pouca chance de ocorrer	0,10

Atenção! Para visualização completa da tabela utilize a rolagem horizontal

GRAUS DE IMPACTO

Definir os graus de impacto:

Grau de impacto	Peso
Muito grande	5
Grande	4
Moderado	3

Pequeno	2
Muito pequeno	1

Atenção! Para visualização completa da tabela utilize a rolagem horizontal

APLICAÇÃO

Podemos, então, aplicar os valores a cada evento:

Evento	Probabilidade	Impacto	Probabilidade x Impacto
1) Ocorrência de um problema de saúde, por causa de um acidente ou ter contraído alguma doença	0,75	5	3,75
1) Cancelamento da passagem pela companhia aérea	0,50	3	1,5
3) A quantidade de dólar adquirida não ser suficiente	0,50	4	2,0

Atenção! Para visualização completa da tabela utilize a rolagem horizontal

Temos agora prioridades! De acordo com a tabela, o evento mais crítico está relacionado com a saúde, em seguida, com a quantidade de dólar e, finalmente, com a passagem.

SAIBA MAIS

A Força Aérea Americana determina em seus projetos de software os seguintes componentes de risco: desempenho, custo, suporte e cronograma. As categorias de impacto incluem: catastrófico, crítico, marginal e negligenciável.

Você pode questionar: como determino a probabilidade e o impacto, valores que agregam incertezas? Lembre-se de que a área de conhecimento Gerenciamento de Risco do Projeto requer, como as demais áreas, especialistas na respectiva área. Os profissionais que trabalham com risco têm a expertise necessária no trato desses números que representam incertezas, principalmente, por experiências em projetos passados semelhantes.

O gerente de projetos é um integrador de conhecimentos e não precisa ter expertise em todas as áreas de conhecimento, mas tem que entender bem dos processos aplicados nas diferentes áreas.

ANÁLISE QUANTITATIVA DOS RISCOS

A Análise Quantitativa dos Riscos é um processo de analisar numericamente os efeitos dos riscos priorizados pela Análise Qualitativa de Riscos, pois estes podem gerar impacto potencial e substancial nos resultados do projeto.

As técnicas aplicadas incluem simulações, árvore de decisão entre outras. No contexto da Engenharia de Software, podemos aplicar a árvore de decisão, a fim de avaliar (1) a contratação de uma fábrica de software para desenvolvimento de um determinado software ou (2) realizar o desenvolvimento na empresa.

Vamos agora dar continuidade ao nosso pequeno exemplo. A tabela a seguir, considerando a simplicidade do cenário adotado, ilustra o resultado da análise quantitativa:

Prioridade	Evento	Custo (R\$)

1	Ocorrência de um problema de saúde, por causa de um acidente ou ter contraído alguma doença	10.000,00
2	A quantidade de dólar adquirida não ser suficiente	1.000,00
3	Cancelamento da passagem pela companhia aérea	3.000,00

Atenção! Para visualização completa da tabela utilize a rolagem horizontal

PLANEJAMENTO DE RESPOSTAS AOS RISCOS

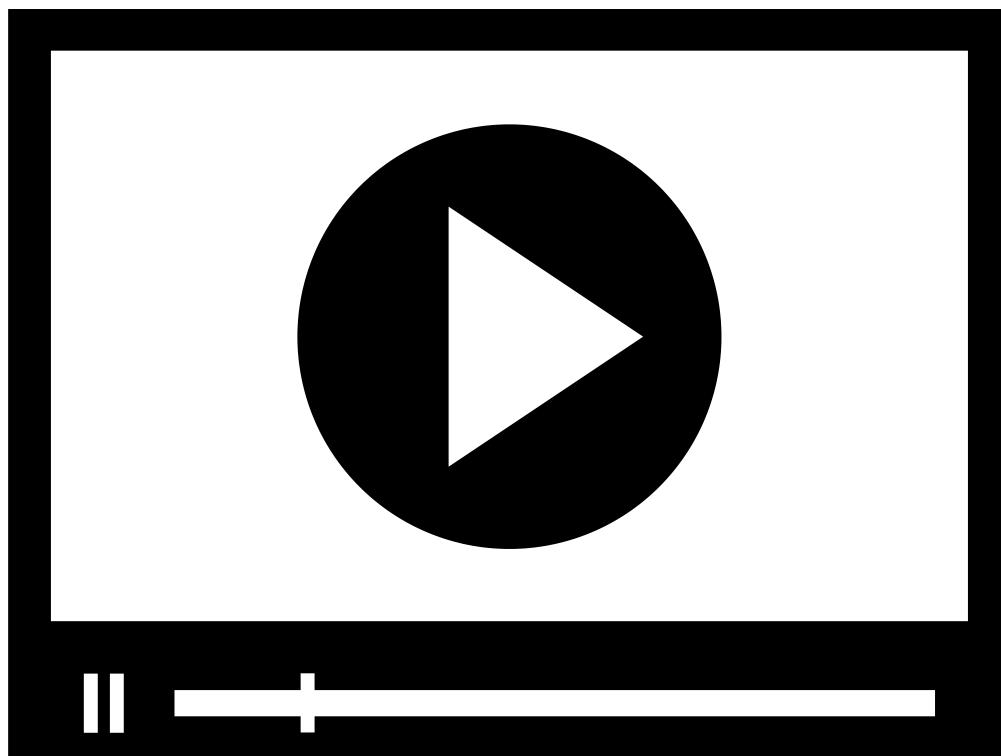
O processo Planejar Respostas aos Riscos inclui o planejamento de ações para o aumento das oportunidades e redução das ameaças aos objetivos do projeto, gerenciando os riscos de acordo com sua prioridade. As referidas ações são representadas pelas respostas adequadas a cada risco, podendo ser incluído o tratamento a ser aplicado, por exemplo, mitigação, quando se busca reduzir o impacto no resultado do projeto, ou eliminação, uma resposta que eliminate o referido impacto.

Retornando ao nosso pequeno exemplo, temos uma proposta de respostas aos riscos identificados na tabela a seguir:

Prioridade	Evento	Tratamento	Resposta
1	Ocorrência de um problema de saúde, por causa de um acidente ou	Eliminação	Contratar o seguro viagem

	ter contraído alguma doença		
2	A quantidade de dólar adquirida não ser suficiente	Eliminação	Atualizar o cartão de crédito para uso internacional
3	Cancelamento da passagem pela companhia aérea	Mitigação	Confirmar a reserva em companhia aérea com boas alternativas de voos

Atenção! Para visualização completa da tabela utilize a rolagem horizontal



Assista agora ao vídeo sobre o **Processo de gerenciamento de riscos na etapa de planejamento.**

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.



RESUMINDO

Neste módulo, podemos destacar a importância da área de conhecimento Gerenciamento dos Riscos do Projeto. Um risco é um evento ou condição incerta que, se ocorrer, provocará um efeito positivo ou negativo em um ou mais objetivos do projeto.

A identificação dos riscos pode ser facilitada pela definição de uma Estrutura Analítica de Riscos (EAR), pois essa permite a categorização dos riscos.

Definidos os riscos, surge a necessidade de priorização destes, pois a tratativa dos riscos poderá impactar o orçamento final do projeto. O processo de análise qualitativa permite a referida priorização definindo para cada risco a sua probabilidade de ocorrência e o grau de impacto, caso ocorra, nos resultados do projeto. Obtidos os dois valores numéricos, pode-se então priorizar os riscos.

Priorizados os riscos, o próximo processo inclui a realização da análise quantitativa dos riscos, permitindo, de acordo com a priorização realizada anteriormente, quantificar os impactos sobre os custos do projeto em função dos riscos.

O planejamento tem como último processo a elaboração do Planejamento de Respostas aos Riscos, incluindo as ações que irão realizar as tratativas que permitirão eliminar ou mitigar os impactos nos resultados do projeto. O referido plano é colocado em execução por meio do processo Implementar Respostas aos Riscos e devidamente monitorado pelo processo Monitorar os Riscos (veja **figura 15**).

O projeto de software possui características que necessitam das melhores práticas de gerenciamento de riscos contidas no PMBOK, com destaque para as atualizações constantes das tecnologias e a alta volatilidade dos requisitos durante o projeto.

Lembre-se, analise os riscos do seu projeto.

VERIFICANDO O APRENDIZADO

1. UM ENGENHEIRO DE SOFTWARE, RESPONSÁVEL PELO GERENCIAMENTO DE RISCOS, DETECTOU UM RISCO RELACIONADO AO USO DE UMA NOVA TECNOLOGIA DE PERSISTÊNCIA DE DADOS NUNCA UTILIZADA NA EMPRESA. EM UMA REUNIÃO DE BRAINSTORMING, PARTICIPANTES DO PROJETO APRESENTARAM OUTROS RISCOS DO PROJETO EM DESENVOLVIMENTO. AO FINAL DA REUNIÃO, CADA RISCO FOI PRIORIZADO EM FUNÇÃO DAS RESPECTIVAS AMEAÇAS AO PROJETO, SENDO GERADA UMA ATA DA REUNIÃO COM O PLANO DE RESPOSTAS A TODOS OS RISCOS. NO CONTEXTO DO GERENCIAMENTO DE RISCO, ANALISE O FINAL DA REFERIDA REUNIÃO E ASSINALE A OPÇÃO CORRETA:

- A)** O gerente de riscos agiu corretamente, gerando uma evidência de tratativa dos riscos, ou seja, o plano de respostas aos riscos.
- B)** O gerente de riscos realizou a análise quantitativa corretamente.
- C)** O gerente de riscos deveria ter realizado a análise quantitativa antes da geração do plano de respostas aos riscos.
- D)** O gerente de riscos elaborou corretamente o plano de respostas a riscos.

2. COVEST- COPSET – 2019 (ADAPTADA) AO FAZER SEU PLANO DE RISCOS, UM ANALISTA ELABOROU UMA MATRIZ DE PROBABILIDADE E IMPACTO. SOBRE O EMPREGO DESTE TIPO DE METODOLOGIA, É CORRETO AFIRMAR QUE:

- A)** Deve-se evitar o uso de probabilidades numéricas, aplicando-se a terminologia “baixo, médio ou alto” para indicar a chance de um determinado risco acontecer.

B) Uma matriz de probabilidade e impacto deve considerar, também, fatores qualitativos como o agente responsável e o plano de ação a ser tomado.

C) Os riscos devem ser previstos e documentados livres de contexto, isto é, da forma mais objetiva possível.

D) Nessa matriz, foram especificadas as combinações de probabilidade e o impacto que levam à classificação dos riscos, podendo estes serem classificados separadamente por objetivo, como custo, tempo e escopo.

GABARITO

1. Um engenheiro de software, responsável pelo gerenciamento de riscos, detectou um risco relacionado ao uso de uma nova tecnologia de persistência de dados nunca utilizada na empresa. Em uma reunião de Brainstorming, participantes do projeto apresentaram outros riscos do projeto em desenvolvimento. Ao final da reunião, cada risco foi priorizado em função das respectivas ameaças ao projeto, sendo gerada uma ata da reunião com o plano de respostas a todos os riscos. No contexto do gerenciamento de risco, analise o final da referida reunião e assinale a opção correta:

A alternativa "C" está correta.

A elaboração do plano de respostas a riscos ocorre após a análise quantitativa, pois essa permite a análise numérica dos riscos que serão inseridos no referido plano.

2. COVEST- COPSET – 2019 (adaptada) Ao fazer seu plano de riscos, um analista elaborou uma matriz de probabilidade e impacto. Sobre o emprego deste tipo de metodologia, é correto afirmar que:

A alternativa "D" está correta.

A Análise qualitativa dos riscos permite definir, para cada risco identificado, uma probabilidade e um grau de impacto, o que permite priorizar os riscos em função de seu efeito sobre os resultados do projeto.

CONCLUSÃO

CONSIDERAÇÕES FINAIS

Apresentamos os principais conceitos da Engenharia de Software relacionados com processo de desenvolvimento de software e Gerenciamento de Projetos.

O processo de desenvolvimento de software permite identificar as atividades que permitem a geração do produto software. Embora existam diferentes modelos de processos, destacamos as atividades típicas ou genéricas que compõem os referidos processos, cujo entendimento permite ao engenheiro de software compreender os possíveis modelos a serem adotados em um determinado projeto de software.

O Gerenciamento de Projetos tem foco na gestão durante o desenvolvimento de software, diferentemente do processo de software, que determina as atividades técnicas. Ele define as atividades de gestão relacionadas à iniciação, ao planejamento, à execução, ao monitoramento e controle e ao encerramento do projeto. Nesse contexto, destacamos o gerenciamento de risco.

Para ouvir um *podcast* sobre o assunto, acesse a versão online deste conteúdo.



REFERÊNCIAS

BEZERRA, E. **Princípios de Análise e Projeto de Sistemas com UML**. 3. ed. Rio de Janeiro: Elsevier, 2014.

PRESSMAN, Roger S.; MAXIM, B. R. **Engenharia de Software**. 8. ed. Porto Alegre: Amgh Editora, 2016.

SOMMERVILLE, I. **Engenharia de Software**. 8. ed. São Paulo: Pearson Prentice Hall, 2007.

PMI. **Um Guia do Conhecimento em Gerenciamento de Projetos**. GUIA PMBOK® 6a. ed. EUA: Project Management Institute, 2017.

EXPLORE+

Para aprofundar seus conhecimentos sobre os assuntos explorados neste tema, leia:

Os capítulos 1, 2, 3, 31 e 35 de:

PRESSMAN, R. S.; MAXIM. B. R. **Engenharia de Software**. 8. ed. Porto Alegre: Amgh Editora, 2016.

Os capítulos 1, 4 e 5 de:

SOMMERVILLE, I. **Engenharia de Software**. 8. ed. São Paulo: Pearson Prentice Hall, 2007.

O capítulo 1 de:

PMI. **Um Guia do Conhecimento em Gerenciamento de Projetos**. GUIA PMBOK® 6. ed. EUA: Project Management Institute, 2017.

CONTEUDISTA

 CURRÍCULO LATTES



Implementação de tratamento de exceções em java

Prof. Marlos de Mendonça Corrêa

Prof. Kleber de Aguiar

Descrição

Exceções em Java: Abordagem de maneira prática dos tipos de exceções e da classe Exception. Sinalização, lançamento, relançamento e tratamento de exceções.

Propósito

O tratamento de exceções é um importante recurso que permite criar programas tolerantes a falhas. Trata-se de um mecanismo que permite resolver ou ao menos lidar com exceções, muitas vezes evitando que a execução do software seja interrompida.

Preparação

Para melhor absorção do conhecimento, recomenda-se o uso de computador com o Java Development Kit (JDK) e um IDE (Integrated Development Environment) instalados.

Objetivos

Módulo 1

Tipos de exceções

Descrever os tipos de exceções em Java.

Módulo 2

A classe Exception

Descrever a classe Exception de Java.

Módulo 3

O mecanismo de tratamento de exceções

Aplicar o mecanismo de tratamento de exceções que Java disponibiliza.



Introdução

A documentação oficial da linguagem Java explica que o termo exceção é uma abreviatura para a frase **evento excepcional** e o define como “um evento, o qual ocorre durante a execução de um programa, que interrompe o fluxo normal das instruções do programa” (ORACLE AMERICA INC., 2021).

A definição de exceção em software apresentada por Java não é específica da linguagem. Sempre que um evento anormal causa a interrupção no fluxo normal das instruções de um software, há uma exceção. Porém, nem todas as linguagens oferecem mecanismos para lidar com tais problemas. Outras oferecem mecanismos menos sofisticados, como a linguagem C++.

A linguagem Java foi concebida com o intuito de permitir o desenvolvimento de programas seguros. Assim, não é de se surpreender que disponibilize um recurso especificamente projetado para permitir o tratamento de exceções de software. Esse será o objeto de nosso estudo, que buscará lançar as bases para que o futuro profissional de programação seja capaz de explorar os recursos da linguagem Java e produzir softwares de qualidade.



1 - Tipos de exceções

Ao final deste módulo, você será capaz de descrever os tipos de exceções em Java.

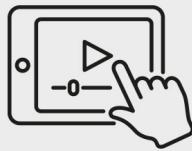
Vamos começar!



Tratamento de exceções em Java

Veja os tipos de exceções e como criar novas exceções em Java.

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.



Visão geral

Conceitos

Uma exceção é uma condição causada por um erro em tempo de execução que interrompe o fluxo normal de execução. Esse tipo de erro pode ter muitas causas, como uma divisão por zero, por exemplo. Quando uma exceção ocorre em Java:

É criado um objeto chamado de exception object, que contém informações sobre o erro, seu tipo e o estado do programa quando o erro ocorreu.

Esse objeto é entregue para o sistema de execução da Máquina Virtual Java (MVJ), processo esse chamado de lançamento de exceção.

Uma vez que a exceção é lançada por um método, o sistema de execução vai procurar na pilha de chamadas (call stack) por um método que contenha um código para tratar essa exceção. O bloco de código que tem por finalidade tratar uma exceção é chamado de: exception handler (tratador de exceções).

A busca seguirá até que o sistema de execução da MVJ encontre um exception handler adequado para tratar a exceção, passando-a para ele.

Quando isso ocorre, é verificado se o tipo do objeto de exceção lançado é o mesmo que o tratador pode tratar. Se for, ele é considerado apropriado para aquela exceção.

Quando o tratador de exceção recebe uma exceção para tratar, diz-se que ele captura (catch) a exceção.

Atenção!

Ao fornecer um código capaz de lidar com a exceção ocorrida, o programador tem a possibilidade de evitar que o programa seja interrompido. Todavia, se nenhum tratador de exceção apropriado for localizado pelo sistema de execução da MVJ, o programa irá terminar.

Um bloco de exceção tem a forma geral mostrada no código 1, a seguir.

Java



Embora o uso de exceções não seja a única forma de se lidar com erros em software, elas oferecem algumas vantagens, como:

A separação do código destinado ao tratamento de erros do código funcional do software. Isso melhora a organização e contribui para facilitar a depuração de problemas.

A possibilidade de se propagar o erro para cima na pilha de chamadas, entregando o objeto da exceção diretamente ao método que tem interesse na sua ocorrência. Tradicionalmente, o código de erro teria de ser propagado método a método, aumentando a complexidade do código.

A possibilidade de agrupar e diferenciar os tipos de erros. Java trata as exceções lançadas como objetos que, naturalmente, podem ser classificados com base na hierarquia de classes. Por exemplo, erros definidos mais abaixo na hierarquia são mais específicos, ao contrário dos que se encontram mais próximos do topo, seguindo o princípio da especialização/generalização da programação orientada a objetos.

Tipos de exceções

Entenda os tipos de exceções em Java.

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.



Subtipos Error e Exception

Todos os tipos de exceção em Java são subclasses da classe `Throwable`. A primeira separação feita agrupa as exceções em dois subtipos:

Error

Agrupa as exceções que, em situações normais, não precisam ser capturadas pelo programa. Em situações normais, não se espera que o programa cause esse tipo de erro, mas ele pode ocorrer e, quando ocorre, causa uma falha catastrófica.

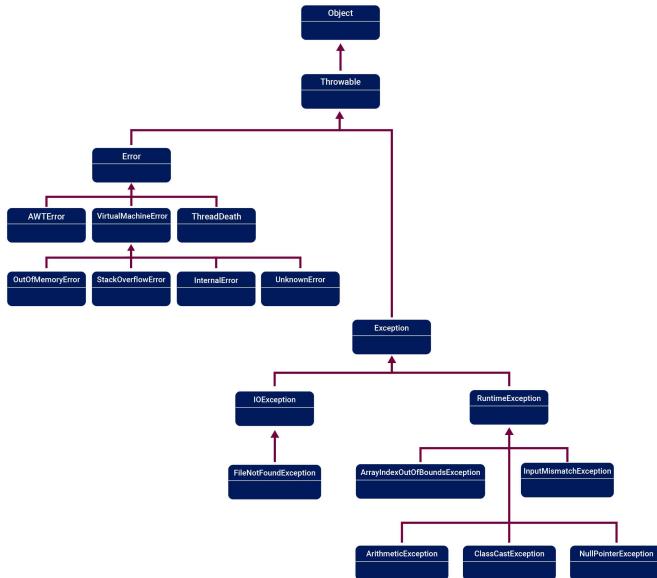
A subclasse `Error` agrupa erros que ocorrem em **tempo de execução** (trata-se do período em que um software permanece em execução). Ela é utilizada para o sistema de execução Java indicar erros relacionados ao ambiente de execução. Um estouro de pilha (*stack overflow*) é um

exemplo de exceção que pertence à subclasse Error.

Exception

Agrupa as exceções que os programas deverão capturar e permite a extensão pelo programador para criar suas próprias exceções. Uma importante subclasse de Exception é a classe **RuntimeException**, que corresponde às exceções como a divisão por zero ou o acesso a índice inválido de array, e que são automaticamente definidas.

A imagem a seguir mostra algumas subclasses da classe **Throwable**.



Hierarquia parcial de classes de exceções em Java.

Quando uma exceção não é adequadamente capturada e tratada, o interpretador irá mostrar uma mensagem de erro com informações sobre o problema ocorrido e encerrará o programa.

Antes de seguir com o exemplo fique atento:

Atenção!

Para a execução correta do programa no emulador, deve-se informar no campo Input o dividendo (número inteiro > 0), o divisor (número inteiro > 0) e "n" ou "s" para a pergunta "Repetir?" (para que o programa saia do laço **do-while** ou continue a executar o laço). Caso o "s" seja escolhido ao invés de "n", será necessário informar um novo dividendo, um novo divisor e uma nova resposta "s" ou "n" no campo Input, antes de clicar no botão Executar do emulador.

A última informação do campo Input deve ser a letra "n", do contrário o programa entrará em loop infinito (ficará executando o laço do-while indefinidamente).

Exemplo de entradas no campo Input do emulador, para que o laço **do-while** seja executado 3 vezes: 1; 0; 2; s; 6; 2; s; 20; 10; n.

Tendo em mente o ponto de atenção anterior, vejamos o exemplo (código 2) mostrado no emulador a seguir.

TUTORIAL COPIAR

Java

null



null



A execução desse código é sujeita a erros ocasionados na execução. Nas condições normais, como dito anteriormente, o programa executará indefinidamente até que escolhamos um valor diferente de "s" para a pergunta "Repetir?". Um dos erros a que ele está sujeito é a divisão por zero. Como não estamos tratando essa exceção, se ela ocorrer, teremos o fim do programa.

Exemplo de um conjunto de entradas que causaria o erro da divisão por zero:

- 8
- 0
- s
- 4
- 2
- n

Volte ao emulador anterior (código 2) e no campo **Input** informe o valor **0** para algum divisor. A seguir, clique em Executar e observe a ocorrência do erro da divisão por zero na prática.

Vendo o campo Console do emulador com o código 2, observamos a mensagem "Exception in thread "main" java.lang.ArithmaticException: / by zero at Main.main(Main.java:14)", que indica o término abrupto do programa por causa da exceção ocorrida na **linha 14**. Vamos, agora, observar como a captura e o tratamento da exceção beneficia o programa.

Modificaremos o código 2, envolvendo a divisão **dividendo / divisor** (linha 14) em um bloco **try**, como está demonstrado no código 3 no próximo emulador.

Comentário

Inserimos o comando de impressão do resultado da divisão, `System.out.println ("O quociente é: " + quociente)`, também dentro do bloco try, para garantir que este seja impresso somente se não ocorrer erro na divisão.

Clique em Executar no emulador a seguir.

TUTORIAL COPIAR

Java

null



null



Eis a saída para o código modificado:

Terminal



Note que agora o programa foi encerrado de maneira normal, apesar da divisão por zero ocorrida (pela inserção do **0** como valor de um divisor no campo Input do emulador com o código 3. Quando o erro ocorreu, a exceção foi lançada e capturada pelo bloco *catch* (linha 17).

Agora que já compreendemos o essencial do tratamento de erros em Java, veremos a seguir os tipos de exceções implícitas, explícitas e throwables.

Exceções implícitas

Exceções implícitas são aquelas definidas nos subtipos **Error** e **RuntimeException** e suas derivadas. Trata-se de exceções ubíquas, quer dizer, que podem ocorrer em qualquer parte do programa e normalmente não são causadas diretamente pelo programador. Por essa razão, possuem um tratamento especial e não precisam ser manualmente lançadas.

Exemplo

Se nos remetermos ao código 2, perceberemos que a divisão por zero não está presente no código. A linha 14 apenas codifica uma operação de divisão. Ela não realiza a divisão que irá produzir a exceção a menos que o usuário, durante a execução, entre com o valor zero para o divisor, situação em que o Java runtime detecta o erro e lança a exceção **ArithmaticException**.

Outra exceção implícita, o problema de estouro de memória (*OutOfMemoryError*) pode acontecer em qualquer momento, com qualquer instrução sendo executada, pois sua causa não é a instrução em si, mas um conjunto de circunstâncias de execução que a causaram. Também são exemplos as exceções **NullPointerException** e **IndexOutOfBoundsException**, entre outras.

Atenção!

As exceções implícitas são lançadas pelo próprio Java runtime, não sendo necessária a declaração de um método *throw* para ela. Quando uma exceção desse tipo ocorre, é gerada uma referência implícita para a instância lançada.

A saída do código 2 mostra o lançamento da exceção que nessa versão não foi tratada, gerando o encerramento anormal do programa.

O código 3 nos mostra um bloco *try-catch*. O bloco *try* indica o bloco de código que será monitorado para ocorrência de exceção e o bloco *catch* captura e trata essa exceção. Mas mesmo nessa versão modificada, o programador não está lançando a exceção, apenas capturando-a. A exceção continua sendo lançada automaticamente pelo Java runtime. Contudo, não há impedimento a que o programador lance manualmente a exceção.

No próximo emulador temos o **código 4**, que modifica o código 3, passando a lançar a exceção.

Atenção!

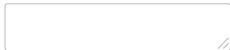
Novamente, você deve informar no campo Input do emulador o dividendo (número inteiro ≥ 0), o divisor (número inteiro ≥ 0) e “n” ou “s” para a pergunta “Repetir?” (para que o programa saia do laço **do-while** ou continue a executar o laço). Lembrando mais uma vez que, caso o “s” seja escolhido ao invés de “n”, será necessário informar um novo dividendo, um novo divisor e uma nova resposta para a pergunta “Repetir?” no campo Input, antes de clicar no botão Executar do emulador.

Atente para que a última informação do campo Input seja a letra “n”, do contrário o programa entrará em loop infinito (ficará executando o laço **do-while** indefinidamente).

 TUTORIAL  COPIAR

Java

null



null



A saída produzida pela execução dessa versão modificada. Observe que na saída presente no campo Console do emulador anterior haverá a seguinte mensagem:

ERRO: Divisão por zero! Divisor nulo.

Embora seja possível ao programador lançar manualmente a exceção, os chamadores do método que a lançam não serão forçados pelo interpretador a tratar a exceção. Apenas exceções que não são implícitas, isto é, lançadas pelo Java runtime, devem obrigatoriamente ser tratadas. Por isso, também dizemos que exceções implícitas são contornáveis, pois podemos simplesmente ignorar seu tratamento. Isso, porém, não tende a ser uma boa ideia, já que elas serão lançadas e terminarão por provocar a saída anormal do programa.

Exceções explícitas

Todas as exceções que não são implícitas são consideradas explícitas. Esse tipo de exceção, de maneira oposta às implícitas, é dito incontornável.

Atenção!

Quando um método usa uma exceção explícita, ele obrigatoriamente deve usar uma instrução throw no corpo do método para criar e lançar a exceção.

O programador pode escolher não capturar essa exceção e tratá-la no método onde ela é lançada. Ou fazê-lo e ainda assim desejar propagar a exceção para os chamadores do método. Em ambos os casos, deve ser usada uma cláusula throws na assinatura do método que declara o tipo de exceção que o método irá lançar se um erro ocorrer. Nesse caso, os chamadores precisarão envolver a chamada em um bloco try-catch.

O **código 5** mostra um exemplo de exceção explícita (*IllegalAccessException*). Essa exceção é uma subclasse de **ReflectiveOperationException** que, por sua vez, descende da classe **Exception**. Logo, não pertence aos subtipos **Error** ou **RuntimeException** que correspondem às exceções explícitas.

Java



Código 5: Exemplo de exceção explícita.

Como é uma exceção explícita, ela precisa ser tratada em algum momento. No caso mostrado no código 5, ela está abarcada por um bloco try-catch, o que livra os métodos chamadores de terem de lidar com a exceção gerada.

E quais seriam as consequências se o programador optasse por não a tratar localmente? Nesse caso, o bloco try-catch seria omitido com a supressão das linhas 4, 7, 8 e 9. A linha 6, que lança a exceção, teria de permanecer, pois, lembremos, é uma exceção explícita. Ao fazer isso, o interpretador Java obrigaria a que uma cláusula throws fosse acrescentada na assinatura do método, informando aos chamadores as exceções que o método pode lançar.

O código 5 pode ser invocado simplesmente fazendo-se como no código 6. Observe que a invocação, vista na linha 5, prescinde do uso de bloco try-catch.

Java



Código 6: Invocando um método que lança uma exceção explícita.

A situação, porém, é diferente se o método “`getELEMENTO (i)`” propagar a exceção por meio de uma cláusula `throws`. Nesse caso, mesmo que a exceção seja tratada localmente, os chamadores deverão envolver a chamada ao método num bloco `try-catch` ou também propagar a exceção.

Criação de novas classes de exceções



Declarando novos tipos de exceções

Entenda os novos tipos de exceções em Java.

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.



Até o momento, todas as exceções que vimos são providas pela própria API Java. Essas exceções cobrem os principais erros que podem ocorrer num software, como erros de operações de entrada e saída (E/S), problemas com operações aritméticas, operações de acesso de memória ilegais e outras. Adicionalmente, fornecedores de bibliotecas costumam prover suas próprias exceções para cobrir situações particulares de seus programas. Então, possivelmente no desenvolvimento de um software simples, isso deve bastar. Mas nada impede que o programador crie sua própria classe de exceções para lidar com situações específicas.

Dica

Se você for prover software para ser usado por outros, como bibliotecas, motores ou outros componentes, declarar seu próprio conjunto de exceções é uma boa prática de programação e agraga qualidade ao produto. Felizmente, isso é possível com o uso do mecanismo de herança.

Para criar uma classe de exceção deve-se, obrigatoriamente, estender-se uma classe de exceção existente, pois isso irá legar à nova classe o mecanismo de manipulação de exceções. Uma classe de exceção não possui qualquer membro a não ser os **4 construtores** a seguir (DEITEL; DEITEL, s.d.):

Um que não recebe argumentos e passa uma String

Mensagem de erro padrão – para o construtor da superclasse.

Um que recebe uma String

Mensagem de erro personalizada – e a passa para o construtor da superclasse.

Um que recebe uma String

Mensagem de erro personalizada – e um objeto Throwable – para encadeamento de exceções – e os passa para o construtor da superclasse.

Um que recebe um objeto Throwable

Para encadeamento de exceções – e o passa para o construtor da superclasse.

Embora o programador possa estender qualquer classe de exceções, é preciso considerar qual superclasse se adequa melhor à situação.

Relembrando

Estamos nos valendo do mecanismo de herança, um importante conceito da programação orientada a objetos. Numa hierarquia de herança, os níveis mais altos generalizam comportamentos, enquanto os mais baixos especializam.

Logo, a classe mais apropriada será aquela de nível mais baixo cujas exceções ainda representem uma generalização das exceções definidas na nova classe. Em última instância, podemos estender diretamente da classe Throwable.

Como qualquer classe derivada de Throwable, um objeto dessa subclasse conterá um instantâneo da pilha de execução de sua thread no momento em que foi criado. Ele também poderá conter uma String com uma mensagem de erro, como vimos antes, a depender do construtor utilizado. Isso também abre a possibilidade para que o objeto tenha uma referência para outro objeto throwable que tenha causado sua instanciação, caso em que há um encadeamento de exceções.

Dica

Ao se criar uma classe de exceção, deve-se considerar a reescrita do método `toString()`. Além de fornecer uma descrição para a exceção, a reescrita do método `toString()` permite ajustar as informações que são impressas pela invocação desse método, potencialmente melhorando a legibilidade das informações mostradas na ocorrência das exceções.

O código 7 mostra um exemplo de exceção criada para indicar problemas na validação do CNPJ. Repare seu uso nas linhas 22 e 61 da classe Juridica (Código 8) e logo após a saída provocada por uma exceção. Veja também que a exceção é lançada e encadeada, ficando o tratamento a cargo de quem invoca `atualizarID()`. Vejamos a seguir o código 7:



Código 7: Nova classe de exceção.

Confira agora o código 8:

Java



Código 8: Classe Jurídica com lançamento de exceção.

Agora vejamos a saída esperada:

Terminal



Saída.

Falta pouco para atingir seus objetivos.

Vamos praticar alguns conceitos?

Questão 1

Considere as afirmações a seguir.

- I – Um desvio no fluxo principal de um programa é uma exceção.
- II – Se uma exceção é lançada pela instrução “throw”, então ela é uma exceção explícita.
- III – Uma exceção explícita não precisa ser tratada localmente no método no qual é lançada.

É (são) verdadeira(s) apenas a

- A I.
- B II.
- C III.
- D I e II.
- E II e III.

Parabéns! A alternativa C está correta.

Uma exceção é um desvio não previsto no fluxo do programa. Desvios no fluxo principal para fluxos alternativos são condições normais e previstas e não são exceções. Além disso, uma exceção implícita também pode ser lançada pela instrução “throw”. Mesmo uma instrução explícita pode ser lançada e propagada, não sendo obrigatório seu tratamento no método onde ocorre.

Questão 2

Sobre exceções implícitas, é correto afirmar apenas que

- A exceções implícitas não podem ser propagadas.
- B exceções definidas na classe UnknownError são implícitas.

- C se uma exceção implícita não for capturada, será gerado erro de compilação.
- D uma exceção implícita é sempre consequência direta da programação.
- E exceções implícitas ocorrem sempre em tempo de compilação.

Parabéns! A alternativa B está correta.

Exceções são implícitas quando definidas nas classes Error e RuntimeException e suas derivadas. A classe UnknownError é uma subclasse de Error.



2 - A classe Exception

Ao final deste módulo, você será capaz de descrever a classe Exception de Java.

Visão geral

O tratamento de exceções em Java é um mecanismo flexível que agrega ao programador um importante recurso. Para extrair tudo que ele tem a oferecer, faz-se necessário conhecer mais detalhadamente o seu funcionamento. Já dissemos que as exceções provocam um desvio na lógica de execução do programa. Mas esse desvio pode trazer consequências indesejadas. A linguagem, porém, oferece uma maneira de remediar tal situação.

Conceitos

Três cláusulas se destacam quando se busca compreender o mecanismo de tratamento de exceções de Java:

Finally

A instrução finally oferece uma maneira de lidar com certos problemas gerados pelo desvio indesejado no fluxo do programa.

Throw

A instrução throw é básica, por meio dela podemos lançar as exceções explícitas e, manualmente, as implícitas.

Throws

A instrução throws pode ser conjugada para alterar a abordagem no tratamento das exceções lançadas.

Nas próximas seções abordaremos cada instrução e, na última seção, analisaremos um mecanismo adicionado a partir da versão 1.4 da Java 2 chamado Encadeamento de Exceção.

Tipos de comando



Os comandos finally, throw e throws

Acompanhe os comandos finally, throw e throws e o encadeamento de exceções.

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.



Comando finally

Vamos relacionar o comando finally com o seguinte exemplo:

Exemplo

Imagine que um programador esteja manipulando um grande volume de dados. Para isso, ele está usando estruturas de dados que ocupam significativo espaço de memória. De repente, o software sofre uma falha catastrófica, por um acesso ilegal de memória, por exemplo, e encerra anormalmente. O que ocorre com toda a memória alocada?

Em linguagens como o C++, que não implementam um coletor de lixo, a porção de memória reservada para uso permanecerá alocada e indisponível até que o sistema seja reiniciado. Como a Máquina Virtual Java implementa o coletor de lixo, o mesmo não ocorre. A porção de memória permanecerá alocada somente até a próxima execução do coletor. Entretanto, vazamentos ainda podem ocorrer se o programador, equivocadamente, mantiver referências para objetos não utilizados.

O problema que acabamos de descrever é chamado de vazamento de memória e é um dos diversos tipos de vazamento de recursos. Podemos ter outros vazamentos, como conexões não encerradas, acessos a arquivos não fechados, dispositivos não liberados etc. Infelizmente, nesses casos, o coletor de lixo não resolve a questão.

Java oferece, porém, um mecanismo para lidar com tais situações: O **bloco finally**. Esse bloco será executado independentemente de uma exceção ser lançada no bloco try ao qual ele está ligado. Aliás, mesmo que haja instruções return, break ou continue no bloco try, finally será executado. A única exceção é se a saída se der pela invocação de System.exit, o que força o término do programa.

Por essa característica, podemos utilizar o bloco finally para liberar os recursos, evitando o vazamento. Quando nenhuma exceção é lançada no bloco try, os blocos catch não são executados e o compilador segue para a execução do bloco finally. Alternativamente, uma exceção pode ser lançada. Nessa situação, o bloco catch correspondente é executado e a exceção é tratada. Em seguida, o bloco finally é executado e, se possuir algum código de liberação de recursos, o recurso é liberado. Uma última situação se dá com a não captura de uma exceção. Também nesse caso, o bloco finally será executado e o recurso poderá ser liberado.

Atenção!

O emulador a seguir mostra um exemplo de uso do finally na linha 21. Como em códigos semelhantes (Códigos 2, 3 e 4), informe no campo Input do emulador: dividendo (número inteiro ≥ 0), divisor (número inteiro ≥ 0) e a resposta para a pergunta “Repetir?” (“s” para que o programa saia do laço **do-while** ou “n” para que o laço continue a ser executado). Caso o “s” seja escolhido ao invés de “n”, será necessário informar um novo dividendo, um novo divisor e uma nova resposta para a pergunta “Repetir” no campo Input, antes de clicar no botão Executar do emulador.

Reiterando que a última informação do campo Input tem que ser a letra “n”, para que o programa não entre em loop infinito.

Vejamos o código 9:

Java

 TUTORIAL  COPIAR

null



null



Observando a saída no Console do emulador anterior, fica claro que finally foi executado independentemente de a exceção ter sido lançada (linhas 15 e 16 do código 9) ou não.

O bloco finally é opcional. Entretanto, um bloco try exige pelo menos um bloco catch ou um bloco finally.

Isto é, se usarmos um bloco try, podemos omitir o bloco catch desde que tenhamos um bloco finally. No entanto, diferentemente de catch, um try pode ter como correspondente uma, e somente uma, cláusula finally. Assim, se suprimíssemos o bloco catch da linha 19 do código 9, mas mantivéssemos o bloco finally, o programa compilaria sem problema.

Relembrando

Mesmo que um desvio seja causado por break, return ou continue, finally é executado. Logo, se inserirmos uma instrução “return;” após a linha 18, ainda veremos a saída “Bloco finally” sendo impressa. Mas, que tal verificar isso por si mesmo? Retorne ao emulador com o código 9 e faça essa experiência!



Throw e Throws

Entenda os comandos throw e throws.

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.



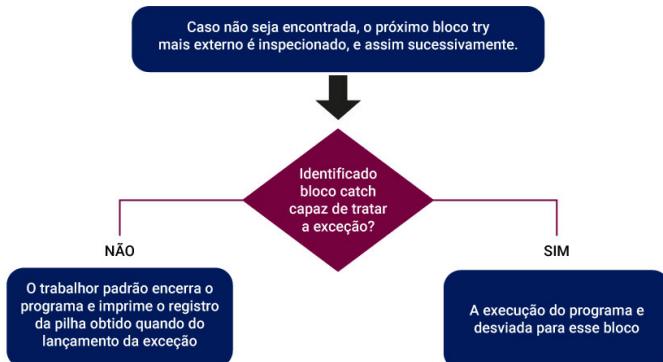
Comando throw

Já vimos que Java lança automaticamente as exceções da classe Error e RuntimeException, as chamadas exceções implícitas. Mas para lançarmos manualmente uma exceção, precisamos nos valer do **comando throw**. Por meio dele, lançamos as chamadas exceções explícitas. A sintaxe de throw é bem simples e consiste no comando seguido da exceção que se quer lançar.

O objeto lançado por throw deve, sempre, ser um objeto da classe Throwable ou de uma de suas subclasses. Ou seja, deve ser uma exceção.

Não é possível lançar tipos primitivos – int, string, char – ou mesmo objetos da classe Object. Tais elementos não podem ser usados como exceção. Uma instância throwable pode ser obtida por meio da passagem de parâmetro na cláusula **catch** ou com o uso do operador **new**.

O efeito quando o compilador encontra uma cláusula throw é de desviar a execução do programa. Isso significa que nenhuma instrução existente após a cláusula será executada. Nesse ponto:



Vamos modificar o código 9 movendo a lógica do programa para a classe Calculadora, conforme mostrado, primeiramente, no código 10 e, posteriormente, no código 11.



Código 10: Classe Calculadora.

Vejamos a seguir o código 11:

Java



Código 11: Classe Principal modificada.

Nessa versão modificada, a exceção é lançada na linha 6 do código 10. É possível ver que a criação da exceção se dá por meio do operador new, que instancia a classe ArithmeticException. Uma vez que a cláusula throw é executada e a instrução lançada, o interpretador Java busca o bloco try-catch mais próximo, que no caso está na linha 4. Esse bloco define um contexto de exceção que, como vemos na linha 8 (bloco catch correspondente), é capaz de tratar a exceção lançada. A execução do programa é, então, desviada para o bloco catch, que recebe a referência do objeto da exceção como parâmetro. A linha 10 imprime, por meio do método getMessage () – definido na classe Throwable –, a mensagem passada como parâmetro para o construtor na linha 6.

Exemplo

O que aconteceria se removêssemos o bloco try-catch, mantendo apenas o lançamento de exceção? Se não houver um bloco catch que trate a exceção lançada, ela será passada para o tratador padrão de exceções, que encerrará o programa e imprimirá o registro da pilha de execução.

Outra forma de lidar com isso seria definir um contexto de exceção no chamador (linha 13 do código 11) e propagar a exceção para que fosse tratada externamente. O mecanismo de propagação de exceções é justamente o que veremos na próxima seção.

Comando throws

Vimos que um método pode lançar uma exceção, mas ele não é obrigado a tratá-la. Ele pode transferir essa responsabilidade para o chamador, que também pode transferir para o seu próprio chamador, e assim repetidamente.

Atenção!

Mas quando um método pode lançar uma exceção e não a trata, ele deve informar isso aos seus chamadores explicitamente, permitindo que o chamador se prepare para lidar com a possibilidade do lançamento de uma exceção.

Essa notificação é feita pela adição da **cláusula throws** na assinatura do método, após o parêntese de fechamento da lista de parâmetros e antes das chaves de início de bloco.

A cláusula throws deve listar todas as exceções explícitas que podem ser lançadas no método, mas que não serão tratadas. Não listar tais exceções irá gerar erro em tempo de compilação. Exceções das classes Error e RuntimeException, bem como de suas subclasses, não precisam ser listadas. A forma geral de throws é:

```
<modificadores> <nome do método> <lista de parâmetros> throws <lista de exceções explícitas não tratadas> { <corpo do método> }
```

A lista de exceções explícitas não tratadas é formada por elementos separados por vírgulas. Eis um exemplo de declaração de métodos com o uso de throws:

```
public int acessaDB ( int arg1 , String aux ) throws ArithmeticException ,  
IllegalAccessException { ...}
```

Observe que, embora não seja necessário listar uma exceção implícita, isso não é um erro e nem obrigará o chamador a definir um contexto de exceção para essa exceção especificamente. No entanto, ele deverá definir um contexto de exceção para tratar da exceção explícita listada.

Vamos modificar a classe Calculadora conforme mostrado no código 12. Observando a linha 2, constatamos a instrução throws. Notamos, também, que o método "divisão ()" ainda pode lançar uma exceção, mas agora não há mais um bloco try-catch para capturá-la. Em consequência, precisamos ajustar a classe Principal como mostrado no código 13. É possível ver, como sugerimos no final da seção anterior, a definição de um bloco try-catch (linha 13) para capturar e tratar a exceção lançada pelo método "divisão ()". Vejamos o código 12:

Java



Código 12: Classe Calculadora com método "divisão" modificado para propagar a exceção.

Observe o código 13 na sequência:

Java



Código 13: Classe Principal com contexto de tratamento de exceção definido para o método "divisão ()".

Encadeamento de exceções

Entenda o encadeamento de exceções em Java.

Para assistir a um vídeo sobre o assunto, accesse a versão online deste conteúdo.



Pode acontecer de, ao responder a uma exceção, um método lançar outra exceção distinta. É fácil entender como isso ocorre. Em nossos exemplos, o bloco catch sempre se limitou a imprimir uma mensagem. Mas isso se deu por motivos didáticos. Nada impede que o tratamento exija um processamento mais complexo, o que pode levar ao lançamento de exceções implícitas ou mesmo explícitas.

Ocorre que se uma exceção for lançada durante o tratamento de outra, as informações como o registro da pilha de execução e o tipo de exceção da exceção anterior são perdidas. Nas primeiras versões, a linguagem Java não provia um método capaz de manter essas informações, agregando-as pertencentes à nova exceção. A consequência dessa limitação é óbvia: a depuração do código se torna mais difícil, já que as informações sobre a causa original do erro se perdem.

Mecanismo de encadeamento de exceções

Felizmente, as versões mais recentes provêm um mecanismo chamado **Encadeamento de Exceção**. Por meio dele, um objeto de exceção pode manter todas as informações relativas à exceção original. A chave para isso são os dois construtores, dos quatro que vimos antes, que admitem a passagem de um objeto Throwable como parâmetro.

O encadeamento de exceção permite associar uma exceção com a exceção corrente, de maneira que a exceção que ocorreu no contexto atual da execução pode registrar a exceção que lhe deu causa. Além da utilização dos construtores que admitem um objeto Throwable como parâmetro, isso também é feito utilizando dois outros métodos da classe Throwable:

getCause ()

Retorna a exceção subjacente à exceção corrente, se houver, caso contrário, retorna null.

initCause ()

Permite associar o objeto Throwable com a exceção que o invoca, retornando uma referência.

Assim, utilizando o método **initCause ()**, cria-se a associação entre uma exceção e outra que lhe originou após a sua criação. Em outras palavras, permite fazer a mesma associação que o construtor, mas após a exceção ter sido criada.

Atenção!

Após uma exceção ter sido associada a outra, não é possível modificar tal associação. Logo, não se pode invocar mais de uma vez o método **initCause ()** e nem o chamar se o construtor já tiver sido empregado para criar a associação.

Falta pouco para atingir seus objetivos.

Vamos praticar alguns conceitos?

Questão 1

Um programador criou seu conjunto de exceções por meio da extensão da classe Error, com a finalidade de tratar erros de socket em conexões com bancos de dados. O comando que pode ser empregado para garantir o fechamento correto da conexão, mesmo em caso de ocorrência de exceção, é

- A try.
- B catch.
- C throw.
- D throws.
- E finally.

Parabéns! A alternativa E está correta.

A ocorrência de exceção causa um desvio não desejado no fluxo do programa, podendo impedir o encerramento da conexão e causando um vazamento de recurso. O comando **finally**, porém, é executado independentemente do desvio causado pela ocorrência de exceção, o que o torna adequado para a tarefa.

Questão 2

Sobre o comando throw, são feitas as seguintes afirmações:

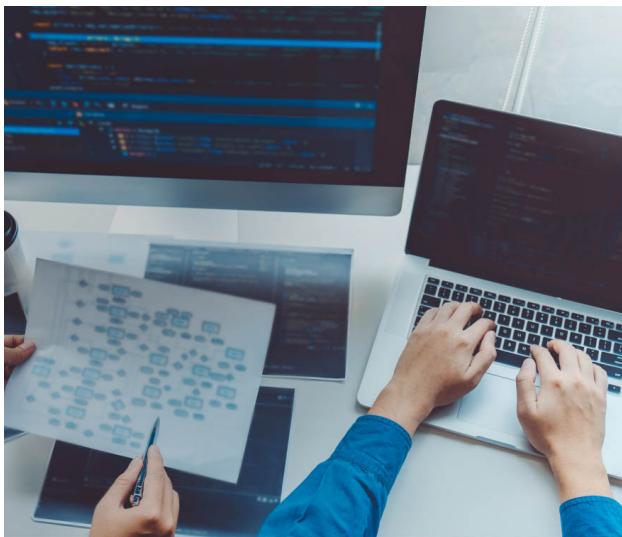
- I – Não pode ser usado com o comando throws.
- II – Só pode ser utilizado dentro do bloco try ou do bloco catch.
- III – Pode ser utilizado para lançar exceções definidas nas subclasses de Error.

É correto apenas o que se afirma em

- A I.
- B II.
- C III.
- D I e II.
- E II e III.

Parabéns! A alternativa C está correta.

O comando throw pode ser usado para lançar exceções fora do bloco try-catch e pode ser usado em combinação com throws, cujo objetivo é notificar o chamador de que uma exceção pode ser lançada, mas não será tratada no local. As exceções derivadas de Error e suas subclasses são implícitas, mas throw pode ser usado para lançá-las manualmente.



3 - O mecanismo de tratamento de exceções

Ao final deste módulo, você será capaz de aplicar o mecanismo de tratamento de exceções que Java disponibiliza.

Visão geral

Conceitos

Vamos agora refletir um pouco mais sobre o mecanismo de tratamentos de exceção de Java. Até o momento, falamos sobre exceções e mencionamos que elas são anormalidades – erros – experimentados pelo software durante sua execução. Não falamos, contudo, dos impactos que o tratamento de exceção produz.

Certamente os erros causados quando se tenta ler além do fim de um array, quando ocorre uma divisão por zero ou mesmo uma tentativa falha de abrir um arquivo que já não existe mais geram problemas que precisam ser enfrentados. Mas, para todos esses casos, técnicas tradicionais podem ser empregadas.

Um ponto que também deve ser considerado é o impacto que o tratamento de exceções traz ao desempenho do software. Num artigo intitulado *Efficient Java Exception Handling in Just-in-Time Compilation* (SEUNGIL et al., 2000), os autores argumentam que o tratamento de exceções impede o compilador Java de fazer algumas otimizações, impactando o desempenho do programa.

Dica

A lição é que tudo possui prós e contras que devem ser pesados segundo a situação específica que se está enfrentando. Não se questiona que o mecanismo de tratamento de exceções possua diversas vantagens. Mas em situações nas quais o desempenho seja crítico, deve-se refletir se seu uso é a abordagem mais adequada.

Outra consideração a ser feita é a suposta distinção entre a sinalização e o lançamento de exceções. Uma rápida pesquisa na internet mostra que, em português, o conceito de sinalização de exceções aparece pouco. Em inglês, praticamente inexiste. Estudando um pouco mais o assunto, podemos ver que o uso do termo sinalização de exceção aparece com mais frequência na disciplina de Linguagem de Programação, que estuda não uma linguagem específica, mas os conceitos envolvidos no projeto de uma linguagem.

Para a disciplina de Linguagem de Programação, sinalizar uma exceção é um conceito genérico que indica o mecanismo pelo qual uma exceção é criada. Em Java, é o que se chama de lançamento de exceção. Outros autores consideram que a sinalização é o mecanismo de instanciação de exceções implícitas, diferenciando-o do mecanismo que cria uma exceção explícita.

Comentário

Para evitar a confusão que essa mescla de conceitos provoca, vamos definir a sinalização de uma exceção como o mecanismo pelo qual um método cria uma instância de uma exceção. Ou seja, mantemo-nos alinhados com o conceito de Linguagem de Programação sobre o mecanismo de tratamento de exceção. Sinalização e lançamento de exceção tornam-se sinônimos.

A razão de fazermos isso é porque alguns profissionais chamam ao mecanismo pelo qual um método Java notifica o chamador das exceções que pode lançar como sinalização de exceções. Isso, contudo, é uma abordagem predisposta à confusão. Tal mecanismo, para fins de discussão, será referenciado como mecanismo de notificação de exceção.

Veremos nas seções seguintes, em mais detalhes, os processos que formam o tratamento de exceções em Java. Abordaremos a notificação de exceção, o seu lançamento e como relançar uma exceção, e concluiremos falando sobre o tratamento.

Tratamento de exceções



Entendendo o tratamento de exceções em Java

Comprenda a notificação, o lançamento, relançamento e tratamento de exceções em Java.

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.



Notificação de exceção

A notificação é o procedimento pelo qual um método avisa ao chamador das exceções que pode lançar. Ela é obrigatória se estivermos lançando uma exceção explícita, e sua ausência irá gerar erro de compilação. Contudo, mesmo se lançarmos uma exceção implícita manualmente, ela não é obrigatória. Fazemos a notificação utilizando a cláusula throws ao fim da assinatura do método. A razão para isso é simples:

As exceções explícitas precisam ser tratadas pelo programador, enquanto as implícitas podem ser simplesmente deixadas para o tratador de exceções padrão de Java. Quando uma exceção implícita é lançada, se nenhum tratador adequado for localizado, a exceção é passada para o tratador padrão, que força o encerramento do programa.

Pela mesma razão, ainda que sejam notificadas pelo método, as exceções implícitas não obrigam o chamador a definir um contexto de tratamento de exceções para invocar o método. Isso não se passa com as exceções explícitas, pois como o compilador impõe que o programador as trate, uma de duas situações deverá ocorrer: ou se define um contexto para tratamento de exceções ou o chamador notifica, por sua vez, outro método que venha a lhe chamar de que pode lançar uma exceção.

Dica

Uma coisa interessante a se considerar na propagação de exceções é que ela pode ser propagada até o tratador padrão Java, mesmo no caso de ser uma exceção explícita. Basta que ela seja propagada até o método "main ()" e que este, por sua vez, faça a notificação de que pode lançar a exceção. Isso fará com que ela seja passada ao tratador padrão.

Lançamento e relançamento de exceções

Lançamento de exceção

Entenda o lançamento de exceções em Java.

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.



O lançamento de uma exceção pode ocorrer de maneira implícita ou explícita, caso em que é utilizada a instrução throw. Como já vimos, quando uma exceção é lançada, há um desvio indesejado no fluxo de execução do programa. Isso é importante, pois o lançamento de uma exceção pode se dar fora de um bloco try-catch. Nesse caso, o programador deve ter o cuidado de inserir o lançamento em fluxo alternativo, pois, do contrário, a instrução throw estará no fluxo principal e o compilador irá acusar erro de compilação, pois todo o código abaixo dela será inatingível. Um exemplo

desse caso pode ser visto no código 14. Observe que uma exceção é lançada na linha 4, no fluxo principal do programa. Ao atingir a linha 4, a execução é desviada e todo o restante não pode ser alcançado.



Código 14: Exemplo de código que não compila devido a erro no lançamento de exceção.

Situações assim podem ser resolvidas, porém, por meio do aninhamento de blocos try. Quando múltiplos blocos try são aninhados, cada contexto de tratamento de interrupção é empilhado até o mais interno. Quando uma exceção ocorre, os blocos são inspecionados em busca de um bloco catch adequado para tratar a exceção. Esse processo começa no contexto no qual a exceção foi lançada e segue até o mais externo. Se nenhum bloco catch for adequado, a exceção é passada para o tratador padrão e o programa é encerrado.

Saiba mais

Uma maneira menos óbvia em que o aninhamento se dá é quando há um encadeamento de chamadas a métodos onde cada método tenha definido seu próprio contexto de tratamento de exceção. Esse é exatamente o caso se combinarmos o código 10 e o código 13. Veja que, na linha 4 do código 10, há um bloco try, que forma o contexto de tratamento de exceção do método (linha 2). Quando esse método é invocado na linha 14 do código 13, ele está dentro de outro bloco try (linha 12) que define outro contexto de tratamento de exceção.

Exceções também podem ser lançadas quando o programador estabelece pré-condições e pós-condições para o método. Trata-se de uma boa prática de programação que contribui para um código bem escrito! Veja:

Pré-condições



São condições estabelecidas pelo programador que devem ser satisfeitas a fim de que o método possa começar sua execução. Quando não são satisfeitas, o comportamento do método é indefinido. Nesse caso, uma exceção pode ser lançada para refletir essa situação e permitir o tratamento adequado ao problema.

Pós-condições

São restrições impostas ao retorno do método e a seus efeitos colaterais possíveis. Elas são verdadeiras se o método, após a sua execução, e dado que as pré-condições tenham sido satisfeitas, tiver levado o sistema ao estado previsto. Ou seja, se os efeitos da execução daquele método correspondem ao projeto. Caso contrário, as pós-condições são falsas e o programador pode lançar exceções para identificar e tratar o problema.

Podemos pensar num método que une (concatena) dois vetores (array) recebidos como parâmetro e retorna uma referência para o vetor resultante como um exemplo didático desse uso de exceções. Podemos estabelecer como pré-condição que nenhuma das referências para os vetores que serão unidos pode ser nula. Se alguma delas o for, então uma exceção `NullPointerException` é lançada. Não sendo nulas, então há, de fato, dois vetores para serem concatenados. O resultado dessa concatenação não pode ultrapassar o tamanho da heap. Se o vetor for maior do que o permitido, uma exceção `OutOfMemoryError` é lançada. Caso contrário, o resultado é válido.

Relançamento de exceção

As situações que examinamos até o momento sempre caíram em uma de duas situações: ou as exceções lançadas eram tratadas, ou elas eram propagadas, podendo ser, em último caso, tratadas pelo tratador padrão da Java. Mas há outra possibilidade. É possível que uma exceção capturada não seja tratada, ou seja tratada parcialmente. Essa situação difere das anteriores, como veremos.

Comentário

Quando propagamos uma exceção lançada ao longo da cadeia de chamadores, fazemos isso até que seja encontrado um bloco catch adequado (ou, como já dissemos, ela seja capturada pelo tratador padrão). Uma vez que um bloco catch capture uma exceção, ele pode decidir tratá-la, tratá-la parcialmente ou não a tratar. Nos dois últimos casos, será preciso que a busca por outro bloco catch adequado seja reiniciada, pois, como a exceção foi capturada, essa busca terminou. Felizmente, é possível reiniciar o processo relançando-se a exceção capturada.

Relançar uma exceção permite postergar seu tratamento, ou parte dele. Assim, um novo bloco catch adequado, associado a um bloco try mais externo, será buscado. É claro que, uma vez relançada a exceção, o procedimento transcorre da mesma forma como ocorreu no lançamento. Um bloco catch adequado é buscado e, se não encontrado, a exceção será passada para o tratador padrão de exceções da Java, que forçará o fim do programa.

O relançamento da exceção é feito pela instrução `throw`, dentro do bloco catch, seguida pela referência para o objeto exceção capturado. A linha 11 do código 15 mostra o relançamento da exceção.

Java



Código 15: Exemplo de relançamentos de exceção na classe Calculadora.

É importante ressaltar que: exceções, contudo, não podem ser relançadas de um bloco finally, pois, nesse caso, a referência ao objeto exceção não está disponível. Observe a linha 21 do código 9. O bloco finally não recebe a referência para a exceção e esta é uma variável local do bloco catch.

Captura de exceção

Entenda o captura de exceções em Java.

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.



A instrução catch define qual tipo de exceção aquele bloco pode tratar. Assim, quando uma exceção é lançada, uma busca é feita do bloco try local para o mais externo, até que um bloco catch adequado seja localizado. Quando isso ocorre, a exceção é capturada por aquele bloco catch que recebe como parâmetro a referência para o objeto exceção que foi lançado.

A captura de uma exceção também significa que os demais blocos catch não serão verificados. Quer dizer que, uma vez que um bloco catch adequado seja identificado, a exceção é entregue a ele para tratamento e os demais blocos são desprezados. Aliás, é por isso que precisamos relançar a exceção, se desejarmos transferir seu tratamento para outro bloco.

Blocos catch, contudo, não podem ocorrer de maneira independente no código. Eles precisam, sempre, estar associados a um bloco try. E também não podem ser aninhados, como os blocos try. Então, como lidar com o código 16?

Java



Código 16: Possibilidade de lançamento de tipos distintos de exceções.

Observe que se a variável **tamnh** for zero, a linha 3 irá gerar uma exceção `ArithmaticException` (divisão por zero), e se tamnh for maior do que o tamanho de arrj, a linha 6 irá gerar uma exceção `ArrayIndexOutOfBoundsException`. Esse é um exemplo de um trecho de código que pode lançar mais de um tipo de exceção.

Uma solução é aninhar os blocos try. Mas a linguagem Java nos dá uma solução mais elegante. Podemos empregar múltiplas cláusulas catch, como mostrado no código 17:



Código 17: Múltiplas cláusulas catch.

Atenção: As exceções de subclasses devem vir antes das exceções da respectiva superclasse. O bloco catch irá capturar as exceções que correspondam à classe listada e a todas as suas subclasses.

Isso se dá porque um objeto de uma subclass é, também, um tipo da superclasse. Uma vez que a exceção seja capturada, as demais cláusulas catch não são verificadas.

Falta pouco para atingir seus objetivos.

Vamos praticar alguns conceitos?

Questão 1

Sobre o relançamento de exceções em Java, assinale a única alternativa correta.

- A O relançamento é feito com a instrução throws.
- B Blocos try aninhados dispensam o relançamento de exceção.
- C O relançamento é feito dentro de um bloco try.
- D Usa-se a instrução throw seguida de new e do tipo de exceção a ser relançado.

- E É feito dentro de um bloco catch e usando a referência da exceção capturada.

Parabéns! A alternativa E está correta.

O relançamento utiliza a referência da exceção capturada junto da instrução throw para lançar novamente essa exceção e é feito dentro de um bloco catch.

Questão 2

Um programador criou uma exceção (NotReferencedException) a partir da classe RuntimeException. Considerando seus conhecimentos de tratamento de exceção em Java, marque a única alternativa correta quando as exceções são empregadas em múltiplas cláusulas catch associadas ao mesmo bloco try para lidar com o lançamento de tipos diferentes de exceção pelo mesmo trecho de código.

- A RuntimeException deve vir antes de NotReferencedException.
- B É indiferente a ordem em que as exceções são usadas.
- C NotReferencedException deve vir antes de RuntimeException.
- D Como NotReferencedException é uma exceção implícita, ela não pode ser usada dessa forma.
- E Exceções implícitas não são capturadas quando há mais de uma cláusula catch associada ao mesmo bloco try.

Parabéns! A alternativa C está correta.

A classe NotReferencedException é uma subclasse de RuntimeException, logo, se RuntimeException estiver antes de NotReferencedException, a exceção será capturada por este bloco catch e nunca chegará ao bloco destinado a tratar exceções do tipo NotReferencedException.

Considerações finais

O tratamento de exceções não é uma exclusividade da linguagem Java. Outras linguagens de programação a implementam. Java inspirou sua implementação no tratamento de exceções da linguagem C++, mas não se limitou a copiá-lo, oferecendo uma abordagem própria.

Iniciamos nosso estudo abordando os tipos de exceções. Compreendemos o que são exceções implícitas e explícitas e discorremos sobre a criação de novos tipos de exceção. Em seguida, analisamos as instruções finally, throw e throws, importantes peças do tratamento de exceção em Java, e finalizamos com um estudo mais detido na notificação, no lançamento, relançamento e tratamento de exceções.

Pudemos ver as diversas formas em que o tratamento de exceções contribui para um código mais robusto, legível e organizado. É, certamente, uma ferramenta a ser empregada por profissionais capacitados e sérios. Mas, para tirar todo o proveito que pode ser obtido, é preciso conhecer as

nuances desse mecanismo!

Podcast

Para encerrar, ouça um resumo sobre implementação de tratamento de exceções em Java.

Para ouvir o áudio, acesse a versão online deste conteúdo.



Explore +

O mecanismo de tratamento de exceções possui muitas nuances interessantes que são úteis para o desenvolvimento de software de qualidade. O encadeamento de exceções é uma das características que podem ajudar, sobretudo no desenvolvimento de bibliotecas, motores e componentes que manipulam recursos. Por isso, estudar a documentação Java sobre o assunto e conhecer melhor a classe Throwable certamente contribuirá para o seu aprendizado.

Referências

DEITEL, P.; DEITEL, H. **Java – How to program.** 11th. ed. [S.I.]: Pearson, [s.d.].

ORACLE AMERICA INC. **What Is an Exception?** (The Java Tutorials > Essential Classes > Exceptions). Consultado na internet em: 01 jun. 2021.

SEUNGIL, L. et. al. **Efficient Java Exception Handling in Just-in-Time Compilation.** Publicado em: 03 jun. 2000. Consultado na internet em: 01 jun. 2021.

Material para download

Clique no botão abaixo para fazer o download do conteúdo completo em formato PDF.

[Download material](#)

O que você achou do conteúdo?



Relatar problema



Introdução à programação OO em Java

Prof. Marlos de Mendonça Corrêa

Prof. Kleber de Aguiar

Descrição

Introdução à programação em Java, com apresentação de classes e objetos, o básico de implementação de herança e polimorfismo, agrupamento de objetos e ambientes de desenvolvimento Java e estruturas básicas da linguagem.

Propósito

Conhecer a linguagem de programação Java, uma das mais populares nos dias atuais por causa da sua portabilidade de código e rapidez de desenvolvimento, o que a torna largamente presente em dispositivos móveis. Juntamente com a orientação a objetos, é elemento indispensável para o profissional de Tecnologia da Informação.

Preparação

Antes de iniciar o conteúdo deste tema, recomenda-se o uso de computador com o Java Development Kit – JDK e um IDE (Integrated Development Environment) instalados.

Objetivos

Módulo 1

Classes e Objetos

Descrever a definição, a manipulação e as nuances de classes e objetos em Java.

Módulo 2

Herança e Polimorfismo

Descrever o mecanismo de herança e polimorfismo em Java.

Módulo 3

Agrupamento de Objetos

Descrever os mecanismos de agrupamento de objetos em Java.

Módulo 4

Ambientes de Desenvolvimento

Reconhecer os ambientes de desenvolvimento em Java e as principais estruturas da linguagem.



Introdução

A Programação Orientada a Objetos (POO) é um paradigma que surgiu em resposta à crise do software. A POO buscou resolver diversos problemas existentes no paradigma de Programação Estruturada, como a manutenibilidade e o reaproveitamento de código. Essas deficiências tiveram papel central na crise, pois causavam o encarecimento do desenvolvimento e tornavam a evolução do software um desafio. A incorporação de novas funções em um software já desenvolvido vinha acompanhada do aumento de sua complexidade, fazendo com que, em certo ponto, fosse mais fácil reconstruir todo o software.

A solução visualizada à época tinha como objetivo facilitar o reaproveitamento do código e reduzir a sua complexidade. As classes e os objetos desempenham esse papel. Uma classe é um modelo a partir do qual se produzem os objetos. Ela define o comportamento e os atributos que todos os objetos produzidos segundo esse modelo terão. Com base nesses conceitos, outros foram desenvolvidos, como a herança, o polimorfismo e a agregação, dando maior flexibilidade e melhorando o reaproveitamento de código e a sua manutenção.

A POO tem nos conceitos de classes e objetos o seu fundamento; eles são centrais para o paradigma. Assim, não é mera coincidência que eles também tenham papel fundamental na linguagem Java.

Quando se diz que uma linguagem é orientada a objetos (OO), diz-se que essa linguagem suporta nativamente os conceitos que formam a POO. Entretanto, a aderência das linguagens aos conceitos OO varia, assim como a forma que elas os implementam. O que deve ficar claro, nesse caso, é que POO é um paradigma de programação, não devendo ser confundido com uma linguagem.

Java possui semelhança com outras linguagens OO, como a C++ ou a C#. Mas, apesar de serem parecidas, as linguagens guardam significativa diferença entre si. Por exemplo, Java é uma linguagem fortemente tipada, ou seja, ela não aceita conversões implícitas entre tipos diferentes de dados como a C++ aceita.

Neste estudo, começaremos pela forma como Java trata e manipula classes e objetos. Com isso, também traremos conceitos de orientação a objetos que são essenciais para compreender o funcionamento de um software em Java. Seguiremos mostrando como Java trata os conceitos de herança e polimorfismo e, em seguida, mostraremos os mecanismos de agrupamento de objetos. Assim, teremos percorrido conceitos e técnicas que formarão a base que o profissional poderá usar para alçar voos mais altos. Finalizaremos

falando sobre ambientes de desenvolvimento em Java, quando abordaremos alguns aspectos úteis para formar um profissional com senso crítico e tecnicamente capacitado.



1 - Classes e Objetos

Ao final deste módulo, você será capaz de descrever a definição, a manipulação e as nuances de classes e objetos em Java.

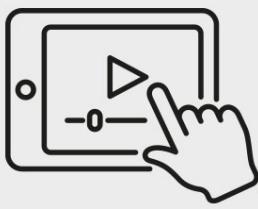
Vamos começar!



Classes e Objetos em Java

Neste vídeo, apresentaremos os conceitos de classes, estado e métodos, passagem de objetos como parâmetros, citando exemplos práticos com a linguagem Java.

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.



Aprender Java sem saber orientação a objeto (OO) é, no mínimo, ilógico.

Comentário

Java é uma linguagem OO pura, logo o estudo dela implica o conhecimento do paradigma OO. Por isso, começaremos explorando os conceitos de classe e objeto ao mesmo tempo em que veremos como Java os trata. Para tanto, veremos o conceito de classes e as formas de emprego, o conceito de objetos e a forma de manipulá-los e, finalmente, um estudo de caso.

Classes e sua realização em Java

Agora, assista o vídeo a seguir:

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.



Em POO (**Programação Orientada a Objetos**), uma classe é uma maneira de se criar objetos que possuem mesmo comportamento e mesma estrutura.

Formalmente falando, uma classe é uma estrutura que define:

- Os dados.
- Os métodos que operam sobre esses dados e formam o comportamento do objeto.
- O mecanismo de instanciação dos objetos.

O conjunto formado pelos dados e métodos (assinatura e semântica) estabelece o contrato existente entre o desenvolvedor da classe e o seu usuário. Isso porque é o método que será invocado pelo usuário e, para tanto, ele precisa saber sua assinatura e conhecer seu comportamento.

O **Código 1** mostra um exemplo de definição de uma classe em Java.

Diferentemente de linguagens como a C e a C++, Java não possui arquivos de cabeçalho, portanto, a implementação dos métodos ocorre junto de sua declaração. Além disso, em Java cada classe pública deve estar num arquivo com o mesmo nome da classe e extensão “java”. Logo, a classe do Código 1 deve ser salva num arquivo de nome “**Aluno.java**”. Confira a seguir:

Java



Código 1: Definição de uma classe em Java.

A classe do Código 1 chama-se “**Aluno**”, possui um atributo do tipo “*String*” (Nome) e dois métodos (*inserirNome* e *recuperarNome*). Mas, além disso, podemos notar as palavras reservadas “*private*” e “*public*”. Essas instruções modificam a visibilidade (ou acessibilidade) de métodos, atributos e classes. O trecho mostrado é um exemplo bem simples de declaração de uma classe em Java. Segundo Gosling et. al. (2020), pela especificação da linguagem há duas formas de declaração de classes: normal e enum. Vamos nos deter apenas na forma normal, mostrada a seguir:

[Modificador] class Identificador [TipoParâmetros] [Superclasse] [Superinterface] { [Corpo da Classe] }

Rotacione a tela.

Na sintaxe mostrada, os colchetes indicam elementos opcionais. Os símbolos que não estão entre colchetes são obrigatórios, sendo que “Identificador” é um literal. Em negrito estão os símbolos reservados da linguagem.

Logo, a forma mais simples possível de se declarar uma classe em Java é: `class Inutil {}`.

Observamos nessa declaração a presença dos símbolos reservados (e obrigatórios) e a existência do Identificador (`Inutil`). É óbvio que essa classe é inútil, uma vez que esse código não faz e não armazena nada. Mas é uma expressão perfeitamente válida na linguagem.

Olhemos agora os demais símbolos da declaração. Os modificadores podem ser qualquer elemento do seguinte conjunto:

{*Annotation, public, protected, private, abstract, static, final, strictfp*}.

ANNOTATION



Não é propriamente um elemento, mas sim uma definição. Sua semântica implementa o conceito de anotações em Java e pode ser substituída por uma anotação padrão ou criada pelo programador.

PUBLIC, PROTECTED E PRIVATE



São os símbolos que veremos quando falarmos de encapsulamento; são modificadores de acesso.

STATIC



Este modificador afeta o ciclo de vida da instância da classe e só pode ser usado em classes membro.

ABSTRACT E FINAL



Já os modificadores *abstract* e *final* relacionam-se com a hierarquia de classes. Todos esses modificadores serão vistos oportunamente.

STRICTFP



Por fim, esse é um modificador que torna a implementação de cálculos de ponto flutuando independentes da plataforma. Sem o uso desse modificador, as operações se tornam dependentes da plataforma sobre a qual a máquina virtual é executada.

É interessante observar que alguns desse modifikadores podem ser compostos com outros. Por exemplo, você pode definir uma classe como `public abstract class Teste {}`.

Outro elemento opcional são os "*TipoParâmetros*". Tais elementos fazem parte da implementação Java da programação genérica e não serão abordados aqui.

O elemento opcional seguinte é a "Superclasse".

Esse parâmetro, assim como o "Superinterface", permite ao Java implementar a herança entre classes e interfaces. O elemento "Superclasse" será sempre do tipo "extends IdentificadorClasse", no qual "extends" (palavra reservada) indica que a classe é uma subclasse de "IdentificadorClasse" (que é um nome de classe). Por sua vez, "IdentificadorClasse" indica uma superclasse da classe.

A sintaxe do elemento "Superinterface" é "*implements IdentificadorInterface*". A palavra reservada "*implements*" indica que a classe implementa a interface "*IdentificadorInterface*".

Vejamos a seguir um exemplo mais completo de declaração de classe em Java.

Java



Objetos: os produtos das classes

Anteriormente mencionamos que as classes eram modelos para a fabricação de objetos. Ao longo da seção anterior, vimos como podemos construir esse modelo, agora veremos como usá-lo.

Nem toda classe permite a criação de um objeto.

Uma classe definida como abstrata não permite a instanciação direta. Não vamos entrar em pormenores de classes abstratas nesse momento, mas é oportuno dizer que ela fornece um modelo de comportamento (e/ou estado) comum a uma coleção de classes. A tentativa de criar um objeto diretamente a partir de uma classe abstrata irá gerar erro de compilação.

Consideraremos, assim, apenas o caso de classes que permitam a sua instanciação.

O que é instanciar?

Instanciar uma classe significa realizar o modelo, e essa realização é chamada de objeto. Para compreender melhor o que é um objeto, vamos analisar seu ciclo de vida.

A criação de um objeto se dá em duas etapas:



Primeiramente, uma variável é declarada como sendo do tipo de alguma classe.



A seguir, o compilador é instruído a gerar um objeto a partir daquela classe, que será rotulado com o identificador que nomeia a variável.

Essas duas etapas podem ocorrer na mesma linha, como no seguinte exemplo:

Java



Criando objeto de uma classe.

Ou esse código pode ser separado nas etapas descritas da seguinte forma:

Java



Declarando e instanciando um objeto.

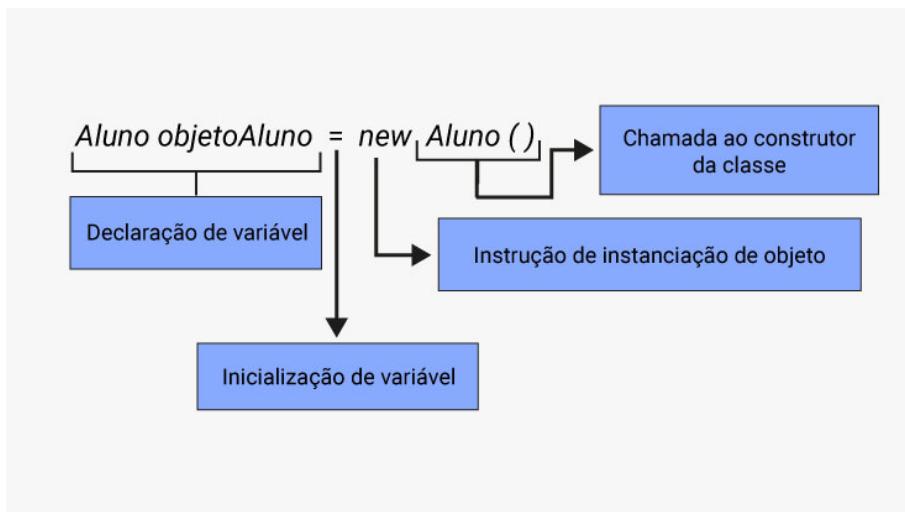
O processo de criação do objeto começa com a alocação do espaço em memória. E prossegue com a execução do código de construção do objeto.

Esse código, obrigatoriamente, deve ser implementado num método especial chamado **construtor**.

O método construtor é sempre executado quando da instanciação de um objeto e obrigatoriamente deve ter nome idêntico ao da classe.

Além disso, ele pode ter um modificador, mas não pode ter tipo de retorno. A instrução “new” é sempre seguida da chamada ao construtor da classe. Finalmente, a atribuição (“=”) inicializa a variável com o retorno (referência para o objeto) de “new”.

Revendo a instanciação, identificamos que ela pode ser decomposta da seguinte forma:



Exemplo de método construtor.

O construtor deve conter o código que precise ser executado quando da criação do objeto. Uma vez que esse código será executado por ocasião da instanciação, fica claro que se tal código for computacionalmente custoso, isso é, exigir muito processamento, a criação do objeto será impactada.

Observe as linhas 12 e 16 do Código 3.

Java



Código 3: Instanciação de objeto com "new".

O que mudaria se a linha 12 fosse suprimida e a linha 16 substituída por “*Random aleatório = new Random ();*”?

Resposta

Nesse caso, a variável “*aleatorio*” seria válida apenas no escopo local do método construtor, não seria um atributo da classe “*Aluno*” e o objeto designado por ela não existiria fora do método “*Aluno (String nome , int idade)*”.

Outra forma de se instanciar um objeto é através da clonagem.

Nesse mecanismo, cópias idênticas do objeto são feitas através da cópia do espaço de memória.

Cada objeto clonado terá um identificador diferente (pois são alocados em diferentes espaços de memória), mas seu conteúdo – isto é, o conteúdo de seus atributos – será igual.

A instanciação através de “new” é mais demorada do que a clonagem, pois executa todos os procedimentos que vimos anteriormente.

Assim, a maior rapidez da clonagem é uma vantagem quando se precisa criar muitos objetos do mesmo tipo.

O estado de um objeto é definido pelos seus atributos, enquanto seu comportamento é determinado pelos seus métodos. Considere a seguinte instanciação a partir da classe do **Código 3**:

Java



Instanciando um objeto.

Após todas as etapas descritas anteriormente serem executadas, haverá um objeto criado e armazenado em memória identificado por “novoAluno”.

O estado desse objeto é definido pelas variáveis “*nome*, *idade*, *codigo* e *aleatorio*”, e seu comportamento é dado pelos métodos “*public void Aluno (String nome , int idade)*”, *public void definirNome (String nome)* e *public void definirIdade (int idade)*”.

Após a instanciação mencionada, o estado do objeto será “*nome = Carlos Alberto*”, “*idade = 16*”. A variável “*codigo*” terá como valor o retorno da função “*nextDouble*”, e a variável “*aleatório*” terá como valor uma referência para o objeto do tipo “*Random*” criado.

O comportamento modelado no construtor permite a construção inicial do objeto. Os métodos “*definirNome*” e “*definirIdade*” permitem, respectivamente, atualizar o nome e a idade. Então, se for executada a chamada “*novoAluno.definirNome (“Carlos Roberto”)*”, a variável “*nome*” será atualizada,

passando a valer “Carlos Roberto”, alterando, assim, o estado do objeto. Repare que a chamada indica ao compilador uma forma inequívoca de invocar o método.

O compilador é informado que está sendo invocado o método “*definirNome*” do objeto “*novoAluno*”.

A última etapa do ciclo de vida de um objeto é sua destruição. Esse processo é necessário para se reciclar o espaço de memória usado. Quando um objeto é destruído, a **JVM** executa diversas operações que culminam com a liberação do espaço de memória usado pelo objeto. Esse é um ponto em que a Java difere de muitas outras linguagens OO.

VM

Java Virtual Machine.

Exemplo

Java não possui o conceito de destrutor. Em outras linguagens, como a C++, o método destrutor é invocado quando um objeto é destruído e a destruição do objeto é feita manualmente pelo programador.

Na linguagem Java, não é possível ao programador manualmente destruir um objeto. Em vez disso, a Java implementa o conceito de coletor de lixo. Periodicamente, a JVM varre o programa verificando objetos que não estejam mais sendo referenciados. Ao encontrar tais objetos, a JVM os destrói e libera a memória. O programador não possui qualquer controle sobre isso.

É possível ao programador, todavia, solicitar à JVM que a coleta de lixo seja realizada. Isso é feito através da invocação do método “*gc ()*” da biblioteca “*System*”.

Essa, porém, é apenas uma solicitação, não é uma ordem de execução, o que significa que a JVM tentará executar a coleta de lixo tão logo quanto possível, mas não necessariamente quando o método foi invocado.

Mas e se o programador desejar executar alguma ação quando um objeto for destruído?

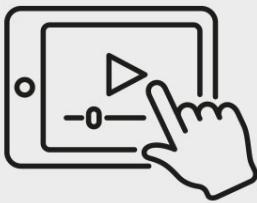
Resposta

Apesar de a Java não implementar o destrutor, ela fornece um mecanismo para esse fim: o método “*finalize ()*”. Este, quando implementado por uma classe, é invocado no momento em que a JVM for reciclar o objeto. Todavia, o método “*finalize ()*” é solicitado quando o objeto estiver para ser reciclado, e não quando ele sair do escopo. Isso implica, segundo Schildt (2014), que o programador não sabe quando – ou mesmo se – o método será executado.

Classes e o encapsulamento de código

Para continuar os seus estudos, assista o vídeo a seguir:

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.



O encapsulamento está intimamente ligado ao conceito de classes e objetos.

Do ponto de vista da POO, o encapsulamento visa ocultar do mundo exterior os atributos e o funcionamento da classe. Dessa maneira, os detalhes de implementação e variáveis ficam isolados do resto do código.

O contato da classe (e, por conseguinte, do objeto) com o resto do mundo se dá por meio dos métodos públicos da classe. Tais métodos formam o chamado **contrato**, que é estabelecido entre a classe e o código que a utiliza.

O conceito de encapsulamento também se liga ao de visibilidade.

A visibilidade de um método ou atributo define quem pode ou não ter acesso a estes. Ou seja, ela afeta a forma como o encapsulamento funciona. Há três tipos de visibilidade, representados pelos modificadores "*private*", "*protected*" e "*public*". Não entraremos em detalhes sobre eles nesse momento, deixando para explorar o assunto quando falarmos de herança e polimorfismo. Mas convém dizer que "*private*" indica que o método ou atributo só pode ser acessado internamente à classe, enquanto "*public*" define que ambos são visíveis para todo o exterior.

Trabalhando com classes e objetos

Assista o vídeo a seguir para aprofundar os seus estudos:

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.



As classes e os objetos são trechos de código que refletem o produto da análise OO e interagem entre si, formando um sistema. Essa interação é a razão pela qual estabelecemos o contrato de uma classe. Esse contrato fornece a interface pela qual outras classes ou outros objetos podem interagir com aquela classe ou seu objeto derivado.

Isso pode ser visto na linha 17 do **Código 3**. Ao invocarmos o método “`nextDouble()`” da classe “`Random`” ao qual o objeto “`aleatorio`” pertence, estamos seguindo o contrato da classe para consumir um serviço prestado por ela mesma. Ainda examinando o Código 3, notamos na linha 10 que a classe “`Aluno`” possui como atributo uma variável do tipo “`Random`” (outra classe). Logo, um objeto do tipo “`Aluno`” conterá um objeto do tipo “`Random`”.

O parágrafo anterior nos mostra uma relação entre objetos. Em OO há diferentes tipos de relações.

ASSOCIAÇÃO

A Associação é semanticamente a mais fraca e se refere a objetos que consomem – usam – serviços ou funcionalidades de outros. Ela pode ocorrer mesmo quando nenhuma classe possui a outra e cada objeto instanciado tem sua existência independente do outro. Essa relação pode ocorrer com cardinalidade 1-1, 1-n, n-1 e n-n.

AGREGAÇÃO

Outro tipo de relacionamento entre classes é a Agregação, que ocorre entre dois ou mais objetos, com cada um tendo seu próprio ciclo de vida, mas com um objeto (pai) contendo os demais (filhos). É importante compreender que, nesse caso, os objetos filhos podem sobreviver à destruição do objeto pai. Um exemplo de Agregação se dá entre escola e aluno: se uma escola deixar de existir, não implica que os alunos irão desaparecer.

COMPOSIÇÃO

A Composição difere sutilmente da Agregação, pois ocorre quando há uma relação de dependência entre o(s) filho(s) e o objeto pai. Ou seja, caso o pai deixe de existir, necessariamente o filho será destruído. Voltando ao exemplo anterior, temos uma composição entre a escola e os departamentos. A extinção da escola traz consigo a extinção dos departamentos.

Os conceitos Associação, Agregação e Composição formam conjuntos que se relacionam, como visto na imagem a seguir. Vemos que a Composição é um caso especial de Agregação e o conceito mais restritivo de todos, enquanto a Associação é o mais abrangente.



Conjunto formado pela definição das relações do tipo Associação, Agregação e Composição.

Essas relações podem ser identificadas no **Código 4**.

Observe as linhas 4, 5 e 6 e os métodos “*criarDepartamento*” e “*matricularAluno*”. Vamos analisá-las.



Código 4: Agregação e Composição.

Primeiro, devemos notar que as linhas 4, 5 e 6 mostram uma Associação entre a classe “Escola” e as classes “Endereco”, “Departamento” e “Aluno”.

Essas linhas apenas declaram variáveis do tipo das classes mencionadas.

A instanciação de objetos ocorre depois.

No caso de “Departamento”, objetos dessa classe são instanciados na linha 21.

A criação de um objeto da classe “Departamento” depende da existência de um objeto da classe “Escola”.

Ou seja, o objeto da classe “Escola” precede a criação de objeto do tipo “Departamento”.

Em segundo lugar, uma vez que o objeto do tipo “Escola” for destruído, necessariamente todos os objetos do tipo “Departamento” também serão destruídos. Isso mostra uma relação forte entre ambas as classes com o ciclo de vida dos objetos de “Departamento” subordinados ao ciclo de vida dos objetos da classe “Escola”, ilustrando uma relação do tipo Composição. É fácil entender essa semântica. Conforme mencionamos anteriormente, basta considerar que, se uma escola deixar de existir, não é possível seus departamentos continuarem funcionando.

Comentário

O caso da classe “Aluno” difere do anterior. Semanticamente, o fechamento de uma escola não causa a morte dos alunos. Estes podem migrar para outro estabelecimento. No caso do Código 4, esse comportamento encontra-se modelado no método “*matricularAluno*”. O que a linha 27 faz é armazenar no vetor *“discentes []”* uma referência para um objeto do tipo “Aluno” criado em outro local. Assim, a criação desse objeto pode ser feita independentemente da instanciação do objeto do tipo “Escola” que o recebe.

Da mesma maneira, a destruição do último não leva necessariamente à destruição do objeto do tipo “Aluno” se este estiver sendo referenciado em outra parte do código. Entretanto, um objeto da classe “Escola” contém objetos do tipo “Aluno”. Trata-se, portanto, de uma relação de Agregação.

Oportunamente essa situação nos permite explorar a referenciação de objetos em Java. Uma referência para um objeto é um valor de endereço de memória que indica a posição em memória na qual um objeto se encontra.

Em outras linguagens, como a C++, esse conceito é comumente implementado por ponteiros, que são variáveis específicas para manipulação de endereços de memória. Por exemplo, uma variável “aux” do tipo inteiro guardará um valor inteiro se fizermos “aux = 0”. Todavia, uma variável ponteiro deve ser usada para guardar o endereço da variável “aux”.

Atenção!

Em Java, não é possível criar variáveis do tipo ponteiro! A linguagem Java oculta esse mecanismo, de forma que toda variável de classe é, de fato, uma referência para o objeto instanciado.

Isso tem implicações importantes na forma de lidar com objetos. A passagem de um objeto como parâmetro em um método, ou o retorno dele, é sempre uma passagem por referência. Isso não ocorre com tipos primitivos, que são sempre passados por valor.

Vejamos o Código 5.



Código 5: Referenciando objetos.

A execução desse código produz a seguinte saída:



Saída do código 5.

Vamos entender o que acontece, seguindo passo a passo a execução do Código 5 a partir da linha 5. Essa linha instrui a JVM instanciar um objeto do tipo Aluno.

Isto é, a JVM reserva um espaço de memória para armazenar um objeto do tipo "Aluno" e o inicializa definindo as variáveis "*nome*" e "*idade*", respectivamente, como "Carlos" e "20".

A linha 6 faz a mesma coisa, instanciando um segundo objeto com "*nome*" recebendo "Ana" e "*idade*" recebendo "23". Após a execução da linha 6, existirão dois objetos distintos ("a1" e "a2"), cujos estados também são distintos.

Por isso, o resultado das linhas 7 e 8 é o mostrado nas saídas 1 e 2, respectivamente.

A linha 9 merece maior atenção. O segredo para entender o que ocorre nela é lembrarmos que "a1" e "a2" são referências para os objetos criados, e não o objeto em si.

Comentário

Para deixar mais claro, quando atribuímos uma variável a outra, estamos atribuindo o seu conteúdo e, no caso, o conteúdo é uma referência para o objeto. Assim, ao fazermos "a2 = a1" estamos fazendo com que "a2" guarde a referência que "a1" possui. Na prática, "a2" passa a referenciar o mesmo objeto referenciado por "a1".

Por isso, na linha 10, ao alterarmos o conteúdo da variável "*nome*" estamos fazendo-o no objeto referenciado por "a1" (e, agora, por "a2").

Por esse motivo, a linha 11 exibe o nome "Flávia". Na verdade, após a linha 9, o uso de "a1" e "a2" é indistinto.

O que acontece na linha 12 não é muito diferente. Como dissemos, a passagem de objetos é sempre feita por referência. Logo, a variável "aluno" na assinatura do método "manipulaAluno" vai receber a referência guardada por "a1". Desse momento em diante, todas as operações feitas usando "aluno" ocorrem no mesmo objeto referenciado por "a1".

Falta pouco para atingir seus objetivos.

Vamos praticar alguns conceitos?

Questão 1

Qual das opções a seguir contém uma declaração de classe válida em linguagem Java?

- A "private class Aluno { }".
- B "class Aluno { }".
- C "protected class Aluno { }".
- D "public Aluno { }".
- E "extends class Aluno { }".

Parabéns! A alternativa B está correta.

Na declaração de uma classe, o modificador "public" é opcional e o único permitido.

Questão 2

Sobre objetos em Java, é correto apenas o que se afirma em:

- A O programador pode determinar o momento exato em que deseja que o objeto seja destruído.

- B Quando um objeto é passado como parâmetro em um método, um clone dele é gerado.
- C O programador não precisa se preocupar em desalocar a memória de um objeto destruído.
- D O método construtor não pode ser privado.
- E O coletor de lixo tem a finalidade de reciclar os objetos destruídos pelo programador.

Parabéns! A alternativa C está correta.

A reciclagem de espaço de memória em Java é feita pelo coletor de lixo.



2 - Herança e Polimorfismo

Ao final deste módulo, você será capaz de descrever o mecanismo de herança e polimorfismo em Java.

Vamos começar!



Herança e Polimorfismo em Java

Para iniciar seus estudos, assista o vídeo a seguir:

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.



Herança

Neste estudo, examinaremos os conceitos de herança e polimorfismo. Começaremos pelos aspectos elementares de herança e, em seguida, vamos aprofundar tal discussão, criando condições para analisarmos o polimorfismo.

Herança: Aspectos elementares

O termo **herança em OO** define um tipo de relação entre objetos e classes, baseado em uma hierarquia.

Dentro dessa relação hierárquica, classes podem herdar características de outras classes situadas acima ou transmitir suas características às classes abaixo.



Uma classe situada hierarquicamente acima é chamada de **superclasse**, enquanto aquelas situadas abaixo chamam-se **subclasses**.



Essas classes podem ser, também, referidas como classe base ou pai (superclasse) e classe derivada ou filha (subclasse).

A herança nos permite reunir os métodos e atributos comuns numa superclasse, que os leva às classes filhas. Isso evita repetir o mesmo código várias vezes.

Outro benefício está na manutenibilidade: caso uma alteração seja necessária, ela só precisará ser feita na classe pai, e será automaticamente propagada para as subclasses.

A imagem a seguir apresenta um diagrama UML que modela uma hierarquia de herança. Nela, vemos que “Empregado” é pai (superclasse) das classes “Professor”, “Diretor”, “Coordenador” e “Secretario”. Essas últimas são filhas (subclasses) da classe “Empregado”.

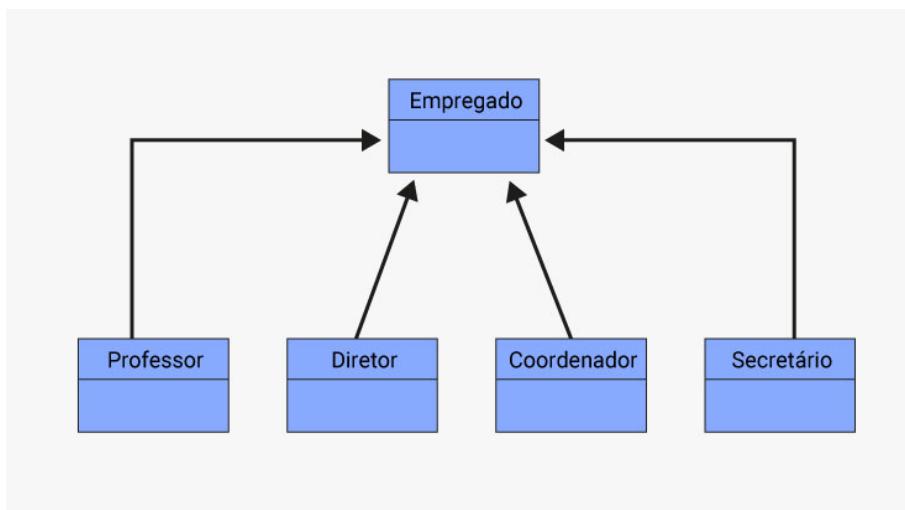


Diagrama de classes – herança.

E quando há mais de uma superclasse?

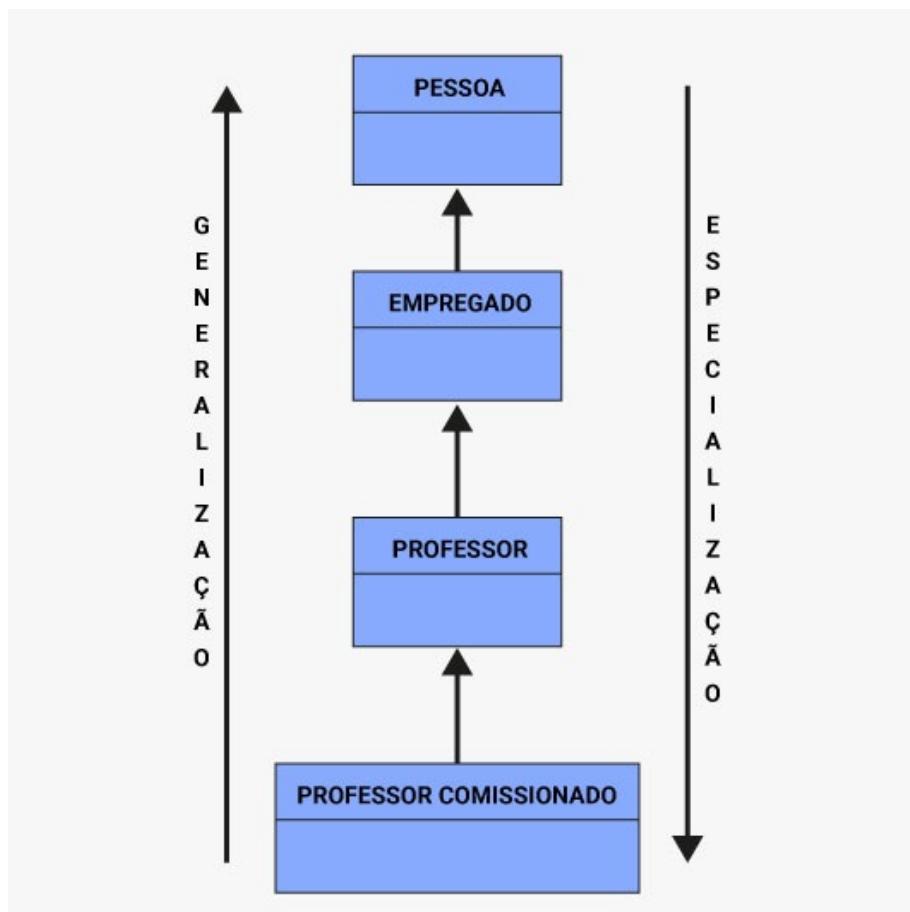
Essa situação é denominada herança múltipla e, apesar de a POO aceitar a herança múltipla, a linguagem Java não a suporta para classes. Veremos as implicações da herança múltipla em outra ocasião, quando explorarmos mais a fundo o conceito de herança.

Comentário

Apesar de não permitir herança múltipla de classes, a linguagem permite que uma classe herde de múltiplas interfaces. Aliás, uma interface pode herdar de mais de uma interface pai. Diferentemente das classes, uma interface não admite a implementação de métodos. Esta é feita pela classe que implementa a interface.

Examinando a genealogia das classes da imagem seguinte, podemos notar que, à medida que descemos na hierarquia, lidamos com classes cada vez mais específicas. De forma oposta, ao subirmos na hierarquia, encontramos classes cada vez mais gerais.

Essas características correspondem aos conceitos de generalização e especialização da OO.



Herança com vários níveis de ancestralidade.

A sintaxe para declarar uma superclasse já foi mostrada anteriormente. O código a seguir mostra a implementação da herança, na imagem anterior, para a classe “ProfessorComissionado”.

Java



Código 6: Herança em classes.

No **Código 6**, podemos observar que a herança é declarada apenas para a classe ancestral imediata.

Em outras palavras:



A classe “**Professor**” deve declarar “**Empregado**” como sua superclasse.



“**Empregado**” deve declarar “**Pessoa**” como sua superclasse.

A sintaxe é análoga para o caso das interfaces, exceto que nesse caso pode haver mais de um identificador de superinterface. O código a seguir mostra um exemplo baseado na imagem Diagrama de classes – herança, considerando que “**ProfessorComissionado**”, “**Professor**” e “**Diretor**” sejam interfaces.

Nesse exemplo, a herança múltipla pode ser vista pela lista de superinterfaces (“**Professor**” e “**Diretor**”) que se segue ao modificador “*extends*”.



Código 7: Herança em interfaces.

Algo interessante de se observar é que em Java todas as classes descendem direta ou indiretamente da classe “Object”. Isso torna os métodos da classe “Object” disponíveis para qualquer classe criada. O método “*equals ()*”, da classe “Object”, por exemplo, pode ser usado para comparar dois objetos da mesma classe.



Se uma classe for declarada sem estender nenhuma outra, então o compilador assume implicitamente que ela estende a classe “Object”.



Se ela estender uma superclasse, como no código, então ela é uma descendente indireta de “Object”.

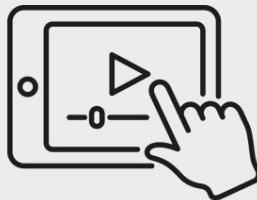
Isso fica claro se nos remetermos à imagem **Herança com vários níveis de ancestralidade**. Nesse caso, a classe “Pessoa” estende implicitamente “Object”, então os métodos desta são legados às subclasses até a

base da hierarquia de classes. Logo, um objeto da classe “ProfessorComissionado” terá à sua disposição os métodos de “Object”.

Herança e visibilidade

Agora, assista o vídeo a seguir:

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.



Quando dizemos que a classe “Pessoa” é uma generalização da classe “Empregado”, isso significa que ela reúne atributos e comportamento que podem ser generalizados para outras subclasses. Esses comportamentos podem ser especializados nas subclasses. Isto é, as subclasses podem sobrescrever o comportamento modelado na superclasse. Nesse caso, a assinatura do método pode ser mantida, mudando-se apenas o código que o implementa.

Neste momento, vamos entender o funcionamento desse mecanismo para compreendermos o que é polimorfismo.

Primeiramente, precisamos compreender como os modificadores de acesso operam. Já vimos que tais modificadores alteram a acessibilidade de classes, métodos e atributos. Há quatro níveis de acesso em Java:

DEFAULT

Assumido quando nenhum modificador é usado. Define a visibilidade como restrita ao pacote.

PRIVADO

Declarado pelo uso do modificador “*private*”. A visibilidade é restrita à classe.

PROTEGIDO

Declarado pelo uso do modificador “*protected*”. A visibilidade é restrita ao pacote e a todas as subclasses.

PÚBLICO

Declarado pelo uso do modificador “*public*”. Não há restrição de visibilidade.

Os modificadores de acessibilidade ou visibilidade operam controlando o escopo no qual se deseja que os elementos (classe, atributo ou método) aos quais estão atrelados sejam visíveis.

O maior escopo é o **global**, que, como o próprio nome indica, abarca todo o programa. Outro escopo é definido pelo pacote.

Um pacote define um espaço de nomes e é usado para agrupar classes relacionadas.

Esse conceito contribui para a melhoria da organização do código de duas formas: permite organizar as classes pelas suas afinidades conceituais e previne conflito de nomes.

Comentário

Evitar conflitos de nomes parece fácil quando se trabalha com algumas dezenas de entidades, mas esse trabalho se torna desafiador num software que envolva diversos desenvolvedores e centenas de entidades e funções.

Em Java, um pacote é definido pela instrução “*package*” seguida do nome do pacote inserida no arquivo de código-fonte. Todos os arquivos que contiverem essa instrução terão suas classes agrupadas no pacote. Isso significa que todas essas classes, isto é, classes do mesmo pacote, terão acesso aos elementos que tiverem o modificador de acessibilidade “*default*”.

Saiba mais

O modificador “*private*” é o mais restrito, pois limita a visibilidade ao escopo da classe. Isso quer dizer que um atributo ou método definido como privado não pode ser acessado por qualquer outra classe senão

aquela na qual foi declarado. Isso é válido mesmo para classes definidas no mesmo arquivo e para as subclasses.

O acesso aos métodos e atributos da superclasse pode ser concedido pelo uso do modificador “*protected*”. Esse modificador restringe o acesso a todas as classes do mesmo pacote. Classes de outros pacotes têm acesso apenas mediante herança.

Finalmente, o modificador de acesso “*public*” é o menos restritivo. Ele fornece acesso com escopo global. Isto é, qualquer classe do ambiente de desenvolvimento pode acessar as entidades declaradas como públicas.

A tabela a seguir sumariza a relação entre os níveis de acesso e o escopo.

	default	public	private	protected	scope
Subclasse do mesmo pacote	sim	sim	não		s
Subclasse de pacote diferente	não	sim	não		s
Classe (não derivada) do mesmo pacote	sim	sim	não		s
Classe (não derivada) de pacote diferente	não	sim	não		r

Níveis de acesso e escopo.

Marlos de Mendonça.

As restrições impostas pelos modificadores de acessibilidade são afetadas pela herança da seguinte maneira:

- 1) Métodos (e atributos) declarados públicos na superclasse devem ser públicos nas subclasses.
- 2) Métodos (e atributos) declarados protegidos na superclasse devem ser protegidos ou públicos nas subclasses. Eles não podem ser privados.

Atenção!

Métodos e atributos privados não são acessíveis às subclasses, e sua acessibilidade não é afetada pela herança.

Para entender melhor, examinaremos parte do código que implementa o modelo da imagem a seguir:

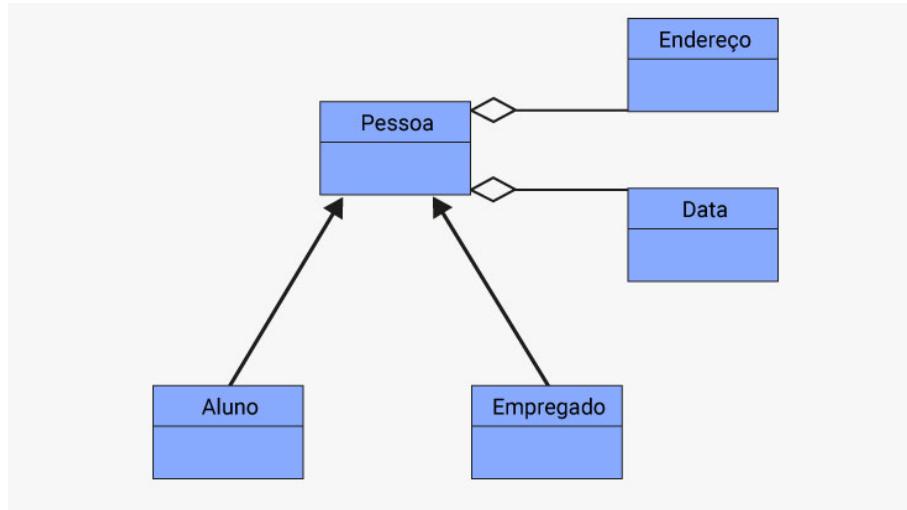


Diagrama de classes parcial de um sistema.

Como podemos observar no diagrama de classes exibido na imagem anterior, a classe “Pessoa” generaliza as classes “Empregado” e “Aluno”.

Os códigos que implementam as classes “Pessoa”, “Empregado”, “Aluno” e a classe principal do programa são mostrados respectivamente nos Código 8, Código 9, Código 10 e Código 11.



Código 8: Classe "Pessoa".

Java



Na sequência temos o código que implementa a classe "Empregado":

Código 9: Classe "Empregado".

A classe "Aluno" é implementada no código. Veja a seguir:

Java



Código 10: Classe "Aluno".

Por fim temos o código da classe "Principal", como podemos conferir a seguir:

Java



Código 11: Classe Principal (código parcial).

As linhas 1 do Código 9 e do Código 10 declaram que as respectivas classes têm "Pessoa" como superclasse. Podemos notar que a classe "Pessoa" contém atributos que são comuns às subclasses, assim como os métodos para manipulá-los.

Outra coisa que podemos ver no Código 8 é que a classe "Pessoa" possui um construtor não vazio. Assim, os construtores das classes derivadas precisam passar para a superclasse os parâmetros exigidos na assinatura do construtor. Isso é feito pela instrução "super", que pode ser notada na linha 7 das classes "Empregado" e "Aluno".

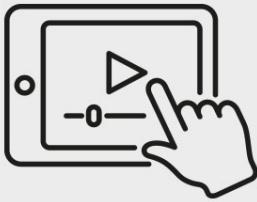
Dissemos anteriormente que os métodos e atributos privados não são acessíveis às subclasses, mas se observarmos as linhas 19 e 20 do código 11, veremos que os atributos "nome, idade, data_nascimento, CPF e endereço", que são privados, são definidos na instanciação da subclasse "Aluno" e imediatamente após. De fato, esses atributos não são herdados, mas a herança define uma relação "é tipo" ou "é um". Ou seja, um objeto do tipo "Aluno" é também do tipo "Pessoa" (o inverso não é verdade).

Uma vez que foram fornecidos métodos protegidos capazes de manipular tais atributos, estes podem ser perfeitamente alterados pela subclasse. Em outras palavras, uma subclasse possui todos os atributos e métodos da superclasse, mas não tem visibilidade daqueles que são privados. Então, podemos entender que a subclasse herda aquilo que lhe é visível (ou acessível). Por isso, a subclasse "Aluno" é capaz de usar o método privado "calcularIdade ()" (linha 41 do Código 8) da superclasse. Mas ela o faz através da invocação do método protegido "atualizarIdade()", como vemos na linha 20 da classe "Principal".

Polimorfismo

Para continuar os seus estudos, assista o vídeo a seguir:

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.



Em OO, **polimorfismo** é a capacidade de um objeto se comportar de diferentes maneiras.

Mais uma vez convém destacar que se trata de um princípio de OO que Java implementa, assim como várias linguagens OO.

O polimorfismo pode se expressar de diversas maneiras. A sobrecarga de função, assim como a herança, são formas de dar ao objeto uma capacidade polimórfica. No caso da herança, o polimorfismo surge justamente porque um objeto pode se comportar também como definido na superclasse.

Exemplo

Considere um objeto do tipo "Aluno". Como vimos, todo objeto do tipo "Aluno" é do tipo "Pessoa". Logo, ele pode se comportar como "Aluno" ou como "Pessoa".

Todo objeto que possui uma superclasse tem capacidade de ser polimórfico. Por essa razão, todo objeto em Java é polimórfico, mesmo que ele não estenda explicitamente outra classe. A justificativa, como já dissemos, é que toda classe em Java descende direta ou indiretamente da classe "Object".

O polimorfismo permite o desenvolvimento de códigos facilmente extensíveis, pois novas classes podem ser adicionadas com baixo impacto para o restante do software. Basta que as novas classes sejam derivadas daquelas que implementam comportamentos gerais, como no caso da classe "Pessoa".

Essas novas classes podem especializar os comportamentos da superclasse, isto é, alterar a sua implementação para refletir sua especificidade, e isso não impactará as demais partes do programa que se valem dos comportamentos da superclasse.

O **código 12** mostra a classe “Diretor”, que é subclasse de “Empregado”, e o **código 13** mostra a classe “Principal” modificada.

Java



Código 12: Classe "Diretor".

A seguir, a implementação da Classe "Principal" alterada:

Java



Código 13: Classe "Principal" alterada.

A execução desse código produz como saída:

A matrícula do Diretor é: E-096d9a3d-98e9-4af1-af61-a03d97525429A matrícula do Empregado é: Matrícula não definida.

Observe que estamos invocando o método “*gerarMatricula ()*” com referências do tipo da superclasse (vide linha 3 do Código 13). A variável diretor, porém, está se referindo a um objeto da subclasse (vide linha 12 do código 13) e o método em questão possui uma versão especializada na classe “Diretor” (ela sobrescreve o método “*gerarMatricula ()*” da superclasse), conforme observamos na linha 8 do código 12. Dessa maneira, durante a execução, o método da subclasse será chamado.

Outra forma de polimorfismo pode ser obtida por meio da **sobrecarga de métodos**.

A sobrecarga é uma característica que permite que métodos com o mesmo identificador, mas diferentes parâmetros, sejam implementados na mesma classe.

Ao usar parâmetros distintos em número ou quantidade, o programador permite que o compilador identifique qual método chamar. Veja o exemplo do **código 14**.



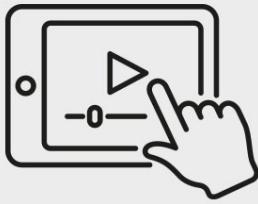
Código 14: Polimorfismo por sobrecarga de método.

Nesse caso, uma chamada do tipo “`alterarMatricula ()`” invocará o método mostrado na linha 8 do **Código 14**. Caso seja feita uma chamada como “`alterarMatricula (“M-202100-1000”)`”, o método chamado será o da linha 14, pois o parâmetro é uma “`String`”.

Polimorfismo e classes abstratas

Assista o vídeo a seguir para aprofundar os seus estudos:

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.



Falta pouco para atingir seus objetivos.

Vamos praticar alguns conceitos?

Questão 1

Sobre herança em Java, é correto afirmar apenas que:

- A Um atributo protegido da superclasse não é visível para a subclasse.

- B Um objeto instanciado da subclasse é também um objeto do tipo da superclasse.
- C A superclasse herda os métodos e atributos públicos da subclasse.
- D Uma superclasse só pode ter uma subclasse.
- E Um objeto instanciado da superclasse é também um objeto do tipo da subclasse.

Parabéns! A alternativa B está correta.

O mecanismo de herança dá subclasse à mesma estrutura da superclasse.

Questão 2

Em um software Java, uma classe chamada “Painel” tem a classe derivada “LCD”, que sobrecarrega um método “acender” de “Painel”. O método é protegido em ambas as classes. A única opção que possui uma afirmativa correta é:

- A O método de “LCD” sobrescreve o de “Painel”.
- B Um objeto do tipo “Painel” pode usar ambas as versões do método “acender”.
- C Trata-se de um caso de polimorfismo.
- D Um objeto do tipo “LCD” só tem acesso ao método “acender” da subclasse.
- E Uma classe derivada de “LCD” terá apenas a versão sobreescrita de “acender”.

Parabéns! A alternativa C está correta.

LCD herdará o método “acender” de “Painel”, ficando com duas versões, o que caracteriza um tipo de polimorfismo.



3 - Agrupamento de Objetos

Ao final deste módulo, você será capaz de descrever os mecanismos de agrupamento de objetos em Java.

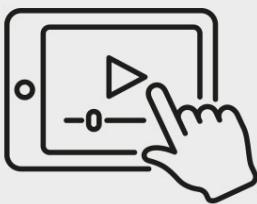
Vamos começar!



Agrupamentos e Coleções em Java

Para iniciar seus estudos, assista o vídeo a seguir:

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.



Agrupamento de objetos em Java

O propósito do agrupamento é permitir que, a partir de um universo de objetos, grupos de objetos afins sejam estabelecidos com base em determinado critério. Não se trata de um conceito de OO. Na verdade, é um conceito comum na interação com banco de dados. A cláusula “GROUB BY” da SQL, usada nas consultas ao banco de dados, faz exatamente isso. Esse é um dos motivos para o agrupamento nos interessar: a interação com banco de dados.

Comentário

A lógica do agrupamento de objetos é simples: dado um universo de objetos e os critérios de particionamento, separá-los em grupos tais que em cada grupo todos os objetos possuam o mesmo valor para os critérios escolhidos. Se a lógica é simples, a implementação tem nuances que precisam ser observadas, como a checagem de referência para evitar acessos ilegais (*NullPointerException*). Isso porque, para fazer o particionamento, teremos de utilizar uma lista de listas ou outra estrutura de dados similar.

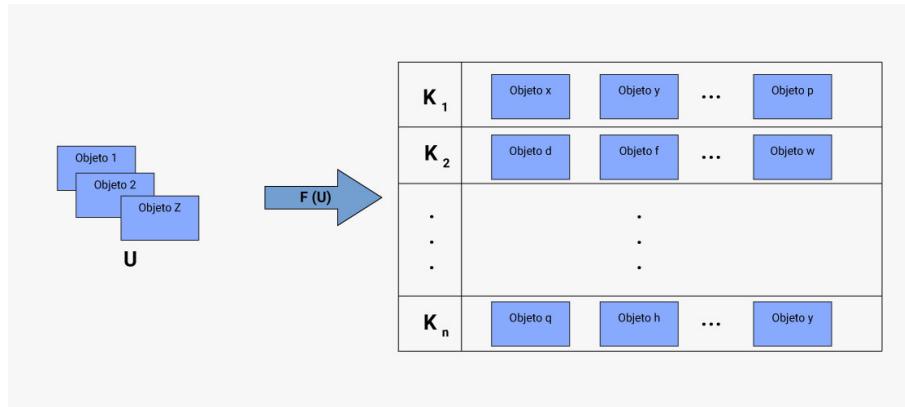
Apesar dos pontos levantados, é perfeitamente possível implementar o particionamento de objetos. Isso pode ser feito em poucas linhas (se você se valer das estruturas oferecidas pela API Java). De fato, essa era a maneira de se agrupar objetos até o lançamento da Java 8. A partir daí, a API Java passou a oferecer um mecanismo de agrupamento, simplificando o trabalho de desenvolvimento.

Implementando o agrupamento de objetos

No agrupamento, o estado final desejado é ter os objetos agrupados, e cada agrupamento deve estar mapeado para a chave usada como critério. Em outras palavras, buscamos construir uma função tal que, a

partir de um universo de objetos de entrada, tenha como saída “n” pares ordenados formados pela chave de particionamento e a lista de objetos agrupados: $F(U) = \{ [k_1, , [k_2, , \langle \text{lista agrupada} \rangle, \dots, [k_n, , \langle \text{lista agrupada} \rangle] \}$.

A próxima imagem dá uma ideia do que se pretende.



Representação gráfica do agrupamento de objetos.

Felizmente, a Java API oferece estruturas que facilitam o trabalho. Para manter e manipular os objetos usaremos o container “List”, que cria uma lista de objetos. Essa estrutura já possui métodos de inserção e remoção e pode ser expandida ou reduzida conforme a necessidade.

Para mantermos os pares de particionamento, usaremos o container “Map”, que faz o mapeamento entre uma chave e um valor. No nosso caso, a chave é o critério de particionamento e o valor é a lista de objetos particionados, exatamente conforme vemos na imagem anterior. A estrutura “Map”, além de possuir métodos que nos auxiliarão, não permite a existência de chaves duplicadas. Vejamos, então, um exemplo.



Código 15: Classe "Aluno" modificada.

Java



Código 16: Classe "Escola" – agrupamento de objetos pelo programador.

Código 16.

Código 17: Classe Principal.

Java



Código 17.

É o **Código 16** que merece nossa atenção. Na linha 6, é criada uma lista de objetos do tipo *Aluno*. Para aplicarmos nossa função de agrupamento, precisaremos varrer essa lista, fazendo a separação de cada objeto encontrado segundo o critério de agrupamento.

O método de agrupamento encontra-se na linha 27. Na linha 28, temos a declaração de uma estrutura do tipo “*Map*” e a instanciação da classe pelo objeto “*agrupamentoPorNaturalidade*”. Podemos observar que será mapeado um objeto do tipo “*String*” a uma lista de objetos do tipo “*Aluno*” (“*Map < String , List < Aluno >*”).

Em seguida, na linha 29, o laço implementa a varredura sobre toda a lista. A cada iteração, o valor da variável “*naturalidade*” é recuperado, e a função “*containsKey*” verifica se a chave já existe no mapa. Se não existir, ela é inserida. A linha 30, finalmente, adiciona o objeto à lista correspondente à chave existente no mapa.

Clique e veja a saída:



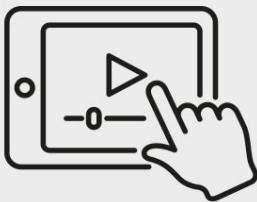
```
Resultado do agrupamento por naturalidade: {  
São Paulo=[Marcio Gomes(São Paulo), César Augusto(São Paulo),  
Castelo Branco(São Paulo)], Rio de Janeiro=[Marco Antônio(Rio de Janeiro),  
Clara Silva(Rio de Janeiro)], Madri=[Alejandra Gomez(Madri)],  
Sorocaba=[Marcos Cintra(Sorocaba), João Carlos(Sorocaba)],  
Barra do Píraí=[Ana Beatriz(Barra do Píraí)]  
}
```

Vemos que nossa função agrupou corretamente os objetos. A chave é mostrada à esquerda do sinal de “=” e, à direita, entre colchetes, estão as listas de objetos, nas quais cada objeto encontra-se separado por vírgula.

Agrupando objetos com a classe “collectors” da API Java

Agora, assista o vídeo a seguir:

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.



A classe “*Collectors*” é uma classe da API Java que implementa vários métodos úteis de operações de redução, como, por exemplo, o agrupamento dos objetos em coleções, de acordo com a Oracle America Inc. (2021). Com o auxílio da API Java, não precisaremos varrer a lista de objetos. Vamos transformar esses objetos em um fluxo (“*stream*”) que será lido pela classe “*Collectors*”, a qual realizará todo o trabalho de classificação dos objetos, nos devolvendo um mapeamento como na situação anterior.

A operação de agrupamento é feita pelo método “*groupingBy*”.

Esse método possui três variantes sobrecarregadas, cujas assinaturas exibimos a:

Java



Operação de agrupamento: *groupingBy*.

O agrupamento da classe “*Collectors*” usa uma função de classificação que retorna um objeto classificador para cada objeto no fluxo de entrada. Esses objetos classificadores formam o universo de chaves de particionamento. Isto é, são os rótulos ou as chaves de cada grupo ou coleção de objetos formados.

Conforme o agrupador da classe “*Collectors*” vai lendo os objetos do fluxo de entrada, ele vai criando coleções de objetos correspondentes a cada classificador. O resultado é um par ordenado (Chave, Coleção), que é armazenado em uma estrutura de mapeamento “*Map*”.

Na assinatura 1, identificamos com mais facilidade que o método “*groupingBy*” recebe como parâmetro uma referência para uma função capaz de mapear T em K.

A cláusula “*static Collector>>*” é o método (“*Collector*”) que retorna uma estrutura “*Map*”, formada pelos pares “K” e uma lista de objetos “T”. “K” é a chave de agrupamento e “T”, obviamente, um objeto agrupado. Então a função cuja referência é passada para o método “*groupingBy*” é capaz de mapear o objeto na chave de agrupamento.

A modificação trazida pela segunda assinatura é a possibilidade de o programador decidir como a coleção será criada na estrutura de mapeamento. Ele pode decidir usar outras estruturas ao invés de “*List*”, como a “*Set*”, por exemplo.

A terceira versão é a mais genérica. Nela, além de poder decidir a estrutura utilizada para implementar as coleções, o programador pode decidir sobre qual mecanismo de “*Map*” será utilizado para o mapeamento.

Vejamos o código 18, que é uma versão modificada do Nessa nova versão, o método “*agruparAlunos*” foi reescrito usando a primeira assinatura de “*groupingBy*”.

Java



Código 18: Agrupamento com o uso do método "groupingBy" (assinatura 1).

A execução desse código produz a saída a seguir:

Resultado do agrupamento por naturalidade:

São Paulo = [Marcio Gomes(São Paulo), César Augusto(São Paulo), Castelo Branco(São Paulo)]

Rio de Janeiro = [Marco Antônio(Rio de Janeiro), Clara Silva(Rio de Janeiro)]

Madri = [Alejandra Gomez(Madri)]

Sorocaba = [Marcos Cintra(Sorocaba), João Carlos(Sorocaba)]

Barra do Píraí = [Ana Beatriz(Barra do Píraí)]

Comparando essa saída com a produzida pelo **Código 16**, podemos perceber que os agrupamentos de objetos são rigorosamente os mesmos.

Analisemos a linha 28 do **Código 18**.

Inicialmente, identificamos que “agrupamento” é uma estrutura do tipo “Map” que armazena pares do tipo “String” e lista de “Aluno” (“List<Aluno>”).

O objeto “String” é a chave de agrupamento, enquanto a lista compõe a coleção de objetos.

A seguir, vemos o uso do método “stream”, que produz um fluxo de objetos a partir da lista “discentes” e o passa para “Collectors”.

Por fim, “groupingBy” recebe uma referência para o método que permite o mapeamento entre o objeto e a chave de agrupamento.

No nosso exemplo, esse método (“Aluno::recuperarNaturalidade”) é o que retorna o valor da variável “naturalidade” dos objetos alunos. Na prática, estamos agrupando os alunos pela sua naturalidade. Essa função é justamente a que mapeia o objeto “Aluno” à sua naturalidade (chave de agrupamento).

Vejamos agora o uso das demais assinaturas. Por simplicidade, iremos mostrar apenas a sobrecarga do método “agruparAluno”, uma vez que o restante da classe permanecerá inalterado.

Java



Código 19: Agrupamento com o uso do método “groupingBy” (assinatura 2).

Podemos ver que o **Código 19** utiliza uma estrutura do tipo “Set”, em vez de “List”, para criar as coleções. Consequentemente, o método “groupingBy” passou a contar com mais um argumento – “Collectors.toSet()” – que retorna um “Collector” que acumula os objetos em uma estrutura “Set”. A saída é a mesma mostrada para a execução do **código 18**.

O **Código 20** mostra uma sobrecarga do método “agruparAlunos”, que usa a terceira assinatura de “groupingBy”.



Código 20: Agrupamento com o uso do método "groupingBy" (assinatura 3).

A diferença sintática para a segunda assinatura é apenas a existência de um terceiro parâmetro no método "groupingBy": "TreeMap::new". Esse parâmetro vai instruir o uso do mecanismo "TreeMap" na instanciação de "Map" ("agrupamento").

Veja agora a saída desse código:

Resultado do agrupamento por naturalidade:

Barra do Pirai = [Ana Beatriz(Barra do Pirai)]

Madri = [Alejandra Gomez(Madri)]

Rio de Janeiro = [Clara Silva(Rio de Janeiro), Marco Antônio(Rio de Janeiro)]

Sorocaba = [João Carlos(Sorocaba), Marcos Cintra(Sorocaba)]

São Paulo = [Castelo Branco(São Paulo), César Augusto(São Paulo), Marcio Gomes(São Paulo)]

Podemos notar que a ordem das coleções está diferente do caso anterior. Isso porque o mecanismo "TreeMap" mantém as suas entradas ordenadas. Todavia, podemos perceber que os agrupamentos são iguais.

Uma observação sobre o método "groupingBy" é que ele não é otimizado para execuções concorrentes. Caso você precise trabalhar com agrupamento de objetos e concorrência, a API Java fornece uma versão

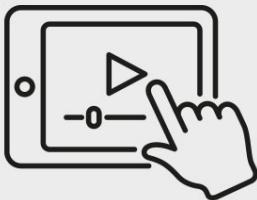
apropriada para esse caso.

Todas as três variantes possuem uma contraparte chamada “*groupingByConcurrent*”, destinada ao uso num ambiente *multithread*. As assinaturas, os parâmetros e o retorno – e, portanto, o uso – são exatamente os mesmos que na versão para desenvolvimento não paralelizado.

Coleções

Para continuar os seus estudos, assista o vídeo a seguir:

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.



Coleções, por vezes chamadas de *Containers*, são objetos capazes de agrupar múltiplos elementos em uma única unidade.

Sua finalidade é armazenar, manipular e comunicar dados agregados, de acordo com o Oracle America Inc., (2021). Neste módulo, nós as usamos (“*List*” e “*Set*”) para armazenar os agrupamentos criados, independentemente dos métodos que empregamos.

Ainda de acordo com a Oracle America Inc. (2021), a API Java provê uma interface de coleções chamada *Collection Interface*, que encapsula diferentes tipos de coleção: “*Set*”, “*List*”, “*Queue* e “*Deque*” (“*Map*” não é verdadeiramente uma coleção). Há, ainda, as coleções “*SortedSet*” e “*SortedMap*”, que são, essencialmente, versões ordenadas de “*Set*” e “*Map*”, respectivamente.

As interfaces providas são genéricas, o que significa que precisarão ser especializadas. Vimos isso ocorrer na linha 2 do 20, por exemplo, quando fizemos “*Set < Aluno>*”. Nesse caso, instruímos o compilador a especializar a coleção “*Set*” para a classe “*Aluno*”.

Cada coleção encapsulada possui um mecanismo de funcionamento, apresentado brevemente a seguir:

SET



Trata-se de uma abstração matemática de conjuntos. É usada para representar conjuntos e não admite elementos duplicados.

LIST

Implementa o conceito de listas e admite duplicidade de elementos. É uma coleção ordenada e permite o acesso direto ao elemento armazenado, assim como a definição da posição onde armazená-lo. O conceito de Vetor fornece uma boa noção de como essa coleção funciona.

QUEUE

Embora o nome remeta ao conceito de filas, trata-se de uma coleção que implementa algo mais genérico. Uma “Queue” pode ser usada para criar uma fila (FIFO), mas também pode implementar uma lista de prioridades, na qual os elementos são ordenados e consumidos segundo a prioridade e não na ordem de chegada. Essa coleção admite a criação de outros tipos de filas com outras regras de ordenação.

DEQUE

Implementa a estrutura de dados conhecida como Deque (*Double Ended Queue*). Pode ser usada como uma fila (FIFO) ou uma pilha (LIFO). Admite a inserção e a retirada em ambas as extremidades.

Como dissemos, “Map” não é verdadeiramente uma coleção. Esse tipo de classe cria um mapeamento entre chaves e valores, conforme vimos nos diversos exemplos da subseção anterior. Além de não admitir chaves duplicadas, a interface “Map” mapeia uma chave para um único valor.

Se observarmos a linha 2 do Código 20 por exemplo, veremos que a chave é mapeada em um único objeto que, no caso, é uma lista de objetos “Aluno”. Uma tabela hash fornece uma boa noção do funcionamento de “Map”.

Não é nosso propósito examinar em detalhe cada uma dessas coleções. A melhor forma de aprender sobre elas é consultando a documentação da API, mas uma noção de seu funcionamento foi mostrada nos

exemplos anteriores. As linhas 4 do código 20 e 5 do código 19 mostram formas de percorrer as estruturas.

Comentário

Repare que o método “foreach”, no primeiro caso, percorre todo o mapeamento sem que seja necessária a criação de um laço. Da mesma maneira, no segundo exemplo, nenhum controle de iteração é necessário para que o laço funcione e percorra todos os elementos da lista. Esses são apenas alguns exemplos das funcionalidades providas por essas estruturas.

Na linha 31 do 16, o método “put” é utilizado para inserir um elemento no mapeamento. A linha 33 desse código é ainda mais interessante: o método “get” é usado para recuperar a referência do objeto mapeado pela chave passada com parâmetro em “get”. Esse objeto é uma lista, que utiliza o método “add” para adicionar o novo elemento.

Falta pouco para atingir seus objetivos.

Vamos praticar alguns conceitos?

Questão 1

Avalie as seguintes afirmações feitas acerca da linguagem Java:

- I. Ao utilizar o método “groupingBy” da classe “Collectors”, o programador tem de informar o atributo a ser usado para o agrupamento.
- II. Os objetos agrupados são armazenados em um container que é mapeado para a chave de agrupamento.
- III. O método “groupingBy” só armazena os objetos em coleções do tipo “List”.

Está correto apenas o que se afirma em:

A I

B

II

C

III

D

I e II.

E

II e III.

Parabéns! A alternativa B está correta.

O retorno do método “groupingBy” é um “Collector” que cria uma estrutura “Map”. Essa estrutura mantém um mapeamento entre a chave de agrupamento e o container que contém os objetos agrupados. As demais afirmativas são falsas.

Questão 2

Escolha a única alternativa verdadeira:

A

As coleções em Java não admitem elementos duplicados.

B

O container “Queue” é uma fila FIFO.

C

Uma pilha pode ser implementada com o container “Deque”.

D

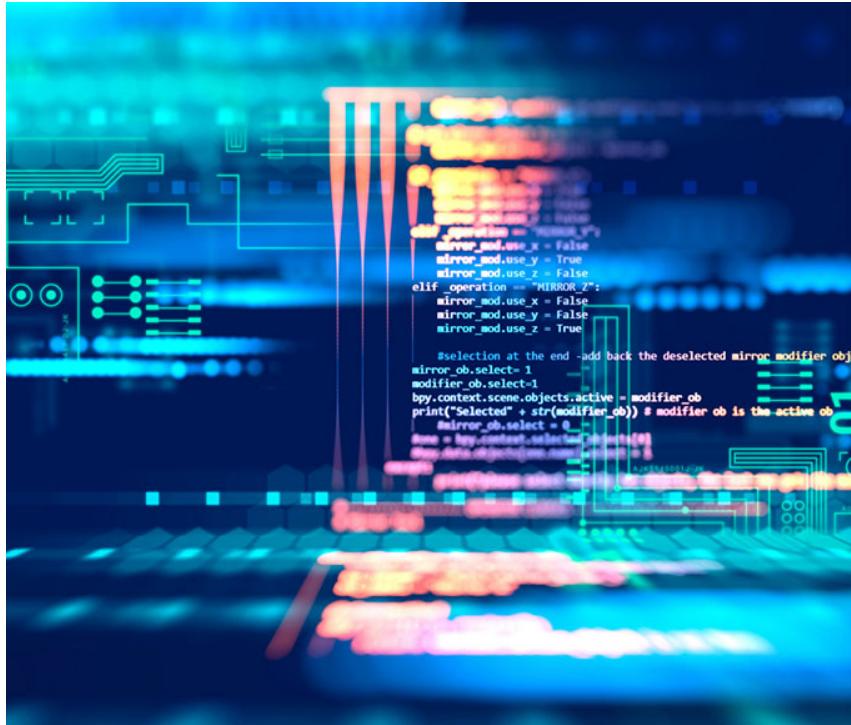
Nenhum container permite definir a posição do objeto a ser inserido.

E

Os containers não podem ser usados em programação concorrente.

Parabéns! A alternativa C está correta.

O container “Deque” possibilita inserção e remoção em ambas as extremidades, o que permite implementar uma pilha.



4 - Ambientes de Desenvolvimento

Ao final deste módulo, você será capaz de reconhecer os ambientes de desenvolvimento em Java e as principais estruturas da linguagem.

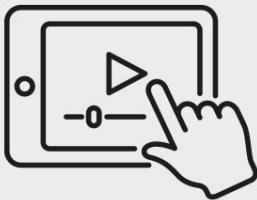
Vamos começar!



Paradigmas de Programação e Ambientes de Desenvolvimento em Java

Para iniciar seus estudos, assista o vídeo a seguir:

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.



Java versus C/C++: um breve comparativo

Neste estudo, falaremos sobre ambientes de desenvolvimento Java. Começaremos abordando alguns aspectos comerciais e prosseguiremos fazendo uma breve comparação com a linguagem C/C++. Em seguida, apresentaremos os ambientes de desenvolvimentos – sem esgotá-los –, alguns *Integrated Development Environment* (IDE) e a estrutura e os principais comandos de um programa Java.

Comentário

Segundo seu criador, Bjarne Stroustrup (2020), a linguagem C++ e a Java não deveriam ser comparadas, pois têm condicionantes de desenvolvimento distintas. Stroustrup aponta, também, que muito da simplicidade de Java é uma ilusão e que, ao longo do tempo, a linguagem se tornará cada vez mais dependente de implementações.

As diferenças mencionadas por Stroustrup se mostram presentes, por exemplo, no fato de a C++ ter uma checagem estática fraca, enquanto a Java impõe mais restrições. A razão de C++ ser assim é a performance ao interfacear com o hardware, enquanto Java privilegia um desenvolvimento mais seguro contra erros de programação. Nesse sentido, James Gosling, um dos desenvolvedores da Java, aponta a checagem de limites de arrays, feita pela Java, como um ponto forte no desenvolvimento de código seguro, algo que a C++ não faz, conforme explica Sutter (2021).

Talvez um dos pontos mais interessantes seja a questão da portabilidade. A C++ buscou alcançá-la através da padronização.

Em 2021, a linguagem C++ segue o padrão internacional ISO/IEC 14882:2020, definido pelo *International Organization for Standardization*, e possui a *Standard Template Library*, uma biblioteca padrão que oferece várias funcionalidades adicionais como *containers*, algoritmos, iteradores e funções.

Logo, um programa C++, escrito usando apenas os padrões e as bibliotecas padrões da linguagem, em tese, possui portabilidade.

Saiba mais

A portabilidade é um dos pontos fortes da Java.

Todavia, conforme Stroustrup (2020) aponta:

A Java não é uma linguagem independente de plataforma, ela é uma plataforma!

O motivo dessa afirmação é que um software Java não é executado pelo hardware, mas sim pela Máquina Virtual Java, que é uma plataforma emulada que abstrai o hardware subjacente. A JVM expõe sempre para o programa a mesma interface, enquanto ela própria é um software acoplado a uma plataforma de hardware específica. Se essa abordagem permite ao software Java rodar em diferentes hardwares, ela traz considerações de desempenho que merecem atenção.

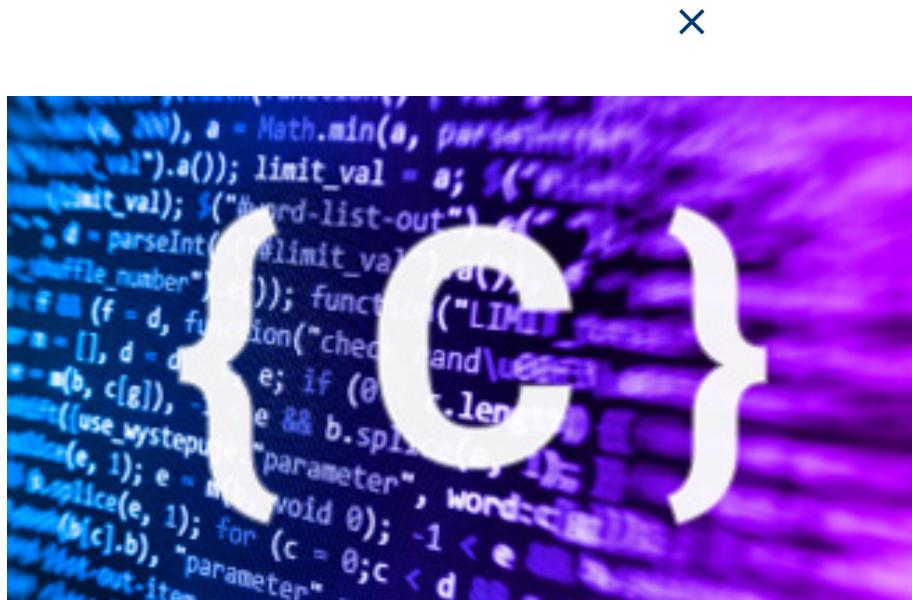
Mas e a linguagem C?

Comparar Java com C é comparar duas coisas totalmente diferentes que têm em comum apenas o fato de serem linguagens de programação. A linguagem C tem como ponto forte a interação com sistemas de baixo nível. Por isso, é muito utilizada em drivers de dispositivo.

A distância que separa Java de C é ainda maior do que a distância entre Java e C++.



Java e C++ são linguagens OO (Orientadas a Objeto).



A linguagem C é aderente ao paradigma de programação estruturado. Não possui conceito de classes e objetos.

Em contrapartida, Java é uma linguagem puramente OO. Qualquer programa em Java precisa ter ao menos uma classe e nada pode existir fora da classe (na verdade, as únicas declarações permitidas no arquivo são classe, interface ou enum).

Isso quer dizer que não é possível aplicar o paradigma estruturado em Java.

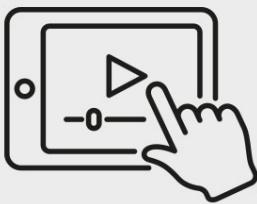
[Saiba mais](#)

Comparar Java e C, portanto, serve apenas para apontar as diferenças dos paradigmas OO e estruturado.

Ambientes de desenvolvimento Java

Agora, assista o vídeo a seguir:

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.



Conceitos

Para falarmos de ambientes de desenvolvimento, precisamos esclarecer alguns conceitos, como a Máquina Virtual Java (**MVJ**) – em inglês, *Java Virtual Machine* (JVM) –, o *Java Runtime Environment* (**JRE**) e o principal para nosso intento: o *Java Development Kit* (**JDK**).

Comecemos pela JVM, que já dissemos ser uma abstração da plataforma.

Conforme explicado pelo Oracle America Inc. (2015), trata-se de uma especificação feita inicialmente pela Sun Microsystems, atualmente incorporada pela Oracle. A abstração procura ocultar do software Java detalhes específicos da plataforma, como o tamanho dos registradores da CPU ou sua arquitetura – RISC ou CISC.

Saiba mais

A JVM é um software que implementa a especificação mencionada e é sempre aderente à plataforma física. Contudo, ela provê para os programas uma plataforma padronizada, garantindo que o código Java sempre possa ser executado, desde que exista uma JVM. Ela não executa as instruções da linguagem Java, mas sim os bytecodes gerados pelo compilador Java.

Adicionalmente, para que um programa seja executado, são necessárias bibliotecas que permitam realizar operações de entrada e saída, entre outras. Assim, o conjunto dessas bibliotecas e outros arquivos formam o ambiente de execução juntamente com a JVM. Esse ambiente é chamado de JRE, que é o elemento que gerencia a execução do código, inclusive chamando a JVM.

Com o JRE, pode-se executar um código Java, mas não se pode desenvolvê-lo.

Para isso, precisamos do JDK, que é um ambiente de desenvolvimento de software usado para criar aplicativos e applets. O **JDK** engloba o JRE e mais um conjunto de ferramentas de desenvolvimento, como um interpretador Java (`java`), um compilador (`javac`), um programa de arquivamento (`jar`), um gerador de documentação (`javadoc`) e outros.

Dois JDK muito utilizados são o Oracle JDK e o OpenJDK. A imagem a seguir mostra um aviso legal exibido durante a instalação do Oracle JDK (Java SE 15). Trata-se de um software cujo uso gratuito é restrito.



Aviso legal na instalação do Oracle JDK.

A Oracle também desenvolve uma implementação de referência livre chamada OpenJDK.

Saiba mais

Há, contudo, diferenças entre o OpenJDK e o Oracle JDK (o coletor de lixo possui mais opções no último). Tais diferenças, porém, não impedem o uso do OpenJDK.

Uma vez instalado um ambiente de desenvolvimento, é possível começar o trabalho de desenvolvimento. Basta ter disponível um editor de texto. Os programas do ambiente podem ser executados a partir de chamadas feitas diretamente no console. Por exemplo, “`javac aluno.java`” irá compilar a classe “Aluno”, gerando um arquivo chamado “`aluno.class`”.

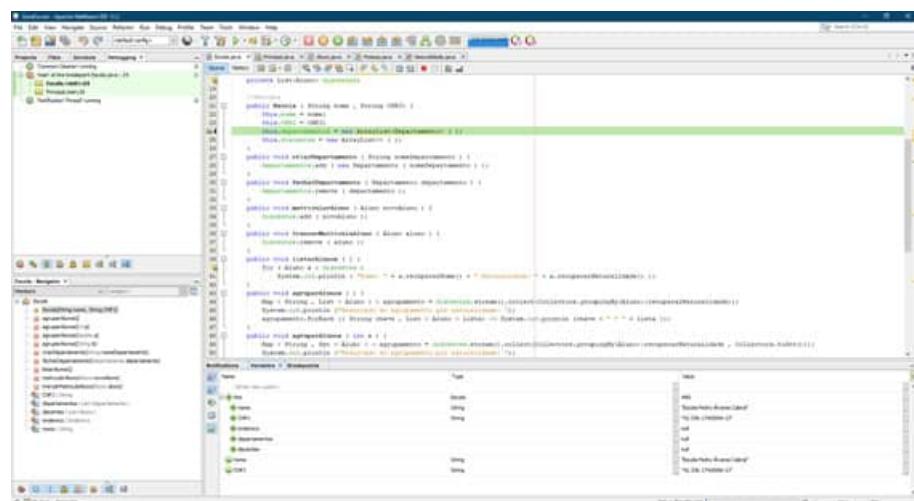
Integrated development environment (IDE)

Um Integrated Development Environment, ou ambiente integrado de desenvolvimento, é um software que reúne ferramentas de apoio e funcionalidades com o objetivo de facilitar e acelerar o desenvolvimento de software.

Ele normalmente engloba um editor de código, as interfaces para os softwares de compilação e um depurador, mas pode incluir também uma ferramenta de modelagem (para criação de classes e métodos), refatoração de código, gerador de documentação e outros.

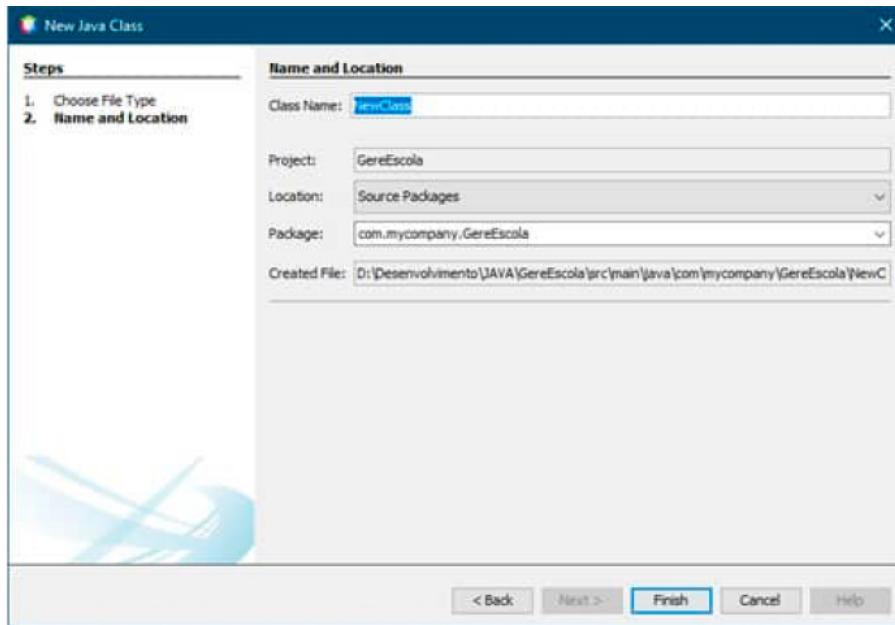
O Netbeans é um IDE mantido pela The Apache Software Foundation e licenciado segundo a licença Apache versão 2.0. De acordo com o site do IDE, ele é o IDE oficial da Java 8, mas também permite desenvolver em HTML, JavaScript, PHP, C/C++, XML, JSP e Groovy. É um IDE multiplataforma que pode ter suas funcionalidades ampliadas pela instalação de plugins.

A imagem a seguir mostra o IDE durante uma depuração.



IDE Netbeans.

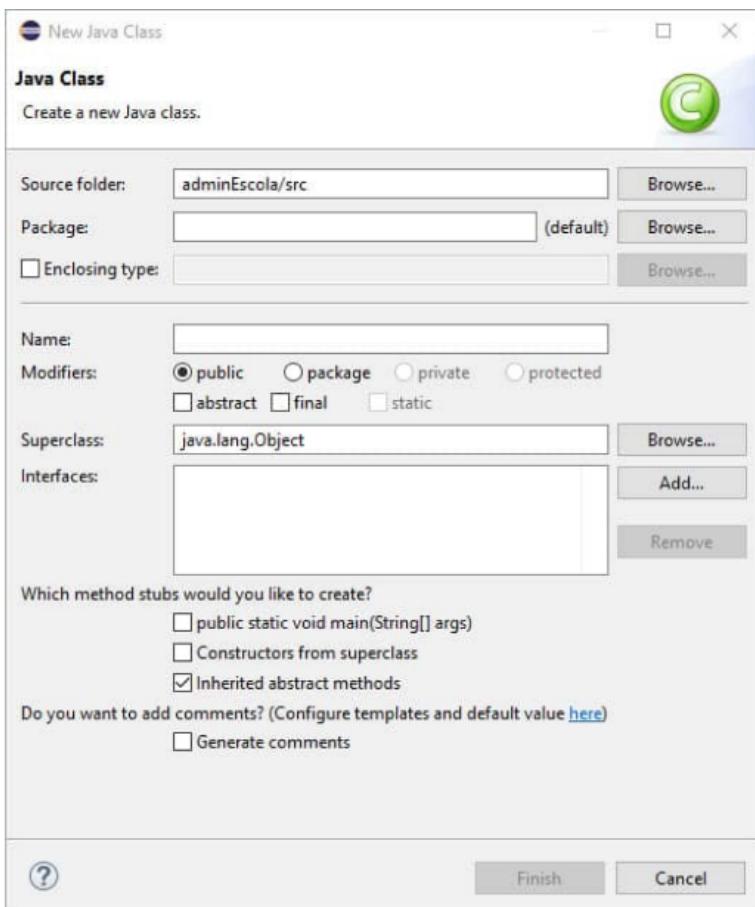
Apesar de possuir muitas funcionalidades, a parte de modelagem realiza apenas a declaração de classes, sem criação automática de construtor ou métodos, conforme se nota na imagem seguinte.



Criação de classe no Netbeans.

A parte de modelagem do IDE Eclipse, porém, é mais completa. Pode-se especificar o modificador da classe, declará-la como “*abstract*” ou “*final*”, especificar sua superclasse e passar parâmetros para o construtor da superclasse automaticamente.

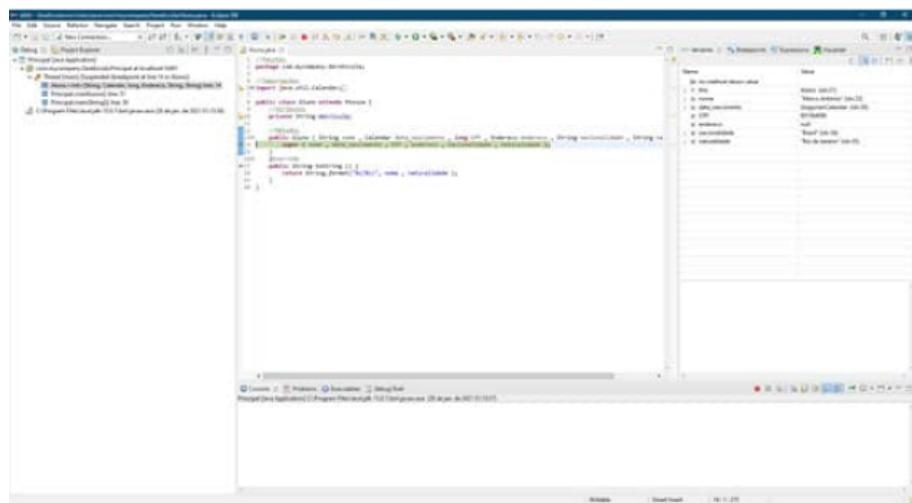
Veja a imagem seguir:



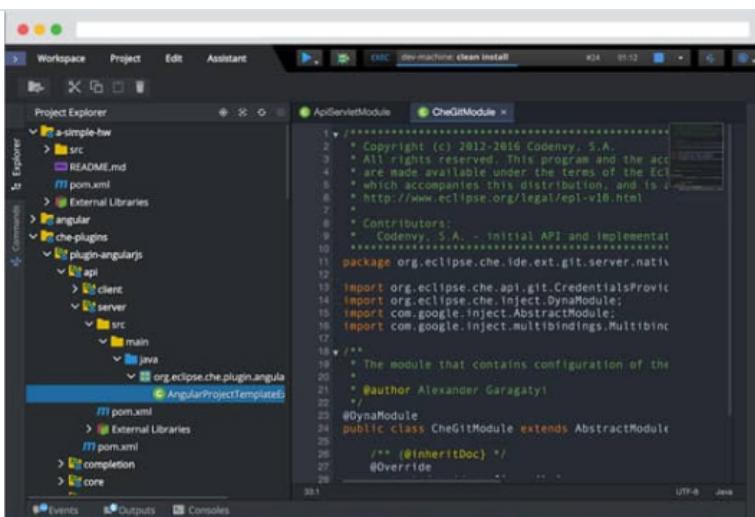
Criação de classe no Eclipse.

O Eclipse é mantido pela Eclipse Foundation, que possui membros como IBM, Huawei, Red Hat, Oracle e outros.

Trata-se de um projeto de código aberto que, da mesma maneira que o Netbeans, suporta uma variedade de linguagens além da Java. O IDE também oferece suporte a plugins, e o editor de código possui características semelhantes ao do Netbeans. O Eclipse provê também um Web IDE chamado Eclipse Che, voltado para DevOps. Trata-se de uma ferramenta rica, que oferece terminal/SSH, depuradores, flexibilidade de trabalhar com execução multicontainer e outras características. A imagem IDE Eclipse mostra um exemplo de depuração no Eclipse, e a imagem IDE Eclipse Che mostra a tela do Eclipse Che. Veja a seguir.



IDE Eclipse.



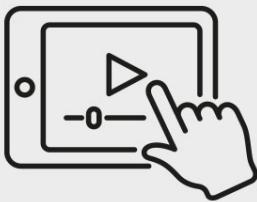
IDE Eclipse Che.

Estrutura e principais comandos de um

programa em Java

Para continuar os seus estudos, assista o vídeo a seguir:

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.



Como um programa em Java começa a sua execução?

Na linguagem C/C++, o ponto de entrada para a execução do programa é a função “main”.



Na Java, o ponto de entrada não é obrigatório para compilar o programa.

Isso quer dizer que, caso seu código possua apenas as classes “Aluno”, “Escola”, “Departamento” e “Pessoa”, por exemplo, ele irá compilar, mas não poderá ser executado.

Quando se trata de uma aplicação *standalone*, um ponto de entrada pode ser definido pelo método “*main*”. Entretanto, diferentemente de C/C++, esse método deverá pertencer a uma classe.

Veja o código 21. A linha 2 mostra a função “*main*”, que define o ponto de entrada da aplicação. Esse será o primeiro método a ser executado pela JVM.



Código 21: Definição do ponto de entrada de um aplicativo Java.

A sintaxe mostrada na linha 2 é a assinatura do método “*main*”. Qualquer modificação nela fará com que o método não seja mais reconhecido como ponto inicial de execução.

Mas há outro caso a ser considerado: as *Applets*, que são programas executados por outros programas, normalmente um navegador ou um *appletviewer*. Geralmente se apresentando embutidas em páginas HTML, as *Applets* não utilizam o método “*main*” como ponto de entrada.

Comentário

Nesse caso, a classe deverá estender a classe “*Applet*”, que fornece os métodos “*start*”, “*init*”, “*stop*” e “*destroy*”. O método “*start*” é invocado quando a *Applet* é visitada. Em seguida, o método “*init*” é chamado no início da execução da *Applet*. Se o navegador sair da página onde a *Applet* está hospedada, é chamado o método “*stop*”. Por fim, quando o navegador for encerrado, será chamado o método “*destroy*”.

Outro elemento importante da estrutura de um programa Java são os métodos de acesso. Estes não são comandos Java e nem uma propriedade da linguagem, e sim consequência do encapsulamento. Eles apareceram em alguns códigos mostrados aqui, embora sem os nomes comumente usados em inglês.

Os **métodos de acesso** são as mensagens trocadas entre objetos para alterar seu estado. Eles, assim como os demais métodos e atributos, estão sujeitos aos modificadores de acesso, que já vimos anteriormente.

Todos os métodos utilizados para recuperar valor de variáveis ou para defini-los são métodos de acesso!

Na prática, é muito provável, mas não obrigatório, que um atributo dê origem a dois métodos: um para obter seu valor (“*get*”) e outro para inseri-lo (“*set*”). Assim, no **Código 8**, as linhas 17 e 20 mostram métodos de acesso ao atributo “*nome*”.

Vejamos agora alguns dos principais comandos em Java.

Vamos começar pelas estruturas de desvio, que realizam o desvio da execução do código com base na avaliação de uma expressão booleana. Java possui o tipo de dado “*boolean*”, que pode assumir os valores “*true*” ou “*false*”. Repare que, diferentemente de C/C++, um número inteiro não é convertido automaticamente pelo compilador em “*boolean*”. Logo, o **código 22** gera erro de compilação em Java.



Código 22: Estrutura de desvio – if.

A sintaxe do comando “if” é a seguinte:



A sintaxe do comando "if".

A expressão deve sempre ser reduzida a um valor booleano. Assim, se substituirmos a linha 2 do **código 22** por “`private int temp = 1, aux = 0;`”, “`temp > aux`” é um exemplo de expressão válida. Da mesma maneira, se a substituição for por “`private boolean temp = true;`”, a linha 5 do código se torna válida. Quando a expressão for verdadeira, executa-se o “bloco” que se segue ao comando “`if`”. A cláusula “`else`” é opcional e desvia a execução para o “bloco” que a segue quando a expressão é falsa. Caso ela seja omitida, o programa continua na próxima linha. Quando “bloco” for composto por um único comando, o uso de chaves (“`{ }`”) não é necessário, mas é obrigatório se houver mais de uma instrução. O comando “`if`” pode ser aninhado, como vemos no **código 23**.

Java



Código 23: "if" aninhado.

Embora o aninhamento do comando “`if`” seja útil, muitos casos podem ser melhor resolvidos pelo comando “`switch`”. A sintaxe de “`switch`” é a seguinte:

Java (Pseudocódigo)



Código 24: sintaxe de “switch”.

No caso do comando “switch”, a expressão pode ser, por exemplo, uma “String”, “byte”, “int”, “char” ou “short”. O valor da expressão será comparado com os valores em cada cláusula “case” até que um casamento seja encontrado. Então, o “bloco” correspondente ao “case” coincidente é executado. Se nenhum valor coincidir com a expressão, é executado o bloco da cláusula “default”.

É interessante notar que tanto as cláusulas “break” quanto a “default” são opcionais, mas seu uso é uma boa prática de programação. Veja um exemplo no **código 25**. Além disso, “switch” também pode ser aninhado. Para isso, basta usar um comando “switch” em um dos blocos mostrados na sintaxe dele.

Java



Código 25: Estrutura de desvio – “switch”.

Outro grupo de comandos importante são as estruturas de repetição. As três estruturas providas pela Java são “while”, “do-while” e o laço “for”.

A estrutura “while” tem a seguinte sintaxe:

Java (Pseudocódigo)



Código 26: A estrutura do “while”.

Observe que, nesse comando, antes de executar o “bloco” a expressão é avaliada. O “bloco” será executado apenas se a expressão for verdadeira e, nesse caso, a execução se repete até que a expressão assuma valor falso. O exemplo do **Código 27** conta de 0 até 9, pois quando “controle” assume valor 10, a expressão “*controle < 10*” se torna falsa e as linhas 5 e 6 não são executadas.

Java



Código 27: Estrutura de repetição – "while".

A estrutura “*do-while*” tem funcionamento parecido. Todavia, ao contrário de “*while*”, nessa estrutura o bloco sempre será executado ao menos uma vez. Veja sua sintaxe:

Java (Pseudocódigo)



Código 28: A estrutura do “*do-while*”.

Como vemos pela sintaxe, o primeiro “bloco” é executado e, somente após, a expressão é avaliada. Se for verdadeira, a repetição continua até que a expressão seja falsa. Caso contrário, a execução do programa continua na linha seguinte.

Vejamos, no Código 29, o **código 27** modificado para o uso de “*do-while*”.

Java



Código 29: Estruturas de repetição – "do-while".

A última estrutura de repetição que veremos é o laço “for”. A sua sintaxe é:

Java (Pseudocódigo)



Código 30: estrutura de repetição: “for”.

O parâmetro “inicialização” determina a condição inicial e é executado assim que o laço se inicia. A “expressão” é avaliada em seguida. Se for verdadeira, a repetição ocorre até que se torne falsa. Caso seja falsa no início, o “bloco” não é executado nenhuma vez e a execução do programa saltará para a próxima

linha após o laço. O último item, “iteração”, é executado após a execução do “bloco”. No caso de uma “expressão” falsa, “iteração” não é executado.

Vejamos, no **Código 31**, um exemplo a partir da modificação do **Código 16**.

Java



Código 31: Estruturas de repetição – "for".

O laço “for” passou a ter uma segunda forma, desde a Java 5, chamada de “for-each”. Esse laço é empregado para iterar sobre uma coleção de objetos, de maneira sequencial, como vimos ao estudarmos o agrupamento de objetos. Um exemplo do seu emprego pode ser visto na linha 29 do **código 16**. A sua sintaxe é:

Java (Pseudocódigo)



Código 32: exemplo do código 16.

Por fim, observamos que quando o “bloco” for formado por apenas uma instrução, o uso das chaves (“{ } ”) é opcional.

Falta pouco para atingir seus objetivos.

Vamos praticar alguns conceitos?

Questão 1

A única assinatura de método que cria um ponto de entrada para a execução de um programa Java é:

- A “public static int main (String args[])”.

- B “protected static void main (String args[])”.

- C “public static void main (int args[])”.

- D “public static void main (String args[])”.

- E "protected static void main (int args[])".

Parabéns! A alternativa D está correta.

O método “main” em Java não admite sobrecarga. Sua assinatura é fixa, inclusive quanto aos parâmetros.

Questão 2

Para realizar um desenvolvimento em Java, são imprescindíveis todos os itens da alternativa:

- A JRE, IDE e Máquina Virtual Java.
- B Máquina Virtual Java, IDE e Editor de Código.
- C JDK e Editor de Código.
- D JRE, IDE e Editor de Código.
- E JDK, JRE e Máquina Virtual Java.

Parabéns! A alternativa C está correta.

O JDK contém o JRE e a Máquina Virtual Java, mas não possui aplicativo de edição de código, que precisa ser complementado.

Considerações finais

Ao longo deste estudo, você desenvolveu uma percepção crítica sobre a linguagem, que deve extrapolar para todas as linguagens. O bom profissional avalia tecnicamente as ferramentas à sua disposição e o problema a ser resolvido.

Nós apenas tocamos a superfície do desenvolvimento em Java, mas esta é uma linguagem com muitos recursos e que não se tornou tão popular por mero acaso! Introduzimos os conceitos de classe e objetos e a forma com trabalhá-los em Java. Prosseguimos no estudo aprendendo sobre o agrupamento de objetos, as formas mais complexas de manipulação e as ferramentas disponíveis para isso. Por fim, tecemos uma importante discussão sobre a linguagem e passamos por exercícios para sedimentar o conhecimento. Agora, você está pronto para se aprofundar na programação em Java!



Agora, apresentaremos um histórico da linguagem Java e orientação objeto, as principais aplicações e perspectivas futuras.

Para ouvir o áudio, acesse a versão online deste conteúdo.



Explore +

Para saber mais sobre os assuntos tratados neste tema, pesquise na internet:

- Os processos envolvidos na implementação do método “*groupingBy*”. A documentação da API Java é um bom ponto de partida!
- Artigos e entrevistas sobre os criadores do C++ (Bjarne Stroustrup) e da Java (James Gosling).
- As disputas comerciais envolvendo a Oracle e o Google em torno do uso da Java no Android.

Referências

ECLIPSE FOUNDATION. **Eclipse Che**. Consultado em meio eletrônico em: 12 fev. 2021.

GOSLING, J. et.al. **The Java® Language Specification**: Java SE 15 Edition. In: Oracle America Inc, 2020.

ORACLE AMERICA INC. **Java SE**: Chapter 2 – The Structure of the Java Virtual Machine. 2015.

ORACLE AMERICA INC. **Java® Platform, Standard Edition & Java Development Kit Specifications – Version 15**. Consultado em meio eletrônico em: 12 fev. 2021.

ORACLE AMERICA INC. **The Java™ Tutorials**. Lesson: Introduction to Collections. Consultado em meio eletrônico em: 12 fev. 2021.

SCHILD, Herbert. **Java**: The Complete Reference. 9th. ed. Nova Iorque: McGraw Hill Education, 2014.

STROUSTRUP, Bjarne. **Stroustrup**: FAQ. 2020.

SUTTER, Herb. **The C Family of Languages**: Interview with Dennis Ritchie, Bjarne Stroustrup, James Gosling. Consultado em meio eletrônico em: 12 fev. 2021.

THE APACHE FOUNDATION. **NetBeans IDE**: Overview. Consultado em meio eletrônico em: 12 fev. 2021.

Material para download

Clique no botão abaixo para fazer o download do conteúdo completo em formato PDF.

[Download material](#)

O que você achou do conteúdo?



Relatar problema