UNIFEOB

CENTRO UNIVERSITÁRIO DA FUNDAÇÃO DE ENSINO OCTÁVIO BASTOS

ESCOLA DE NEGÓCIOS

ANÁLISE E DESENVOLVIMENTO DE SISTEMAS

GESTÃO DE TECNOLOGIA DA INFORMAÇÃO

PROJETO INTEGRADO

SISTEMA EMPRESARIAL

SÃO JOÃO DA BOA VISTA, SP

ABRIL 2023

UNIFEOB

CENTRO UNIVERSITÁRIO DA FUNDAÇÃO DE ENSINO OCTÁVIO BASTOS

ESCOLA DE NEGÓCIOS

ANÁLISE E DESENVOLVIMENTO DE SISTEMAS

GESTÃO DE TECNOLOGIA DA INFORMAÇÃO

PROJETO INTEGRADO

SISTEMA EMPRESARIAL

MÓDULO DESENVOLVIMENTO DESKTOP

Banco de Dados – Prof. Sidney Gitcoff Telles

Programação Orientada a Objeto – Prof. Sidney Gitcoff Telles

Projeto de Desenvolvimento Desktop – Prof. Sidney Gitcoff Telles

Estudantes:

Jhonatan Ferreira da Silva
Jonas Juan Pereira Gomes
Lucca Schoneborn de Castro
Michael Amorim André
Suzana Sampaio Braga dos Santos de Oliveira

Grupo 8

SÃO JOÃO DA BOA VISTA, SP
ABRIL, 2023

Etapa de desenvolvimento do projeto:

- ➔ Primeira reunião da equipe via meet dia 14/03: link https://meet.google.com/ebe-mqoz-ouu

J.A.Madeiras LTDA - Poços de Caldas - MG

Foi Verificado junto a empresa a necessidade de um sistema para cadastro de funcionários, onde RH fez um pedido, que fosse feito um sistema para que eles pudessem cadastrar e alterar dados dos funcionários

PROBLEMA DE NEGÓCIO

Devido ao alto número de entradas de funcionários, a empresa decidiu pela criação de um sistema base de controle de funcionários.

Com a possibilidade de realizar o cadastro de funcionários e a alterações de entrada e saída de funcionários. assim automatizando o processo de contratação e pagamento via sistema

REQUISITOS DE SISTEMA

R1 - Permissão de entrada

O sistema deve permitir que os usuários criem uma conta inserindo seu nome de usuário e senha

R1.1 - Permissão de login

O sistema deve permitir que os usuários façam login usando seu nome de usuário e senha

R1.2 - Mensagem de sucesso ou erro

O sistema deve exibir mensagem de erro claras e informativas para os usuário em caso de erro no login

R1.3 - Sistema de recuperação de senhas

O sistema deve permitir que os usuários resistam sua senha por meio de um processo de recuperação de senha.

R1.4 - Armazenamento de dados

O sistema deve armazenar informações de login do usuário com segurança e privacidade

REQUISITOS NÃO FUNCIONAIS

RN1 - Sistema

RN1.1 - Sistema confiável

O sistema deve ser confiável e estar disponível a qualquer momento, garantindo um tempo de atividade mínim

RN1.2 - Quantidade de Usuários

O sistema deve ser capaz de suportar um grande número de usuários simultaneamente.

RN1.3 - Usabilidade do sistema

O sistema deve ser fácil de usar e intuitivo, com uma interface de usuário clara e organizada.

RN1.4 - Segurança

O sistema deve ter um alto nível de segurança, usando técnicas de criptografia e autenticação para proteger as informações do usuário.

RN1.5 - Escalabilidade

O sistema deve ser escalável e modular, permitindo uma fácil manutenção e expansão do sistema

REGRAS DE NEGÓCIO

RN1 - Segurança

O sistema deve garantir que os usuários criem senhas fortes e exclusivas para evitar ataques de hacker

RN1.1 Compartilhamento de Senha

Os usuários não devem compartilhar suas senhas com outras pessoas e devem manter suas informações de login em sigilo

RN1.2 - Tentativas de login

.O sistema deve impor limites para o número de tentativas de login incorretas, após o qual o usuário deve aguardar um determinado período antes de tentar fazer login novamente.

RN 1.3 - Log de login

O sistema deve registrar todas as tentativas de login, bem como todas as alterações de senha e enviar notificações de e-mail para o usuário, caso sua conta seja acessada de um local diferente ou em horários incomuns.

RN1.4 - LGPD
O sistema deve estar em conformidade com as leis e regulamentos de privacidade de dados aplicáveis, protegendo as informações pessoais dos usuários e obtendo seu consentimento para coletar e processar seus dados.

Programação orientada a objetos e programação estruturada

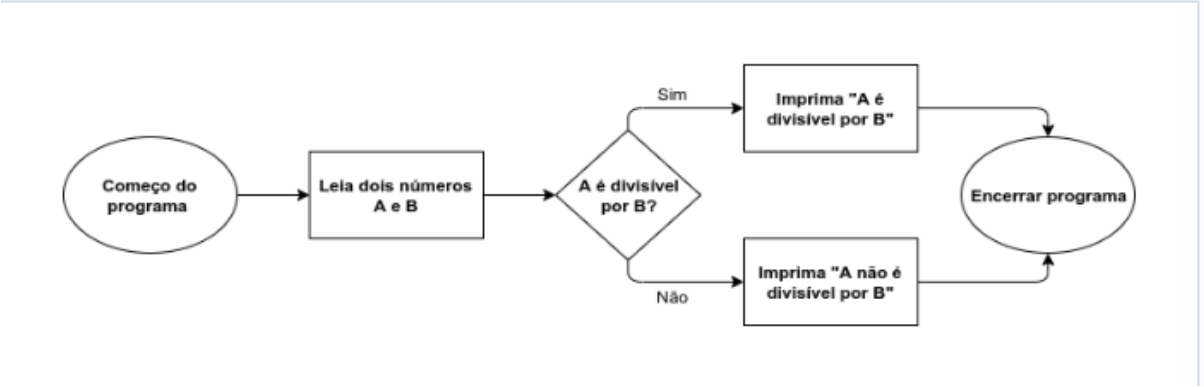
Como a maioria das atividades que fazemos no dia a dia, programar também possui modos diferentes de se fazer. Esses modos são chamados de **paradigmas de programação** e, entre eles, estão a **programação orientada a objetos** (POO) e a programação estruturada. Quando começamos a utilizar linguagens como Java, C#, Python e outras que possibilitam o paradigma orientado a objetos, é comum errarmos e aplicarmos a programação estruturada achando que estamos usando recursos da orientação a objetos.

Na programação estruturada, um programa é composto por três tipos básicos de estruturas:

- sequências: são os comandos a serem executados
- condições: sequências que só devem ser executadas se uma condição for satisfeita (exemplos: if-else, switch e comandos parecidos)
- repetições: sequências que devem ser executadas repetidamente até uma condição for satisfeita (for, while, do-while etc)

Essas estruturas são usadas para processar a entrada do programa, alterando os dados até que a saída esperada seja gerada. Até aí, nada que a programação orientada a objetos não faça, também, certo?

A diferença principal é que na programação estruturada, um programa é tipicamente escrito em uma única rotina (ou função) podendo, é claro, ser quebrado em subrotinas. Mas o fluxo do programa continua o mesmo, como se pudéssemos copiar e colar o código das subrotinas diretamente nas rotinas que as chamam, de tal forma que, no final, só haja uma grande rotina que execute todo o programa.



Além disso, o acesso às variáveis não possuem muitas restrições na programação estruturada. Em linguagens fortemente baseadas nesse paradigma, restringir o acesso à uma variável se limita a dizer se ela é visível ou não dentro de uma função (ou módulo, como no uso da palavra-chave static, na linguagem C), mas não se consegue dizer de forma nativa que uma variável pode ser acessada por apenas algumas rotinas do programa. O contorno para situações como essas envolve práticas de programação danosas ao desenvolvimento do sistema, como o uso excessivo de variáveis globais. Vale lembrar que variáveis globais são usadas tipicamente para manter estados no programa, marcando em qual parte dele a execução se encontra.

A **programação orientada a objetos** surgiu como uma alternativa a essas características da programação estruturada. O intuito da sua criação também foi o de aproximar o manuseio das estruturas de um programa ao manuseio das coisas do mundo real, daí o nome "objeto" como algo genérico, que pode representar qualquer coisa tangível.

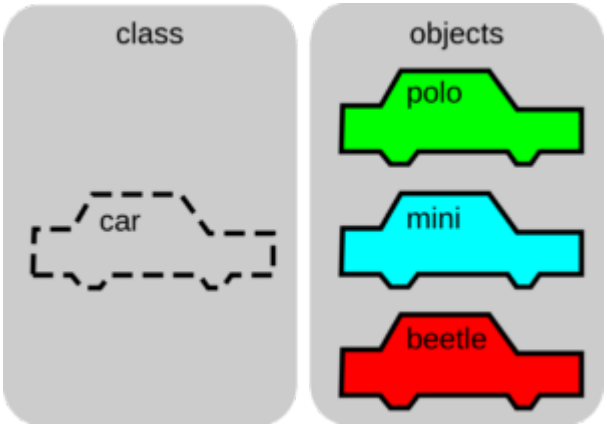
Esse novo paradigma se baseia principalmente em dois conceitos chave: **classes** e **objetos**. Todos os outros conceitos, igualmente importantes, são construídos em cima desses dois.

O que são classes e objetos?

Imagine que você comprou um carro recentemente e decide modelar esse carro usando programação orientada a objetos. O seu carro tem as características que você estava procurando: um motor 2.0 híbrido, azul escuro, quatro portas, câmbio automático etc. Ele também possui comportamentos que, provavelmente, foram o motivo de sua compra, como acelerar, desacelerar, acender os faróis, buzinar e

tocar música. Podemos dizer que o carro novo é um *objeto*, onde suas características são seus *atributos* (dados atrelados ao objeto) e seus comportamentos são ações ou *métodos*.

Seu carro é um objeto seu mas na loja onde você o comprou existiam vários outros, muito similares, com quatro rodas, volante, câmbio, retrovisores, faróis, dentre outras partes. Observe que, apesar do seu carro ser único (por exemplo, possui um registro único no Departamento de Trânsito), podem existir outros com exatamente os mesmos atributos, ou parecidos, ou mesmo totalmente diferentes, mas que ainda são considerados *carros*. Podemos dizer então que seu objeto pode ser classificado (isto é, seu *objeto pertence à uma classe*) como um carro, e que seu carro nada mais é que uma *instância* dessa *classe* chamada "carro".



Assim, abstraindo um pouco a analogia, uma classe é um conjunto de características e comportamentos que definem o conjunto de objetos pertencentes à essa classe. Repare que a classe em si é um conceito abstrato, como um molde, que se torna concreto e palpável através da criação de um objeto. Chamamos essa criação de *instância da classe*, como se estivéssemos usando esse molde (classe) para criar um objeto.

```
public class Carro {
    Double velocidade;
    String modelo;

    public Carro(String modelo) {
        this.modelo = modelo;
        this.velocidade = 0;
    }

    public void acelerar() {
```

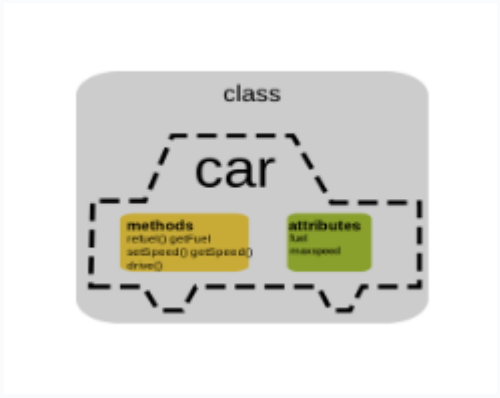
```
        /* código do carro para acelerar */
    }

    public void frear() {
        /* código do carro para frear */
    }

    public void acenderFarol() {
        /* código do carro para acender o farol */
    }
}
```

Encapsulamento, herança e polimorfismo: as principais características da POO

As duas bases da **POO** são os conceitos de classe e objeto. Desses conceitos, derivam alguns outros conceitos extremamente importantes ao paradigma, que não só o definem como são as soluções de alguns problemas da **programação estruturada**. Os conceitos em questão são o *encapsulamento*, a *herança*, as *interfaces* e o *polimorfismo*.

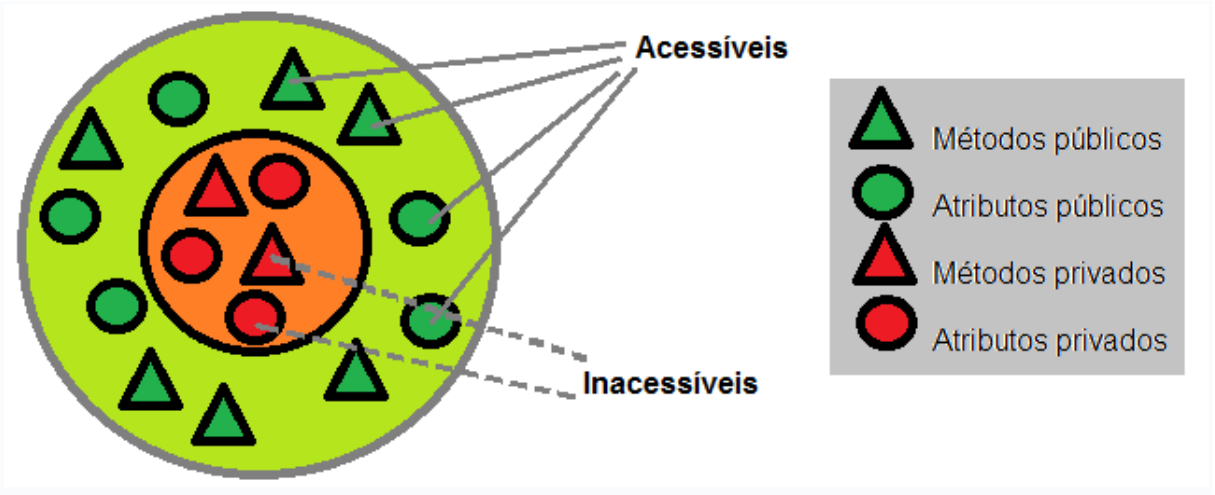
Encapsulamento

Ainda usando a analogia do carro, sabemos que ele possui atributos e métodos, ou seja, características e comportamentos. Os métodos do carro, como acelerar, podem usar atributos e outros métodos do carro como o tanque de gasolina e o mecanismo de injeção de combustível, respectivamente, uma vez que acelerar gasta combustível.



No entanto, se alguns desses atributos ou métodos forem facilmente visíveis e modificáveis, como o mecanismo de aceleração do carro, isso pode dar liberdade para que alterações sejam feitas, resultando em efeitos colaterais imprevisíveis. Nessa analogia, uma pessoa pode não estar satisfeita com a aceleração do carro e modifica a forma como ela ocorre, criando efeitos colaterais que podem fazer o carro nem andar, por exemplo.

Dizemos, nesse caso, que o método de aceleração do seu carro não é visível por fora do próprio carro. Na POO, um atributo ou método que não é visível de fora do próprio objeto é chamado de "privado" e quando é visível, é chamado de "público".



Mas então, como sabemos como o nosso carro acelera? É simples: não sabemos. Nós só sabemos que para acelerar, devemos pisar no acelerador e de resto o objeto sabe como executar essa ação sem expor como o faz. Dizemos que a aceleração do carro está *encapsulada*, pois sabemos o que ele vai fazer ao executarmos esse método, mas não sabemos como - e na verdade, não importa para o programa como o objeto o faz, só que ele o faça.

O mesmo vale para atributos. Por exemplo: não sabemos como o carro sabe qual velocidade mostrar no velocímetro ou como ele calcula sua velocidade, mas não precisamos saber como isso é feito. Só precisamos saber que ele vai nos dar a velocidade certa. Ler ou alterar um atributo encapsulado pode ser feito a partir de *getters e setters* (colocar referência).

Esse *encapsulamento* de atributos e métodos impede o chamado *vazamento de escopo*, onde um atributo ou método é visível por alguém que não deveria vê-lo, como outro objeto ou classe. Isso evita a confusão do uso de variáveis globais no programa, deixando mais fácil identificar em qual estado cada variável vai estar a cada momento do programa, já que a restrição de acesso nos permite identificar quem consegue modificá-la.

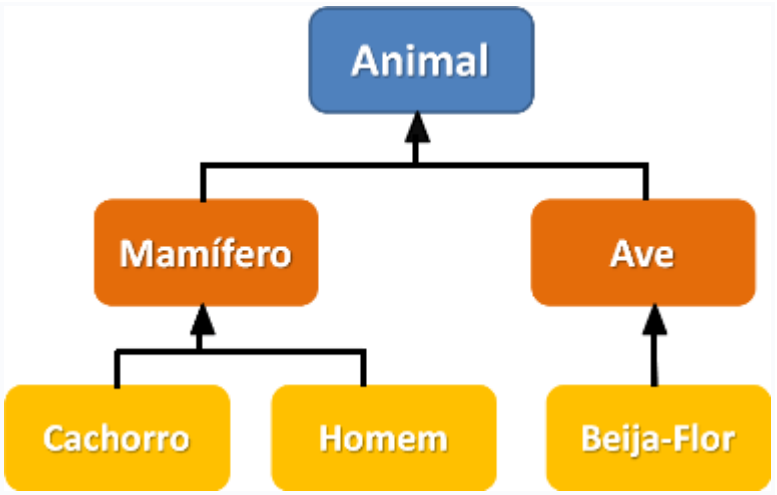
Exemplo em Java

```
public class Carro {
    private Double velocidade;
    private String modelo;
    private MecanismoAceleracao mecanismoAceleracao;
    private String cor;

     /* Repare que o mecanismo de aceleração é inserido no carro ao ser construído, e
          não o vemos nem podemos modificá-lo, isto é, não tem getter nem setter.
        Já o "modelo" pode ser visto, mas não alterado. */
                    public Carro(String modelo, MecanismoAceleracao mecanismoAceleracao) {
        this.modelo = modelo;
        this.mecanismoAceleracao = mecanismoAceleracao;
        this.velocidade = 0;
    }

    public void acelerar() {
        this.mecanismoAceleracao.acelerar();
    }

    public void frear() {
        /* código do carro para frear */
    }

    public void acenderFarol() {
        /* código do carro para acender o farol */
    }
```

```
    public Double getVelocidade() {
        return this.velocidade
    }

    private void setVelocidade() {
        /* código para alterar a velocidade do carro */
        /* Como só o próprio carro deve calcular a velocidade,
            esse método não pode ser chamado de fora, por isso é "private" */
    }

    public String getModelo() {
        return this.modelo;
    }

    public String getCor() {
        return this.cor;
    }

    /* podemos mudar a cor do carro quando quisermos */
    public void setCor(String cor) {
        this.cor = cor;
    }
}
```

Herança

No nosso exemplo, você acabou de comprar um carro com os atributos que procurava. Apesar de ser único, existem carros com exatamente os mesmos atributos ou formas modificadas. Digamos que você tenha comprado o modelo Fit, da Honda. Esse modelo possui uma outra versão, chamada WR-V (ou "Honda Fit Cross Style"), que possui muitos dos atributos da versão clássica, mas com algumas diferenças bem grandes para transitar em estradas de terra: o motor é híbrido (aceita álcool e gasolina), possui um sistema de suspensão diferente, e vamos supor que além disso ele tenha um sistema de tração diferente (tração nas quatro rodas, por exemplo). Vemos então que não só alguns atributos como também alguns mecanismos (ou métodos, traduzindo para POO) mudam, mas essa versão "cross" ainda é do modelo Honda Fit, ou melhor, *é um tipo* do modelo.



Quando dizemos que uma classe A *é um tipo de* classe B, dizemos que a classe A *herda* as características da classe B e que a classe B é *mãe* da classe A, estabelecendo então uma relação de **herança** entre elas. No caso do carro, dizemos então que um Honda Fit "Cross" *é um tipo de* Honda Fit, e o que muda são alguns atributos (paralama reforçado, altura da suspensão etc), e um dos métodos da classe (acelerar, pois agora há tração nas quatro rodas), mas todo o resto permanece o mesmo, e o novo modelo recebe os mesmos atributos e métodos do modelo clássico.

Exemplo em Java

```
// "extends" estabelece a relação de herança dom a classe Carro
public class HondaFit extends Carro {

    public HondaFit(MecanismoAceleracao mecanismoAceleracao) {
        String modelo = "Honda Fit";
        // chama o construtor da classe mãe, ou seja, da classe "Carro"
        super(modelo, mecanismoAceleracao);
    }
}
```

Polimorfismo

Vamos dizer que um dos motivos de você ter comprado um carro foi a qualidade do sistema de som dele. Mas, no seu caso, digamos que a reprodução só pode ser feita via rádio ou *bluetooth*, enquanto que no seu antigo carro, podia ser feita apenas via cartão SD e *pendrive*. Em ambos os carros está presente o método "tocar música" mas, como o sistema de som deles é diferente, a forma como o carro

toca as músicas é diferente. Dizemos que o método "tocar música" é uma forma de **polimorfismo**, pois dois objetos, de duas classes diferentes, têm um mesmo método que é implementado de formas diferentes, ou seja, um método possui *várias formas*, várias implementações diferentes em classes diferentes, mas que possuem o mesmo efeito ("polimorfismo" vem do grego *poli* = muitas, *morphos* = forma).

Exemplo em Java

```
public class Main {

   public static void main(String[] args) {

                        Automovel moto = new Moto("Yamaha XPTO-100", new
MecanismoDeAceleracaoDeMotos())

                              Automovel carro = new Carro("Honda Fit", new
MecanismoDeAceleracaoDeCarros())

      List<Automovel> listaAutomoveis = Arrays.asList(moto, carro);

      for (Automovel automovel : listaAutomoveis) {

         automovel.acelerar();

         automovel.acenderFarol();

      }

   }
```

Design Patterns

Alguns problemas aparecem com tanta frequência em **POO** que suas soluções se tornaram padrões de design de sistemas e modelagem de código orientado a objeto, a fim de resolvê-los. Esses **padrões de projeto**, (ou **design patterns**) nada mais são do que formas padronizadas de resolver problemas comuns em linguagens orientadas a objetos. O livro "Design Patterns", conhecido como **Gof:Gang of Four**, é a principal referência nesse assunto, contendo os principais

padrões usados em grandes projetos. A Alura também oferece cursos de *Design Patterns* em linguagens de programação como Java, Python e C#.

O que é um Banco de Dados?

Banco de dados definido

Um banco de dados é uma coleção organizada de informações - ou dados - estruturadas, normalmente armazenadas eletronicamente em um sistema de computador. Um banco de dados é geralmente controlado por um sistema de gerenciamento de banco de dados (DBMS). Juntos, os dados e o DBMS, juntamente com os aplicativos associados a eles, são chamados de sistema de banco de dados, geralmente abreviados para apenas banco de dados.

Os dados nos tipos mais comuns de bancos de dados em operação atualmente são modelados em linhas e colunas em uma série de tabelas para tornar o processamento e a consulta de dados eficientes. Os dados podem ser facilmente acessados, gerenciados, modificados, atualizados, controlados e organizados. A maioria dos bancos de dados usa a linguagem de consulta estruturada (SQL) para escrever e consultar dados.

O que é SQL (Structured Query Language, Linguagem de consulta estruturada)?

SQL é uma linguagem de programação usada por quase todos os bancos de dados relacionais para consultar, manipular e definir dados e fornecer controle de acesso. O SQL foi desenvolvido pela primeira vez na IBM nos anos 1970, com a Oracle como principal contribuinte, o que levou à implementação do padrão SQL ANSI; o SQL estimulou muitas extensões de empresas como IBM, Oracle e Microsoft. Embora o SQL ainda seja amplamente usado hoje em dia, novas linguagens de programação estão começando a aparecer.

Evolução do banco de dados

Os bancos de dados evoluíram muito desde a sua criação no início dos anos 1960. Bancos de dados de navegação, como o banco de dados hierárquico (que se baseava em um modelo de árvore e permitia apenas um relacionamento um-para-muitos), e o banco de dados de rede (um modelo mais flexível que permitia múltiplos relacionamentos) eram os sistemas originais usados para armazenar e manipular dados. Embora simples, esses primeiros sistemas eram inflexíveis. Nos anos 1980, bancos de dados relacionais tornaram-se populares, seguidos por bancos de dados orientados a objetos na década de 1990. Mais recentemente, bancos de dados NoSQL surgiram como uma resposta ao crescimento da internet e à necessidade de maior velocidade e processamento de dados não estruturados. Hoje, bancos de dados na nuvem e bancos de dados autônomos estão abrindo novos caminhos quando se trata de como os dados são coletados, armazenados, gerenciados e utilizados.

Qual é a diferença entre um banco de dados e uma planilha?

Bancos de dados e planilhas (como o Microsoft Excel) são modos convenientes de armazenar informações. As principais diferenças entre os dois são:

Como os dados são armazenados e manipulados
Quem pode acessar os dados
Quantos dados podem ser armazenados

As planilhas foram originalmente projetadas para um usuário e suas características refletem isso. São ótimos para um único usuário ou um pequeno número de usuários que não precisam fazer a manipulação de dados muito complicada. Bancos de dados, por outro lado, são projetados para conter coleções muito maiores de informações organizadas - quantidades enormes, às vezes. Os bancos de dados permitem que vários usuários, ao mesmo tempo, acessem e consultem com rapidez e segurança os dados usando lógica e linguagem altamente complexas.

Tipos de bancos de dados

Existem muitos tipos diferentes de bancos de dados. O melhor banco de dados para uma organização específica depende de como a organização pretende usar os dados.

Bancos de dados relacionais Bancos de dados relacionais se tornaram dominantes na década de 1980. Os itens em um banco de dados relacional são organizados como um conjunto de tabelas com colunas e linhas. A tecnologia de banco de dados relacional fornece a maneira mais eficiente e flexível de acessar informações estruturadas.

Bancos de dados orientados a objeto As informações em um banco de dados orientado a objetos são representadas na forma de objetos, como na programação orientada a objetos.

Bancos de dados distribuídos banco de dados distribuído consiste em dois ou mais arquivos localizados em sites diferentes. O banco de dados pode ser armazenado em vários computadores, localizados no mesmo local físico ou espalhados por diferentes redes.

Data warehouses Um repositório central de dados, um data warehouse é um tipo de banco de dados projetado especificamente para consultas e análises rápidas.

Bancos de dados NoSQL NoSQL, ou banco de dados não relacional, permite que dados não estruturados e semiestruturados sejam armazenados e manipulados (em contraste com um banco de dados relacional, que define como todos os dados inseridos no banco de dados devem ser compostos). Os bancos de dados NoSQL se tornaram populares à medida que os aplicativos web se tornaram mais comuns e mais complexos.

Bancos de dados gráficos. Um banco de dados gráfico armazena dados em termos de entidades e os relacionamentos entre entidades.

Bancos de dados OLTP. Um banco de dados OLTP é um banco de dados rápido e analítico projetado para um grande número de transações realizadas por vários usuários. Esses são apenas alguns dos vários tipos de bancos de dados em uso atualmente. Outros bancos de dados menos comuns são adaptados para funções científicas, financeiras ou outras

muito específicas. Além dos diferentes tipos de banco de dados, as mudanças nas abordagens de desenvolvimento de tecnologia e os avanços dramáticos, como a nuvem e a automação, estão impulsionando os bancos de dados em direções totalmente novas. Alguns dos mais recentes bancos de dados incluem

Bancos de dados de código aberto. Um sistema de banco de dados de código aberto é aquele cujo código-fonte é código aberto; esses bancos de dados podem ser bancos de dados SQL e NoSQL.

Bancos de dados em nuvem. Um banco de dados em nuvem é uma coleção de dados, estruturados ou não estruturados, que residem em uma plataforma de computação em nuvem privada, pública ou híbrida. Existem dois tipos de modelos de banco de dados em nuvem: tradicional e banco de dados como serviço (DBaaS). Com o DBaaS, as tarefas administrativas e a manutenção são executadas por um provedor de serviços.

Banco de dados multi modelo. Bancos de dados multi modelo combinam diferentes tipos de modelos de banco de dados em um back-end único e integrado. Isso significa que eles podem acomodar vários tipos de dados.

Banco de dados de documentos/JSON. Projetado para armazenamento, recuperação e gerenciamento de informações orientadas a documentos, os bancos de dados de documentos são uma maneira moderna de armazenar dados no formato JSON, em vez de linhas e colunas.

Bancos de dados autônomos. Os bancos de dados independentes mais novos e inovadores (também conhecidos como bancos de dados autônomos) são baseados em nuvem e usam machine learning para automatizar o ajuste de banco de dados, segurança, backups, atualizações e outras tarefas de gerenciamento de rotina tradicionalmente executadas por administradores de banco de dados.

O que é um software de banco de dados?

O software de banco de dados é usado para criar, editar e manter arquivos e registros de banco de dados, facilitando a criação de arquivos e registros, entrada

de dados, edição, atualização e relatórios de dados. O software também processa armazenamento de dados, backup e relatórios, controle multi acesso e segurança. A segurança forte do banco de dados é especialmente importante hoje, porque o roubo de dados se torna mais frequente. O software de banco de dados às vezes também é conhecido como "sistema de gerenciamento de banco de dados" (DBMS).

O software de banco de dados simplifica o gerenciamento de dados, permitindo que os usuários armazenem dados em um formulário estruturado e depois os acessem. Ele normalmente tem uma interface gráfica para ajudar a criar e gerenciar os dados e, em alguns casos, os usuários podem construir os próprios bancos de dados usando o software do banco de dados.

O que é um sistema de gerenciamento de banco de dados (DBMS)?

Um banco de dados normalmente requer um programa abrangente de banco de dados, conhecido como sistema de gerenciamento de banco de dados (DBMS). Um DBMS serve como uma interface entre o banco de dados e seus usuários finais ou programas, permitindo que os usuários recuperem, atualizem e gerenciem como as informações são organizadas e otimizadas. Um DBMS também facilita a supervisão e o controle de bancos de dados, permitindo uma variedade de operações administrativas, como monitoramento de desempenho, ajuste e backup e recuperação.

Alguns exemplos de softwares de bancos de dados populares ou DBMSs incluem MySQL, Microsoft Access, Microsoft SQL Server, FileMaker Pro, Oracle Database e dBASE.

O que é um MySQL Database?

MySQL é um sistema de gerenciamento de banco de dados relacional de código aberto baseado em SQL. Ele foi projetado e otimizado para aplicativos da web e pode ser executado em qualquer plataforma. Como surgiram requisitos novos e diferentes com a internet, o MySQL tornou-se a plataforma preferida para desenvolvedores da web e aplicativos baseados na web. Como foi projetado para processar milhões de consultas e milhares de transações, o MySQL é uma escolha popular para empresas de comércio eletrônico que precisam gerenciar várias

transferências de dinheiro. A flexibilidade sob demanda é o principal recurso do MySQL.

O MySQL é o DBMS por trás de alguns dos principais sites e aplicativos baseados na web do mundo, incluindo Airbnb, Uber, LinkedIn, Facebook, Twitter e YouTube.

Bancos de dados para aprimorar o desempenho e a tomada de decisões nos negócios

Com a coleta maciça de dados da Internet das Coisas, transformando a vida e o setor em todo o mundo, as empresas hoje têm acesso a mais dados do que nunca. Organizações inovadoras agora podem usar bancos de dados que vão além do armazenamento de dados e de transações básicas para analisar grandes quantidades de dados de vários sistemas. Ao usar bancos de dados e outras ferramentas de business intelligence e computação, as organizações aproveitam dados que coletam para executar funções com mais eficiência, possibilitar melhor tomada de decisões e serem mais rápidas e escalonáveis. A otimização do acesso e do throughput aos dados é fundamental para as empresas de hoje, pois há mais volume de dados a ser rastreado. É fundamental ter uma plataforma que possa oferecer o desempenho, a escala e a agilidade necessários às empresas à medida que crescem com o tempo.

O banco de dados autônomo está pronto para fornecer um impulso significativo a esses recursos. Como os bancos de dados autônomos automatizam processos manuais caros e demorados, eles liberam utilizadores de negócios para se tornarem mais proativos com seus dados. Por ter controle direto sobre a capacidade de criar e usar bancos de dados, os usuários ganham controle e autonomia enquanto mantêm importantes padrões de segurança.

Desafios do banco de dados

Os grandes bancos de dados empresariais atuais geralmente suportam consultas muito complexas e devem fornecer respostas quase instantâneas a essas consultas. Como resultado, os administradores de bancos de dados são constantemente chamados para empregar uma ampla variedade de métodos que

ajudam a melhorar o desempenho. Alguns desafios comuns que eles enfrentam incluem:

Absorção de aumentos significativos no volume de dados. A explosão de dados provenientes de sensores, máquinas conectadas e dezenas de outras fontes mantém os administradores de bancos de dados lutando para gerenciar e organizar os dados de suas empresas com eficiência.

Garantia da segurança de dados. Violações de dados estão acontecendo em todos os lugares nos dias de hoje, e os hackers estão ficando mais inventivos. É mais importante do que nunca garantir que os dados estejam seguros, mas também acessíveis aos usuários.

Acompanhando a demanda. No atual ambiente de negócios de rápido movimento, as empresas precisam de acesso em tempo real aos seus dados para apoiar a tomada de decisões em tempo hábil e aproveitar novas oportunidades.

Gerenciamento e manutenção do banco de dados e da infraestrutura. Os administradores de banco de dados devem observar continuamente o banco de dados em busca de problemas e executar a manutenção preventiva, bem como aplicar atualizações e correções de software. À medida que os bancos de dados se tornam mais complexos e o volume de dados aumenta, as empresas enfrentam a despesa de contratar mais talentos para monitorar e ajustar seus bancos de dados.

Remoção de limites na escalabilidade. Uma empresa precisa crescer se quiser sobreviver, e seu gerenciamento de dados deve crescer junto com ela. Mas é muito difícil para os administradores de banco de dados prever a capacidade que a empresa precisará, principalmente com bancos de dados on-premise.

Garantir residência, soberania de dados ou requisitos de latência. Algumas organizações têm casos de uso mais adequados para sua execução on-premises. Nesses casos, os sistemas projetados pré-configurados e pré-otimizados para a execução do banco de dados são ideais.

Resolver todos esses desafios pode consumir muito tempo e impedir que os administradores de banco de dados executem mais funções estratégicas.

Como a tecnologia autônoma está aprimorando o gerenciamento de banco de dados

Os bancos de dados autônomos são a onda do futuro - e oferecem uma possibilidade intrigante para as organizações que desejam usar a melhor tecnologia de banco de dados disponível sem as dores de cabeça da execução e da operação dessa tecnologia.

Os bancos de dados autônomos usam tecnologia baseada em nuvem e machine learning para automatizar muitas das tarefas de rotina necessárias para gerenciar bancos de dados, como ajuste, segurança, backups, atualizações e outras tarefas de gerenciamento de rotina. Com essas tarefas tediosas automatizadas, os administradores de banco de dados ficam livres para fazer um trabalho mais estratégico. Os recursos autônomos de autocondução, autoproteção e autorreparo dos bancos de dados independentes estão prestes a revolucionar a forma como as empresas gerenciam e protegem seus dados, possibilitando vantagens de desempenho, custos mais baixos e segurança aprimorada.

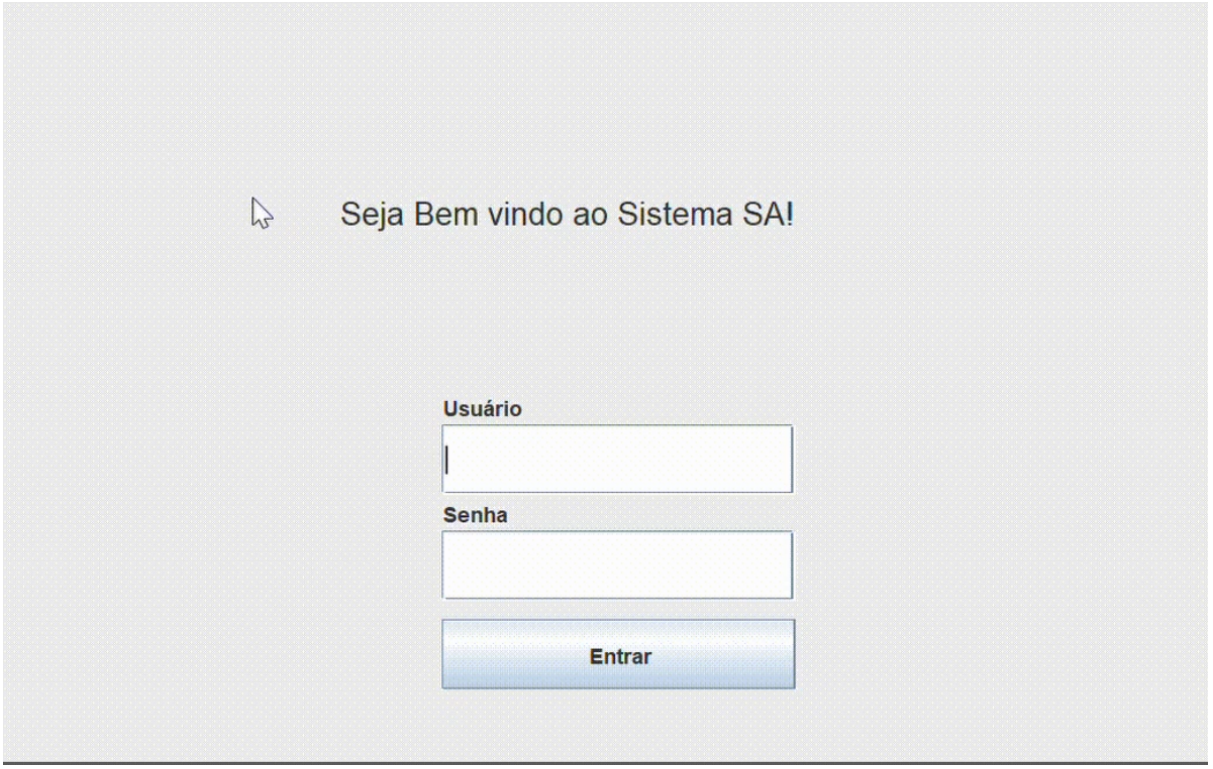
Futuro dos bancos de dados e bancos de dados autônomos

O primeiro banco de dados autônomo foi anunciado no final de 2017, e vários analistas independentes do setor rapidamente reconheceram a tecnologia e seu impacto potencial na computação.
A partir das informações adquiridas nas reuniões decidimos criar um banco de dados utilizando MySql na parte de interface e código utilizamos da linguagem JAVA por ser uma linguagem muito utilizada com POO.

Nesta primeira parte criamos o banco de dados, Segue a imagem abaixo.

Field	Type	Null	Key	Default	Extra
nome	varchar(50)	NO		NULL	
sobrenome	varchar(50)	NO		NULL	
data_nascimento	varchar(50)	NO		NULL	
email	varchar(50)	YES		NULL	
cargo	int	NO		NULL	
salario	decimal(10,0)	YES		NULL	
id	int	NO	PRI	NULL	auto_increment

Já na segunda parte fizemos a UI do nosso sistema utilizando a ferramenta Eclipse onde fizemos um sistema de login bem simples e com uma interface fácil de se utilizar.



Logo em seguida criamos a ligação do banco de dados com o sistema onde foi se utilizado o comando “GET” para puxar a conexão com o servidor do banco de dados.

```java
package conectividade;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

public class Main {
    public static void main(String[] args){
        String driver = "com.mysql.jdbc.Driver";
        String servidor = "jdbc:mysql://localhost:3306/sistema_de_funcionarios?useTimezone=true&serverTimezone=UTC&useSSL=false";
        String usuario = "root";
        String senha = "q1w2e3r4";

        Connection conexao;

        Statement instrucaoSQL;

        ResultSet resultados;

        try{
            conexao = DriverManager.getConnection(servidor, usuario, senha);

            instrucaoSQL = conexao.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE, ResultSet.CONCUR_READ_ONLY);
            resultados = instrucaoSQL.executeQuery("SELECT * FROM funcionarios");
            System.out.println("deu certo");
        } catch(SQLException erro){
            System.out.println(erro.getMessage());
        }
    }
}
```

Nesta parte a mostra código quando um funcionário e inserido no sistema

```
+-------------+------------------+------+-----+--------------------+--------------------+
| Field       | Type             | Null | Key | Default            | Extra              |
+-------------+------------------+------+-----+--------------------+--------------------+
| id          | bigint unsigned  | NO   | PRI | NULL               | auto_increment     |
| nome_cargo  | varchar(50)      | NO   |     | NULL               |                    |
| created_at  | timestamp        | YES  |     | CURRENT_TIMESTAMP  | DEFAULT_GENERATED  |
+-------------+------------------+------+-----+--------------------+--------------------+
```

Aqui é a parte de código da "CLASSE" do funcionário.

```java
package entidade;

public class Funcionario {
    //dados funcionario
    private int id;
    private String nome;
    private String sobrenome;
    private String dataNascimento;
    private String email;
    private int cargo;
    private double salario;

    public int getId() {
        return id;
    }

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

    public String getNome() {
        return nome;
    }

    public void setNome(String nome) {
        this.nome = nome;
    }

    public String getSobrenome() {
        return sobrenome;
    }

    public void setSobrenome(String sobrenome) {
        this.sobrenome = sobrenome;
    }
```

Bibliografia:

https://www.oracle.com/br/autonomous-database/what-is-autonomous-database/

https://www.oracle.com/br/database

https://youtu.be/jpuJ1qrluoU

POO: o que é programação orientada a objetos? | Alura