DESENVOLVIMENTO WEB – BACK-END

AULA 1

Profª Luciane Yanase Hirabara Kanashiro

CONVERSA INICIAL

Nesta etapa, veremos um pouco sobre a arquitetura de *software*, mais precisamente sobre arquitetura de 3 camadas e conheceremos um pouco sobre a plataforma Java e a arquitetura Java. Vamos explorar alguns conceitos fundamentais e mergulhar em alguns pontos-chave:

No tópico 1, abordaremos a relação entre a arquitetura de *software* e padrões de projetos. Nos tópicos 2 e 3, aprenderemos sobre a Arquitetura Java EE, seus componentes, serviços e bibliotecas. No tópico 4, abordaremos o Eclipse Jakarta EE e o impacto prático da mudança de Java EE para Jakarta EE para os programadores Java. Por fim, no tópico 5, apesar de estarmos em um assunto mais avançado, iremos revisitar algumas classes do Java, pois é sempre benéfico revisitar os fundamentos.

TEMA 1 – RELAÇÃO ENTRE ARQUITETURA DE SOFTWARE E PADRÃO DE PROJETOS

A arquitetura de software e os padrões de projetos estão intimamente relacionados no desenvolvimento de software. A arquitetura de *software* refere-se à estrutura fundamental do sistema, que compreende componentes, relacionamentos, princípios de *design* e diretrizes organizacionais. Ela se concentra em decisões de alto nível que afetam a estrutura geral do sistema, como a escolha de padrões arquiteturais, a distribuição de responsabilidades e a comunicação entre os componentes. Como exemplo de arquitetura de *software* temos: Arquiteturas de três camadas, microsserviços, arquitetura orientada a eventos, entre outras.

Já os padrões de projeto são soluções reutilizáveis para problemas recorrentes no *design* de *software*. Eles são descrições de boas práticas de *design* que foram comprovadas ao longo do tempo por desenvolvedores experientes.

Os padrões de projeto concentram-se em níveis mais baixos de *design*, oferecendo soluções específicas para problemas de implementação. Eles ajudam a lidar com detalhes da implementação de classes e objetos.

Como exemplos de Padrões de projeto temos: Singleton, Factory Method, Observer, MVC (Model-View-Controller), Strategy, entre outros.

Sendo assim, temos essa relação íntima entre arquitetura de *software* e padrões de projeto, pois esta define a estrutura global do sistema, enquanto os padrões de projeto oferecem soluções específicas para problemas de *design* dentro dessa estrutura. Ambas as áreas, arquitetura de *software* e padrões de projeto, colaboram para criar sistemas robustos e flexíveis. Uma boa arquitetura fornece a estrutura geral, enquanto os padrões de projeto oferecem soluções específicas para problemas de implementação, promovendo a coesão, reusabilidade e manutenibilidade do código.

Nesta sessão não abordaremos os padrões de projetos, alguns deles serão discutidos em momento oportuno.

1.1 Arquitetura multicamadas

Com frequência falamos em arquitetura multicamadas. Mas você sabe o que é isso?

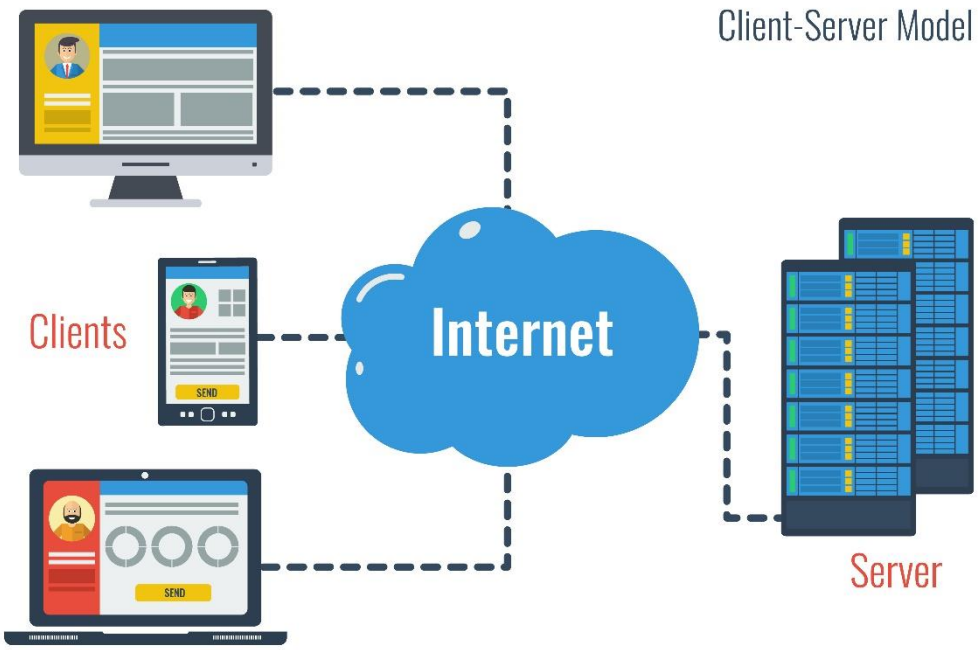
Uma **arquitetura multicamadas** é qualquer arquitetura que tenha mais de uma camada. Essa definição não parece ser muito esclarecedora, mas vamos a alguns exemplos para clarificar.

Uma arquitetura muito conhecida e que deu origem a arquitetura de 3 (ou +) camadas, é o **modelo cliente servidor**. O modelo cliente servidor é uma arquitetura de 2 camadas. De acordo com Fowler (2007):

> A noção de camadas se tornou mais visível nos anos 1990, com o advento dos sistemas cliente-servidores. Estes eram sistemas em duas camadas: o cliente mantinha a interface com o usuário e um ou outro código da aplicação, e o servidor era normalmente um banco de dados relacional. Ferramentas comuns para o lado cliente eram, por exemplo, VB, Powerbuilder e Delphi. Essas ferramentas tornaram particularmente fácil criar aplicações que faziam uso intensivo de dados, uma vez que elas disponibilizavam componentes visuais que trabalhavam com SQL. Assim, você podia criar uma tela arrastando controles para uma área de desenho e então usando páginas de propriedades para conectar os controles ao banco de dados.

A Figura 1 ilustra uma arquitetura cliente servidor.

Figura 1 – Arquitetura cliente servidor

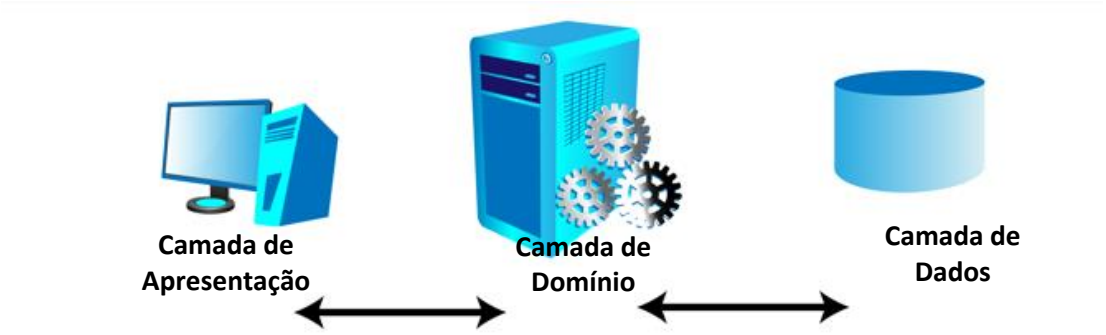


Crédito: VasutinSergey/ShutterStock.

Os sistemas cliente servidor funcionavam bem para aplicações que apenas exibiam e faziam alterações simples no Banco de Dados, porém a necessidade de um modelo com mais camadas começou a surgir quando as primeiras aplicações web começaram a aparecer. Essas aplicações web requerem uso de funcionalidades e camadas distintas, ou seja, temos que ter um servidor *web*, servidor de aplicações, servidor de banco de dados, *softwares* clientes.

Assim, surgiu o conceito de modelo multicamadas, também conhecido como **arquitetura em camadas**, podendo ser definido como um estilo de *design* de *software* em que a funcionalidade de um sistema é dividida em camadas distintas, cada uma responsável por uma parte específica da lógica de negócios ou da funcionalidade do aplicativo. Cada camada fornece serviços para a camada acima dela e consome serviços da camada abaixo, criando uma estrutura hierárquica. As principais camadas em um modelo multicamadas podem incluir: camada de apresentação, camada de domínio e camada de dados. A Figura 2 ilustra uma aplicação de 3 camadas.

Figura 2 – Aplicação de 3 camadas



Crédito: TechnoVectors/Shutterstock.

1.2 Camada lógica *versus* camada física

De acordo com Fowler (2007), quando o tema é a criação de camadas, é comum observar certa confusão entre os termos *layer* e *tier*. Frequentemente, as pessoas os utilizam como sinônimos, mas muitos consideram que *tier* implica uma separação física. Sistemas cliente-servidor são frequentemente categorizados como sistemas em duas camadas (*two-tier* systems), em que a separação é física: o cliente reside em um *desktop*, enquanto o servidor está em uma máquina dedicada. A opção por utilizar *layer* é para enfatizar que não é necessário executar as camadas em máquinas diferentes. Uma camada específica de lógica de domínio pode operar tanto em um *desktop* quanto no servidor de banco de dados. Nessa configuração, temos dois nós, mas três camadas distintas. Mesmo ao executar todas as três camadas em um único laptop com um banco de dados local, ainda mantemos três camadas claramente definidas. Sendo assim podemos encontrar duas definições distintas:

- **Tier (camada)**: é a camada física, executada em uma infraestrutura separada das outras camadas. A camada física representa a implementação concreta do sistema e como os componentes são distribuídos fisicamente em hardware ou ambientes de execução. Aqui, a atenção está voltada para questões como servidores, redes, banco de dados, máquinas virtuais, balanceamento de carga, entre outros. Trata-se da infraestrutura real que suporta a execução da aplicação;
- **Layer (nível)**: é a camada lógica, refere-se a divisão funcional do software. A camada lógica representa a estrutura e a organização do sistema a partir de uma perspectiva funcional e conceitual. Ela está

relacionada à lógica de negócios, aos serviços e às funcionalidades da aplicação. A camada lógica proporciona uma visão mais abstrata do sistema, focando nas funcionalidades e nas regras de negócios, independentemente de detalhes de implementação.

Como as palavras *tier* e *layer* podem ser traduzidas para o português como camada, muitas vezes encontramos ainda na literatura a palavra *camada* tanto para definir camada física quanto a camada lógica, porém agora você sabe a definição correta. Em resumo, a camada lógica concentra-se na estrutura e nas funcionalidades do sistema de uma perspectiva conceitual, enquanto a camada física trata da implementação concreta e da distribuição física dos componentes para execução. Ambas são essenciais para o *design* e a operação eficientes de sistemas de *software*.

1.3 Descrição das camadas em um modelo multicamadas

A criação de camadas é uma das técnicas mais comuns que os projetistas de ***software*** usam para quebrar em pedaços um sistema complexo de *software*. A parte mais difícil de uma arquitetura em camadas é decidir quais camadas são necessárias e quais as responsabilidades que cada uma deve receber. As camadas do modelo multicamadas foram criadas de modo a interagirem entre si, criando um sistema modular mais fácil de entender e manter. Estão apresentados abaixo as definições de cada camada em um modelo multicamadas de 3 camadas, conforme definidas por Fowler (2007):

- **Camada de apresentação:** a lógica de apresentação diz respeito a como tratar a interação entre o usuário e o *software*. Isso pode ser tão simples quanto uma linha de comando ou um menu baseado em texto, porém hoje é mais provável que seja uma interface gráfica em um cliente rico (uma interface com o usuário ao estilo Windows ou Swing, em vez de um navegador HTML) ou um navegador com interface baseada em HTML. As responsabilidades primárias da camada de apresentação são exibir informações para o usuário e traduzir comandos do usuário em ações sobre o domínio e a camada de dados;
- **Camada de dados:** a lógica da camada de dados diz respeito à comunicação com outros sistemas que executam tarefas no interesse da aplicação. Estes podem ser monitores de transações, outras aplicações,

sistemas de mensagens e assim por diante. Para a maioria das aplicações corporativas, a maior parte da lógica de dados é um banco de dados responsável, inicialmente, pelo armazenamento de dados persistentes;

- **Camada de domínio:** o resto é a lógica de domínio, também chamada de logica de negócio. Este é o trabalho que essa aplicação tem de fazer para o domínio com o qual você está trabalhando. Envolve cálculos baseados nas entradas e em dados armazenados, validação de quaisquer dados provenientes da camada de apresentação e a compreensão exata de qual lógica de dados executar, dependendo dos comandos recebidos da apresentação.

Os principais benefícios da arquitetura multicamada estão relacionadas com alguns dos fatores de qualidade de *software* descritos a seguir:

- **Modularidade e manutenibilidade:** Cada camada é responsável por uma parte específica do sistema, facilitando a manutenção e a evolução do *software*;
- **Reusabilidade:** as camadas podem ser projetadas para serem reutilizáveis em diferentes contextos ou em diferentes projetos;
- **Escalabilidade:** o modelo facilita a escalabilidade, pois cada camada pode ser dimensionada independentemente conforme necessário;
- **Padrões de desenvolvimento:** permite o uso eficaz de padrões de desenvolvimento, como MVC (*Model, View* e *Controller*)

1.4 Um pouco sobre padrões

Esta subseção não tem o intuito de aprofundar nos padrões de projeto, mas sim dar uma visão geral sobre alguns padrões de projeto que serão utilizados no decorrer da disciplina. Não se preocupe se não entender o relacionamento entre uma aplicação prática e os padrões de projeto nesse momento. Esses padrões serão resgatados em momentos oportunos.

Os padrões são maneiras testadas e documentadas de se alcançar objetivos determinados. Podemos ainda dizer que é como se fosse um vocabulário comum para conversar sobre projetos de *software*. A partir de 1995, o desenvolvimento de *software* passou a ter o seu primeiro catálogo de soluções para projeto de *software*: o livro GoF. A partir de então, nos padrões de projeto, soluções que não tinham nome passam a ter nome e em vez de se discutir um

sistema em termos de estrutura de dados, passa-se a falar de coisas de muito mais alto nível como fábricas, fachadas, observador, estratégia etc. A maioria dos autores desse livro eram entusiastas de Smalltalk e escreveram o livro baseado em C++ (Padrões de projetos: soluções reutilizáveis de *software* orientados a objetos).

Saiba mais

GAMMA, E. et al. **Padrões de projetos**: soluções reutilizáveis de *software* orientados a objetos. Porto Alegre: Bookman, 2007.

Na subseção sobre arquitetura multicamadas descrevemos as camadas de um modelo multicamadas, mais especificamente o modelo de 3 camadas modelo (MVC – Modelo, Visão, Controlador). Nesta seção descreveremos essas camadas focando nos padrões de projeto estruturais e comportamentais que tem relação com o modelo MVC. Os padrões arquitetural e estrutural (e comportamental) podem ser complementares, sendo usados em diferentes níveis de abstração durante o design de um sistema.

A principal diferença entre padrões arquiteturais, estruturais e comportamentais está no nível de abstração e na escala em que operam. A seguir segue a descrição de cada padrão.

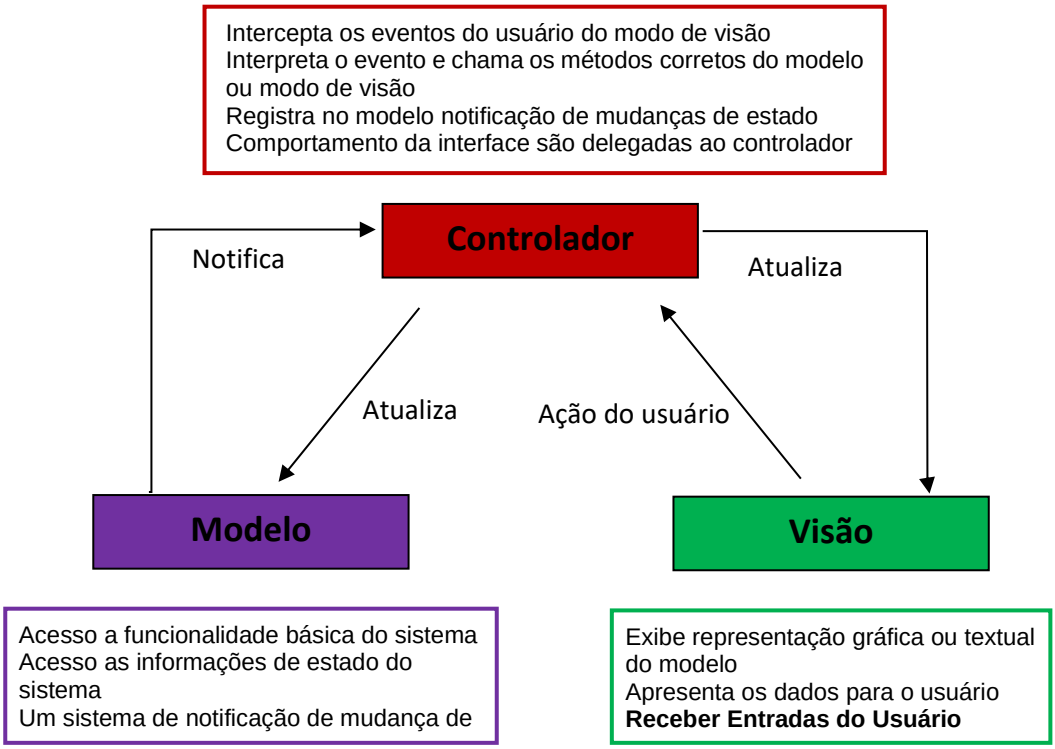
- **Padrões arquiteturais:** dizem respeito à organização global e à estrutura fundamental de um sistema. Eles lidam com a divisão de responsabilidades, a comunicação entre componentes e a visão geral da arquitetura do sistema;
- **Padrões estruturais:** dizem respeito à organização interna de classes e objetos dentro de um sistema. Eles se concentram em como as partes individuais se relacionam e se compõem para formar uma estrutura funcional. Exemplo: Composite;
- **Comportamentais:** dizem respeito a padrões de interação e responsabilidades entre objetos. Descrevem como os objetos colaboram e interagem para distribuir responsabilidades. Exemplo: Strategy e Observer

Dito isso, vamos à abordagem do modelo MVC. O modelo MVC utiliza três outros padrões de projeto: Observer, composite e Strategy. Cada um desses padrões está sucintamente descrito abaixo.

- **O padrão Composite** agrupa objetos de mesma interface em estrutura de árvore para tratar todos eles como se fossem uma única entidade. Permite construir hierarquias de *Views*, onde uma *View* composta pode conter outras *Views* simples ou compostas. Isso é útil para criar interfaces de usuário complexas e reutilizáveis;
- **O padrão Observer** define uma dependência um-para-muitos entre objetos tal que quando um objeto muda de estado, todos são notificados. A View (Visão) observa (é um observador) as mudanças no estado do *Model* (Modelo). Quando o *Model* (Modelo) muda, todas as *Views* registradas são notificadas para se atualizarem automaticamente;
- **O padrão Strategy** define uma família de algoritmos, encapsula em objetos e permite que sejam intercambiados. É uma ótima alternativa a herança. O *Controller* (Controlador) representa a lógica de controle e pode ser projetado usando o padrão Strategy. Diferentes *Controllers* (estratégias) podem ser intercambiáveis para manipular diferentes tipos de interações do usuário.

O modelo MVC desacopla a UI (interface do usuário) do sistema, dividindo-o em 3 partes: a camada de Modelo que representa o sistema, a camada de visão que exibe os dados ao usuário e a camada de Controle que processa as entradas do usuário. O padrão MVC em projetos web fundamenta-se na quebra da comunicação direta entre as camadas de visualização e as camadas de modelo. Essas duas camadas permanecem separadas, interagindo exclusivamente por meio de uma camada intermediária conhecida como controle. A Figura 3 ilustra esse padrão MVC para projetos *web*.

Figura 3 – Padrão MVC



TEMA 2 – ARQUITETURA JAVA EE

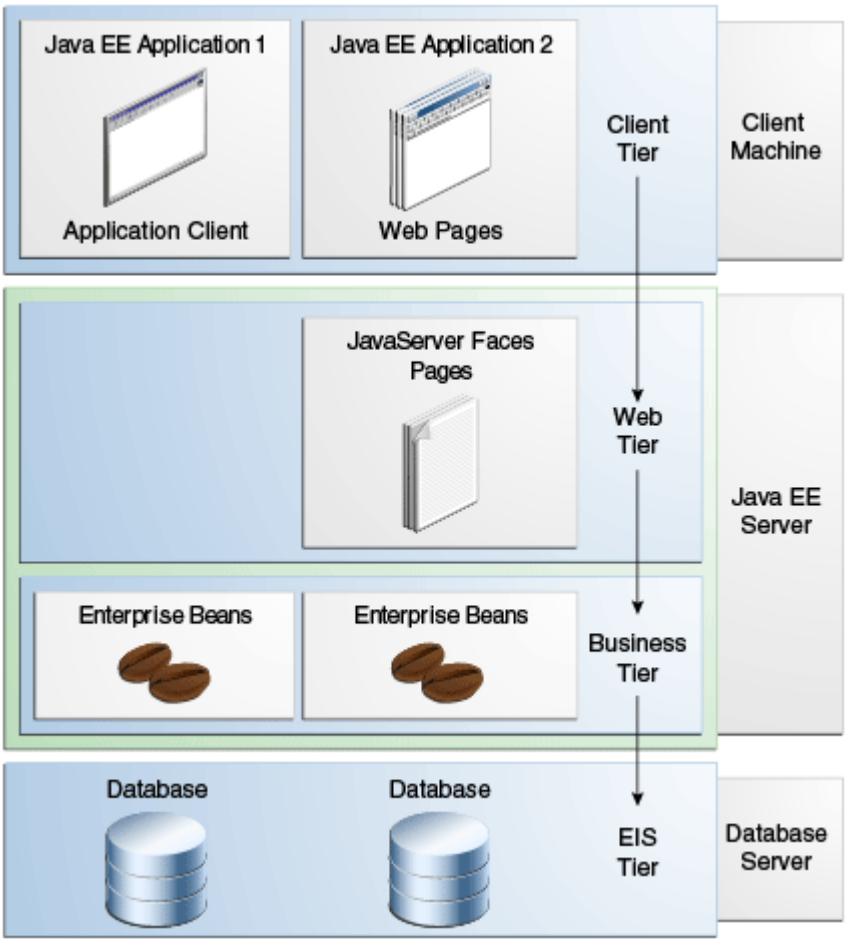
A plataforma Java EE (Java Enterprise Edition) é uma plataforma de desenvolvimento para a construção de aplicativos corporativos em Java. O Java EE usa um modelo de aplicativo distribuído de várias camadas para aplicativos corporativos. Essa divisão é feita de acordo com a função de cada componente. Sendo assim uma aplicação Java EE é composta por componentes, unidades de *software* funcional independentes que são executados dentro de uma aplicação Java EE. Nessa aula aprenderemos um pouco mais sobre o Java e sua arquitetura.

Temos na arquitetura Java EE 4 camadas lógicas: camada do cliente, camada *web*, camada de negócios, camada de sistema de informação corporativa.

Segundo a própria Oracle, embora a arquitetura da aplicação Java EE possa ser distribuída em multicamadas, com as 4 camadas citadas anteriormente, ela é considerada uma aplicação de 3 camadas porque são distribuídos em três locais: máquinas clientes, a máquina servidor Java EE e o banco de dados ou máquinas legadas no *back end*. **Uma aplicação JavaEE deve conter ao menos três camadas lógicas: apresentação, domínio e fonte de dados.**

A Figura 4 ilustra a arquitetura Java EE. Observe que, apesar de a figura utilizar a palavra em inglês *tier*, nessa arquitetura elas se referem a camadas lógicas.

Figura 4 – Arquitetura Java EE



Fonte: Java Plataform, S.d.

Como citado anteriormente, temos 4 camadas na arquitetura JAVA EE: camada cliente, camada *web*, camada de negócios e camada do sistema de informações corporativas (EIS). A seguir veremos mais sobre cada uma delas. Cabe ressaltar que as tecnologias associadas a cada camada serão discutidas posteriormente.

- **Camada Cliente:** os componentes da camada cliente são executados na máquina cliente. A camada do cliente é a interface de interação direta com o usuário final. Pode incluir diferentes tipos de clientes, como clientes web, clientes móveis ou clientes *desktop*. As tecnologias associadas são: para clientes web, tecnologias como HTML, CSS, Javascript e AJAX podem

ser utilizadas. Para clientes móveis, podem ser empregadas tecnologias específicas, como Android ou iOS;

- **Camada Web:** os componentes da camada da Web são executados no servidor Java EE. A camada *web* lida com a apresentação e interação do usuário através de navegadores *web*. Ela recebe as requisições do cliente, processa essas requisições e envia as respostas de volta para o cliente. As tecnologias associadas são: Servlets, JSP (JavaServer Pages), JSF (JavaServer Faces), e JSTL (JavaServer Pages Standard Tag Library) são algumas das tecnologias comumente usadas nesta camada. Ela também pode incluir *frameworks* de *front end*, como Angular, React ou Vue.js;
- **Camada de negócios:** os componentes da camada de negócios são executados no servidor Java EE. A camada de negócios contém a lógica de negócios da aplicação, separando-a da camada de apresentação. Ela processa as requisições recebidas da camada web, realiza as operações necessárias e interage com a camada de acesso a dados. As tecnologias associadas a esta camada são: Enterprise JavaBeans (EJB), CDI (Contexts and Dependency Injection), e outros *frameworks* de injeção de dependência podem ser usados nesta camada. Ela também pode envolver serviços e regras de negócios específicos;
- **Camada do Sistema de Informações Corporativas (EIS):** O software da camada do Sistema de Informações Corporativas (ou Empresariais) – EIS é executado no servidor EIS. A camada de EIS lida com a integração de sistemas corporativos e a persistência de dados. Ela gerencia o acesso a bancos de dados, sistemas de mensageria, e outras fontes de dados corporativas. As tecnologias associadas a esta camada são: JPA (Java Persistence API) é comumente utilizada para o mapeamento objeto-relacional nesta camada. Além disso, tecnologias como JMS (Java Message Service) e JCA (Java Connector Architecture) são empregadas para integração com sistemas empresariais legados.

As camadas trabalham juntas para criar uma arquitetura de aplicação robusta, modular e escalável, permitindo o desenvolvimento eficiente de aplicações corporativas complexas na Plataforma Java EE. Vale ressaltar que, com base no Java EE 8, a Oracle transferiu as tecnologias Java EE para a Eclipse Foundation, resultando no Eclipse Jakarta EE, uma continuação do Java

EE sob um novo nome. Falaremos sobre o Jakarta EE em um momento oportuno.

TEMA 3 – TECNOLOGIAS JAVA EE

As tecnologias Java EE (Enterprise Edition) referem-se a um conjunto de especificações, APIs (Interfaces de Programação de Aplicações - conjunto de rotinas e padrões de programação) e tecnologias desenvolvidas para facilitar o desenvolvimento e a implantação de aplicativos corporativos em Java. As tecnologias abrangem componentes e serviços que juntos fornecem um ambiente robusto para desenvolver sistemas empresariais de grande porte.

3.1 Componentes

A plataforma Java EE constitui-se de vários componentes que fornecem serviços específicos para o desenvolvimento de aplicações empresariais. Os componentes são baseados em APIs (Interfaces de Programação de Aplicações) e projetados para abordar diferentes aspectos das aplicações empresariais.

Esses componentes são construídos seguindo o modelo de programação baseado em componentes, onde as aplicações são compostas por módulos independentes e reutilizáveis. Cada componente representa uma parte específica da lógica da aplicação e é desenvolvido para ser implantado em um ambiente de execução Java EE. Os aplicativos Java EE são compostos de componentes. Um componente Java EE é uma unidade de *software* funcional independente que é montada em um aplicativo Java EE com suas classes e arquivos relacionados e que se comunica com outros componentes. A especificação Java EE define os seguintes componentes Java EE:

- **Clientes aplicativos e mini aplicativos:** um cliente de aplicativo em Java refere-se a um programa ou *software* que é desenvolvido usando a linguagem de programação Java e é executado no lado do cliente;
- **Enterprise JavaBeans (EJB):** os EJB são componentes que representam a lógica de negócios em uma aplicação Java EE. Existem três tipos principais: EJB de Sessão, EJB de Entidade e EJB de Mensagem. Componentes EJB (*enterprise beans*) são componentes de negócios executados no servidor;

- **Servlets e JavaServer Pages (JSP):** componentes usados para desenvolver a camada de apresentação em aplicações *web*. Os Servlets (“servidorezinhos”) podem ser definidos como classes Java que estendem a funcionalidade de servidores *web* para processar solicitações HTTP, permitindo a criação de aplicativos *web* dinâmicos. Já o JSP simplifica o desenvolvimento de páginas web dinâmicas usando Java. Ele permite a incorporação de código Java diretamente em páginas HTML. Os componentes de tecnologia Java Servlet, JavaServer Faces e JavaServer Pages (JSP) são componentes da *web* executados no servidor;
- **Managed Beans (CDI):** componentes gerenciados pelo CDI (Contexts and Dependency Injection), que são usados para armazenar e gerenciar estados em aplicações Java EE. No manual de Managed Beans da Plataforma Jakarta™ EE os Managed Beans são definidos da seguinte forma (Jakarta EE, 2020):

> Os **Managed Beans** são objetos gerenciados por contêiner com requisitos mínimos, também conhecidos como a sigla “POJOs” (Plain Old Java Objects). Eles suportam um pequeno conjunto de serviços básicos, como injeção de recursos, retornos de chamada de ciclo de vida e interceptadores. Outros aspectos mais avançados serão introduzidos nas especificações complementares, de modo a manter o modelo básico tão simples e tão universalmente útil possível.

O exemplo a seguir ilustra uma classe POJO.

```
public class Pessoa {
    // Atributos
    private String nome;
    private int idade;

    // Construtores
    public Pessoa() {
        // Construtor padrão
    }

    public Pessoa(String nome, int idade) {
        this.nome = nome;
        this.idade = idade;
    }

    // Métodos de Acesso (Getters e Setters)
    public String getNome() {
        return nome;
    }

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

```
    }

    public int getIdade() {
        return idade;
    }

    public void setIdade(int idade) {
        this.idade = idade;
    }
}
```

3.2 Serviços e bibliotecas

Serviços e bibliotecas são conjuntos de APIs e implementações que fornecem funcionalidades específicas para facilitar o desenvolvimento de aplicações empresariais. Diferentemente dos componentes, que são unidades individuais de funcionalidade lógica, serviços e bibliotecas são conjuntos de funcionalidades compartilhadas por diferentes partes de uma aplicação. Alguns exemplos de serviços e bibliotecas Java EE incluem:

- **Java Persistence API (JPA):** o JPA é uma API para mapeamento objeto-relacional em Java, que facilita a persistência de dados em bancos de dados. O JPA fornece um *framework* de mapeamento objeto-relacional (ORM) para mapear objetos Java para tabelas em bancos de dados relacionais;
- **Java Message Service (JMS):** uma API para comunicação assíncrona entre componentes distribuídos, permitindo o envio e recebimento de mensagens;
- **Java Naming and Directory Interface (JNDI):** um serviço de diretório que permite que aplicações Java obtenham informações de configuração e serviços a partir de um diretório;
- **JavaMail API:** o JavaMail é uma API para envio e recebimento de e-mails em aplicações Java. Com a JavaMail API, as funcionalidades de *email* são facilmente incorporadas nas aplicações Java;
- **JavaServer Faces (JSF):** um *framework* para o desenvolvimento de interfaces *web* baseadas em componentes. O JSF simplifica a construção de aplicações *web* baseadas em Java, fornecendo um conjunto de componentes visuais reutilizáveis e um modelo de programação baseado em eventos.

TEMA 4 – ECLIPSE JAKARTA EE

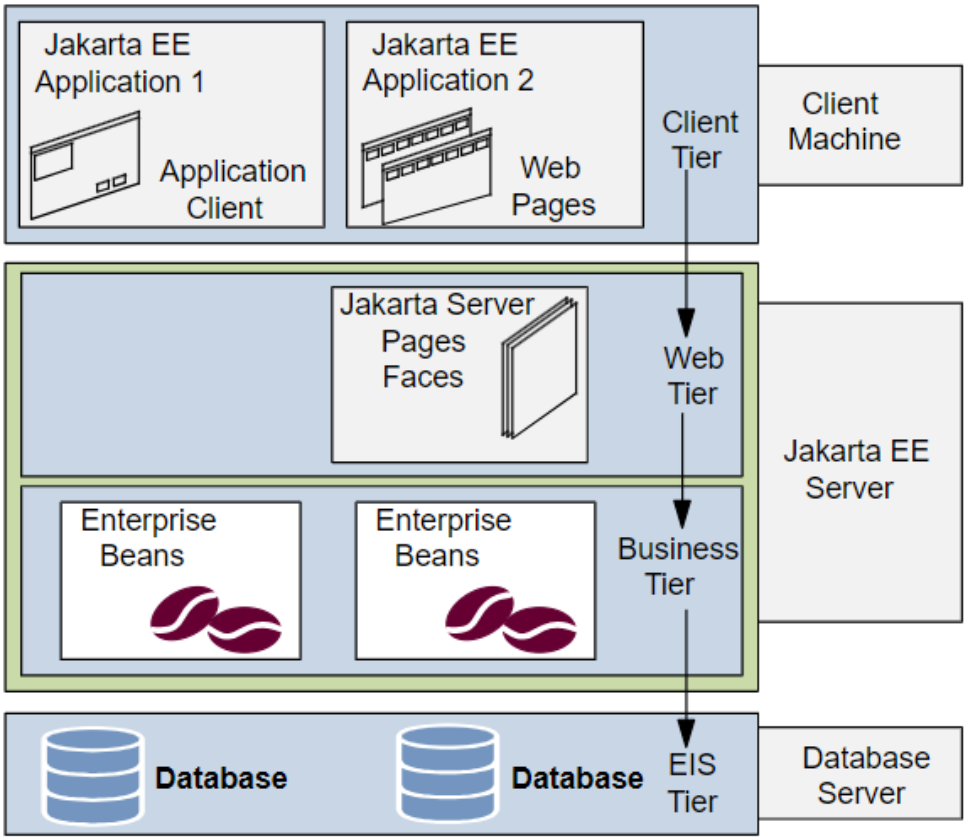
O Eclipse Jakarta EE é uma plataforma de desenvolvimento para construção de aplicações empresariais em Java. O Eclipse Jakarta EE fornece um conjunto de especificações, APIs e ferramentas para facilitar o desenvolvimento de aplicações corporativas robustas voltadas para *web*

A definição acima deve parecer familiar para você, pois o Eclipse Jakarta EE, continua o trabalho anteriormente realizado sob o nome de Java EE (Enterprise Edition). Java EE (Enterprise Edition) e Jakarta EE são essencialmente a mesma plataforma para o desenvolvimento de aplicações empresariais em Java, mas com uma diferença significativa no nome devido a mudanças na governança e no desenvolvimento da tecnologia. Portanto se ouvir qualquer um dos nomes: Java EE e Jakarta EE, se referem praticamente a mesma plataforma, com uma nomenclatura diferente.

Dessa forma as tecnologias Java EE como citadas anteriormente foram mantidas. Os manuais foram atualizados, mas os conceitos permanecem os mesmos.

A Figura 5 que representa uma aplicação multicamada foi retirada do manual atualizado do Jakarta EE, observe que, apesar de a imagem estar atualizada, o conceito permanece o mesmo.

Figura 5 – Aplicação multicamada



Fonte: The Jakarta, S.d.

4.1. Breve história do JAVA EE até sua evolução para Jakarta EE

O Java EE (Enterprise Edition) surgiu como uma extensão do Java Standard Edition (Java SE) para atender às necessidades específicas de desenvolvimento de aplicações corporativas e distribuídas. O Java EE teve suas raízes na plataforma J2EE, que foi lançada pela Sun Microsystems (agora parte da Oracle Corporation) como uma extensão do Java 2 Platform, Standard Edition (J2SE). A versão inicial, J2EE 1.2, foi lançada em dezembro de 1999. Ao longo do tempo, o J2EE passou por várias versões, cada uma introduzindo melhorias, novas tecnologias e atualizações. Com o tempo, a Sun Microsystems decidiu simplificar a nomenclatura e passou a se referir à plataforma como Java EE a partir da versão 5.0, que foi lançada em maio de 2006. Essa mudança refletia a evolução da plataforma e sua abrangência além do ambiente empresarial.

Em 2017, a Oracle anunciou a intenção de transferir as tecnologias do Java EE para uma organização de código aberto, visando uma governança mais comunitária e promovendo a neutralidade de fornecedores

Como parte desse processo, as tecnologias do Java EE foram transferidas para a Eclipse Foundation, e o nome foi alterado para Eclipse Jakarta EE. O termo *Jakarta* foi escolhido como uma homenagem à comunidade Java na Indonésia, refletindo a natureza global da comunidade de desenvolvimento Java.

Jakarta EE mantém as metas e objetivos do Java EE, continuando a oferecer uma plataforma robusta e escalável para o desenvolvimento de aplicações empresariais em Java. A mudança para a Eclipse Foundation trouxe uma governança mais comunitária, com contribuições de várias organizações e desenvolvedores. Jakarta EE continua a evoluir, introduzindo atualizações regulares e colaborando com outros projetos relacionados, como o Eclipse MicroProfile. O Eclipse MicroProfile é uma iniciativa de código aberto que visa facilitar o desenvolvimento de micro serviços em Java.

Portanto, a principal diferença entre Java EE e Jakarta EE é o nome e a mudança na governança, com Jakarta EE sendo o sucessor direto do Java EE, mantendo a compatibilidade e adicionando melhorias contínuas. Para desenvolvedores e empresas que estavam familiarizados com o Java EE, a transição para Jakarta EE geralmente envolve atualizações de nome e algumas modificações nos processos de desenvolvimento, mas a abordagem geral para a criação de aplicações empresariais em Java permanece consistente.

4.2 Considerações Práticas para o Programador Java

A migração Gradual do Java EE para Jakarta EE está sendo gradual. Portanto se você estiver trabalhando em um projeto existente baseado em Java EE, a transição para Jakarta EE pode ser uma migração gradual, pois as versões iniciais do Jakarta EE mantêm compatibilidade com Java EE.

Um grande benefício nessa mudança de governança é o fato de que o Jakarta EE oferece oportunidades para os programadores se envolverem mais ativamente no desenvolvimento e influência da plataforma, devido à sua governança aberta e natureza de código aberto.

Cabe ressaltar que, no final de 2022, as mudanças tornaram-se bem mais visíveis aos programadores que se depararam com a mudança física de alguns pacotes do javax para o jakarta e algumas alterações na nomenclatura das anotações. Sendo assim para obter as últimas atualizações e recursos,

especialmente à medida que Jakarta EE evolui, os programadores java devem considerar a migração para as versões mais recentes da plataforma.

TEMA 5 – REVISITANDO O JAVA BÁSICO

Não importa o quão avançados sejamos, é sempre benéfico revisitar os fundamentos. Essa sessão tem o objetivo de revisitar algumas classes do Java que já conhecemos, mas que agora veremos com um outro olhar, pois já temos aqui uma bagagem suficiente de conhecimento para compreendermos um pouco mais velhos conceitos.

A linguagem Java possui um conjunto extenso de classes e bibliotecas que são essenciais para o desenvolvimento de aplicações. Algumas classes da biblioteca Java Standard Edition (Java SE) são consideradas fundamentais, pois fornecem funcionalidades básicas e são amplamente utilizadas. Veremos algumas dessas classes a seguir. As definições e conceitos foram retirados do próprio manual do Java, disponibilizado pela Oracle.

5.1 Classe Object

Mãe de todas as outras classes em Java. A classe Object é a classe base para todos os objetos em Java. Ela fornece métodos fundamentais, como equals(), hashCode(), e toString(), que são frequentemente sobrescritos por outras classes.

O método toString() retorna uma *string* representando o objeto. O método toString() retorna uma *string* que normalmente contém informações sobre o estado do objeto. O propósito principal do método toString() é fornecer uma representação legível do objeto, que pode ser útil para depuração, registro ou simplesmente para exibição.

Por padrão, a implementação padrão do método toString() na classe Object retorna uma *string* que consiste no nome da classe seguido pelo *"@"* e a representação hexadecimal do *hashcode* do objeto.

Considere a classe Pessoa que está dentro do pacote modelo. A classe Pessoa tem 2 atributos: nome e idade. Os métodos *getters* e *setters* já se encontram implementados assim como um construtor alternativo.

Classe Pessoa

```
public class Pessoa {

	private String nome;
	private int idade;

	public Pessoa(){
		this.nome = “”;
		this.idade = 1;
	}

//Construtor alternativo
public Pessoa(String nome, int idade) {

		this.nome = nome;
		this.idade = idade;
	}
	}
//métodos getters e setters
	public String getNome() {
		return nome;
	}
	public void setNome(String nome) {
		this.nome = nome;
	}
	public int getIdade() {
		return idade;
	}
	public void setIdade(int idade) {
		this.idade = idade;
}
}


//Construtor alternativo
public Pessoa(String nome, int idade) {

		this.nome = nome;
		this.idade = idade;
	}
	}
//métodos getters e setters
	public String getNome() {
		return nome;
	}
	public void setNome(String nome) {
		this.nome = nome;
	}
	public int getIdade() {
		return idade;
	}
	public void setIdade(int idade) {
```

```
            this.idade = idade;
}
}
```

Na nossa classe de teste, imprimiremos nosso objeto *pessoa a partir da linha de comando:*

```
System.out.println(pessoa);
```

Classe de teste

```
package com.uninter.modelo;

public class TestePessoa {

      public static void main(String[] args) {

            Pessoa pessoa = new Pessoa("Maria", 20);
            System.out.println(pessoa);


      }

}
```

Após a execução do programa a saída será algo parecido com:

```
com.uninter.modelo.Pessoa@2c7b84de
```

Isso não nos parece muito útil não é mesmo? Mas é isso o que acontece se o método ToString não for sobrescrito. O referido método poderia ser reescrito da seguinte forma:

```
    // Exemplo de toString para facilitar a visualização dos
dados
@Override
      public String toString() {
            return "Pessoa [nome=" + nome + ", idade=" + idade +
"]";
      }
```

Executando novamente a classe de teste, a saída será algo parecido com:

```
Pessoa [nome=Maria, idade=20]
```

Método equals compara se dois objetos são iguais, e retorna true se os objetos forem o mesmo, e false se não forem o mesmo. Ou seja, verifica se dois objetos apontam para o mesmo local na memória.

Ao sobrescrever o método *equals*, é uma boa prática também sobrescrever o método hashCode para garantir o correto funcionamento em estruturas de dados baseadas em *hash*, como HashMap e HashSet.

Ao se testar se dois objetos da classe Pessoa são iguais, se no método equals não for sobrescrito, a saída para essa comparação será *false*.

```
public class TestePessoa{
    public static void main(String[] args) throws Exception {
        Pessoa pessoa1 = new Pessoa("Maria", 20);
        Pessoa pessoa2 = new Pessoa("Maria", 20);
        System.out.println(pessoa1.equals(pessoa2));
    }
}
```

Método equals sobreescrito:

```
@Override
     public boolean equals(Object obj) {
          if (this == obj)
               return true;
          if (obj == null)
               return false;
          if (getClass() != obj.getClass())
               return false;
          Pessoa other = (Pessoa) obj;
          return idade == other.idade && Objects.equals(nome,
other.nome);
     }
```

Método hashCode() retorna um código *hash* para o objeto. Muito usado em Collections, retorna um valor inteiro (um código *hash*) representando o estado do objeto. O propósito principal do método hashCode() é fornecer um valor numérico que identifica exclusivamente o estado do objeto. Este valor é frequentemente usado em estruturas de dados baseadas em *hash*, como HashMap ou HashSet, para otimizar a pesquisa e recuperação de objetos.

A implementação padrão do método hashCode() na classe Object gera um código *hash* baseado no endereço de memória do objeto.

O código a seguir imprime o *hashcode* dos objetos pessoa1 e pessoa2.

```
public class TestePessoa{
    public static void main(String[] args) throws Exception {
        Pessoa pessoa1 = new Pessoa("Maria", 20);
        Pessoa pessoa2 = new Pessoa("Maria", 20);
        System.out.println(pessoa1.hashCode() + " | "
+ pessoa2.hashCode());
```

```
    }
}
```

Se não sobrescrevermos o método hashCode, a saída será algo parecido com a linha abaixo. Observe que são *hashcodes* diferentes.

```
746292446 | 1072591677
```

O código abaixo representa a sobrescrita do método hashCode:

```
@Override
    public int hashCode() {
        return Objects.hash(idade, nome);
    }
```

Ao se sobrescrever o método *hashCode* a saída será algo parecido com a linha abaixo. Observe agora que os valores são iguais.

```
74115331 | 74115331
```

5.2 Classe System

A classe System contém vários campos e métodos de classe úteis. Ela fornece métodos e propriedades que lidam com propriedades do sistema, interação com o ambiente de execução e manipulação de fluxos de entrada e saída. É uma classe que não pode ser instanciada, porque todos seus métodos e variáveis são estáticos.

Uma das coisas mais úteis prestados por esta classe: entrada e saída de dados.

- Fluxos de entrada: utilizados para ler dados;
- Fluxos de saída: usados para impressão na tela.

Como exemplo clássico, temos na linha de código a seguir a impressão na tela da frase *Olá Mundo!*

```
        System.out.print("Olá Mundo!");
```

5.3 Classe Scanner

A classe Scanner é usada para ler dados de diferentes fontes, como entrada do teclado, arquivos ou *strings*. Ela fornece métodos para analisar e

converter *tokens* (sequências de caracteres) em tipos primitivos, como inteiros, ponto flutuantes, ou *strings*.

A classe oferece um scanner de texto simples que pode analisar tipos primitivos e *strings* usando expressões regulares. Por exemplo, o código abaixo permite que um usuário leia um número de System.in:

```
Scanner sc = new Scanner(System.in);
     int i = sc.nextInt();
```

5.4 Classe *String*

A classe *String* representa cadeias de caracteres. Todos os literais de *string* em programas Java, como “abc”, são implementados como instâncias desta classe.

As *strings* são constantes; seus valores não podem ser alterados após serem criados. *Buffers* de *string* suportam *strings* mutáveis. Como os objetos *String* são imutáveis, eles podem ser compartilhados. Por exemplo, a *string* a seguir:

```
String str = "abc";
```

É equivalente a :

```
      char data[] = {'a', 'b', 'c'};
    String str = new String(data);
```

5.5 Classe Math

Nesta classe existe uma série de métodos estáticos que fazem operações com números. A classe *Math* contém métodos para realizar operações numéricas básicas, como funções exponenciais elementares, logaritmos, raízes quadradas e trigonométricas.

Exemplo:

```
double d = 4;
double i = Math.sqrt(d);
```

5.6 Classes Wrapper

As classes *Wrapper* permitem a manipulação de valores dos tipos literais da linguagem como se fossem objetos. Frequentemente é necessário

representar um valor de tipo primitivo como se fosse um objeto. As classes *Wrapper Boolean, Character, Integer, Long, Float* e *Double* atendem a esse propósito. Um objeto do tipo *double*, por exemplo, contém um campo cujo tipo é *double*, representando aquele valor de tal forma que uma referência a ele pode ser armazenada em uma variável do tipo referência. Essas classes também fornecem vários métodos para conversão entre valores primitivos, além de oferecer suporte a métodos padrão como *equals* e *hashCode*. A classe *Void* é uma classe não instanciável que contém uma referência a um objeto *Class* que representa o tipo *void*.

Tabela 1 – Tabela de tipo de dados primitivos e sua classe empacotadora equivalente

Tipo de Dado Primitivo	Classe Wrapper correspondente
boolean	Boolean
char	Character
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double

Como exemplo, temos a Classe Integer que permite trabalhar com objetos que representam o tipo primitivo int. O método parseInt transforma diretamente uma cadeia de caracteres num tipo int.

Exemplo:

```
String itxt = "123";
int i = Integer.parseInt( itxt );
System.out.println(i + "\n ");
```

5.7 Collection Framework

A plataforma Java inclui uma estrutura de coleções. Uma coleção é um objeto que representa um grupo de objetos (como a clássica classe Vector). Uma estrutura de coleções é uma arquitetura unificada para representar e manipular coleções, permitindo que as coleções sejam manipuladas independentemente dos detalhes de implementação. A essa série de classes e interfaces para representar e manipular coleções de objetos, como listas, conjuntos e mapas é que se dá o nome de Collections Framework.

As implementações de uso geral estão resumidas na tabela a seguir:

Tabela 2 – Tabela contendo as interfaces e as classes de implementação

Interface	Hash Table	Resizible Array	Balanced Tree	Linked List	Hash Table + Linked List
Set	HashSet		TreeSet		LinkedHashSet
List		ArrayList		LinkedList	
Deque		ArrayDeque		LinkedList	
Map	HashMap		TreeMap		LinkedHashMap

A seguir, uma descrição sucinta de cada interface:

- **Collection (**java.util.Collection**):** a interface base para representar grupos de objetos. Inclui métodos comuns a todas as coleções, como add, remove, size, entre outros;
- **List** (java.util.List): uma coleção ordenada onde os elementos podem ser acessados por índice. Implementações conhecidas incluem ArrayList e LinkedList;
- **Set** (java.util.Set): uma coleção que não permite elementos duplicados. Implementações incluem HashSet e TreeSet;
- **Queue** (java.util.Queue): uma coleção projetada para manipular elementos em uma ordem específica. Geralmente segue a lógica FIFO (First-In-First-Out). Implementações incluem LinkedList (também pode ser usada como uma fila) e PriorityQueue;
- **Deque** (java.util.Deque): representa uma estrutura de dados de deque, onde os elementos podem ser inseridos e removidos tanto no final quanto no início. Pode ser utilizado como uma fila (FIFO) ou uma pilha (LIFO - Last-In-First-Out). Deque inclui todas as operações de Queue e adiciona operações adicionais;
- **Map** (java.util.Map): associação de chaves a valores únicos. Implementações conhecidas incluem HashMap, TreeMap e LinkedHashMap.

FINALIZANDO

Esta jornada nos proporcionou uma visão abrangente do ecossistema Java, desde suas bases até as tecnologias avançadas e ferramentas modernas. Vimos um pouco sobre a arquitetura de *software* que é a espinha dorsal de qualquer aplicativo robusto. Exploramos as camadas que constituem a

arquitetura Java EE. Exploramos componentes, serviços e bibliotecas que fornecem funcionalidades essenciais para a construção de aplicativos robustos e conformes com os padrões Java EE. Vimos sobre o Jakarta EE e como essa ferramenta evoluiu da antiga Java EE. Por fim, destacamos a importância de revisitar os conceitos básicos do Java. Mesmo ao explorar tópicos avançados, nunca devemos subestimar a relevância de compreender velhos conceitos. À medida que avançarmos nessa disciplina, a utilização dos componentes, serviços e biblioteca ficará mais clara ainda.

REFERÊNCIAS

DEITEL, P. J.; DEITEL, H. M. **Java**: como programar. 10. ed. São Paulo: Pearson, 2017.

FOWLER, M. **Padrões de arquitetura de aplicações corporativas**. Porto Alegre: Bookman, 2007.

GAMMA, E. et al. **Padrões de projetos**: soluções reutilizáveis de *software* orientados a objetos. Porto Alegre: Bookman, 2007.

JAKARTA. The Jakarta EE Tutorial. Disponível em: <https://eclipse-ee4j.github.io/jakartaee-tutorial/#distributed-multitiered-applications>. Acesso em: 18 mar. 2024.

JAKARTA EE. Jakarta Managed Beans, **Jakarta**, 15 out. 2020. Disponível em: <https://jakarta.ee/specifications/managedbeans/2.0/jakarta-managed-beans-spec-2.0.pdf>. Acesso em: 18 mar. 2024.

JAVA PLATFORM. The JavaEE Tutorial – Distributed Multitiered Applications. **Java Platform**, S.d. Disponível em: <https://javaee.github.io/tutorial/overview004.html>. Acesso em: 18 mar. 2024.

PRESSMAN, R. S.; Maxim, B. R. **Engenharia de *software***. 9. ed. São Paulo: Grupo A, 2021.