FACULDADE FACI | WYDEN

Curso Superior de Tecnologia em Análise de Desenvolvimento De Sistemas

Wagner Roberto Silva Carneiro Filho – RA: 202051261382

Atividade Avaliativa - Desenvolvimento de Software INTEGRADA

Trabalho acadêmico apresentado como requisito para aprovação da disciplina Temas Tecnológicos em Desenvolvimento e Segurança do curso de Análise e Desenvolvimento de Sistemas

Belém – PA

2022

Atividade Avaliativa - Desenvolvimento de Software INTEGRADA

Wagner Roberto Silva Carneiro Filho – RA: 202051261382

Trabalho acadêmico apresentado como requisito para aprovação da disciplina Temas Tecnológicos em Desenvolvimento e Segurança do curso de Análise e Desenvolvimento de Sistemas

Belém – PA

2022

SUMÁRIO

Arquitetura Web: Fundamentos; Modelos Arquiteturais

Fundamentos:

Introdução

Os três padrões mais comuns são:

Cliente Web Thin - Utilizado principalmente para aplicativos com base na Internet, em que há pouco controle da configuração do cliente. O cliente requer apenas um navegador padrão da Web (com capacidade para formulários). Toda a lógica do negócio é executada no servidor.

Cliente Thick Web - Uma parte significativa, em termos de arquitetura, da lógica de negócios é executada na máquina cliente. Em geral, o cliente utiliza HTML Dinâmico, Applets Java ou controles ActiveX para executar a lógica do negócio. A comunicação com o servidor ainda é feita via HTTP.

Entrega pela Web - Além de utilizar o protocolo HTTP para a comunicação de cliente e servidor, é possível utilizar outros protocolos, como IIOP e DCOM, para suportar um sistema de objetos distribuídos. O navegador da Web age principalmente como um dispositivo de entrega e de contêiner de um sistema de objetos distribuídos.

Essa lista não pode ser considerada completa, especialmente nos setores da indústria em que as revoluções tecnológicas parecem ocorrer a cada ano. Ela representa, em alto nível, os padrões mais comuns de arquitetura de aplicativos da Web. Assim como ocorre com qualquer padrão, é aceitável a aplicação de vários deles a uma única arquitetura.

Cliente Web Thin

O padrão arquitetural Cliente Web Thin é muito útil para aplicativos baseados na Internet, para os quais pode-se garantir apenas a configuração mínima no cliente. Toda a lógica do negócio é executada no servidor durante o processamento das solicitações de página do navegador cliente.

Aplicabilidade

Esse padrão é mais adequado a aplicativos da Web baseados na Internet ou a ambientes em que o cliente tenha uma capacidade mínima de computação ou não tenha nenhum controle sobre a configuração.

Usos Conhecidos

A maioria dos aplicativos de comércio eletrônico para a Internet usa esse padrão, pois não há muito sentido, comercialmente falando, em eliminar qualquer nicho de consumidores simplesmente porque não têm recursos suficientes no cliente. Um aplicativo típico de comércio eletrônico tenta abranger o maior conjunto possível de consumidores, afinal, o dinheiro de um usuário do Commodore Amiga é tão bom quanto o de um usuário do Windows NT.

Estrutura

Os principais componentes do padrão de arquitetura Cliente Web Thin estão no servidor. Em muitos aspectos, essa arquitetura representa a arquitetura mínima do aplicativo da Web. Estes são os principais componentes:

Navegador de cliente - Qualquer navegador HTML padrão com capacidade para formulários. O navegador age como um dispositivo genérico da interface de usuário. Quando usado em uma arquitetura Cliente Web Thin, o único outro serviço que ele fornece é a capacidade de aceitar e retornar cookies. O usuário do aplicativo utiliza o navegador para solicitar páginas da Web: HTML ou servidor. A página retornada contém uma interface de usuário totalmente formatada, com controles de entrada e texto, que é convertida pelo navegador no modo de exibição do cliente. Todas as interações do usuário com o sistema são feitas por meio do navegador.

Servidor Web - O principal ponto de acesso para todos os navegadores de cliente. Os navegadores de cliente, na arquitetura Cliente Web Thin, acessam o sistema somente por meio do servidor Web, que aceita pedidos de páginas da Web - páginas em HTML estático ou páginas do servidor. Dependendo da solicitação, o servidor da Web pode iniciar algum processamento no próprio servidor. Se o pedido de página for para um módulo CGI, ISAPI ou NSAPI da página com scripts do servidor, o servidor Web delegará o processamento para o interpretador de script ou módulo executável apropriado. De qualquer forma, o resultado será uma página em formato HTML, apropriada para ser processada por um navegador HTML.

Conexão HTTP - O protocolo mais comum em uso entre os navegadores de cliente e os servidores da Web. Esse elemento de arquitetura representa um tipo de comunicação sem conexão entre cliente e servidor. Sempre que o cliente ou o servidor enviar informações um para o outro, uma nova conexão separada é estabelecida entre os dois. Uma variação da conexão HTTP é a conexão segura HTTP via Camada de Soquetes de Segurança (SSL). Esse

tipo de conexão criptografa as informações que estão sendo transmitidas entre cliente e servidor, usando a tecnologia de criptografia de chave pública/privada.

Página HTML - Uma página da Web com interface com usuário e informações de conteúdo que não passam por nenhum processamento no lado do servidor. Essas páginas geralmente contêm texto explicativo como, por exemplo, direções e informações de ajuda, ou formulários HTML de entrada. Quando o servidor da Web recebe uma solicitação de uma página HTML, ele simplesmente recupera o arquivo e o envia para o cliente solicitante, sem filtrá-lo.

Página de servidor - Páginas da Web que passam por alguma forma de processamento no lado do servidor. Em geral, essas páginas são implementadas no servidor como páginas com scripts (Páginas Active Server, Páginas Java Server, páginas Cold Fusion), que são processadas por meio de um filtro no servidor do aplicativo ou de módulos executáveis (ISAPI ou NSAPI). Elas têm possibilidade de acesso a todos os recursos do servidor, incluindo componentes da lógica do negócio, bancos de dados, sistemas legados e sistemas de contabilidade comercial.

Servidor de aplicativos - O mecanismo principal para executar a lógica de negócios no lado do servidor. O servidor de aplicativo é responsável pela execução do código nas páginas de servidor. Ele pode estar localizado no mesmo equipamento do servidor da Web e também pode ser executado no mesmo espaço de processamento. O servidor de aplicativo é, em termos de lógica, um elemento de arquitetura separado, pois está envolvido somente na execução da lógica do negócio e pode usar uma tecnologia completamente diferente da usada no servidor da Web.

Arquiteturas Monolíticas

Esta é a arquitetura mais comum e mais utilizada para desenvolvimento de aplicações web devido a sua simplicidade e também por ser a mais antiga utilizada. De forma geral, uma aplicação monolítica tem as características:

- Pode suportar diferentes tipos de cliente como desktop/mobile
- Pode exportar APIs para comunicação com terceiros
- Pode integrar outras aplicações utilizando serviços REST/SOAP ou filas de mensagens

- Pode tratar requisições HTTP, executar regras de negócio, acessar banco de dados e trocar informações com outros sistemas
- Podem escalar verticalmente aumentando o poder das máquinas em que a aplicação roda ou horizontalmente com a adição instâncias atrás de um Load Balancer

Arquitetura de Micro Serviços

Agora falando da febre do momento, que está ganhando cada vez mais popularidade entre arquitetos e desenvolvedores de software. Este tipo de arquitetura pode oferecer quase todas as funcionalidades de uma arquitetura monolítica. Adicionalmente, oferece muitas outras funcionalidades e maior flexibilidade e consequentemente é geralmente considerada uma escolha superior para aplicações complexas. Diferentemente da arquitetura monolítica é difícil generalizar esta opção pois ela pode variar muito conforme o caso de uso e aplicações. De qualquer forma, também existem algumas vantagens e desvantagens que devem ser observadas na hora da decisão.

De forma geral, esta arquitetura compartilha de alguns benefícios:

- Os componentes tem baixo acoplamento. Podem ser desenvolvidos, testados, “deployados” e escalados de forma independente
- Os componentes podem ser desenvolvidos cada um com um stack de tecnologias própria
- Eles geralmente implementam funcionalidades avançadas e padrões como o service discovery, circuit breaking, load balancing etc.
- São mais leves e com funcionalidades especificas. Um serviço de autenticação, por exemplo, só sabe fazer autenticação.

- Geralmente tem um longo e extenso setup de monitoramento e troubleshooting

HTML: Exemplo de construção de cadastro de formulário

Código HTML do nosso formulário

Inicialmente, vamos desenvolver a estrutura do nosso HTML. Lembre-se de salvar esse arquivo com a extensão .html ou .php. No caso do nosso exemplos, utilizamos o nome index.php como nome do arquivo. Vejamos então o nosso código inicial:

```
<!DOCTYPE html>
<html lang="pt-br">
<head>
    <title>Formulário HTML</title>
    <meta charset="utf-8"/>
    <link rel="stylesheet" type="text/css"
href="estilo.css">
</head>
<body>
    <form class="formulario" method="post">
        <p> Envie uma mensagem preenchendo o formulário
abaixo</p>

        <div class="field">
            <label for="nome">Seu Nome:</label>
            <input type="text" id="nome" name="nome"
placeholder="Digite seu nome*" required>
        </div>

        <div class="field">
            <label for="telefone">Seu Telefone:</label>
            <input type="text" id="telefone"
name="telefone" placeholder="Digite seu Telefone">
        </div>
```

```
        <div class="field">
            <label for="email">Seu E-Mail:</label>
            <input type="email" id="email" name="email"
placeholder="Digite seu E-Mail*" required>
        </div>
        <div class="field radiobox">
            <span>Deseja receber nossas novidades?</span>
            <input type="radio" name="novidades" id="sim"
value="sim" checked><label for="sim">Sim</label>
            <input type="radio" name="novidades" id="nao"
value="nao"><label for="nao">Não</label>
        </div>
        <div class="field">
            <label for="mensagem">Sua mensagem:</label>
            <textarea name="mensagem" id="mensagem"
placeholder="Mensagem*" required></textarea>
        </div>

        <input type="submit" name="acao" value="Enviar">
    </form>
</body>
</html>
```

Descrever as diferenças entre Servlets e Scripts CGI

Definição de CGI

O **CGI (Common Gateway Interface)** é uma interface que lida com programas externos (scripts CGI) em um servidor da Web para permitir a execução de páginas da Web interativas. O CGI foi desenvolvido pelo **NCSA (Centro Nacional de Aplicações de Supercomputação)** em 1993. Ele reside no lado do servidor e permite que os navegadores da Web interajam com os programas no servidor da web. Por exemplo, se uma página da Web consultar um banco de dados ou um usuário estiver enviando as informações do formulário para o servidor, os scripts CGI serão invocados. O servidor passa essas informações para um aplicativo de duas maneiras, **GET** ou **POST**, e o aplicativo responde ao servidor de volta ao navegador. Dessa maneira, os navegadores obtêm alguns resultados para o usuário.

O CGI é uma estipulação, para descrever um método para executar scripts CGI e fornecer os resultados de volta ao servidor para esses programas específicos. A função de um CGI é escanear as informações recebidas do navegador e produzir uma resposta apropriada, após a conclusão da tarefa o script CGI é finalizado. Na Common Gateway Interface, o comum significa que

pode funcionar bem em qualquer sistema operacional ou linguagem de programação.

Anteriormente, os shell scripts do UNIX e o PERL eram usados para escrever os programas CGI, que é a razão pela qual eles são chamados de **“scripts”** CGI. Mas, agora, qualquer uma das linguagens como C, C ++, Perl, Visual Basic ou Python pode ser usada. Embora o CGI possa liderar os possíveis problemas de segurança, como o servidor da Web também pode tratar arquivos executáveis como programas CGI em alguns diretórios específicos. O CGI manipula cada solicitação do cliente por um processo separado, o que aumenta a carga do servidor, tornando-a mais lenta.

Definição de Servlet

Um **Servlet** é um componente da Web baseado em Java que atua como um programa intermediário que facilita a interação entre o navegador da Web ou o cliente HTTP e o servidor HTTP. Semelhante ao CGI, os servlets também podem ser usados para reunir as informações pelos formulários de páginas da Web, mostrar os registros do banco de dados e gerar páginas da Web dinâmicas com a ajuda de um contêiner. Servlet é uma classe Java que não depende da plataforma e ainda é compilada para o bytecode que é independente de plataforma. O bytecode neutro da plataforma pode ser armazenado e executado dinamicamente pelo servidor da Web Java. O servlet usa o mecanismo de soquete e **RMI** para estabelecer a conexão entre applets, bancos de dados ou outros programas de banco de dados.

O contêiner de servlet é uma parte do servidor da Web que suporta os **protocolos HTTP e HTTPS** . Ele emprega o método de solicitação / resposta usando o protocolo HTTP e HTTPS para permitir a interação com os clientes da web. Ao contrário do CGI, o servlet é executado dentro do espaço de endereço do servidor da web, onde cada cliente não é necessariamente tratado separadamente. No servlet, uma coleção de restrições é aplicada no servidor para proteger os recursos no servidor.

Principais diferenças entre o CGI e o Servlet

1. Os scripts CGI são gravados no sistema operacional nativo e armazenados em determinado diretório. Por outro lado, os programas de servlet geralmente são escritos em Java, o qual é compilado no bytecode de Java e executado na JVM.
2. O CGI é específico da plataforma, o que dificulta a alternância entre os sistemas operacionais. Por outro lado, os Servlets podem ser executados em qualquer sistema operacional que tenha instalado o JVM, portanto, ele é independente de plataforma.
3. Em CGI, cada solicitação de cliente que chega pode gerar um processo separado durante o servlet, os processos não são criados desnecessariamente e compartilham o espaço de memória da JVM.

4. Scripts CGI são programas executáveis escritos no SO nativo do servidor. Por outro lado, os servlets são compilados para o bytecode Java que é executado na JVM.
5. O servlet é mais seguro que o CGI, pois usa o Java.
6. A velocidade, desempenho e eficiência do servlet é melhor que o CGI.
7. Scripts CGI podem ser processados diretamente. Pelo contrário, o servlet primeiro traduz e compila o programa e o processa.
8. Quando se trata de portabilidade, o servlet é portátil, enquanto o CGI não é.

Conclusão

O CGI e o Servlet funcionam da mesma maneira, mas o uso do servlet é vantajoso em relação ao CGI, pois o servlet é rápido, seguro, independente de plataforma, facilmente desenvolvido, acessível por meio de várias APIs e suportado por vários servidores da web.

Introdução ao JSP (Java Server Pages)

Introdução ao JSP

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.

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

Um pouco sobre Polimorfismo Java

O paradigma da Orientação a Objetos traz um ganho significativo na **qualidade da produção de software**, porém grandes benefícios são alcançados quando as técnicas de programação OO são colocadas em prática com o uso de uma tecnologia que nos permita usar todas as características da OO; além de agregar à programação o uso de **boas práticas de programação** e padrões de projeto, **design patterns**. Um objeto é uma entidade do mundo real que tem uma identidade. Objetos podem representar entidades concretas, um arquivo no meu computador, uma bicicleta ou entidades conceituais, uma estratégia de jogo, uma política de escalonamento em um sistema operacional. Cada objeto ter sua identidade significa que, dois objetos são distintos mesmo que eles apresentem exatamente as mesmas características.

Objetos são instâncias de classes, que determinam qual informação um objeto contém e como ele pode manipulá-la. Um programa desenvolvido com uma linguagem de programação orientada a objetos manipula estruturas de dados através dos objetos da mesma forma que um programa em linguagem tradicional utiliza variáveis.

Em orientação a objeto, uma classe é uma estrutura que abstrai um conjunto de objetos com características similares. Uma classe define o comportamento de seus objetos através de métodos e os estados possíveis destes objetos através de atributos. Em outros termos, uma classe descreve os serviços providos por seus objetos e quais informações eles podem armazenar. Classes não são diretamente suportadas em todas as linguagens, e são necessárias para que uma linguagem seja orientada a objetos. A programação orientada a

objeto tem três pilares, **encapsulamento**, herança e **Polimorfismo**, mas antes de tratarmos destes assuntos se faz necessário o entendimento de alguns conceitos iniciais para que tudo possa ficar claro à medida que a aula for dando andamento.

Encapsulamento

Encapsulamento vem de encapsular, que em programação orientada a objetos significa separar o programa em partes, o mais isolado possível. A idéia é tornar o software mais flexível, fácil de modificar e de criar novas implementações. O Encapsulamento serve para controlar o acesso aos atributos e métodos de uma classe. É uma forma eficiente de proteger os dados manipulados dentro da classe, além de determinar onde esta classe poderá ser manipulada. Usamos o nível de acesso mais restritivo, private, que faça sentido para um membro particular. Sempre usamos private, a menos que tenhamos um bom motivo para deixá-lo com outro nível de acesso. Não devemos permitir o acesso público aos membros, exceto em caso de ser constantes. Isso porque membros públicos tendem a nos ligar a uma implementação em particular e limita a nossa flexibilidade em mudar o código. O encapsulamento que é dividido em dois níveis:

- Nível de classe: Quando determinamos o acesso de uma classe inteira que pode ser public ou Package-Private (padrão);
- Nível de membro: Quando determinamos o acesso de atributos ou métodos de uma classe que podem ser public, private, protected ou Package-Private (padrão).

Um pouco sobre Polimorfismo Java

O paradigma da Orientação a Objetos traz um ganho significativo na **qualidade da produção de software**, porém grandes benefícios são alcançados quando as técnicas de programação OO são colocadas em prática com o uso de uma tecnologia que nos permita usar todas as características da OO; além de agregar à programação o uso de **boas práticas de programação** e padrões de projeto, **design patterns**. Um objeto é uma entidade do mundo real que tem uma identidade. Objetos podem representar entidades concretas, um arquivo no meu computador, uma bicicleta ou entidades conceituais, uma estratégia de jogo, uma política de escalonamento em um sistema operacional. Cada objeto ter sua identidade significa que, dois objetos são distintos mesmo que eles apresentem exatamente as mesmas características.

Objetos são instâncias de classes, que determinam qual informação um objeto contém e como ele pode manipulá-la. Um programa desenvolvido com uma linguagem de programação orientada a objetos manipula estruturas de dados através dos objetos da mesma forma que um programa em linguagem tradicional utiliza variáveis.

Em orientação a objeto, uma classe é uma estrutura que abstrai um conjunto de objetos com características similares. Uma classe define o comportamento de seus objetos através de métodos e os estados possíveis destes objetos através de atributos. Em outros termos, uma classe descreve os serviços providos por seus objetos e quais informações eles podem armazenar. Classes não são diretamente suportadas em todas as linguagens, e são necessárias para que uma linguagem seja orientada a objetos. A programação orientada a objeto tem três pilares, **encapsulamento**, herança e **Polimorfismo**, mas antes de tratarmos destes assuntos se faz necessário o entendimento de alguns conceitos iniciais para que tudo possa ficar claro à medida que a aula for dando andamento.

Relacionado: **Por onde começar no Java - Devcast**

Uma interface nada mais é do que um bloco de código definindo um tipo e os métodos e atributos que esse tipo deve possuir. Na prática o que acontece é que qualquer classe que quiser ser do tipo definido pela interface deve implementar os métodos dessa interface. A interface não contém nenhum código de implementação, apenas assinaturas de métodos e/ou atributos que devem ter seu código implementado nas classes que implementarem essa interface. A Interface define um padrão para especificação do comportamento de classes. Porém, os métodos de uma interface são implementados de maneira particular a cada classe; ou seja, permitem expressar comportamento sem se preocupar com a implementação. Uma interface não possui atributos. Uma classe pode implementar várias interfaces, mas pode ter apenas uma superclasse.

Encapsulamento

Encapsulamento vem de encapsular, que em programação orientada a objetos significa separar o programa em partes, o mais isolado possível. A idéia é tornar o software mais flexível, fácil de modificar e de criar novas implementações. O Encapsulamento serve para controlar o acesso aos atributos e métodos de uma classe. É uma forma eficiente de proteger os dados manipulados dentro da classe, além de determinar onde esta classe poderá ser manipulada. Usamos o nível de acesso mais restritivo, private, que faça sentido para um membro particular. Sempre usamos private, a menos que tenhamos um bom motivo para deixá-lo com outro nível de acesso. Não devemos permitir o acesso público aos membros, exceto em caso de ser constantes. Isso porque membros públicos tendem a nos ligar a uma implementação em particular e limita a nossa flexibilidade em mudar o código. O encapsulamento que é dividido em dois níveis:

- Nível de classe: Quando determinamos o acesso de uma classe inteira que pode ser public ou Package-Private (padrão);
- Nível de membro: Quando determinamos o acesso de atributos ou métodos de uma classe que podem ser public, private, protected ou Package-Private (padrão).

Então para ter um método encapsulado utilizamos um modificador de acesso que geralmente é public, além do tipo de retorno dele. Para se ter acesso a algum atributo ou método que esteja encapsulado utiliza-se o conceito de get e set. Por definição, com SET é feita uma atribuição a algum atributo, ou seja, define, diz o valor que algum atributo deve ter. E com GET é possível recuperar esse valor.

Herança

A herança é um mecanismo da Orientação a Objeto que permite criar novas classes a partir de classes já existentes, aproveitando-se das características existentes na classe a ser estendida. Este mecanismo é muito interessante, pois promove um grande reuso e reaproveitamento de código existente. Com a herança é possível criar classes derivadas, subclasses, a partir de classes bases, superclasses. As subclasses são mais especializadas do que as suas superclasses, mais genéricas. As subclasses herdam todas as características de suas superclasses, como suas variáveis e métodos. A linguagem Java permite o uso de herança simples, mas não permite a implementação de herança múltipla. Para superar essa limitação o Java faz uso de interfaces, o qual pode ser visto como uma “promessa”que certos métodos com características previamente estabelecidas serão implementados, usando inclusive a palavra reservada implements para garantir esta implementação. As interfaces possuem sintaxe similar as classes, no entanto apresentam apenas a especificação das funcionalidades que uma classe deve conter, sem determinar como essa funcionalidade deve ser implementada. Apresentam apenas protótipos dos métodos.

Por exemplo, Imagine que dentro de uma organização empresarial, o sistema de RH tenha que trabalhar com os diferentes níveis hierárquicos da empresa, desde o funcionário de baixo escalão até o seu presidente. Todos são funcionários da empresa, porém cada um com um cargo diferente. Mesmo a secretária, o pessoal da limpeza, o diretor e o presidente possuem um número de identificação, além de salário e outras características em comum. Essas características em comum podem ser reunidas em um tipo de classe em comum, e cada nível da hierarquia ser tratado como um novo tipo, mas aproveitando-se dos tipos já criados, através da herança. **Os subtipos, além de herdarem todas as características de seus supertipos, também podem adicionar mais características, seja na forma de variáveis e/ou métodos adicionais, bem como reescrever métodos já existentes na superclasse, polimorfismo**. A herança permite vários níveis na hierarquia de classes, podendo criar tantos subtipos quanto necessário, até se chegar ao nível de especialização desejado. Podemos tratar subtipos como se fossem seus supertipos, por exemplo, o sistema de RH pode tratar uma instância de Presidente como se fosse um objeto do tipo Funcionário, em determinada funcionalidade. Porém não é possível tratar um supertipo como se fosse um subtipo, a não ser que o objeto em questão seja realmente do subtipo desejado e a linguagem suporte este tipo de tratamento, seja por meio de conversão de tipos ou outro mecanismo. Algumas linguagens de programação permitem

herança múltipla, ou seja, uma classe pode estender características de várias classes ao mesmo tempo. É o caso do C++. Outras linguagens não permitem herança múltipla, por se tratar de algo perigo se não usada corretamente. É o caso do Java. Na Orientação a Objetos as palavras classe base, supertipo, superclasse, classe pai e classe mãe são sinônimos, bem como as palavras classe derivada, subtipo, subclasse e classe filha também são sinônimos.

Polimorfismo

Polimorfismo é o princípio pelo qual duas ou mais classes derivadas de uma mesma superclasse podem invocar métodos que têm a mesma identificação, assinatura, mas comportamentos distintos, especializados para cada classe derivada, usando para tanto uma referência a um objeto do tipo da superclasse. O overload não é um tipo de polimorfismo, pois com overload a assinatura do método obrigatoriamente tem que ter argumentos diferentes, requisito que fere o conceito de Polimorfismo citado acima.

De forma genérica, **polimorfismo significa várias formas**. No caso da Orientação a Objetos, polimorfismo denota uma situação na qual um objeto pode se comportar de maneiras diferentes ao receber uma mensagem, dependendo do seu tipo de criação.

Por exemplo, a operação move quando aplicada a uma janela de um sistema de interfaces tem um comportamento distinto do que quando aplicada a uma peça de um jogo de xadrez. Um método é uma implementação específica de uma operação para certa classe. Polimorfismo também implica que uma operação de uma mesma classe pode ser implementada por mais de um método. O usuário não precisa saber quantas implementações existem para uma operação, ou explicitar qual método deve ser utilizado: a linguagem de programação deve ser capaz de selecionar o método correto a partir do nome da operação, classe do objeto e argumentos para a operação. Desta forma, novas classes podem ser adicionadas sem necessidade de modificação de código já existente, pois cada classe apenas define os seus métodos e atributos. **Em Java, o polimorfismo se manifesta apenas em chamadas de métodos**.

A decisão sobre qual o método que deve ser selecionado, de acordo com o tipo da classe derivada, é tomada em tempo de execução, através do mecanismo de ligação tardia. A ligação tardia ocorre quando o método a ser invocado é definido durante a execução do programa. Através do mecanismo de sobrecarga, dois métodos de uma classe podem ter o mesmo nome, desde que suas assinaturas sejam diferentes, entretanto isso não é polimorfismo. Como dito anteriormente, tal situação não gera conflito, pois o compilador é capaz de detectar qual método deve ser escolhido a partir da análise dos tipos dos argumentos do método. Nesse caso, diz-se que ocorre a ligação

prematura para o método correto. Em Java, todas as determinações de métodos a executar ocorrem através de ligação tardia exceto em dois casos:

1. Métodos declarados como final não podem ser redefinidos e, portanto não são passíveis de invocação polimórfica da parte de seus descendentes; e
2. Métodos declarados como private são implicitamente finais.

No caso de polimorfismo, é necessário que os métodos tenham exatamente a mesma identificação, sendo utilizado o mecanismo de redefinição de métodos, que é o mesmo que sobrescrita de métodos em classes derivadas. A redefinição ocorre quando um método cuja assinatura já tenha sido especificada recebe uma nova definição, ou seja, um novo corpo, em uma classe derivada. É importante observar que, quando polimorfismo está sendo utilizado, o comportamento que será adotado por um método só será definido durante a execução. Embora em geral esse seja um mecanismo que facilite o desenvolvimento e a compreensão do código orientado a objetos, há algumas situações onde o resultado da execução pode ser não-intuitivo.

Introdução a linguagem UML.

A **UML permite que você "desenhe" uma "planta" do seu sistema**. A comparação ideal é a de um construtor que vai realizar um projeto sem antes ter toda a planta que defina estrutura a ser construída. A experiência do construtor garante, até certo ponto, o sucesso do projeto. Mas, com certeza, uma vez feito o planejamento, o "cálculo estrutural", o desenho da planta, a garantia de sucesso antes, durante e depois da efetivação da construção é incomparavelmente maior. O mesmo acontece com um projeto de software.

A experiência do desenvolvedor ou analista, não pode substituir a necessidade de um projeto que defina uma "planta" da solução como um todo. Esta "planta" garante, em todas as fases do projeto, seja na definição, desenvolvimento, homologação, distribuição, utilização e manutenção do mesmo, uma maior clareza e objetividade para execução de cada ação, e, com certeza, quanto maior a solução, maior a necessidade de um projeto definido adequadamente. **Desta forma, a UML é uma linguagem padrão para visualização, especificação, construção e documentação de um aplicativo ou projeto de software**, e objetiva aumentar a produtividade, otimizar as etapas que envolvem o desenvolvimento de um sistema, aumentando assim a qualidade do produto a ser implementado. Ela independe da ferramenta em que o aplicativo será desenvolvido. A idéia e prover uma visão lógica de todo o processo de forma a facilitar a implementação física do mesmo.

A **UML disponibiliza, através de conceitos, objetos, símbolos e diagramas, uma forma simples, mas objetiva e funcional, de documentação e entendimento de um sistema**. Você pode utilizar os diagramas e arquivos que compõe um modelo UML para o desenvolvimento, apresentação, treinamento e manutenção durante todo o ciclo de vida da sua aplicação. Ela é mais completa

que outras metodologias empregadas para a modelagem de dados pois, tem em seu conjunto todos os recursos necessários para suprir as necessidade de todas as etapas que compõe um projeto, desde a definição, implementação, criação do modelo de banco de dados, distribuição, enfim, proporcionando sem qualquer outra ferramenta ou metodologia adicional, um total controle do projeto.

A UML implementa uma modelagem com uma visão **orientada a objetos**. Através dela podemos definir as classes que compõe a nossa solução, seu atributos, métodos e como elas interagem entre si. Apesar da **UML ter como base a orientação a objetos, não significa que a ferramenta e a linguagem utilizada para a implementação do modelo seja também orientada a objetos**, embora seja recomendável. Este artigo não irá explorar os conceitos de orientação a objetos, e sim a **implementação de um modelo UML simples**, para início da documentação de um sistema, utilizando dois diagramas implementados pela UML que são o "**Diagrama de Casos e Uso" e o "Diagrama de Classes".**

Os diagramas têm como objetivo representar, através de um conjunto de elementos, como o sistema irá funcionar e como cada peça do sistema ira trabalhar e interagir com as outras. Outra vantagem vem da facilidade de leitura dos diagramas que compõe a UML, além da facilidade de confeccioná-los, pois existem inúmeras ferramentas para modelagem de dados orientados a objetos (ferramentas Case), dentre elas o Rational Rose, o Model Maker, e o Poseidom UML. **Além dos diagramas citados a UML disponibiliza outros diagramas, dentre os quais podemos citar o Diagrama de Objetos, Diagrama de Seqüência, Diagrama de Colaboração, Diagrama de Estado, Diagrama de Atividade e Diagrama de Componentes.**

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

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.

Neste artigo baseamos nossas explicações conforme desenvolvemos algumas funcionalidades em um sistema web. Portanto é fundamental que o leitor "brinque" um pouco com os recursos aqui exemplificados e adquira conhecimento a partir dessas experiências.

Para isso, desenvolveremos algumas funcionalidades básicas de um sistema de aluguel de carros. Primeiro aprenderemos como criar um pacote JSF 2 instalável em um servidor Java EE. Posteriormente criaremos uma funcionalidade exemplo (um cadastro de modelos de carros), demonstrando alguns componentes básicos do JSF e como funciona o ciclo de vida de uma chamada. Após isto, transformaremos pouco a pouco nossa aplicação, passando por conceitos como navegação, internacionalização, validadores, conversores, layouts, AJAX, etc.

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.

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.

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

- **Listas**
    - Listas encadeadas (Linked Lists)
    - Listas duplamente encadeadas (Doubly-Linked Lists)
- **Filas** (Queues)
- **Pilhas** (Stacks)

Uma **lista** é uma série de elementos ligados. Uma **lista encadeada** ou **lista ligada**, é uma sequência finita de elementos ligadas entre si, onde uma célula da lista aponta para a próxima célula sequencialmente. As listas encadeadas são úteis para representar conjuntos dinâmicos de dados. Ou seja, você não precisa definir um tamanho máximo para uma lista.

As **pilhas** e as **filas** são consideradas listas especializadas por possuírem características de uso próprias. Seus elementos são organizados em função de um critério que regulamenta a entrada e saída dos elementos.

Uma **fila** (Queue) estabelece uma política de entrada e saída **FIFO** (first in, first out), ou seja, o primeiro elemento a entrar na lista é o primeiro a sair. Trata-se de uma lista encadeada onde as regras de inserção e remoção são bem definidas. Os elementos são sempre inseridos no início da lista e removidos do final da lista.

Uma **pilha** estabelece uma política de entrada e saída **LIFO** (last in, first out), o último elemento a entrar é o primeiro a sair. Os elementos são empilhados um a um. Apenas o último elemento
empilhado pode ser removido da pilha, pois se tentarmos remover um elemento, por exemplo, que se encontra no meio da pilha, corremos o risco de desestruturar toda a estrutura.

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

Conceitos

As árvores são estruturas de dados baseadas em listas encadeadas que possuem um nó superior também chamado de raiz que aponta para outros nós, chamados de nós filhos, que podem ser pais de outros nós.

Uma árvore de busca binária tem as seguintes propriedades:

- todos os elementos na subárvore esquerda de um determinado nó n são menores que n;
- todos os elementos na subárvore direita de um determinado nó n são maiores ou iguais a n.

Introdução de estrutura de dados grafos e exemplos

Grafos – Definições

Um grafo G = (V, A) é constituído de um conjunto de vértices e um conjunto de arestas conectando pares de vértices.

Pode ser direcionado, não direcionado, ponderado

Muitas aplicações em computação precisam considerar um conjunto de conexões entre pares de objetos

• Existe um caminho para ir de um objeto a outro seguindo as conexões?

• Qual a menor distância entre um objeto e outro?

• Quantos objetos podem ser alcançados a partir de um determinado objeto?

Alguns problemas práticos que podemos resolver com grafos

• Ajudar máquinas de busca a localizar informação relevante na Web.

• Descobrir os melhores casamentos entre posições disponíveis em empresas e pessoas que aplicaram para as posições de interesse.

• Descobrir qual o roteiro mais curto para visitar as principais cidades de uma regiões turística.

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

O problema do caminho mais curto é um tópico atrativo para pesquisadores e profissionais, pois sua resolução permite obter soluções eficientes a importantes e freqüentes problemas reais ao determinar de modo confiável a forma mais rápida e econômica de realizar uma atividade determinada. Outro ponto que faz com que o tema seja atrativo para pesquisadores é porque tem a característica de que a partir de modelos de redes simples podem-se elaborar modelos mais complexos que são estudados na área de otimização combinatória. Não obstante que a solução de problemas do caminho mais curto seja relativamente fácil, o desenho e a análise de algoritmos eficientes de solução exigem muito engenho, (Ahuja et al., 1993).

Complexidade computacional do problema do caixeiro:

O problema do caixeiro é um clássico exemplo de problema de otimização combinatória. A primeira coisa que podemos pensar para resolver esse tipo de problema é reduzí-lo a um **problema de enumeração**: achamos todas as rotas possíveis e, usando um computador, calculamos o comprimento de cada uma delas e então vemos qual a menor. (É claro que se acharmos todas as rotas estaremos contando-as, daí podermos dizer que estamos reduzindo o problema de otimização a um de enumeração).

Para acharmos o número **R(n)** de rotas para o caso de **n** cidades, basta fazer um raciocínio combinatório simples e clássico. Por exemplo, no caso de n = 4 cidades, a primeira e última posição são fixas, de modo que elas não afetam o cálculo; na segunda posição podemos colocar qualquer uma das 3 cidades restantes B, C e D, e uma vez escolhida uma delas, podemos colocar qualquer uma das 2 restantes na terceira posição; na quarta posição não teríamos nenhuma escolha, pois sobrou apenas uma cidade; consequentemente, o número de rotas é 3 x 2 x 1 = 6, resultado que tinhamos obtido antes contando diretamente a lista de rotas acima.
De modo semelhante, para o caso de n cidades, como a primeira é fixa, o leitor não terá nenhuma dificuldade em ver que o número total de escolhas que podemos fazer é (n-1) x (n-2) x ... x 2 x 1. De modo que, usando a notação de fatorial: **R(n) = (n - 1)!**.

Assim que nossa estratêgia reducionista consiste em gerar cada uma dessas R(n) = (n - 1)! rotas, calcular o comprimento total das viagens de cada rota e ver qual delas tem o menor comprimento total. Trabalho fácil para o computador, diria alguém. Bem, talvez não. Vejamos o porquê.

Suponhamos temos um muito veloz computador, capaz de fazer 1 bilhão de adições por segundo. Isso parece uma velocidade imensa, capaz de tudo. Com efeito, no caso de 20 cidades, o computador precisa apenas de 19 adições para dizer qual o comprimento de uma rota e então será capaz de calcular 10^9 / 19 = 53 milhões de rotas por segundo. Contudo, essa imensa velocidade é um nada frente à imensidão do número 19! de rotas que precisará examinar. Com efeito, acredite se puder, o valor de 19! é 121 645 100 408 832 000 (ou , aproximadamente, 1.2 x 10^{17} em notação científica). Consequentemente, ele precisará de 1.2 x 10^{17} / (53 milhões) = 2.3 x 10^9 segundos

para completar sua tarefa, o que equivale a cerca de **73 anos** . O problema é que **a quantidade (n - 1)! cresce com uma velocidade alarmante, sendo que muito rapidamente o computador torna-se incapaz de executar o que lhe pedimos.** Constate isso mais claramente na tabela a seguir:

n	rotas por segundo	(n - 1)!	cálculo total
5	250 milhoes	24	insignific
10	110 milhoes	362 880	0.003 seg
15	71 milhoes	87 bilhoes	20 min

20	53 milhoes	1.2×10^{17}	73 anos
25	42 milhoes	6.2×10^{23}	470 milhoes de anos

ALGORITMO DE DIJKSTRA O algoritmo de Dijkstra encontra o menor caminho de um nó fonte ou origem s até os outros nós de uma rede orientada para o caso em que todos os pesos dos arcos sejam não negativos. Mantém a distância rotulada d(i) para cada nó i, e é o limite superior do caminho mais curto até o nó i. O algoritmo divide os nós em dois grupos: rotulados permanentemente (ou permanentes) e rotulados temporariamente (temporários). A distância rotulada dos nós permanentes representa a menor distância do nó fonte até um outro nó. No caso dos nós temporários, esta distância representa o limite superior da distância do caminho mais curto até o nó. O algoritmo examina todos os nós sem incluir o nó fonte s e rotula permanentemente os nós em ordem das distâncias ao nó fonte. Inicia-se o algoritmo com d(s) = 0 e , d i)(= ∞ para todo j que pertence à rede. Em cada iteração o rótulo do nó i é o menor caminho desde o nó fonte ao longo de um caminho que contém outros nós intermediários. O algoritmo escolhe o nó i com a mínima distância rotulada temporária para fazê-la permanente e visita os outros nós, ou seja, examina os arcos de A(i) e atualiza as distâncias rotuladas dos nós adjacentes. O algoritmo termina quando todos os nós são designados como permanentes. A correção do algoritmo baseia-se na observação que é sempre possível designar o nó com a mínima distância temporária como permanente. O algoritmo de Dijkstra finaliza quando não existirem nós com rótulos temporários (caminho mais curto do nó s para todos os outros) ou quando o rótulo do nó i passar a permanente (caminho mais curto do nó s para o nó i). Este algoritmo mantém a árvore orientada T com raiz o nó fonte e que abrange os nós com distâncias finitas rotuladas. Para i,(, neste algoritmo pred(j) = i, é registrado, e j) ∈T para cada arco i,(, j) ∈T ij d(. No fim do algoritmo quando as distâncias rotulad j) = d i)(+ c as representam os caminhos mais curtos, T é a árvore de caminho mínimo. No algoritmo de Dijkstra a operação de seleção da mínima distância rotulada temporária conhece-se como “operação de seleção de nós”. A operação de verificação de rótulos atuais dos nós i e j que satisfaçam a condição ij d(, e se isto acontece j) > d i)(+ c então ij d(, conhece-se como operação de “atualização de dist j) = d i)(+ c âncias”. Pela forma como o algoritmo determina o caminho mais curto, o número de operações realizadas vem determinado a partir de duas operações: seleção de nós e atualização de distâncias. Na seleção de nós esta operação é realizada em n nós e a partir de cada um destes são analisados (n-1) nós associados aos (n-1) arcos que saem de cada nó. Esta operação não é feita nos nós que têm o rótulo permanente por tanto a partir do k-ésimo nó selecionado são analisados n-k nós. Assim o número de operações da seleção de nós é: $n + (n - 1) + (n - 2) + K + 1 = O(n^2)$

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

Variáveis

As variáveis são posições na memória do computador que podem armazenar dados. As variáveis são formadas por quatro elementos: **nome, tipo, tamanho e valor**.

Dependendo da programação, o básico de uma declaração de variável pode ter somente um tipo, um nome e um valor.

O Java possui dois tipos de dados que são divididos em **por valor** (tipos primitivos) e **por referência** (tipos por referência).

Os tipos primitivos são **boolean, byte, char, short, int, long, float e double**. Os tipos por referência, são classes que especificam os tipos de objeto **Strings, Arrays Primitivos e Objetos**.

Uma variável do tipo primitivo pode armazenar exatamente um valor de seu tipo declarado por vez, quando outro valor for atribuído a essa variável, seu valor inicial será substituído.

As variáveis de instância de tipo primitivo são inicializadas por padrão, as variáveis dos tipos byte, char, short, int, long, float e double são inicializadas como 0, e as variáveis do tipo boolean são inicializadas como **false**. Esses tipos podem especificar seu próprio valor inicial para uma variável do tipo primitivo atribuindo à variável um valor na sua declaração.

O Java fornece dois tipos primitivos para armazenar números de ponto flutuante na memória, o tipo float e double.

A diferença entre eles é que as variáveis double podem armazenar números com maior magnitude e mais detalhes, ou seja, armazena mais dígitos à direita do ponto de fração decimal, do que as variáveis float. As variáveis do tipo float representam números de ponto flutuante de precisão simples e podem representar até 7 dígitos.

As variáveis do tipo double representam números de ponto flutuante de precisão dupla, onde precisam duas vezes a quantidade de memória das variáveis float fornecendo 15 dígitos, sendo o dobro da precisão de variáveis float. Os valores do tipo double são conhecidos como literais de ponto flutuante. Para números de ponto flutuante precisos, o Java fornece a classe BigDecimal (pacote java.math).

Constantes

As constantes são melhor entendidas junto com o conceito de orientação a objetos e classes, porém vamos apresentar a sua definição mas não se preocupe, pois elas serão melhor abordadas mais a frente.

Assim como uma variável final, uma constante é declarada quando precisamos lidar com dados que não devem ser alterados durante a execução do programa. No Java declaramos uma constante utilizando as palavras-chave static final antes do tipo da variável

Estrutura de Seleção

A **estrutura de seqüência:** as ações são executadas, uma por vez, de forma encadeada, na ordem definida no programa.

A estrutura de seleção: a partir da verificação de uma condição, o programa realiza ou não uma ação e volta à seqüência do programa.

A estrutura de repetição: um bloco de ações é repetido um número de vezes conforme se deseje, e após isso

o controle volta à seqüência do programa.