FACULDADE IDEAL FACI WYDEN

SILVIO CEZAR DE SOUZA TEIXEIRA

MAINÁ DA SILVA TOMAZ

DESENVOLVIMENTO DE SOFTWARE

Belém

2022

Sumário

RESUMO

O desenvolvimento de software possibilita unir tecnologia e estratégia como solução para diversos problemas de um negócio. Cada software possui seus objetivos e tem capacidade de aumentar o nível de produtividade das atividades em que for empregado. Isso porque a tecnologia de um sistema é capaz de permitir maior controle dos processos da empresa. O que possibilita que tarefas, antes manuais, sejam feitas com eficácia e assertividade, minimizando erros, por exemplo. Por envolver muito planejamento e estratégia, além de muitas análises, manutenções, implementações e testes, o software possibilita automatizar diversas aplicações. Cada codificação tem suas particularidades justamente para personalizar o produto de acordo com as necessidades do cliente. Ou seja, a importância do desenvolvimento de sistemas está na eficiência de solucionar problemas de modo automatizado. Isso contribui para a otimização de tempo, serviços e funções dentro da empresa. Inclusive, possibilita que haja redução de custos a longo prazo. Agora que você já sabe os benefícios desse tipo de aplicação, saiba que um sistema desse porte só é um bom investimento se o processo de desenvolvimento de software seguir as melhores práticas da área. Análise econômica Uma das etapas do processo de desenvolvimento e software é realizar uma análise econômica do projeto. Afinal, o desenvolvimento do sistema precisa atender aos requisitos do cliente e usuários com qualidade. Ao mesmo tempo, ele não deve ultrapassar o teto estipulado

1. **Arquitetura Web:**

É possível identificar diversas soluções que poderiam ser utilizadas visando resolvê-lo. Contudo, outros fatores como custo e eficiência influenciam na escolha da solução a ser adotada. No contexto do desenvolvimento de software, o mesmo pode ser observado ao se analisar os requisitos visando a construção de um software: várias soluções computacionais podem ser definidas para atender a esses requisitos, mas uma análise deve ser feita para definir a mais adequada ao contexto de desenvolvimento da aplicação. Para se representar essas soluções, a arquitetura de software é uma das abordagens que podem ser usadas.

Em que situação o tema é útil:

No entendimento dos fundamentos da arquitetura de software. Conhecimento este fundamental na elaboração da arquitetura de aplicações em projetos reais.

Quando tentamos solucionar um problema, é possível identificar diversas soluções que poderiam ser utilizadas visando resolvê-lo. Contudo, outros fatores como custo e eficiência influenciam na escolha da solução a ser adotada. No contexto do desenvolvimento de software, o mesmo pode ser observado ao se analisar os requisitos visando a construção de um software: várias soluções computacionais podem ser definidas para atender a esses requisitos, mas uma análise deve ser feita para definir a mais adequada ao contexto de desenvolvimento da aplicação.

Para se representar essas soluções, a arquitetura de software é uma das abordagens que podem ser usadas. Com isso, para se obter a arquitetura (solução) mais adequada para atender aos requisitos do software (problema), uma avaliação dessa estrutura deve ser realizada.

A arquitetura consiste em um modelo de alto nível que possibilita um entendimento e uma análise mais fácil do software a ser desenvolvido. O uso de arquitetura para representar soluções de software foi incentivada principalmente por duas tendências (GARLAN e PERRY, 1995; KAZMAN, 2001): (1) o reconhecimento por parte dos projetistas que o uso de abstrações facilita a visualização e o entendimento de certas propriedades do software, e (2) a exploração cada vez maior de frameworks visando diminuir o esforço de construção de produtos através da integração de partes previamente desenvolvidas.

Outra propriedade da arquitetura é a possibilidade de usá-la como ferramenta para comunicar a solução projetada aos diversos stakeholders que participam do processo de desenvolvimento

do software (GARLAN, 2000). Contudo, para que essa comunicação seja possível, a arquitetura deve ser representada através de um documento, conhecido como documento arquitetural.

Para se construir a arquitetura de um software, e por conseqüência o documento arquitetural que a representa, os requisitos são as principais informações usadas. Durante o processo de especificação arquitetural (Figura 1), além dos requisitos,

outras fontes de conhecimento podem ser utilizadas para definir os elementos arquiteturais e a forma como eles devem estar organizados. Entre essas fontes de conhecimento se destacam principalmente a experiência do arquiteto, o raciocínio sobre os requisitos, e os estilos e as táticas arquiteturais.

1. **HTML: Exemplo de construção de cadastro de formulário**
   É um formulário de preenchimento de dados ou que resulta em uma ação desejada utilizando a linguagem de marcação HTML. É formado por um ou mais widgets. Esses widgets são campos de textos, caixas de seleção, botões, radio buttons e checkboxes utilizando ferramentas do próprio HTML. Dessa forma, o usuário pode interagir com a página ao executar ações através desses formulários.

   Esse recurso é muito utilizado para a criação de formulários de contato, formulários para captura de leads, e também para criação de sistemas, como por exemplo a criação de um modal de login.

   Portanto, vejamos abaixo alguns exemplos de formulários HTML que podemos encontrar na página inicial da HomeHost:

   Exemplo 01 de Formulário HTML
   A caixa de texto, assim como o botão de busca, e a caixa de seleção para escolher a extensão de domínio são elementos de um formulário HTML. Porém, além disso, percebe-se que há efeitos e estilização, elaborados através de outras ferramentas, como o CSS.

   Ainda na página da HomeHost, na categoria Suporte, podemos encontrar mais dois exemplos típicos de formulários. O primeiro exemplo representa um formulário de contato, enquanto o segundo representa um formulário de busca.

   Como inserir um formulário HTML
   O formulário HTML é representado pela tag de abertura <form> e a de fechamento </form>. Dentro dessas tags, serão colocados todos os elementos que compõem este formulário. Posteriormente, vamos explicar melhor como incluir esses elementos.

   Para a tag <form> podem ser atribuídos o atributo method e o atributo action.

   O atributo action define o local (através de uma URL) ao qual serão enviados todos os dados recolhidos do formulário.

   O atributo method define o método HTTP com que o formulário HTML irá lidar com os dados recebidos. São eles: o método GET e o método POST. Posteriormente vamos entender melhor como funcionam esses dois métodos.

   Dessa forma, vejam exemplos abaixo do uso da tag <form> :

   Os métodos GET e POST
   Para entender a diferença entre esses dois métodos, é necessário compreender como funcionam as requisições HTTP. De uma forma resumida, sempre que você acessa um

recurso através da web, o navegador envia uma requisição através da URL. Portanto o HTTP é uma requisição que consiste em duas partes, o cabeçalho e o corpo. O cabeçalho contém um conjunto de metadados globais envolvendo os recursos do navegador. Já o corpo pode conter informações necessárias para o servidor conseguir processar a solicitação anterior.

O método GET
O método GET é usado pelo navegador para solicitar que o servidor envie de volta um determinado recurso. Portanto, é como se falássemos ao servidor "Servidor, eu quero obter este recurso.". Nesse caso, o navegador envia um corpo vazio. Portanto, já que o corpo fica vazio, se um formulário for enviado através do método GET, os dados serão reconhecidos pelo servidor através da URL.

Dessa forma, podemos perceber que ao enviar um formulário com os campos Nome e Idade, a URL iria se parecer com algo como: url?nome=valor&idade=value.

Esse é o método padrão do formulário HTML. Caso não seja declarado o atributo method, o formulário funcionará através do método GET.

Vejamos então, algumas observações a respeito do método GET:

O tamanho de uma URL é limitado a cerca de 3000 caracteres;
Nunca use o GET para enviar dados confidenciais, pois esses dados ficarão visíveis na URL;
É útil para envios de formulários em que um usuário deseja marcar o resultado;
GET é melhor para dados não seguros, como strings de consulta no Google.

O método POST
O método POST é utilizado no navegador para conversar com o servidor. Os dados são enviados ao servidor através do corpo da solicitação HTTP. Também é realizada uma solicitação de resposta. Portanto, é como se falássemos ao servidor "Servidor, verifique esses dados e me retorne um resultado adequado". Dessa forma, ao enviar um formulário através do método POST, os dados serão anexados ao corpo da solicitação HTTP.

Diferentemente do método GET, explicado anteriormente, o método POST não inclui o corpo na URL. Portanto, os dados enviados não ficarão visíveis na URL.

Vejamos então, algumas observações a respeito do método POST:

Anexa dados de formulário dentro do corpo da solicitação de HTTP. Portanto, os dados não são mostrados na URL;
Não tem limitações de tamanho;
Os envios de formulários com o POST não podem ser marcados.

2. Descrever as diferenças entre Servlets e Scripts CGI

CGI (Common Gateway Interface) é a primeira tentativa de fornecer aos usuários conteúdo dinâmico. Ele permite que os usuários executem um programa que reside no servidor para processar dados e até acessar bancos de dados para produzir o conteúdo relevante. Como esses são programas, eles são gravados no sistema operacional nativo e armazenados em um diretório específico. Um servlet é uma implementação de Java que visa fornecer o mesmo serviço que o CGI, mas, em vez de programas compilados no sistema operacional nativo, ele é compilado no bytecode Java, que é executado na máquina virtual Java. Embora os programas Java possam ser compilados no código nativo, eles ainda preferem compilar no bytecode Java.

A primeira vantagem dos servlets sobre o CGI está na independência da plataforma. Os servlets podem ser executados em qualquer sistema operacional enquanto a JVM estiver instalada, o que significa que você não teria nenhum problema, mesmo se optar por mudar de sistema operacional. Com o CGI, alternar sistema operacional é um processo difícil e trabalhoso, pois você precisa recompilar os programas no novo sistema operacional.

Como você está executando programas independentes em CGI, eles criam seu próprio processo quando são executados, algo que não acontece com os servlets, pois eles apenas compartilham o espaço de memória da JVM. Isso pode levar a problemas relacionados à sobrecarga, especialmente quando você aumenta o número de usuários exponencialmente. Ele também cria problemas de vulnerabilidade, pois o programa não é controlado de forma alguma quando é executado no servidor.

Posteriormente, o método mais comum ao usar CGI é via scripts. Isso reduz o tempo necessário na criação de programas e geralmente é mais seguro. Com o CGI, você pode executar scripts imediatamente, enquanto servlets, você precisará converter o script em Java e compilá-lo em um servlet, o que adiciona um pouco ao tempo de carregamento.

3. Introdução ao JSP (Java Server Pages)

JSP é o acrônimo para Java Server Pages, uma linguagem criada pela SUN gratuita, JSP é uma linguagem de script com especificação aberta que tem como objetivo primário a geração de conteúdo dinâmico para páginas da Internet. Podemos ao invés de utilizar HTML para desenvolver páginas Web estáticas e sem funcionalidade, utilizar o JSP para criar dinamismo. É possível escrever HTML com códigos JSP embutidos. Como o HTML é uma linguagem estática, o JSP será o responsável por criar dinamismo. Por ser gratuita e possuir especificação aberta possui diversos servidores que suportam a linguagem, entre eles temos: Tomcat, GlassFish, JBoss, entre outros. O JSP necessita de servidor para funcionar por ser uma linguagem Server-side script, o usuário não consegue ver a codificação JSP, pois esta é convertida diretamente pelo servidor, sendo apresentado ao usuário apenas codificação HTML.

Uma pagina JSP possui extensão .jsp e consiste em uma página com codificação HTML e com codificação Java, inserida entre as tag´s <% e %>, denominada scriptlets e funcionando da seguinte forma: o servidor recebe uma requisição para uma página JSP, interpreta esta página gerando a codificação HTML e retorna ao cliente o resultado de sua solicitação. A página JSP que foi interpretada pelo servidor não precisa ser compilada como aconteceria com um servlet java por exemplo, esta tarefa é realizada em tempo real pelo servidor. É necessário apenas desenvolver as páginas JSP e disponibilizá-las no Servlet Container (Tomcat, por exemplo). O trabalho restante será realizado pelo servidor que faz a compilação em tempo de uso transformando o jsp em bytecode.

Assim, pode-se definir o JSP como uma tecnologia que provê uma maneira simples e prática de desenvolver aplicações dinâmicas baseadas em web, sendo independente de Plataforma de Sistema Operacional.

4. Introdução aos Padrões de Projetos; explicar funcionamento arquitetura MVC

Na fase de Projeto começamos a nos preocupar com a arquitetura da aplicação. Damos realmente valor à tecnologia, diferente da fase de análise onde ainda estamos esboçando o problema a ser resolvido. Definimos a plataforma e como os componentes do sistema se organizarão. Evidentemente que os requisitos ainda são importantes, pois, por exemplo, um sistema Web ou então uma aplicação de tempo real deverá influenciar na arquitetura.

Mesmo não possuindo uma definição consensual, muitos autores definem a arquitetura de software de um sistema computacional como a suas estruturas, que são compostas de elementos de software, de propriedades externamente visíveis de seus componentes e do relacionamento entre eles. Ou seja, a arquitetura define os elementos de software e como eles interagem entre si.

Para realizar a arquitetura de uma aplicação não basta estar num dia inspirado ou acordar com bastante vontade de realizar uma arquitetura, muito pelo contrário, precisamos de bastante experiência em diferentes organizações, diferentes tipos de projetos, conhecimentos de diferentes arquiteturas etc. A experiência prática ainda é a melhor solução, pois o trabalho em equipe também é uma forma excelente de definir uma arquitetura. Muitas vezes, alguns programadores possuem outras experiências ou conhecimentos e a troca dessa experiência é sempre válida, mesmo quando temos um arquiteto bastante experiente na equipe.

A arquitetura de um sistema tem diversos elementos como: elementos utilitários, de interação, elementos que fazem parte do domínio do problema, elementos de conexão, de persistência etc. Dessa forma, na arquitetura sempre definimos os seus elementos que serão utilizados no software e como eles se conectam. Uma arquitetura complexa exige mais tempo para desenvolvimento, porém, através de geração automática de aplicações torna-se mais produtivo. Por isso algumas equipes

definem um framework para uma determinada aplicação e assim podemos utilizar muitas coisas pré-prontas que facilitam o desenvolvimento.

No entanto, alguns padrões arquiteturais já foram pensados para resolver problemas corriqueiros. Alguns projetos ou organizações combinam esses padrões arquiteturais, pois atendem melhor às suas necessidades ou deram certo para o seu tipo de aplicação. Por isso é sempre interessante entender as características básicas de cada um dos estilos e escolher ou combinar aqueles que atendem melhor às necessidades de um projeto específico. Isso tudo deve ser feito após uma análise do sistema a ser desenvolvido. Entre as arquiteturas existentes temos: Cliente-servidor, P2P - Peer to Peer, Dados compartilhados, Máquina virtual, Camadas, MVC e muitos outros.

Algumas vantagens quando utilizamos um ou mais padrões de projeto em desenvolvimento de software:

Baixo acoplamento: é o grau em que uma classe conhece a outra. Se o conhecimento da classe A sobre a classe B for através de sua interface, temos um baixo acoplamento, e isso é bom. Por outro lado, se a classe A depende de membros da classe B que não fazem parte da interface de B, então temos um alto acoplamento, o que é ruim.

Coesão: quando temos uma classe elaborada com um único e bem focado propósito, dizemos que ela tem alta coesão, e isso é bom. Quando temos uma classe com propósitos que não pertencem apenas a ela, temos uma baixa coesão, o que é ruim.

Padrão de Projeto
Alexander descreveu um padrão como sendo um problema que se repete inúmeras vezes em um mesmo contexto e que contenha uma solução para resolve-lo de tal modo que esta solução seja utilizada em diversas situações. O termo padrões de projeto ou Design Patterns, descreve soluções para problemas recorrentes no desenvolvimento de sistemas de software orientados a objetos. O Factory é um destes padrões e é baseado em uma interface para criar objetos e deixar que suas subclasses decidam que classe instanciar. Deste modo, utiliza-se o conceito de fábrica de objetos quando um objeto é utilizado para a criação de outros objetos. Algo assim foi implementado no Framework Hibernate para a criação de uma espécie de fábrica de sessões, ou seja, na criação de uma única SessionFactory teríamos acesso a vários objetos do tipo Session.

Padrões de projeto são estabelecidos por um nome, problema, solução e consequências. O nome deve ser responsável por descrever o problema, a solução e as consequências. Quando alguém na equipe de um projeto se refere a um padrão pelo nome, os demais membros da equipe devem relacionar este nome com um problema encontrado, a solução e as consequências na utilização de tal padrão. Encontrar um nome para um novo padrão é considerada uma etapa difícil, já que o nome deve proporcionar a ideia para a qual o padrão foi criado. O problema descreve quando devemos aplicar o padrão, qual o problema a que se refere e seu contexto. A solução descreve os elementos que fazem parte da implementação, as relações entre eles, suas responsabilidades e colaborações. Uma solução não deve descrever uma implementação concreta em particular e sim proporcionar uma descrição abstrata de

um problema e como resolvê-lo. As consequências são os resultados, vantagens e desvantagens ao utilizar o padrão e são importantes para avaliar as alternativas descritas bem como proporcionar uma ideia de custos e benefícios ao aplicá-lo.

Um exemplo existente e muito utilizado de padrão é o Data Access Object, ou simplesmente DAO. Foi uma solução encontrada para resolver problemas referentes à separação das classes de acesso a banco de dados das classes responsáveis pelas regras de negócio.

Já o conceito principal do modelo MVC é utilizar uma solução já definida para separar partes distintas do projeto reduzindo suas dependências ao máximo.

Desenvolver uma aplicação utilizando algum padrão de projeto pode trazer alguns dos seguintes benefícios:

- Aumento de produtividade;
- Uniformidade na estrutura do software;
- Redução de complexidade no código;
- As aplicações ficam mais fácies de manter;
- Facilita a documentação;
- Estabelece um vocabulário comum de projeto entre desenvolvedores;
- Permite a reutilização de módulos do sistema em outros sistemas;
- É considerada uma boa prática utilizar um conjunto de padrões para resolver problemas maiores que, sozinhos, não conseguiriam;
- Ajuda a construir softwares confiáveis com arquiteturas testadas;
- Reduz o tempo de desenvolvimento de um projeto.
- Nos dias atuais existem diversos padrões e cada um possui uma função bem específica dentro da estrutura do projeto. Deste modo, podemos utilizar em um mesmo projeto de software mais de um padrão simultaneamente. Poderíamos fazer uso simultâneo, por exemplo:

  Do Factory para criar uma fábrica de sessões com o banco de dados;
  O DAO para separar as classes de CRUD das regras de negócios e
  O MVC para dividir e diminuir a dependência entre módulos do sistema.
  CRUD: A sigla tem origem na língua inglesa para Create, Retrieve, Update e Delete, em português teria o significado de Criar, Recuperar, Atualizar e Excluir. São as quatro operações básicas em um banco de dados.
  O Padrão MVC (Model-View-Controller)
  O MVC é utilizado em muitos projetos devido a arquitetura que possui, o que possibilita a divisão do projeto em camadas muito bem definidas. Cada uma delas, o Model, o Controller e a View, executa o que lhe é definido e nada mais do que isso.

  A utilização do padrão MVC traz como benefício o isolamento das regras de negócios da lógica de apresentação, que é a interface com o usuário. Isto possibilita a existência de várias interfaces com o usuário que podem ser modificadas sem a necessidade de alterar as regras de negócios, proporcionando muito mais flexibilidade e oportunidades de reuso das classes.

Uma das características de um padrão de projeto é poder aplicá-lo em sistemas distintos. O padrão MVC pode ser utilizado em vários tipos de projetos como, por exemplo, desktop, web e mobile.

A comunicação entre interfaces e regras de negócios é definida através de um controlador, que separa as camadas. Quando um evento é executado na interface gráfica, como um clique em um botão, a interface se comunicará com o controlador, que por sua vez se comunica com as regras de negócios.

Imagine uma aplicação financeira que realiza cálculos de diversos tipos, como os de juros. Você pode inserir valores para os cálculos e também escolher que tipo de cálculo será realizado. Isto tudo é feito pela interface gráfica, que para o modelo MVC é conhecida como View. No entanto, o sistema precisa saber que você está requisitando um cálculo, e para isso, terá um botão no sistema que quando clicado gera um evento.

Este evento pode ser uma requisição para um tipo de cálculo específico como o de juros simples ou juros compostos. Fazem parte da requisição os valores digitados no formulário e a seleção do tipo de cálculo que o usuário quer executar sobre o valor informado. O evento do botão é como um pedido a um intermediador (Controller) que prepara as informações para então enviá-las para o cálculo. O controlador é o único no sistema que conhece o responsável pela execução do cálculo, neste caso, a camada que contém as regras de negócios. Esta operação matemática será realizada pelo Model assim que ele receber um pedido do Controller.

O Model realiza a operação matemática e retorna o valor calculado para o Controller, que também é o único que possui conhecimento da existência da camada de visualização. Tendo o valor em mãos, o intermediador o repassa para a interface gráfica que exibirá para o usuário. Caso esta operação deva ser registrada em uma base de dados, o Model se encarrega também desta tarefa.

O MVC inicialmente foi desenvolvido no intuito de mapear o método tradicional de entrada, processamento, e saída que os diversos programas baseados em GUI utilizavam. No padrão MVC, teríamos então o mapeamento de cada uma dessas três partes para o padrão MVC conforme ilustra a Figura 1.

Mapeamento das três partes de uma aplicação para o MVC
Mapeamento das três partes de uma aplicação para o MVC.
demonstra que a entrada do usuário, a modelagem do mundo externo e o feedback visual para o usuário são separados e gerenciados pelos objetos Modelo (Model), Visão (View) e Controlador (Controller).

Objetos utilizados no MVC e suas interações
Objetos utilizados no MVC e suas interações.
Explicando cada um dos objetos do padrão MVC tem-se:

Primeiramente o controlador (Controller), que interpreta as entradas do mouse ou do teclado enviadas pelo usuário e mapeia essas ações do usuário em comandos que são enviados para o modelo (Model) e/ou para a janela de visualização (View) para efetuar a alteração apropriada;
Por sua vez, o modelo (Model) gerencia um ou mais elementos de dados, responde a perguntas sobre o seu estado e responde a instruções para mudar de estado. O modelo sabe o que o aplicativo quer fazer e é a principal estrutura computacional da arquitetura, pois é ele quem modela o problema a ser resolvido;
Por fim, a visão (View) gerencia a área retangular do display e é responsável por apresentar as informações para o usuário através de uma combinação de gráficos e textos. A visão não sabe nada sobre o que a aplicação está atualmente fazendo, pois tudo que ela realmente faz é receber instruções do controle e informações do modelo e então exibi-las. A visão também se comunica de volta com o modelo e com o controlador para reportar o seu estado.
Portanto, a principal ideia do padrão arquitetural MVC é a separação dos conceitos - e do código. O MVC é como a clássica programação orientada a objetos, ou seja, criar objetos que escondem as suas informações e como elas são manipuladas e então apresentadas em uma interface simples para o mundo. Entre as diversas vantagens do padrão MVC estão a possibilidade de reescrita da GUI ou do Controller sem alterar o modelo, reutilização da GUI para diferentes aplicações com pouco esforço, facilidade na manutenção e adição de recursos, reaproveitamento de código, facilidade na manutenção do código sempre limpo etc.

Implementação do MVC
Existem diversos frameworks para Java que implementam o padrão MVC e são muito utilizados em diversos projetos. Entre eles temos o JSF, Struts 1 e Struts 2, Spring MVC, Play Framework, Tapestry, e diversos outros. Existem diversos artigos e sites especializados que comparam as diferenças e vantagens entre esses diferentes frameworks. No entanto, o melhor é sempre verificar o que cada framework disponibiliza para os desenvolvedores e analisar se atende às expectativas.

Outras linguagens/plataformas também possuem frameworks que aderem ao padrão arquitetural MVC. Isso não inviabiliza que uma equipe crie o seu próprio framework, mas é preciso lembrar que um desenvolvedor novo precisa de tempo para aprender a desenvolver em determinada arquitetura e caso a empresa/projeto já utilize um framework bastante popular, a curva de aprendizado será bem menor ou praticamente nula. Isso inclusive ajuda na contratação de novos funcionários, pois pode exigir como pré-requisito conhecimentos neste framework.

5. Introdução ao Orientação a Objetos: Classes; objeto; atributos/propriedades; métodos; níveis de visibilidade; herança; encapsulamento; polimorfismo; extends; implements.

Orientação a objetos é um paradigma aplicado na programação que consiste na interação entre diversas unidades chamadas de objetos.

Depois desta definição formal, você deve estar se perguntando onde e quando podemos utilizar orientação a objetos? É normal este questionamento quando ficamos apenas na abstração.

Porém, se tem algo que é a base de todo programador, é a orientação a objetos, um paradigma criado há anos atrás e usado até hoje, nas principais linguagens e tecnologias do mercado.

Usamos a orientação a objetos para nos basear na vida real e resolver problemas de software, ou pelo menos tentamos. Ela acaba sendo uma base inclusive para outros paradigmas.

orientação a objetos é algo atemporal e não está ligado a uma linguagem, o que significa que você pode investir seu tempo e aprender pois não vai mudar.

Pilares
A OOP (Object Oriented Programming -- Programação Orientada a Objetos) pode ser definida por quatro pilares principais, sendo eles herança, encapsulamento, abstração e polimorfismo.

Encapsulamento
Na definição, dissemos que orientação a objetos é uma forma de interação entre unidades chamadas de objetos. Pois bem, este é o conceito por trás do seu primeiro pilar, o encapsulamento.

É comum ouvirmos o termo "encapsular" no dia-a-dia. "Encapsula isto em uma classe", mas o que de fato da significado a este termo?

Imagine o seguinte programa abaixo:

```
int idade = 34;

...

public void VerificaIdade() {
    if(idade > 18) ...
    }

public void AtribuiIdade(int idade) {
    idade = idade;
    }
```

Neste pequeno cenário, temos uma variável representando uma idade e posteriormente duas funções que leem/escrevem nela. A medida que este programa cresce, as coisas podem se complicar.

Não sabemos onde a variável é usada ou modificada, não sabemos sequer quais funções são pertinentes a idade ou mesmo a que se refere esta idade.

Aplicando o conceito de encapsulamento, fazemos o agrupamento das coisas que fazem sentido estarem juntas, para podermos organizar e reutilizar melhor nosso código.

```
public class Aluno {
public int Idade { get; set; }

public Aluno(int idade) {
Idade = idade;
}

public void VerificaIdade() {
if(idade > 18) ...
}
}
```

Refatorando nosso código, temos então uma classe Aluno, agrupando as variáveis e funções que tinhamos anteriormente.

As variáveis passam então a ser o que chamamos de propriedades e as funções passam a ser o que chamamos de métodos. Esta composição dá origem ao nosso objeto e aplica o conceito de encapsulamento.

As classes na orientação a objetos funcionam como um molde para os objetos. Os objetos são criados a partir de uma classe e muitos deles podem ser feitos da mesma classe.

Herança

Assim como na vida real somos um acúmulo de características de nossos antepassados, na orientação a objetos isto também é possível. Na verdade isto é até uma boa prática.

O conceito de herdar é literalmente ser uma cópia de outra classe com algumas características adicionais.

Vamos tomar como base uma classe de pagamento simples, que contém a data de vencimento deste pagamento.

```
public class Pagamento {
public DateTime Vencimento { get; set; }
}
```

O pagamento por sua vez pode ser via Boleto, Cartão de Crédito, PayPal, Stripe, MoiP e futuros outros métodos de pagamento poderão existir.

Ao invés de tratar todas estas possibilidades dentro da mesma classe, tornando ela grande e complexa de ser gerenciada, podemos aplicar o conceito de herança e criar filhos desta classe.

```
public class PagamentoBoleto : Pagamento {
    public string CodigoBarras { get; set; }
}
```

Neste caso, temos uma nova classe, chamada PagamentoBoleto que herda as características da sua classe pai Pagamento. Desta forma, na classe PagamentoBoleto temos tanto a data de vencimento quanto o código de barras.

Abstração

Na orientação a objetos, o conceito de abtração ou abstrair, significa esconder os detalhes de uma implementação, ou seja, quanto menos souberem sobre nossas classes, mais fácil de consumí-las será.

```
public class PagamentoBoleto : Pagamento {
    public int digitoVerificador = 0;

    public string CodigoBarras { get; set; }

    public int CalcularDigitoVerificador() {
        ...
    }
}
```

No evemplo anterior temos o cálculo do dígito verificador. Isto é exposto tanto pela propriedade digitoVerificador quanto pelo método CalcularDigitoVerificador neste classe.

A pergunta aqui é: É realmente necessário expor estas propriedades e métodos?

Tanto a propriedade quanto o método são usados exclusivamente no boleto, e caso precisem ser alterados, se consumidos externamente, causariam uma refatoração em vários outros componentes.

Desta forma, escondemos tudo que não é necessário o mundo externo ao nosso objeto saber, assim ficamos mais confortáveis com as mudanças, pois elas afetam somente o nosso objeto.

```
public class PagamentoBoleto : Pagamento {
    private int digitoVerificador = 0;

    public string CodigoBarras { get; set; }

    private int CalcularDigitoVerificador() {

    }
}
```

Polimorfismo

Poli significa muitos e morfo significa formas, então temos a possibilidade de um objeto assumir diversas formas diferentes na orientação a objetos.

Novamente no caso dos pagamentos, embora cada pagamento seja pertinente a uma operadora distinta, ainda temos comportamentos que serão padrão em todos eles.

Vamos tomar como base o método PodeSerPago que nos informa se um pagamento está vencido ou não, se pode ser pago ou não.

```
public class Pagamento {
public bool PodeSerPago() {
...
}
}
```

Na classe pai, este comportamento é padrão, ou seja, passado a data de vencimento, caso o pagamento ainda não tenha sido realizado, ele está vencido e não pode ser pago.

Para os pagamentos via Cartão de Crédito e outras formas digitais, não haveria problemas, afinal, os pagamentos neste formato são cobrados em qualquer data e hora.

Porém, os pagamentos via boleto tem um formato diferente onde caso a data de vencimento seja em um fim de semana ou feriado, o mesmo poderá ser pago no próximo dia útil.

Não queremos reescrever a mesma regra de pagamento duas vezes, queremos que ela seja padrão para todos os pagamentos, porém, queremos poder sobrescreve-la caso haja necessidade.

```
public class Pagamento {
public virtual bool PodeSerPago() {
...
}
}

public class PagamentoBoleto : Pagamento {
public override bool PodeSerPago() {
...
}
}
```

Nesta implementação, temos uma regra específica para pagamentos via boleto, que não afeta a regra base para quaisquer outros tipos de pagamento.

6. Introdução a linguagem UML.

Entenda
Utilizaremos o Poseidon UML para implementar o modelo do exemplo. O Poseidon UML é uma ferramenta case que suporta os principais diagramas UML e é desenvolvido em Java. Tem algumas facilidades no que se refere a interação com o Java podendo, a partir de um modelo gerar o código Java das classes definidas no mesmo e também suporta a Engenharia Reversa, que é gerar o modelo a partir das classes implementadas.

O modelo aqui proposto começa a ser implementado a partir de um problema real que é a necessidade de um cliente. O problema proposto é o seguinte:

"Desenvolver um sistema para um caixa eletrônico onde é permitido a um cliente realizar quatro tipos de operações: a de consulta de saldo, solicitação de extrato, depósito e saque. Esse mesmo caixa eletrônico deve ser abastecido de dinheiro e ter os depósitos recolhidos por um funcionário do banco"

Para definição do modelo do nosso sistema, iremos implementar primeiro um Diagrama de Casos de Uso ou Use Cases. Os objetivos principais de um diagrama de Casos de Uso são:

Descrever os requisitos funcionais do sistema de maneira uniforme para usuários e desenvolvedores;
Descrever de forma clara e consistente as responsabilidades a serem cumpridas pelo sistema, formando a base para a fase de projeto;
Oferecer as possíveis situações do mundo real para a fase de testes do sistema.

Um ator é uma entidade externa ao sistema que de alguma forma participa de um caso de uso. Um ator pode ser um ser humano, máquinas, dispositivos, ou outros sistemas. Atores típicos são cliente, usuário, gerente, computador, impressora, etc. Os atores representam um papel e iniciam um caso de uso que após executado, retorna um valor para o ator. Um caso de uso especifica um serviço que será executado ao usuário e é composto por um ou mais cenários. Um cenário é uma narrativa de uma parte do comportamento global do sistema. Para o problema proposto, o Diagrama de Casos de Uso pode ser implementado

No Diagrama de Casos de Uso implementado o sistema é o Caixa Eletrônico, os atores representam o Cliente e o Funcionário do banco. O Cliente interage com os Casos de Uso consulta de saldo, solicitação de extrato, depósito e saque e o Funcionário interage com os Casos de Uso Abastece de dinheiro e Recolher envelopes de depósitos.

Para melhor entendimento do Diagrama de Casos de Uso é necessária a descrição textual do fluxo do Caso de Uso (principal e alternativo) e do Cenário ou dos Cenários que compõe cada Caso de Uso. Iremos descrever o Use Case Solicitação de Extrato bem como o Cenário que o compõe.

7. **Modelar um domínio (livre escolha), apresentar caso de uso, diagrama de classe e sequência**

Modelando
1 - Um item que está na saída mas não está no modelo.
Temos itens na nossa saída que não existem em no modelo de domínio, então os adicionamos:
Note que o salário médio é calculado com base em outros dados; então o chamamos de getAverageSalary para nos lembrar ele é um atributo calculado.
2 - Um item está no modelo mas não na saída
Alguém se empolgou e adicionou a data de nascimento à entidade Empregado
Perguntamos ao diretor de RH se era mesmo necessária a data de nascimento no relatório. Disseram que não, mas gostariam da idade no relatório.
3 - Há dois pedaços de informação em um mesmo lugar
O nome completo é composto pelo nome e sobrenome. Perguntamos ao diretor do RH se desejam que os nomes sejam separados para se distinguir pessoas da mesma família (O nome de família pode ser usado para identificar grupos culturais, por exemplo). Ele não gostou da ideia.
4 - Um entidade não está relacionada com nenhuma outra
Todas as entidades no modelo devem estar ligadas. Perguntamos para os especialistas no negócio se as entidades são ligadas diretamente umas com as outras ou através de alguma outra coisa. Por exemplo, o empregado está ligado diretamente ao departamento ou está em um time e o time está ligado ao departamento?
5 - Relacionamento um-para-um
Como relacionamentos um-para-um costumam constituir um mau cheiro, perguntamos aos especialistas no negócio sobre eles: "Poderia nos dar um exemplo de um empregado com mais de um departamento e um exemplo de departamento com mais de um empregado?"
Ao aplicar esta técnica rigorosamente pode-se gerar perguntas aparentemente bobas mas que na verdade são de bastante valor. Um exemplo: "Poderia me dar um exemplo de empregado com mais de um sexo, ou mais de uma raça?". As perguntas devem se manter abertas até se achar um exemplo para elas. Funcionam como indicadores de risco para avaliar se um sistema pode sofrer alterações
6 - Relacionamentos muitos para muitos
Relacionamentos muitos-para-muitos podem indicar perda de informações. Perguntamos ao especialista de negócio o que essa informação perdida poderia ser. Neste caso, o tempo de trabalho de um empregado pode ser alocado entre um ou mais departamentos:
Para questões de função (ou papel), sexo e raça, o especialista de negócio não consegue pensar em mais nenhuma informação de valor. Então o cheiro permanece, com perguntas abertas que indicam o risco de mudança.
Um website especializado resolveu o problema do relacionamento muitos-para-muitos do sexo, incluindo classificações para cada uma das opcões (masculino, feminino, transsexual feminina, transexual masculino, intersexual). As informações perdidas poderiam ser a data em que o indivíduo decidiu realizar a operação, ou quando a operação foi realizada e o impacto que isso teve nos bônus e aumentos de salários subsequentes.
7 - Funções não definidas

Perguntamos ao especialista no negócio como calcular as funções getAverageSalary e getAge:
Isto leva a criação de de allocation.cost e employee.dateOfBirth, que são adicionados ao modelo.
Não há mais cheiros que sem resolução ou que tiveram seu tratamento adiado. Então podemos parar nosso processo de modelagem de domínio!

8. Introdução a JSF (Java Server Faces)

Desenvolver sistemas web é uma realidade no mercado atual e o framework JavaServer Faces é a opção padrão do Java EE para resolver este tipo de problema. Uma das características do JSF é trabalhar de uma maneira orientada a componentes de tela e seus eventos (por exemplo, cliques). Desta maneira, podemos associar estes componentes a diversos aspectos de nosso sistema, como a execução de operações de negócio, conversões de valores, validações de campos, etc. Este framework também suporta internacionalização, a utilização de layouts unificados, e chamadas assíncronas ao servidor (AJAX).

Desenvolvimento para web é um assunto muito discutido no contexto da tecnologia Java. Há anos a comunidade de software tem se esforçado para projetar e criar melhores formas de criar sistemas para web com esta linguagem. Atualmente, uma das tecnologias mais utilizadas para o desenvolvimento em Java deste tipo de aplicação é o JavaServer Faces (JSF). O JSF é a especificação de um framework para aplicações web definido na padronização do Java Enterprise Edition (Java EE). A nova versão desta tecnologia (o JSF 2.0) traz uma série de recursos úteis para o desenvolvedor de software, como a definição de componentes através de anotações, trabalho nativo com Facelets, AJAX, integração com CDI e diversos outros itens.

Para demonstrar como utilizar este framework, este artigo visa apresentar ao leitor com algum conhecimento de aplicativos web em Java (por exemplo, Servlets e JSPs) as vantagens do JSF, já considerando os recursos desta segunda versão. Também é nosso intuito demonstrar alguns mecanismos úteis desta nova versão para conhecedores de JSF, integrada com o mecanismo padrão de injeção de dependências do Java EE (o CDI – Contexts and Dependency Injection), e a simplificação de sua utilização prática no dia a dia.

9. Apresentação dos conceitos ORM (Object Relacional Mapping) e JPA e Hibernate

ORM (Object Relational Mapper) é uma técnica de mapeamento objeto relacional que permite fazer uma relação dos objetos com os dados que os mesmos representam. Ultimamente tem sido muito utilizada e vem crescendo bastante nos últimos anos.

Este crescimento tem se dado principalmente pelo fato de muitos desenvolvedores não se sentirem a vontade em escrever código SQL e pela produtividade que esta técnica nos proporciona. Existem ótimos ORM´s como Hibernate, NHibernate, Entity Framework e etc.

ORM

Existem dois mundos: o relacional e o orientado a objetos.

No mundo relacional prevalecem princípios matemáticos com a finalidade de armazenar e gerenciar corretamente os dados, de forma segura e se trabalha com a linguagem SQL que é utilizada para dizer o banco de dados “O QUE?” fazer e não como fazer.

Já no mundo orientado a objetos trabalhamos com classes e métodos, ou seja, trabalhamos fundamentados na engenharia de software e seus princípios que nos dizem “COMO” fazer. O ORM é justamente, a ponte entre estes dois mundos, ou seja, é ele quem vai permitir que você armazene os seus objetos no banco de dados.

Para isto precisamos fazer um mapeamento dos seus objetos para as tabelas do banco de dados.

Como o ORM trabalha
Ele faz o mapeamento da sua classe para o banco de dados e cada ORM tem suas particularidades para gerar o SQL referente a inserção do objeto que corresponde a uma tabela no banco de dados e realizar a operação. Utilizando um ORM, também se ganha produtividade, pois deixa-se de escrever os comando SQL para deixar que o próprio ORM

Hibernate e JPA
A diferença entre o Hibernate e o JPA. O Hibernate é de fato o framework ORM, ou seja, a implementação física do que você usará para persistir, remover, atualizar ou buscar dados no SGBD. Por outro lado, o JPA é uma camada que descreve uma interface comum para frameworks ORM.

Você pode desenvolver todo seu sistema sem JPA, apenas com Hibernate ou qualquer outro framework ORM, como o TopLink. Porém você não pode desenvolver o sistema apenas com JPA, pois ele é apenas uma interface a ser utilizada por Frameworks ORM.

A ideia geral é tornar o sistema o mais abstrato possível e passível de mudanças sem grandes impactos. Se você desenvolver todo seu sistema usando JPA com o framework Hibernate e amanhã decide mudar para o TopLink, então as alterações serão mínimas.

10. Introdução a Estrutura de Dados: Tipos de Listas; Fila; Pilha; apresentar uma estrutura de dados implementada em JAVA (TED)

Em computação, normalmente utilizamos os dados de forma conjunta. A forma como estes dados serão agregados e organizados depende muito de como serão utilizados e processados, levando-se em consideração, por exemplo, a eficiência para buscas, o volume dos dados trabalhados, a complexidade da implementação e a forma como os dados se relacionam. Estas diversas formas de organização são as chamadas estruturas de dados.

Podemos afirmar que um programa é composto de algoritmos e estruturas de dados, que juntos fazem com que o programa funcione como deve.

Cada estrutura de dados tem um conjunto de métodos próprios para realizar operações como:

- Inserir ou excluir elementos;
- Buscar e localizar elementos;
- Ordenar (classificar) elementos de acordo com alguma ordem especificada.
- Características das estruturas de dados
-

As estruturas de dados podem ser:

- **lineares** (ex. arrays) ou não lineares (ex. grafos);
- **homogêneas** (todos os dados que compõe a estrutura são do mesmo tipo) ou heterogêneas (podem conter dados de vários tipos);
- **estáticas** (têm tamanho/capacidade de memória fixa) ou dinâmicas (podem expandir).

Array
Também chamado de vetor, matriz ou arranjo, o array é a mais comum das estruturas de dados e normalmente é a primeira que estudamos.

Normalmente trabalha-se com apenas um tipo de dado por array; embora o JavaScript permita a declaração de arrays de mais de um tipo de dado, por exemplo ["banana", 5, true], isso não acontece na maior parte das linguagens de programação.

Por ser uma estrutura de dados básica e muitíssimo utilizada, a linguagens de programação costumam já ter este tipo implementado, com métodos nativos para criação e manipulação de arrays. No caso do JavaScript, você pode consultar os métodos e construtores de array no MDN.

Pilha
Em um array, é possível utilizar funções próprias para manipular elementos em qualquer posição da lista. Porém, há situações (veremos exemplos mais adiante) onde é desejável mais controle sobre as operações que podem ser feitas na estrutura. Aí entra a implementação de estruturas de dados como a pilha (stack) e a fila (queue).

A pilha é uma estrutura de dados que, assim como o array, é similar a uma lista. O paradigma principal por trás da pilha é o LIFO - Last In, First Out, ou “o último a entrar é o primeiro a sair”, em tradução livre.

Para entendermos melhor o que significa isso, pense em uma pilha de livros ou de pratos. Ao empilharmos livros, por exemplo, o primeiro livro a ser retirado da pilha é obrigatoriamente o último que foi colocado; se tentarmos retirar o último livro da pilha, tudo vai desabar. Ou seja, o último livro a ser empilhado é o primeiro a ser retirado.

Abstraindo este princípio para código, percebe-se que há apenas dois métodos possíveis para manipular os dados de uma pilha: 1) inserir um elemento no topo da pilha e 2) remover um elemento do topo da pilha.

Ao contrário do array, as linguagens de programação normalmente não têm métodos nativos para criação e manipulação de pilhas. Porém, é possível usar métodos de array para a implementação de pilhas.

Fila

A fila tem uma estrutura semelhante à pilha, porém com uma diferença conceitual importante: o paradigma por trás da fila é o FIFO - First In, First Out, ou “o primeiro a entrar é o primeiro a sair”, em tradução livre.

Pense em uma fila de bilheteria, por exemplo. A pessoa que chegou antes vai ser atendida (e comprar seu ingresso) antes de quem chegou depois e ficou atrás na fila. A fila como estrutura de dados segue o mesmo princípio.

Sendo assim, também há somente duas formas de se manipular uma fila: 1) Inserir um elemento no final da fila e 2) remover um elemento do início da fila.

Deque

A estrutura de dados deque (abreviação de double-ended queue ou “fila de duas pontas”) é uma variação da fila que aceita inserção e remoção de elementos tanto do início quanto do final da fila.

Podemos comparar, novamente, com uma fila de pessoas em um guichê de atendimento: uma pessoa idosa que chega é atendida antes (ou seja, não pode ser colocada no fim da fila), ao mesmo tempo que uma pessoa que entrou no final da fila pode desistir de esperar e ir embora (nesse caso, não podemos esperar a pessoa chegar na frente da fila para retirá-la de lá).

Uma outra forma de se entender a estrutura deque é como uma junção das estruturas de pilha e fila.

```
public class Arvore<T extends Comparable<T>> {
	No raiz;

	public Arvore() {
		this.raiz = null;
	}

	public No inserirNo(T valor) {
		No<T> n = new No<T>(valor);
		return inserirNo(n, null);

	}
	public No inserirNo(No novo, No pai) {

		if(pai == null)
			pai = raiz;

		if(raiz == null) {
			raiz = novo;
		}else {
			//menor
			if( novo.obterValor().compareTo(pai.obterValor()) == -1) {

				if(pai.obterNoEsquerdo() == null)
					pai.inserirEsquerdo(novo);
				else
					inserirNo(novo, pai.obterNoEsquerdo());

			}else {

				if(pai.obterNoDireito() == null)
					pai.inserirDireito(novo);
				else
					inserirNo(novo, pai.obterNoDireito());
			}
		}

		return novo;

	}


	public No buscarNo(No novo, No pai) {

		if(pai == null)
			pai = raiz;

		if(novo == null){
			return null;
		}else if(novo.obterValor().compareTo(pai.obterValor()) == 0) {
```

```
                return novo;
            }if( novo.obterValor().compareTo(pai.obterValor()) == -1) {

                buscarNo(novo, pai.obterNoEsquerdo());

            }else {

                buscarNo(novo, pai.obterNoDireito());
            }

            return novo;

        }


        public No removerNo(T valor) {
            return removerNo(valor, null);
        }

        public No removerNo(T valor, No currentno) {

            No noret = null;

            if(currentno == null)
                currentno = raiz;

            //igual
            if(currentno.obterValor().compareTo(valor) == 0) {
                //System.out.println(currentno.obterValor() + "é igual");
                //é um nó folha?
                if((currentno.obterNoDireito()== null) &&
(currentno.obterNoEsquerdo() == null)) {

                    if(currentno == this.raiz)
                        this.raiz = null;

                    else if(currentno == currentno.pai.obterNoDireito() )
                        currentno.pai.inserirDireito(null);
                    else
                        currentno.pai.inserirEsquerdo(null);

                //tem apena sum filho à direita?
                }else if (currentno.obterNoDireito() == null){

                    if(currentno == this.raiz)
                        this.raiz = this.raiz.obterNoEsquerdo();

                    else if(currentno == currentno.pai.obterNoDireito() )
                        currentno.pai.inserirDireito(
currentno.obterNoEsquerdo() );

                    else
```

```
                                        currentno.pai.inserirEsquerdo(
currentno.obterNoEsquerdo() );

                        //tem apena sum filho à esquerda?
                        }else if (currentno.obterNoEsquerdo()== null){

                                if(currentno == this.raiz)
                                        this.raiz = this.raiz.obterNoDireito();

                                else if(currentno == currentno.pai.obterNoDireito() )
                                        currentno.pai.inserirDireito(
currentno.obterNoDireito() );

                                else
                                        currentno.pai.inserirEsquerdo(
currentno.obterNoDireito() );

                        //tem dois filhos
                        }else {

                                No sucessor = this.getSucessor(currentno, true);
                                System.out.println("O sucessor é:" + sucessor+"\n");

                                if(sucessor != currentno.obterNoDireito()) {

                                        sucessor.pai.inserirEsquerdo(
sucessor.obterNoDireito() );
                                        sucessor.inserirDireito(
currentno.obterNoDireito() );
                                }


                                //é a raiz
                                if(currentno == this.raiz )
                                  raiz = sucessor;

                                //é o filho a esquerda
                                else if(currentno == currentno.pai.obterNoDireito())
                                                currentno.pai.inserirDireito(sucessor);
                                else

        currentno.pai.inserirEsquerdo(sucessor);

                                sucessor.inserirEsquerdo(
currentno.obterNoEsquerdo() );

                        }



                }else if( currentno.obterValor().compareTo(valor) == -1) {
```

```
                        //System.out.println(currentno.obterValor() + "é menor que
"+valor);
                        removerNo(valor, currentno.obterNoDireito());
                }else {
                        //System.out.println(currentno.obterValor() + "é mairo que
"+valor);
                        removerNo(valor, currentno.obterNoEsquerdo());

                }


                return null;
        }


        public No getSucessor(No atual, Boolean primeiraVez) {

                No sucessor  = null;

                if(primeiraVez)
                        sucessor = atual.obterNoDireito();
                else
                        sucessor = atual;

                if(sucessor.obterNoEsquerdo()!=null) {
                        return getSucessor(sucessor.obterNoEsquerdo(), false);
                }

                return sucessor;
        }


        public void emOrdem(No no) {
                if (no != null) {
                        emOrdem(no.filhoEsquerdo);
                        System.out.println(no.valor);
                        emOrdem(no.filhoDireito);
                }
        }

        public void preOrdem(No no) {
                if (no != null) {
                        System.out.println(no.valor);
                        emOrdem(no.filhoEsquerdo);
                        emOrdem(no.filhoDireito);
                }
        }


        public void posOrdem(No no) {
                if (no != null) {
                        posOrdem(no.filhoEsquerdo);
```

```
                posOrdem(no.filhoDireito);
                System.out.println(no.valor);
            }
        }

}
```

11. Apresentar conceitos de estrutura de dados árvore AVL e um exemplo de implementação em JAVA

Uma árvore AVL é dita balanceada quando a diferença entre as alturas das sub-árvores não é maior do que um. Caso a árvore não estiver balanceada é necessário seu balanceamento através da rotação simples ou rotação dupla. O balanceamento é requerido para as operações de adição e exclusão de elementos. Para definir o balanceamento é utilizado um fator específico para nós.

O fator de balanceamento de um nó é dado pelo seu peso em relação a sua sub-árvore. Um nó com fator balanceado pode conter 1, 0, ou -1 em seu fator. Um nó com fator de balanceamento -2 ou 2 é considerado um árvore não-AVL e requer um balanceamento por rotação ou dupla-rotação.

```
public void inserir(No node, int valor) {
 //verifica se a árvore já foi criada
  if (node != null) {
   //Verifica se o valor a ser inserido é menor que o nodo corrente da árvore, se sim vai para subárvore esquerda
   if (valor < node.valor) {
      //Se tiver elemento no nodo esquerdo continua a busca
      if (node.esquerda != null) {
         inserir(node.esquerda, valor);
      } else {
         //Se nodo esquerdo vazio insere o novo nodo aqui
         System.out.println("  Inserindo " + valor + " a esquerda de " + node.valor);
         node.esquerda = new No(valor);
      }
   //Verifica se o valor a ser inserido é maior que o nodo corrente da árvore, se sim vai para subárvore direita
   } else if (valor > node.valor) {
      //Se tiver elemento no nodo direito continua a busca
      if (node.direita != null) {
         inserir(node.direita, valor);
      } else {
         //Se nodo direito vazio insere o novo nodo aqui
         System.out.println("  Inserindo " + valor + " a direita de " + node.valor);
         node.direita = new No(valor);
      }
   }
```

```
 }
}
```

12. Introdução de estrutura de dados grafos e exemplos

1 - Um grafo é simples (ou regular) se não possuir laços e nem mais de uma aresta ligando dois vértices.

2 - A vizinhança de um nó é definida assim:
N(v) = {w pertence a V | v-w pertence a A}.

Nestes casos podemos dizer que o vértice w é adjacente a v e que a aresta v-w incide no vértice v.

3 - O grau de um vértice é a quantidade de arestas que incidem nele.
• Um vértice é dito isolado se possuir grau zero.

4 - Um grafo completo com n vértices é um grafo simples onde existe uma aresta ligando todo par não ordenado vértices distintos.
•O número máximo de arestas em um grafo com n vértices é n * (n-1) /2.

5 - Um grafo não precisa ser uma árvore, mas toda árvore é um grafo.

Representação de um Grafo
Um grafo pode ser representado por uma matriz N x N, onde N é a quantidade de vértices de um grafo.

Há grafos que possuem pesos associados às arestas. Assim, o peso é armazenado no matriz.

Exemplo de um programa em C para determinar o caminho entre duas cidades

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>

#define TAM 6

void iniciarGrafo(int mat[][TAM]) {
   int i, j;
   for (i=0; i<TAM; i++)
      for (j=0; j<TAM; j++)
         mat[i][j] = 0;
}

void join(int mat[][TAM], int cidade1, int cidade2) {
   mat[cidade1][cidade2]=1;
}
```

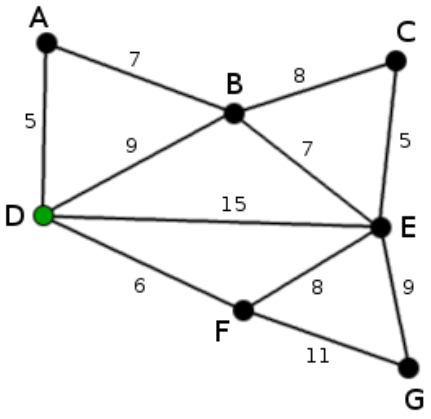
```
int adjacente(int mat[][TAM], int cidade1, int cidade2) {
   if (mat[cidade1][cidade2])
      return 1;
   else return 0;
}

int procuraCaminho(int mat[][TAM], int cidade1, int cidade2, int k) {
   int c;
   if (k ==1) {
      if (adjacente(mat, cidade1, cidade2))
         return 1;
      else return 0;
   }
   for (c=0; c<TAM; ++c)
      if (adjacente(mat, cidade1, c) && procuraCaminho(mat, c, cidade2, k-1))
         return 1;
   return 0;
}

int main() {
   int matriz[TAM][TAM];
   int cidade1, cidade2, k, temCaminho;
   char resp='S', caminho[30]="\0";
   iniciarGrafo(matriz);
   while (toupper(resp) == 'S') {

   do {
      printf("Digite o codigo da cidade de origem: ");
      scanf("%i", &cidade1);
   } while (cidade1 <1 || cidade1 >TAM);

   do {
      printf("Digite o codigo da cidade de destino: ");
      scanf("%i", &cidade2);
   } while (cidade2 <1 || cidade2 >TAM);

   printf("Realizar outro join? ");
   join(matriz, cidade1-1, cidade2-1);
   fflush(stdin);
   scanf("%c", &resp);

   }//fim while

printf("Digite cidade A: ");
scanf("%i", &cidade1);
printf("Digite cidade B: ");
scanf("%i", &cidade2);
printf("Digite tamanho do caminho: ");
```

```
scanf("%i", &k);
temCaminho = procuraCaminho(matriz, cidade1-1, cidade2-1, k);

   if (temCaminho ==1)
      printf("Existe caminho de comprimento %i entre as cidades %i e% i", k, cidade1,
cidade2);
   else
      printf("Nao existe caminho de comprimento %i entre as cidades %i e% i", k,
cidade1, cidade2);
   system("pause");
   return 0;
}
```

13. Descrição da problemática do caminho com custo mínimo; problema do caixeiro viajante, algoritmo Dijkstra

Na teoria de grafos, o problema do caminho mínimo consiste na minimização do custo de travessia de um grafo entre dois nós (ou vértices); custo este dado pela soma dos pesos de cada aresta percorrida.

Formalmente, dado um grafo valorado (ou seja, um conjunto V de vértices, um conjunto A de arestas e uma função de peso {\displaystyle f:A\to \mathbb {R} }{\displaystyle f:A\to \mathbb {R} }) e, dado qualquer elemento v de V, encontrar um caminho P de v para cada v' de V tal que



{\displaystyle \sum _{p\in P}f(p)}{\displaystyle \sum _{p\in P}f(p)}
é mínimo entre todos os caminhos conectando n a n'.

Em programação dinâmica, podemos escolher um subproblema de modo que toda a informação vital seja recordada e levada adiante. Assim, vamos definir que para cada vértice {\displaystyle v}v e cada inteiro {\displaystyle i\leq k}{\displaystyle i\leq k} , {\displaystyle dist(v,i)}{\displaystyle dist(v,i)} como o menor caminho de {\displaystyle s}s até {\displaystyle v}v que usa {\displaystyle i}i arestas. Os valores iniciais de {\displaystyle dist(v,0)}{\displaystyle dist(v,0)} são {\displaystyle \infty }\infty para todos os vértices exceto {\displaystyle s}s, para o qual é 0. E a equação geral de atualização é:

{\displaystyle dist(v,i)=\min _{(u,v)\in E}\{dist(u,i-1)+l(u,v)\}}{\displaystyle dist(v,i)=\min _{(u,v)\in E}\{dist(u,i-1)+l(u,v)\}}

14. Caracterização da linguagem JAVA: técnicas de programação (tipos primitivos; variáveis; constantes; operadores; estruturas de seleção; estruturas de repetição; arrays; functions)

SUPORTE A ORIENTAÇÃO A OBJETOS

A linguagem Java é capaz de representar em termos de código, tudo o que existe no mundo real. Assim, podemos dizer que ela interpreta tudo a sua volta como sendo um objeto. Isto é, a tecnologia de objetos tem como premissa a concentração na modelagem das características e comportamento dos objetos no mundo real.

Os críticos da linguagem ousam dizer que a orientação a objetos é apenas uma forma de organizar o código. E como bem disse Gosling e McGilton, embora tal afirmação não seja uma inverdade, tão pouco conta toda a história. E isso porque você pode obter resultados com técnicas de programação orientada a objetos que não pode com técnicas processuais.

Gosling cita ainda uma característica importante que distingue os objetos de procedimentos e funções comuns. Esta característica é a possibilidade de poder criar objetos em um único local e recuperá-lo em outro local.

Para ser considerada orientada a objetos uma linguagem precisa necessariamente conter quatro características: encapsulamento, polimorfismo, herança e ligação dinâmica.

PORTABILIDADE

A portabilidade é uma caraterística que acompanha o Java desde o primeiro instante da sua criação. Em linhas gerais significa que um sistema desenvolvido em Java poderá ser executado em qualquer sistema e/ou hardware. Lembra da missão do Java, “Write Once, Run Anywhere”.

O próprio ambiente Java é facilmente transportável para novas arquiteturas e sistemas operacionais.

SEGURANÇA

A medida em que ganha escala a internet necessita quase que de forma imediata de um ambiente seguro.

Assim, partindo de tal necessidade e da ideia de que nada é confiável, o Java que segue permanecendo de acordo com este preceito. Buscou meios de manter seu compilador e o sistema em tempo de execução seguros da criação de códigos subversivos.

Esta é uma característica peculiar do Java. A plataforma está preparada para enfrentar os desafios de distribuir software dinamicamente pela rede. E isso, a partir da implementação de várias camadas de defesa contra códigos potencialmente incorretos.

É também por esta característica em particular que o Java é usado pelas maiores empresas do mundo. E se a linguagem não é a principal dentro do ecossistema das gigantes da tecnologia. É certo que ela está presente em algum ambiente ou em alguma funcionalidade. Entre as muitas companhias que usam o Java podemos citar, Google, Amazon, Pinterest, Uber, Spotfy, Banco do Brasil.

LINGUAGEM SIMPLES

Quando falam da simplicidade da linguagem, os seus criadores citam um fragmento do romance de ficção científica. The Rolling Stones de Robert A. Heinlein, onde o autor comenta:

Toda tecnologia passa por três estágios: primeiro, um dispositivo grosseiramente simples e bastante insatisfatório; segundo um grupo enormemente complicado de dispositivos projetados para superar as deficiências do original e, assim, obter um desempenho satisfatório por meio de um compromisso extremamente complexo; terceiro, um projeto final apropriado

A citação serviu de pano de fundo para a comparação da linguagem Java com C/C++. Visto que a primeira ganhou sua simplicidade com a remoção sistemática de recursos de seus antecessores C/C++. Sendo então, a busca pela simplicidade um dos principais objetivos do Java.

Uma forma de explicar a simplicidade do Java é por meio do tradicional e mais simples exemplo, o “Olá Mundo”, vejamos:

E é só o que precisaremos para dizer “olá” ao mundo. O exemplo começa com a declaração de uma classe chamada OlaMundo. Dentro desta classe existe um método de execução, o main. Este método, por sua vez carrega um método para exibir uma String, que neste caso é a saída do programa.

E o método principal faz isso invocando o método público println. Tal método imprime qualquer argumento passado para ele e adiciona uma nova linha à saída. Podemos dizer que este método está contido no método out. O out é do tipo PrintStream da classe final System que é definida no pacote java.lang, e que executa operações de saída.

Em linhas gerais, System.out representa o fluxo de saída padrão, ou seja, permite imprimir uma String, no console (terminal).

ALTA PERFORMANCE

Este é um assunto amplamente discutido nas redes e quase sempre com ideias equivocadas a respeito do desempenho da linguagem Java. É óbvio, que se você não souber o que está fazendo, não há linguagem que possa salvá-lo. Os recursos estão disponíveis, a documentação é vasta, agora cabe ao programador a capacidade de explorar ou não o poder da performance da linguagem.

Enquanto programador, cabe a você escrever seus códigos pensando em performance. A linguagem Java faz a sua parte, oferecendo uma prototipagem rápida e sem medo.

Visto que o garbage collection, por exemplo, remove a responsabilidade do gerenciamento de memória dos ombros do programador.

Enfim, com o Java não é possível apenas criar aplicações de alto desempenho. Mas com todos os recursos da plataforma, o alto desempenho se torna uma característica estendida também ao programador.

DINAMISMO
O Java é organizado de tal forma que qualquer coisa pode ser criada e removida de forma muito dinâmica, rotinas podem ser implementadas a todo momento. Mas o que realmente a torna uma linguagem dinâmica é a sua natureza portátil e interpretada, projetada para se adaptar aos ambientes em evolução.

INTERPRETADA
Ser uma linguagem interpretada significa que ela faz uso do que chamamos de máquina virtual. E é justamente esta máquina virtual, a JVM – Java Virtual Machine, a responsável pela portabilidade das aplicações escritas na linguagem.

Esta já mencionada portabilidade significa que mesmo que a aplicação tenha sido desenvolvida em um ambiente Windows, ela irá rodar tranquilamente em um ambiente Linux.

Assim, depois que você tiver o interpretador da linguagem Java e o suporte em tempo de execução disponíveis em uma determinada plataforma, por exemplo, no Linux, poderá executar qualquer aplicativo da linguagem Java de qualquer lugar. sempre assumindo que o aplicativo específico da linguagem Java seja gravado de maneira portátil.

Entenda, A JVM não entende código Java e não gera código de máquina, em vez disso ela gera um código especifico chamado bytecode – um código independente. Este código é gerado pelo compilador Java (javac) e será traduzido pela Virtual Machine para o código de cada máquina em questão.

DISTRIBUÍDO
O Java é uma tecnologia Open Source, criado pela Sun MicroSystems, empresa comprada pela Oracle. A Oracle é hoje a principal mantenedora da linguagem. Basta baixar a ferramenta e começar a desenvolver.

INDEPENDENTE DA PLATAFORMA
Acredito que seja o que mais repetimos aqui, a portabilidade do Java. Uma aplicação desenvolvida nesta plataforma, pode ser migrada facilmente para uma ampla variedade de sistemas de computadores. E pode ser utilizada em uma grande variedade de arquiteturas de hardware. E em uma grande variedade de arquiteturas de sistemas operacionais.

Ela não depende de uma plataforma exclusiva, ou de plataforma A ou B. Ou seja, uma aplicação escrita em Java simplesmente pode ser executada em qualquer lugar.

MULTITHREAD

Em primeiro lugar vamos entender o que é thread. A documentação do Java nos diz que podemos definir threads como uma determinada função de biblioteca implementada de tal maneira que pode ser executada por vários threads simultâneos de execução. OK, mas nós acreditamos que há um jeito mais simples de defini-las

Grosso modo threads são tarefas, processos que um dado programa executa, é uma forma de dividir a si mesmo em uma ou mais tarefas e realizá-las de forma concorrente. Agora mesmo uma thread está sendo executada no seu computador.

Lembre-se, o nosso contexto é o de programação e estamos desenhando um cenário para que seja de fácil assimilação por todos. Assim, a multithread é a maneira de obter simultaneidade rápida e leve em um único espaço de processo.

JAVA EM POUCAS PALAVRAS

O Java pode fazer muito mais por nós, e nos permite fazer mais, escrevendo menos código, sem dependência de plataformas. Escreva uma vez, execute em qualquer lugar.

E podemos aproveitar todo o poder da linguagem usufruindo de ferramentas de desenvolvimento que fornecem todo o suporte necessário. Suporte para compilar, executar, monitorar, depurar e documentar nossos sistemas. As APIs são um sabor a parte, o Java oferece uma ampla variedade de classes úteis prontas para usarmos.

Tecnologias de implementação, kits de ferramentas da interface do usuário, bibliotecas de integração, entre outros recursos.

15. **Apresentar um exemplo de backend em JAVA utilizando STS Spring Boot**

```
package br.com.treinaweb.springbootapi.controller;

import java.util.List;
import java.util.Optional;

import javax.validation.Valid;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RestController;

import br.com.treinaweb.springbootapi.entity.Pessoa;
import br.com.treinaweb.springbootapi.repository.PessoaRepository;

@RestController
```

```
public class PessoaController {
  @Autowired
  private PessoaRepository _pessoaRepository;

  @RequestMapping(value = "/pessoa", method = RequestMethod.GET)
  public List<Pessoa> Get() {
    return _pessoaRepository.findAll();
  }

  @RequestMapping(value = "/pessoa/{id}", method = RequestMethod.GET)
  public ResponseEntity<Pessoa> GetById(@PathVariable(value = "id") long id)
  {
    Optional<Pessoa> pessoa = _pessoaRepository.findById(id);
    if(pessoa.isPresent())
      return new ResponseEntity<Pessoa>(pessoa.get(), HttpStatus.OK);
    else
      return new ResponseEntity<>(HttpStatus.NOT_FOUND);
  }

  @RequestMapping(value = "/pessoa", method =  RequestMethod.POST)
  public Pessoa Post(@Valid @RequestBody Pessoa pessoa)
  {
    return _pessoaRepository.save(pessoa);
  }

  @RequestMapping(value = "/pessoa/{id}", method =  RequestMethod.PUT)
  public ResponseEntity<Pessoa> Put(@PathVariable(value = "id") long id, @Valid
@RequestBody Pessoa newPessoa)
  {
    Optional<Pessoa> oldPessoa = _pessoaRepository.findById(id);
    if(oldPessoa.isPresent()){
      Pessoa pessoa = oldPessoa.get();
      pessoa.setNome(newPessoa.getNome());
      _pessoaRepository.save(pessoa);
      return new ResponseEntity<Pessoa>(pessoa, HttpStatus.OK);
    }
    else
      return new ResponseEntity<>(HttpStatus.NOT_FOUND);
  }

  @RequestMapping(value = "/pessoa/{id}", method = RequestMethod.DELETE)
  public ResponseEntity<Object> Delete(@PathVariable(value = "id") long id)
  {
    Optional<Pessoa> pessoa = _pessoaRepository.findById(id);
    if(pessoa.isPresent()){
      _pessoaRepository.delete(pessoa.get());
      return new ResponseEntity<>(HttpStatus.OK);
    }
```

```
        else
            return new ResponseEntity<>(HttpStatus.NOT_FOUND);
    }
}
```

16. **Resolver 03 exercícios de uma das listas em JAVA: PUCRS ou UFBA**

01 Escrever um programa para determinar o consumo médio de um automóvel sendo fornecida a distância total percorrida pelo automóvel e o total de combustível gasto.

```
import java.util.Scanner;
public class Exerc3
{
public static void main (String args[])
{
Scanner input = new Scanner(System.in);
double TaxaDeConsumo; // media de combustível por km
double km1; // km inicial
double km2; // km final
double kmtotal; // km percorridos
double litros; // combustivel gasto

System.out.print("Informe o KM inicial: " );
km1 = input.nextDouble();

System.out.print("Informe o KM final: " );
km2 = input.nextDouble();

System.out.print("Informe a quantidade de litros consumidos:" );
litros = input.nextDouble();

kmtotal = km2 - km1;

TaxaDeConsumo = kmtotal / litros;

System.out.println("O total percorrido foi de" +kmtotal+ "Km");
System.out.println("A Taxa media de consumo é de" +TaxaDeConsumo+ "litros por km
percorrido");
}
}
```

2. Escrever um programa que leia o nome de um vendedor, o seu salário fixo e o total de vendas efetuadas por ele no mês (em dinheiro). Sabendo que este vendedor ganha 15% de comissão sobre suas vendas efetuadas, informar o seu nome, o salário fixo e salário no final do mês.

```
import java.util.Scanner;
```

```
public class Exerc4
{
public static void main(String args[])
{
Scanner input = new Scanner(System.in);

String nomeVendedor;
double salarioFixo; // Salario Fixo do vendedor
double vendas; // vendas efetuadas pelo vendedor
double comissao; // comissão de 15% sobre as vendas
double salarioFinal; // salario final ( fixo + comissao )

System.out.print(" Digite o nome do vendedor:" );
nomeVendedor = input.nextLine();

System.out.print("Digite o salário fixo do vendedor:" );
salarioFixo = input.nextDouble();

System.out.print("Digite o total de vendas efetuadas pelo vendedor:" );        vendas =
input.nextDouble();        comissao = ( vendas * 15 ) / 100;
salarioFinal = comissao + salarioFixo;

System.out.println("Nome do Vendedor:" +nomeVendedor);
System.out.println("Salario Fixo:" +salarioFixo+ "reais");
System.out.println("Salario Final:" +salarioFinal+ "reais");

}

}
```

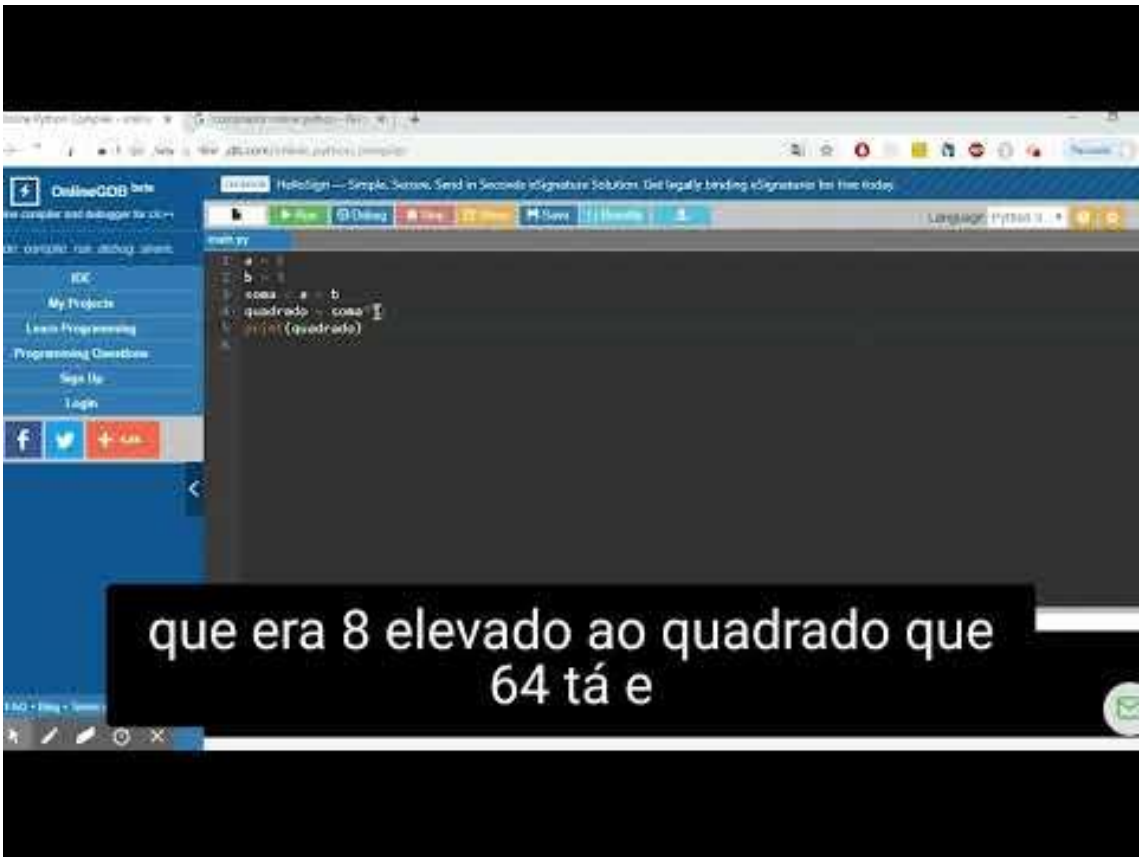
3 (*) E.P.J. para receber o nome de um aluno com suas respectivas 2 notas, em seguida calcular a média do aluno e apresentar ao final a média calculada e a situação de Aprovação do aluno.
(aprovado com média >= 6).

(*) Utilizar o código para uma turma de 30 alunos.
(*) Calcular e mostrar a média geral da turma
(*) Mostrar a maior média da turma
(*) Mostrar a menor média da turma

```
package br.estacio.pri.exercicio;

import java.util.*;

public class TurmaAlunos
{
  public static void main(String a[])
```

```
  {
    Scanner teclado;
    teclado = new Scanner(System.in);

    String aluno;
    float nota1, nota2, media, soma, mediaTurma;

    soma = 0;
    for(int i=0; i<3; i++)
    {
      System.out.print("Nome do Aluno: ");
      aluno = teclado.nextLine();
      System.out.print("Nota 1: ");
      nota1 = teclado.nextFloat();
      System.out.print("Nota 2: ");
      nota2 = teclado.nextFloat();
      teclado.nextLine();

      media = (nota1+nota2)/2;
      soma = soma + media;
      System.out.printf("Média do aluno é %.1f\n",  media);
      if (media >= 6)
        System.out.println("Aluno Aprovado. Parabéns.");
      else
        System.out.println("Reprovado! Estude mais.");
    }

    mediaTurma = soma/3;
    System.out.printf("Média da Turma = %.1f\n",  mediaTurma);

    teclado.close();
  }
}
```

17. **VIDEO –** DESCULPA QUALQUER ERRO ORTOGRÁFICO POIS A LEGENDA FOI EDITADO POR UM SITE.



CONCLUSÃO

Analisar o lucro e o custo previamente é um passo importante ao desenvolver um software. Por isso, questões como tarefas, exigências, recursos, objetivos e utilidades precisam ser verificadas para que haja um levantamento de gastos. Isso vai possibilitar que as melhores metodologias sejam adotadas no desenvolver do projeto. Análise de requisitos de software É importante saber que os requisitos de um projeto são as especificações do sistema. Ou seja, o que ele precisa solucionar, quando e como. Basicamente, é pensar nos objetivos que precisa alcançar, utilidade para o usuário final e o retorno que deve atingir. Para que um processo de desenvolvimento de software seja eficiente, uma análise de requisitos é fundamental. Isso porque é por meio dela que se torna possível garantir que as necessidades do cliente sejam atendidas. É uma etapa capaz de direcionar as melhores abordagens para as demandas e tornar um produto final funcional e seguro. Especificações de usuário Nesta parte, as solicitações do cliente passam a ser formalizadas. Isso quer dizer que as especificações do usuário precisam ser estudadas e detalhadas junto ao levantamento de requisitos realizado anteriormente. Desse modo, torna-se possível arquitetar modelos do sistema. Então, a finalidade dessa etapa é traçar as melhores estratégias para solucionar os problemas e evidenciar o que, de fato, deve ser feito. Esse processo deve ser capaz de permitir visualizar como o sistema vai funcionar diante dos requisitos exigidos. Arquitetura de software Chegamos à parte em que o processo de desenvolvimento de software permite que o sistema comece a ser desenhado. A arquitetura de software é a base de todo o projeto. Isso porque ela define o funcionamento interno do sistema para que todas as especificações sejam atendidas. Esse processo envolve a própria estrutura do sistema, além de questões referentes a linguagem de programação, banco de dados, design de interface, entre outras. Normalmente, isso é feito por um arquiteto de softwares, que direciona todos os componentes citados anteriormente. Implementação Já nesta fase, a codificação, de fato, começa a ser feita. Isso sempre seguindo as especificações realizadas nas etapas anteriores. Ou seja, é o momento em que o sistema é codificado em uma linguagem de programação. Esta, por sua vez, compila e gera o código-executável para o desenvolvimento do software. Há várias linguagens que podem ser utilizadas no processo de desenvolvimento de software, como Delphi, C++ e

REFERÊNCIAS (SITES)

- Extreme Programming: a Gentle Introduction
- XProgramming.com: An eXtreme Programming Resource
- Agile Alliance.
- Agile Modeling
- Scrum
- Dynamic System Development Method (DSDM)
- Site de Martin Fowler, vários artigos sobre processos
- The Rational Edge, e-magazine da Rational
- Rational Unified Process
- Catalysis Website
- OPEN Process Framework
- IEEE Software Magazine
- Software Engeneering Information Reposiroty
- Software Engineering Institute
- CMMI Institute
- Site Oficial do Modelo IDEAL