

CDI

Integre as dependências e contextos
do seu código Java



Sumário

1	Antes de falar de CDI...	1
1.1	Boas práticas de orientação a objetos	2
1.2	Lidando com o acoplamento	7
1.3	Evolução das ferramentas de gestão de dependência	11
1.4	O bom e velho arquivo properties	11
1.5	Buscando as dependências com Spring e XML	14
1.6	Buscando as dependências com Seam e anotações	17
1.7	De onde vimos?	19
2	O que é e para que serve a CDI	21
2.1	Para onde vamos?	21
2.2	Evitando que a complexidade do código sempre aumente	21
2.3	Introdução à injeção de dependências	31
2.4	A motivação da CDI	36
2.5	O que é e para que serve uma especificação?	36
2.6	A CDI é só para Java EE mesmo?	38
3	Iniciando um projeto com CDI	39
3.1	Olá CDI	39
3.2	O que é um pacote CDI?	47
3.3	Usando injeção em propriedades, construtores e métodos inicializadores	48
3.4	As formas de se injetar dependências	49
3.5	Recapitulando	52

4	Resolvendo dependências com tipagem forte	53
4.1	Lidando com a ambiguidade nas dependências	54
4.2	Os qualificadores	58
4.3	Alternatives: eliminando ambiguidade e tornando um bean opcional	65
4.4	Prioridade: novidade da CDI 1.1, parte do Java EE 7	68
4.5	Beans nomeados	70
4.6	Trabalhando com herança entre beans	72
4.7	Restringindo o tipo dos beans	74
4.8	Resolução de dependência sob demanda e lookup programático . . .	76
4.9	Resumo do funcionamento da resolução de dependências	80
5	O ciclo de vida dos objetos gerenciados pela CDI	83
5.1	Métodos produtores	84
5.2	Escopos da CDI	89
5.3	Escopo de requisição com @RequestScoped	94
5.4	Escopo de sessão com o @SessionScoped	97
5.5	@ApplicationScoped: O maior escopo possível	99
5.6	@ConversationScoped: Você criando seu próprio escopo	101
5.7	@Dependent: O escopo padrão do CDI	108
5.8	Métodos finalizadores	111
6	Interceptors e Decorators	113
6.1	Implementando requisitos transversais com interceptors	113
6.2	Interceptadores de ciclo de vida	121
6.3	Estendendo um bean existente com Decorators	124
6.4	Vinculação dinâmica entre o objeto decorado e seu decorador	126
7	Definindo Estereótipos e diminuindo o acoplamento utilizando Eventos	133
7.1	Definindo estereótipos	134
7.2	Utilizando Eventos para obter um baixíssimo acoplamento	137
7.3	Observadores síncronos de eventos	141
7.4	Eventos e qualificadores	141
7.5	Observadores assíncronos de eventos	146
7.6	Injetando dependências nos observadores	148

8	A relação da CDI com as demais especificações Java EE	151
8.1	Relação entre CDI e EJB/JPA	152
8.2	Relação entre CDI e JSF	156
8.3	Relação entre CDI e JMS	163
8.4	Empacotando as melhorias	169
9	Extensões CDI	171
9.1	Do Seam à CDI	172
9.2	Extensões portáveis disponíveis	174
9.3	Criando nossas extensões	175
9.4	Extensão portátil: usando convenção em vez de configuração	177
9.5	Extensão não portátil: eventos assíncronos fora do Java EE	180
10	Executando CDI em diferentes ambientes	189
10.1	Executando CDI em servlet containers	190
10.2	Executando CDI a partir de uma classe main	193
10.3	E como testar uma aplicação CDI?	195
10.4	Desenvolvendo testes de integração com persistência	201
	Índice Remissivo	208

CAPÍTULO 1

Antes de falar de CDI...

O assunto principal deste livro é a CDI, mas não iremos começar nossos estudos diretamente nela. Ter em mãos uma boa ferramenta é muito bom, porém, antes de utilizá-la, precisamos ver sobre desenvolvimento de software como um todo. Mesmo que você seja um programador experiente, vale a pena revisitar tópicos importantes como esses. Até porque ser experiente não é sinônimo de saber tudo.

O bom desenvolvimento de software não é uma tarefa trivial. Equilibrar a balança que tem, de um lado, a alta coesão, e de outro, o baixo acoplamento é algo que se torna mais simples com a experiência e observando bons códigos.

Não é possível justificar a má qualidade de um software pela quantidade de mudanças que o mesmo precisou sofrer por alterações no negócio, ou simplesmente por mudança de opinião do cliente. Isso vai acontecer, e cabe a nós, desenvolvedores, projetarmos nossas aplicações de forma a acomodar da melhor forma possível essas mudanças.

A menos que o software que estivermos desenvolvendo seja algo para uma necessidade temporária, e não tenha um uso permanente, certamente ele mudará, e

não apenas uma vez, e com isso algumas partes dele irão sofrer com essa mudança. Pode ser até que, dependendo da urgência, a parte afetada pela mudança precise ser melhorada depois, mas não podemos deixar que esse efeito seja propagado para toda a aplicação.

Tomemos como exemplo uma aplicação que controle tudo relacionado aos recursos humanos de uma empresa, desde o cadastro detalhado dos dados dos funcionários até a folha de pagamento. Agora vamos considerar uma mudança de legislação que nos obrigue a mudar a forma como a folha de pagamento é calculada, e essa seja simplesmente a parte mais complexa de todo o sistema, uma vez que além do salário de cada funcionário, é preciso também calcular todos os impostos e descontos de plano de saúde, empréstimos com desconto em folha e tudo mais.

Nesse cenário, é possível argumentar que a mudança é inviável por mexer no cerne do sistema? Obviamente, não. Então nosso sistema precisa ser *resiliente*, ou seja, suportar bem as mudanças.

No cenário apresentado, a mudança de todo o cálculo da folha de pagamento provavelmente não será a pior parte do trabalho. Geralmente a equipe sabe programar, escrever código, e consequentemente vai saber passar para a IDE as alterações na legislação. A pior parte costuma ser o reflexo dessa mudança no restante do software. Apesar de não ser o foco do livro, veremos um pouco sobre testes no capítulo 10, e então entenderemos o quanto essa é uma ferramenta importante para minimizar os riscos desses comportamentos não previstos após uma alteração. Mas antes mesmo de escrever testes para o sistema, temos que projetar esse sistema de forma a não propagarmos as mudanças de uma determinada parte para o sistema todo. Esse efeito cascata é que costuma ser a pior parte de uma mudança grande.

1.1 BOAS PRÁTICAS DE ORIENTAÇÃO A OBJETOS

A baixa coesão e o alto acoplamento são as principais causas de uma mudança, que deveria ficar retida em uma parte, sair destruindo o sistema todo. Mas o que realmente é isso? Ouvimos tanto falar em coesão e acoplamento mas as vezes não temos certeza do que isso significa.

Coesão

A coesão é a capacidade de um objeto estar focado na sua tarefa específica. Se seu papel é calcular o salário do funcionário, então que faça isso. Ele não deve calcular o salário e calcular o desconto do imposto de renda, e calcular o desconto do plano

de saúde, e calcular as férias proporcionais daquele mês, e mais um monte de coisas. Isso porque se uma dessas coisas mudar, nosso código muda.

Não existe porém uma regra que determina até que ponto algo deve estar junto, e a partir desse ponto as coisas devem ser separadas. No exemplo anterior, podemos ter tudo isso num mesmo objeto se julgarmos o melhor local, mas seria interessante termos métodos específicos separando cada tarefa, ainda que sejam privados, e depois ter um método que combine todos esses subcálculos.

Podemos perceber que, dessa maneira, criamos níveis de retenção da mudança. Se o cálculo de imposto de renda mudar, mudamos apenas o método específico e os demais não são afetados. Se todo o cálculo da folha de pagamento mudar, talvez tenhamos que alterar a classe toda, mas essa mudança ainda ficaria restrita a essa classe, não afetando as demais do sistema.

Granularidade

E quanto ao fato de deixarmos os subcálculos em métodos específicos dentro de uma mesma classe, não seria mais adequado colocarmos cada um deles em uma classe diferente? Sem fazer isso não estaríamos criando uma classe pouco coesa, que faz coisas demais?

Novamente, não existe uma resposta para qualquer cenário. Essa medida de tamanho de cada unidade, como método, classe, subsistema, chamamos de granularidade. Se tivermos classes menores, cada uma contendo um subcálculo desses, podemos dizer que estamos trabalhando com uma granularidade mais fina. É só imaginar cada classe como um grão de areia. Nesse caso teríamos uma areia bem fina.

Agora se fizermos cada parte do cálculo maior em um método diferente, mas na mesma classe, e formos aplicando a mesma ideia em todo nosso sistema, nossas classes serão como grãos maiores de areia — logo teremos uma granularidade mais grossa.

E qual o tamanho ideal? A partir de qual momento precisamos transformar partes do nosso método em métodos separados, e depois separar esses métodos em classes específicas? Como não há resposta mágica, costumamos deixar nosso método no tamanho máximo de uma tela de um monitor médio. Então, se formos analisar o código de um método, podemos fazê-lo sem rolar a imagem no monitor. Mas note que consideramos um monitor médio, pois de nada adianta você ter um monitor de 30” com uma resolução absurda enquanto os outros membros da equipe têm monitores de 17” ou 19” com uma resolução bem menor. E pense, ainda que todos tenham

essa tela enorme, será que para enchê-la toda com código, não teremos informação demais para analisar? A analogia do método do tamanho de uma tela é para ajudar que ele seja pequeno, de fácil compreensão. Você tem que olhar e logo entender. Se a tela for muito grande e a fonte muito pequena você pode conseguir colocar uma bíblia inteira nela sem precisar utilizar a rolagem, mas isso não vai ajudar muito.

De forma semelhante, devemos cuidar do tamanho das nossas classes. Não há uma regra baseada na quantidade de métodos, pois o tamanho deles pode variar bastante. Podemos ter métodos com meia dúzia de linhas e outros do tamanho da nossa tela. Mas é interessante definir uma meta comum para a toda a equipe, ou para você, caso trabalhe sozinho. Defina por exemplo 150 ou 200 linhas e veja se funciona bem. Utilize ferramentas na IDE ou no servidor de integração contínua que faça relatórios automáticos de quais classes ou métodos estão muito grandes, isso vai te ajudar no começo, mas lembre-se que o mais importante é gerenciarmos o significado dos métodos e das classes. Você não vai quebrar um método ou classe porque uma ferramenta está mandando, você vai usar esse indicador como um alerta te avisando que o método ou classe está grande demais, e então você vai readequá-los para que se mantenham sempre simples de entender e fáceis de alterar, pois mudanças sempre existirão.

OUTROS CENÁRIOS ONDE AVALIAMOS A GRANULARIDADE

Esse conceito de granularidade é importante também em outros cenários, como na programação distribuída. Na programação local (não envolvendo passagem de valores pela rede) podemos chamar diversos métodos, cada uma realizando uma pequena parte do trabalho e juntando o resultado em milissegundos. Quando fazemos uma programação distribuída, chamando um método que está rodando em outra máquina, precisamos que esse método faça mais coisas, para precisarmos chamar menos métodos diferentes para obter o mesmo resultado, pois cada invocação de método diferente passa pela rede para chegar ao outro computador, então é melhor um número menor de invocações.

Nesse caso da programação local versus a programação distribuída, quando os métodos nos dão respostas menores, temos uma granularidade mais fina, e quando as respostas são mais completas, temos uma granularidade mais grossa. Qual a melhor? Geralmente quando programamos local, uma granularidade mais fina nos dará métodos menores, e consequentemente mais fáceis de entender. Já quando temos uma chamada de método remota, é melhor uma granularidade mais grossa, pois temos a latência de rede no meio do caminho. Mesmo nesse tipo de programação, devemos ter a separação de cada parte do trabalho feita da mesma maneira que faríamos na programação local, facilitando o entendimento do código. A diferença é que não chamaremos cada um desses métodos, e sim um método que combine todos eles.

Acoplamento

Enquanto a coesão é a medida do quanto um objeto é focado em sua tarefa, o acoplamento é a medida da sociabilidade desse objeto, ou o quanto ele se relaciona com os outros objetos e a força desses relacionamentos. Por exemplo, receber um objeto do tipo `Funcionario` como parâmetro de um método é um acoplamento menor que ter dentro da nossa classe uma propriedade desse tipo, que por sua vez é menor também do que nossa classe ser um subtipo de `Funcionario`.

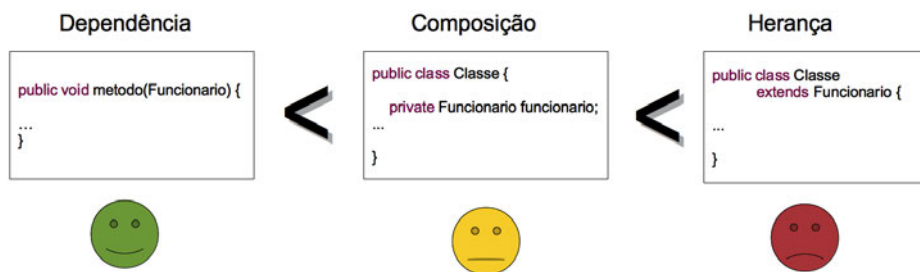


Figura 1.1: Ilustração referente ao acoplamento, quando menor, melhor

Coesão e acoplamento são lados opostos de uma balança. À medida que procuramos manter uma alta coesão, deixaremos de ter classes “sabe-tudo” e teremos mais relacionamentos entre classes com papéis específicos. Mas se tentarmos diminuir demais o acoplamento, voltaremos a ter classes que agregam muitas funções, visando diminuir as dependências entre elas, mas acabando com sua coesão.

Para nossa sorte, essa equação já vem sendo resolvida há bastante tempo, e a forma como isso tem sido feito é focar na coesão, e delimitar bem as fronteiras entre as classes para que suas inter-relações não causem um acoplamento negativo. Um acoplamento negativo é aquele em que as classes que se relacionam conhecem demais o trabalho da outra. Assim, por mais que as responsabilidades estejam (parcialmente) separadas, a classe `A` sabe como a classe `B` está implementada e toma diversas decisões levando isso em consideração. Na prática, a implementação de uma é continuação da outra.

Ao definir fronteiras entre as classes, devemos cuidar para que uma funcione de forma totalmente independente da outra. Por mais que a mesma pessoa codifique ambas, é importante que as decisões internas de uma não se propagem para a outra. Assim, se uma mudar, a outra pode continuar igual.

Por exemplo, vamos considerar uma classe que calcula toda a folha de pagamento de uma empresa, e outra classe que exibe os dez maiores salários para o presidente da empresa. Imagine que, ao calcular a folha de pagamento, a primeira classe armazena uma lista dos funcionários que tiveram seus salários calculados em ordem decrescente de salário. E a segunda classe simplesmente exibe os primeiros dez elementos dessa lista, pois sabe que após o cálculo a lista estará ordenada.

Ocorre porém que, por questões de performance, a primeira classe é alterada, de

modo a realizar o cálculo dos salários de forma paralela, e após a junção do trabalho de cada `Thread` o resultado não está mais ordenado. Nesse caso, a classe que exibe os dez maiores salários sofrerá um impacto por estar acoplada demais com a primeira. Esse tipo de acoplamento é um que devemos evitar.

Defina interfaces claras entre as classes, deixando nítido o que cada uma recebe e devolve de informações para a outra, e implemente cada classe de forma que somente a interface da outra seja conhecida, e sua implementação interna seja totalmente ignorada.

1.2 LIDANDO COM O ACOPLAMENTO

Como dito anteriormente, lidar de forma eficiente com acoplamento é uma matéria que vem sendo aprimorada há muitos anos. Existem diversos padrões de projeto que nos ajudam a gerenciá-lo de forma eficiente, e também ferramentas — ou frameworks — que nos ajudam a implementar esses padrões de uma maneira simples.

Esse não é um livro de padrões de projetos, mas veremos alguns para nos ajudar a entender na prática esses conceitos. Um padrão fácil de entender e que servirá para nosso exemplo inicial é o *Strategy*. Imagine, por exemplo, que precisamos compactar um arquivo, que será representado por um array de bytes. Precisamos de uma classe que receba esse array e devolva um novo array que represente o arquivo já compactado, porém temos diferentes formas, ou estratégias de compactação, como o zip, o rar e o bzip2. O interessante desse padrão é que podemos facilmente trocar a estratégia, ou algoritmo utilizado para resolver o mesmo problema, sem alterar nosso código para isso.

Trazendo para um exemplo mais próximo da nossa realidade, vamos considerar o nosso cálculo de salários. Podemos ter diferentes implementações de uma calculadora de salário no nosso sistema, uma para cada versão do plano de cargos e salários da empresa. Provavelmente nesse caso não teremos essas diferentes versões executando em um mesmo momento, já apenas um plano de cargos fica em vigor por vez. Mesmo assim aplicaremos o mesmo conceito pois a interface continuará a mesma, e apenas a implementação será alterada.

Para exemplificar vamos considerar a interface a seguir.

```
public interface CalculadoraDeSalarios {  
  
    double calculaSalario(Funcionario funcionario);  
  
}
```

E vamos considerar que no ano 2000, a empresa possuía um salário único para os funcionários, todos recebiam R\$ 2.000,00.

```
public class CalculadoraDeSalariosPadrao
    implements CalculadoraDeSalarios {

    public double calculaSalario(Funcionario funcionario) {
        return 2000.0;
    }
}
```

Já iniciamos bem, utilizando uma interface que define o que se espera de uma calculadora de salários, mas mesmo assim precisaremos instanciar um objeto da nossa classe quando formos utilizar a calculadora. Logo, teríamos um trecho de código parecido com o seguinte.

```
...
Funcionario funcionario = buscaAlgumFuncionarioDeAlgumLugar();
CalculadoraDeSalarios calculadora = new CalculadoraDeSalariosPadrao();
double salario = calculadora.calculaSalario(funcionario);
...
```

Toda vez que precisássemos de uma instância da `CalculadoraDeSalarios` seria necessário usar o operador `new` e criar uma nova instância da implementação que temos. Mas, por mais que a interface facilite bastante na evolução do sistema, permitindo-nos trocar a implementação sem quebrar os clientes, quando mudarmos essa implementação, precisaremos alterar todo nosso sistema, buscando onde a implementação antiga estava sendo usada e trocar pela nova. Essa nova implementação pode ser, por exemplo, com base em um novo plano de cargos e salários e tenha saído em 2002.

```
public class CalculadoraDeSalariosPlano2002
    implements CalculadoraDeSalarios{

    public double calculaSalario(Funcionario funcionario) {
        if(funcionario.getEscolaridade() == Escolaridade.SUPERIOR){
            return 3000.0;
        }
        else{
            return 2000.0;
        }
    }
}
```

```
}  
  
}
```

Com base no novo plano, funcionários com nível superior passam a ganhar R\$ 3.000,00, enquanto os demais continuam ganhando R\$ 2.000,00. Não se preocupe com a implementação, ela está propositalmente simples, a ideia é percebermos o problema que teremos, pois precisaremos fazer um *find replace* na nossa aplicação trocando todos os lugares onde a implementação antiga era usada.

Por esse problema, geralmente aliado ao padrão *Strategy*, costumamos utilizar o padrão *Factory Method*. Assim temos um método que sabe criar objetos da forma correta. No nosso caso, esse objeto é a implementação de `CalculadoraDeSalarios`. A nossa implementação desse método construtor de objetos pode ser algo bem simples como o que segue.

```
public class CalculadoraDeSalariosFactory {  
    public CalculadoraDeSalarios criaCalculadora(){  
        return new CalculadoraDeSalariosPlano2002();  
    }  
}
```

E seu uso ficaria da seguinte forma.

```
...  
Funcionario funcionario = buscaAlgumFuncionarioDeAlgumLugar();  
CalculadoraDeSalariosFactory factory =  
    new CalculadoraDeSalariosFactory();  
CalculadoraDeSalarios calculadora = factory.criaCalculadora();  
double salario = calculadora.calculaSalario(funcionario);  
...
```

A vantagem de termos um meio “terceirizado” de criar nossos objetos, é que qualquer mudança na forma de criá-los fica escondida dentro dele. Até agora, a mudança que tivemos foi apenas trocar a implementação da `CalculadoraDeSalarios`, mas a alteração poderia ser maior.

Apenas para evoluir nessa linha de raciocínio, consideremos que, mesmo após a reestruturação do plano de cargos da empresa, em 2002, os funcionários reivindicaram que fosse seguido o piso salarial da categoria, e que essa solicitação foi atendida. Logo, como desenvolvedores do sistema que calcula a folha de pagamento, precisaremos utilizar essa referência salarial para calcular os salários, e para tal, definimos a seguinte interface.

```
public interface TabelaDeReferenciaSalarial {  
    double buscaPisoSalarial(Cargo cargo);  
}
```

Assim, teremos que garantir que o salário do funcionário nunca seja menor que o piso do cargo que ele ocupa. Como esse é um exemplo apenas didático, que nem fará parte do projeto que desenvolveremos, vamos ignorar sua implementação, considerando apenas que ela se chama `TabelaDeReferenciaSalarialPadrao`. Agora precisamos apenas passar essa tabela para nossa `CalculadoraDeSalarios`. Para que essa dependência fique mais explícita podemos definir um método `setTabelaDeReferenciaSalarial` na interface `CalculadoraDeSalarios`. Dessa forma, quem implementar essa interface saberá que obrigatoriamente precisa receber essa tabela.

```
public interface CalculadoraDeSalarios {  
  
    double calculaSalario(Funcionario funcionario);  
    void setTabelaDeReferenciaSalarial(  
        TabelaDeReferenciaSalarial tabela);  
  
}  
  
public class CalculadoraDeSalariosFactory {  
    public CalculadoraDeSalarios criaCalculadora(){  
        TabelaDeReferenciaSalarial pisosSalariais =  
            new TabelaDeReferenciaSalarialPadrao();  
  
        CalculadoraDeSalarios calculadora =  
            new CalculadoraDeSalariosPlano2002();  
  
        calculadora.setTabelaDeReferenciaSalarial(pisosSalariais);  
        return calculadora;  
    }  
}
```

Veja que interessante, dessa vez não trocamos a implementação da `CalculadoraDeSalarios`, mas mudamos a forma como ela é construída. Ainda assim não precisamos sair alterando nosso código em todo lugar, e isso é muito bom. Contudo, quase sempre há espaço para melhorias.

1.3 EVOLUÇÃO DAS FERRAMENTAS DE GESTÃO DE DEPENDÊNCIA

Apesar de muito simples, o que fizemos com nossa *factory* foi uma forma de gerenciar as dependências da nossa aplicação, especificamente as implementações de *CalculadoraDeSalarios* e *TabelaDeReferenciaSalarial*, esta última é uma dependência indireta da nossa aplicação, pois quem depende dela é a calculadora.

O trabalho está indo bem, mas nosso objetivo é conseguir trocar essas dependências sem precisar alterar nossa aplicação, nem que essa alteração esteja restrita a uma única classe. Em outras palavras, veremos o que será preciso fazer para trocarmos os componentes (dependências) da nossa aplicação sem a necessidade de recompilação alguma.

Para exemplificar, imagine que depois de muitos anos do sistema em execução, o sindicato disponibiliza um webservice com as referências salariais, e você crie uma nova implementação da *TabelaDeReferenciaSalarial* com base nesse serviço web. Com as alterações que faremos a partir de agora, será possível colocar um novo jar com a implementação nova no classpath da aplicação, alterar um arquivo de configuração (ou as vezes nem isso), e ao reiniciar a aplicação a nova implementação já está em uso.

Já foi falado que existem frameworks que nos ajudam nesse trabalho, e veremos os principais precursores da CDI nas seções a seguir; mas antes disso veremos como fazer essa gestão manualmente.

1.4 O BOM E VELHO ARQUIVO PROPERTIES

A forma mais simples de tirarmos a instanciação das dependências de dentro do código, é usando arquivo de propriedades, no formato chave=valor. Para trabalhar com esses arquivos, usaremos a classe `java.util.ResourceBundle`. Sua utilização é bem simples, e veremos melhor através de um exemplo.

Primeiro precisaremos criar um arquivo na raiz do classpath da nossa aplicação. Para isso, coloque o arquivo na pasta `src` do seu projeto do eclipse, ou `src/java` se estiver utilizando o netbeans. Esses são os diretórios padrões, mas você pode especificar outro diretório dentro da IDE. O importante é que o arquivo com nome `dependencias.properties` fique no pacote *default*.

Após criar esse arquivo, que é um arquivo texto simples, vamos deixá-lo com o seguinte conteúdo.


```
CalculadoraDeSalarios br.com...CalculadoraDeSalariosPlano2002  
TabelaDeReferenciaSalarial br.com...TabelaDeReferenciaSalarialPadrao
```

O divisor entre a chave e o valor pode ser um `=` ou simplesmente um espaço em branco, que foi o utilizado. E como vamos instanciar essas classes de forma dinâmica, é importante colocarmos o nome completo da classe, que inclui o pacote. No trecho de exemplo os nomes foram encurtados para ser mais facilmente visualizado.

A seguir o trecho que cria a instância da classe a partir da leitura de seu nome nesse arquivo properties.

```
public class CalculadoraDeSalariosFactory {  
  
    private ResourceBundle bundle =  
        ResourceBundle.getBundle("dependencias");  
  
    public CalculadoraDeSalarios criaCalculadora(){  
        String nomeDaClasse =  
            bundle.getString("calculadora_de_salarios");  
        try {  
            Class<?> classe = Class.forName(nomeDaClasse);  
            return classe.asSubclass(CalculadoraDeSalarios.class)  
                           .newInstance();  
        } catch (Exception e) {  
            return null;  
        }  
    }  
}
```

Como podemos ver, obtemos uma instância de `ResourceBundle` através do método `getBundle`, que recebe como parâmetro o nome do arquivo properties que criamos, sem a extensão `.properties`. Com esse objeto em mãos, conseguimos recuperar qualquer valor de dentro do arquivo passando sua respectiva chave para o método `getString`. Por fim, dentro de um `try/catch`, fizemos a instanciação da classe a partir de seu nome, e depois a instanciação do objeto a partir da instância da classe. O método `asSubclass` está sendo usado apenas para retornarmos o tipo correto da classe, caso contrário o método `newInstance` devolveria um `Object` em vez do tipo que esperamos, e então teríamos que fazer um cast.

O uso do `try/catch` é obrigatório pois estamos lidando com nome de classes que podem estar digitados incorretamente por exemplo, ou a classe pode não ter ser visível ou não ter um construtor padrão visível.

Esse trecho serviu para entendermos como instanciar um objeto a partir de um arquivo `properties`, mas agora precisamos terminar de criar nossa `CalculadoraDeSalarios` de forma correta, pois ainda falta instanciarmos a `TabelaDeReferenciaSalarial`. Para que o código não fique muito repetitivo, vamos criar um método privado chamado `criaInstancia`. Esse será um método genérico nos dois sentidos da palavra, pois usa *generics* e serve para qualquer tipo de classe.

```
private <T> T criaInstancia(Class<T> classe){
    String nomeDaClasse = bundle.getString(classe.getSimpleName());
    try {
        Class<?> clazz = Class.forName(nomeDaClasse);
        return clazz.asSubclass(classe).newInstance();
    } catch (Exception e) {
        return null;
    }
}
```

Tanto nesse código, quanto no anterior, o ideal é que usemos uma API de `Logger` para registrar um possível erro nesse processo, e não apenas devolver `null` sem sabermos o real motivo de isso ter ocorrido.

Analizando esse código, percebemos que utilizamos o nome simples da classe como chave a ser buscada no `dependencias.properties`. Isso deu certo porque utilizamos o mesmo nome da interface que desejamos instanciar como chave dentro do arquivo, o que facilitou nosso trabalho agora. O restante do código é praticamente o mesmo que vimos anteriormente.

Agora que já temos uma estrutura para instanciar nossos objetos, podemos voltar ao método `criaCalculadora` da nossa `factory`, que é o que realmente devolve a dependência que precisamos.

```
public CalculadoraDeSalarios criaCalculadora(){
    TabelaDeReferenciaSalarial pisosSalariais =
        criaInstancia(TabelaDeReferenciaSalarial.class);

    CalculadoraDeSalarios calculadora =
        criaInstancia(CalculadoraDeSalarios.class);

    calculadora.setTabelaDeReferenciaSalarial(pisosSalariais);
}
```

```
    return calculadora;  
}
```

Uma vez que criamos o método genérico `criaInstancia`, ficou bem simples criar instâncias de qualquer classe. Em um sistema real, esse método provavelmente ficaria em uma classe utilitária que seria utilizada por qualquer factory, ou ainda evoluiríamos mais esse método para que pudéssemos ter um factory completamente genérica, que utilizasse *reflection* para buscar subdependências como a `TabelaDeReferenciaSalarial` dentro da nossa `CalculadoraDeSalarios`.

Outra opção seria também trocar os arquivos de propriedades por registro em banco de dados. Nesse caso, se as classes já estivessem no classpath da aplicação, não seria necessário nem mesmo reiniciá-la para trocarmos a implementação de uma interface. Porém, independentemente de como continuaríamos o desenvolvimento da nossa aplicação, já vimos o que seria necessário para gerenciarmos manualmente a implementação das dependências das nossas classes. Agora passaremos a ver como podemos fazer essa gestão através de frameworks.

1.5 BUSCANDO AS DEPENDÊNCIAS COM SPRING E XML

O Spring é um framework muito importante para a história do Java, pois enquanto o JEE do passado, chamado à época de J2EE, era muito verboso no seu uso, o Spring facilitou bastante as coisas, dando uma alternativa por fora da “linha oficial” do Java. Seria como comprar um carro mas ir a uma boa oficina não oficial, pois a oficial era demasiadamente burocrática e levávamos muito tempo para conseguir agendar uma revisão. Se analisarmos os códigos feitos até alguns anos usando Spring veremos muitos arquivos XML, o que acabou inclusive na época caracterizando pejorativamente o framework, e até certo ponto a própria plataforma Java como sendo orientada a arquivos XML.

Nessa seção, estaremos vendo o Spring do ponto de vista histórico, então, por mais que hoje seja possível utilizar anotações em sua configuração, no momento histórico em que ele foi mais importante a configuração era basicamente por XML. Porém se formos iniciar um projeto utilizando Spring hoje em dia, certamente não faríamos como veremos a seguir.

Outro ponto importante de considerarmos é que estaremos interagindo com a API do Spring de uma forma desanexada de seu contexto. Estaremos basicamente utilizando o Spring como factory. Faremos isso apenas para simplificar o entendimento, e não visando comparações com Seam ou CDI. Quando digo fora do contexto

significa que partiremos de uma classe simples, ou até mesmo um método *main*, e então chamaremos o primeiro bean através do contexto do Spring.

Para continuar na mesma linha do exemplo anterior, vamos considerar a mesma factory que cria `CalculadoraDeSalarios`. Porém agora iremos implementá-la usando Spring.

```
import org.springframework.context.ApplicationContext;
import
    org.springframework.context.support.ClassPathXmlApplicationContext;

public class CalculadoraDeSalariosFactory {

    private ApplicationContext context =
        new ClassPathXmlApplicationContext("dependencias.xml");

    public CalculadoraDeSalarios criaCalculadora(){
        CalculadoraDeSalarios calculadora =
            (CalculadoraDeSalarios) context.getBean("calculadora");

        return calculadora;
    }
}
```

Não veremos aqui como configurar um ambiente para executar o Spring, mas na documentação do framework é possível ver os jars necessários para sua utilização. O que nos importa é a instanciação do seu contexto e a obtenção de um objeto de dentro dele através do método `getBean`. Esse contexto foi criado tendo como base o arquivo `dependencias.xml`, que geralmente é deixado na raiz do classpath da aplicação, assim como fizemos com o arquivo `properties`.

Pode não parecer, mas nossa factory está pronta. Não precisamos instanciar a `TabelaDeReferenciaSalarial` e colocá-la dentro da calculadora, pois o Spring faz isso por nós. Para tal, precisamos deixar o arquivo `dependencias.xml` como o trecho a seguir.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">
```

```
<bean id="calculadora"
      class="br.com.casadocodigo...CalculadoraDeSalariosPlano2002">
  <property name="tabelaDeReferenciaSalarial" >
    <ref bean="tabela"/>
  </property>
</bean>

<bean id="tabela"
      class="br.com.casadocodigo...TabelaDeReferenciaSalarialPadrao">
</bean>
</beans>
```

Podemos perceber que dentro do XML fazemos a referência do bean com id "tabela" dentro da propriedade `tabelaDeReferenciaSalarial` do bean "calculadora". Com isso, ao pedirmos a calculadora, o Spring instancia também a tabela e através do método `setTabelaDeReferenciaSalarial` vincula os dois objetos.

Um framework como esse facilita e muito quando temos uma estrutura mais complexa, na qual objetos dependem de vários outros objetos, e suas dependências possuem dependências. Fazer uma factory na mão, como vimos antes através de arquivos properties, ficaria extremamente complexo e suscetível a erros. Para automatizar um pouco mais o trabalho teríamos que usar reflexão, e ainda assim não chegaríamos ao nível de refinamento que um framework específico para esse fim pode nos proporcionar.

Obviamente nesse exemplo mostramos algo muito simples. Assim como Seam e CDI, o Spring tem muito mais que isso. Os beans possuem escopos, podem ter interceptadores e método dinâmicos de resolução de dependência. Isso tudo porque a partir do momento em que chamamos o método `getBean`, entramos no contexto do Spring, e dentro desse contexto é possível fazer muita coisa, como integração com módulos Java EE ou outros frameworks. Mas obviamente não iremos entrar nesses detalhes, pois seria assunto para um livro específico.

O que precisamos ter em mente é que frameworks modernos quase sempre nos oferecem, cada um à sua maneira, as mesmas funcionalidades. A proposta dessa seção é apenas perceber, do ponto de vista histórico, como era o "jeitão" Spring de resolver dependências.

1.6 BUSCANDO AS DEPENDÊNCIAS COM SEAM E ANOTAÇÕES

Com o Java 5 e a inclusão de anotações, as coisas ficaram mais fáceis para o desenvolvedor. Novos frameworks surgiram com uma filosofia diferente, a da configuração simplificada. Enquanto antes a quantidade excessiva de configuração permitia uma extrema flexibilidade à custa da simplicidade, a nova geração de frameworks permitia fazer muito mais configurando apenas o essencial ou a exceção. O JBoss Seam foi um framework que nasceu com essa filosofia. Apesar de possuir alguns passos de configuração, por já ter nascido na era das anotações, ele já era bem mais simples de se utilizar.

Outro diferencial do JBoss Seam foi seu foco inicial na integração, possível graças ao período em que ele surgiu, pois nesse momento o Java EE como um todo já tinha evoluído muito, principalmente através do EJB 3 e JPA. Isso permitiu um desenvolvimento puro em Java EE tão, ou mais simples do que fazer utilizando Spring. Dessa forma o Seam não nasceu focado em oferecer uma *stack* alternativa ao Java EE como fez o Spring, mas sim em integrar todas as tecnologias já existentes de uma forma simples e elegante, coisa que ainda não existia no Java Enterprise.

Assim como no exemplo utilizando Spring, não entraremos nas integrações do Seam com o Java EE, nem como ele auxilia a tratar transação ou qualquer outro assunto comum ao Seam, Spring ou CDI. O foco é apenas analisarmos como Seam trata a questão da resolução de dependências.

Apesar de ser possível configurar utilizando XML, o Seam já nasceu com as anotações, então utilizá-las é o mais comum quando usamos esse framework. Dessa forma, assim como fizemos no XML do Spring (ou faríamos no XML do Seam), vamos dar nomes aos beans usando a anotação `@org.jboss.seam.annotations.Name`. Além disso, vamos usar também a anotação `@org.jboss.seam.annotations.AutoCreate` para que o bean seja automaticamente criado quando for demandado. Sem isso, o Seam usa o bean disponível no escopo ou devolve `null`. Porém como não estamos usando escopos, para simplificar o exemplo, precisamos de mais essa anotação.

```
@Name("calculadoraDeSalarios") @AutoCreate
public class CalculadoraDeSalariosPlano2002
    implements CalculadoraDeSalarios{
    ...
}

@Name("tabelaDeReferenciaSalarial") @AutoCreate
```

```
public class TabelaDeReferenciaSalarialPadrao
    implements TabelaDeReferenciaSalarial{
    ...
}
```

Agora que os beans já possuem nomes, precisamos referenciar a `TabelaDeReferenciaSalarial` dentro da `CalculadoraDeSalarios`. Para isso, podemos anotar tanto uma propriedade quanto seu método *setter* com `@org.jboss.seam.annotations.In`. Então, alterando mais uma vez nossa calculadora teremos o seguinte código.

```
@Name("calculadoraDeSalarios") @AutoCreate
public class CalculadoraDeSalariosPlano2002
    implements CalculadoraDeSalarios{

    @In
    public void setTabelaDeReferenciaSalarial(
        TabelaDeReferenciaSalarial t){
        ...
    }
    ...
}
```

Apesar de ficar bastante parecido com o código CDI que veremos no decorrer deste livro, o Seam tem uma séria limitação. A resolução da dependência é feita através do nome do componente, e não pelo tipo. Apesar de não termos especificado nenhum nome na anotação `@In`, o Seam considera o nome do componente a ser injetado como igual ao nome da propriedade anotada. Como anotamos um *setter*, o prefixo “set” é removido e a primeira letra é passada para minúsculo para obter o nome da propriedade correspondente. Então, na prática, é como se tivéssemos colocado o seguinte código `@In("tabelaDeReferenciaSalarial")`.

Agora que a configuração dos beans está feita, precisamos voltar à nossa *factory* e pedir a `CalculadoraDeSalarios` ao contexto do Seam.

```
import org.jboss.seam.Component;

public class CalculadoraDeSalariosFactory {

    public CalculadoraDeSalarios criaCalculadora(){
        CalculadoraDeSalarios calculadora = (CalculadoraDeSalarios)
```

```
        Component.getInstance("calculadoraDeSalarios");  
  
        return calculadora;  
    }  
}
```

Apesar de usarmos menos classes, não é tão simples rodar o Seam dentro de uma main como conseguimos com Spring. No entanto, quero que você se foque na qualidade do código e simplicidade que teve o nosso programa, analisando como se deu a evolução desde o arquivo properties, passando por XML, e chegando à anotação. Contudo, é importante sempre lembrarmos que o Seam usa as anotações para configurar seu contexto, mas a ligação entre os beans é feita por nomes, e não por tipos como é na CDI.

1.7 DE ONDE VIMOS?

O objetivo deste capítulo foi introduzir questões básicas de OO, que são independentes do uso de frameworks, mas que ao serem desprezadas pode não fazer sentido a utilização de um. Depois passamos por um breve histórico com a evolução da resolução de dependências, no qual em momento algum tivemos objetivo de comparar os frameworks, já que cada um pode ser utilizado de diversas maneiras e não necessariamente pararam no ponto em que foram apresentados aqui.

Assim como podemos tirar proveito de um carro moderno sem nos lembrarmos da evolução dos carros carburados, para a injeção eletrônica, e depois os aceleradores eletrônicos, também podemos tirar proveito da CDI sem essa contextualização histórica. Mas quando temos esse conhecimento, entendemos melhor porque algumas coisas são como são, e temos um pouco mais de poder argumentativo.

CAPÍTULO 2

O que é e para que serve a CDI

2.1 PARA ONDE VAMOS?

Neste capítulo veremos o desafio que é deixar simples de manter um sistema com uma regra complexa. Tal desafio é o dia a dia de muitos desenvolvedores, e provavelmente em algum momento passaremos por ele, afinal, por mais que muitos sistemas tenham várias e várias telas de cadastro, sempre que temos que lidar com a parte central do sistema, onde fica sua inteligência de negócio, passaremos pelos problemas aqui apresentados.

2.2 EVITANDO QUE A COMPLEXIDADE DO CÓDIGO SEMPRE AUMENTE

Para iniciar nossos estudos, vamos tomar como exemplo um sistema que calcula o imposto de renda dos funcionários de uma empresa para informar à Receita Federal.

Porém, como nada é simples, antes precisamos calcular o salário de cada fun-

cionário, e para isso será preciso analisar critérios como: cargo ou função exercida dentro da empresa, escolaridade, qualificações, e por fim, um critério não muito utilizado mas que serve para enriquecer nosso exemplo, o tempo de empresa.

Facilmente percebemos que esse é um cenário de complexidade razoável, então iremos por partes. Para começarmos, precisaremos coletar informações básicas, como o salário base do cargo ou função, a escolaridade do funcionário, e a escolaridade esperada para o cargo.

```
public double calculaImpostoDeRenda(Funcionario funcionario){
    double salario = funcionario.getCargo().getSalarioBase();
    Escolaridade escolaridadeFuncionario =
        funcionario.getEscolaridade();
    Escolaridade escolaridadeCargo =
        funcionario.getCargo().getEscolaridadeDesejada();

    ...
}
```

A escolaridade do funcionário e a escolaridade do cargo servem para verificarmos se ele tem os requisitos esperados para a função. Isso ocorre mais frequentemente quando tempos reformulações no plano de carreira de uma empresa. Se, por exemplo, um cargo exigia nível superior e depois passa a exigir pós-graduação, as pessoas com nível superior não perderão a função, mas também não receberão o mesmo salário dos novos funcionários que deverão ser contratados seguindo as novas exigências e com salários um pouco mais altos.

No nosso exemplo, estamos considerando uma variação de 20% (vinte por cento) tanto para quem está abaixo do mínimo esperado para o cargo, quanto para quem tem escolaridade acima do esperado, como um mestrado por exemplo. O cálculo dessa diferença poderia ser feito assim:

```
//se o funcionário tem escolaridade inferior à esperada para o cargo
if(escolaridadeFuncionario.compareTo(escolaridadeCargo) < 0){
    salario = salario * 0.8;
}
//se o funcionário tem escolaridade superior à esperada para o cargo
else if(escolaridadeFuncionario.compareTo(escolaridadeCargo) > 0){
    salario = salario * 1.2;
}
```

Agora iremos calcular uma gratificação por tempo de serviço na empresa. Usamos uma fórmula simples, na qual, para cada ano trabalhado daremos um aumento

de 1% (um por cento), e em caso de permanência de pelo menos 5 anos na empresa, haverá um aumento extra de 10% (dez por cento).

```
int anoAtual = getAnoAtual();
int anoAdmissao = funcionario.getAnoAdmissao();

//dá 1% de aumento para cada ano trabalhado na empresa
double anosTrabalhados = anoAtual - anoAdmissao;
double aumentoAntiguidade = anosTrabalhados / 100;
salario = salario * (1 + aumentoAntiguidade);

//se tem mais de 5 anos na empresa tem aumento de 10%
if(anosTrabalhados > 5){
    salario = salario * 1.1;
}
```

Na maioria das vezes vale muito mais um código legível do que um número menor de linhas. Então, não se preocupe demais em fazer contas em uma única linha comprometendo a legibilidade e a manutenibilidade do seu código. Crie variáveis que expliquem o que está sendo feito, e se algo ficar muito grande, separe em métodos.

Agora que já temos o salário bruto calculado, precisamos descontar o imposto de renda. No mundo real teríamos mais descontos, mas como estamos criando nosso próprio universo de exemplo, vamos deixar o `funcionario` ser mais feliz e ter menos descontos no seu suado salário.

No exemplo a seguir, utilizaremos a tabela de IR de 2013, mas os dados não influenciam muito — o mais importante é percebermos o quanto nosso código vai ficando complexo, e depois veremos como resolver isso.

```
...
//tabela de IR de 2013
double aliquota = 0.0;
double parcelaDeduzir = 0.0;

//ifs estão de forma mais didática, na prática poderiam ser reduzidos
if(salario <= 1710.78){
    aliquota = 0.0;
    parcelaDeduzir = 0.0;
}
else if(salario > 1710.78 && salario <= 2563.91){
    aliquota = 7.5/100;
```

```
        parcelaDeduzir = 128.31;
    }
    else if(salario > 2563.91 && salario <= 3418.59){
        aliquota = 15.0/100;
        parcelaDeduzir = 320.60;
    }
    else if(salario > 3418.59 && salario <= 4271.59){
        aliquota = 22.5/100;
        parcelaDeduzir = 577.0;
    }
    else if(salario > 4271.59){
        aliquota = 27.5/100;
        parcelaDeduzir = 790.58;
    }

    double impostoSemDesconto = salario * aliquota;

    return impostoSemDesconto - parcelaDeduzir;
} //final do método calculaImpostoDeRenda
```

O cálculo de imposto de renda não é o mais simples do mundo, mas também não é tão complicado. Se, por exemplo, tivermos um salário de até R\$ 1.710,78, estamos isentos do imposto, mas se tivermos um salário acima de R\$ 4.271,59 estaremos na faixa de 27,5% do imposto.

Mas alguém que ganha R\$ 10.000,00 não vai pagar R\$ 2.750,00 de IR, que seria o valor guardado na variável `impostoSemDesconto`; pois o desconto de 27,5% só incide na parcela do salário acima dos R\$ 4.271,59. Por esse motivo temos a `parcelaDeduzir`, que é o valor que temos que descontar desses R\$ 2.750,00 para chegar no imposto correto.

Outra forma de chegar no imposto seria dividir o salário em cada faixa e calcular o imposto proporcional de cada uma delas e depois somar tudo. Mas na prática é para isso que a “parcela a deduzir” faz, já é o cálculo pronto dessa diferença.

Juntando tudo, ao final teríamos o seguinte código:

```
public double calculaImpostoDeRenda(Funcionario funcionario){
    double salario = funcionario.getCargo().getSalarioBase();
    Escolaridade escolaridadeFuncionario =
        funcionario.getEscolaridade();
    Escolaridade escolaridadeCargo =
        funcionario.getCargo().getEscolaridadeDesejada();
```

```
// se o funcionário tem escolaridade inferior
// à esperada para o cargo
if(escolaridadeFuncionario.compareTo(escolaridadeCargo) < 0){
    salario = salario * 0.8;
}

// se o funcionário tem escolaridade superior
// à esperada para o cargo
else if(escolaridadeFuncionario.compareTo(escolaridadeCargo) > 0) {
    salario = salario * 1.2;
}

int anoAtual = getAnoAtual();
int anoAdmissao = funcionario.getAnoAdmissao();

//dá 1% de aumento para cada ano trabalhado na empresa
double anosTrabalhados = anoAtual - anoAdmissao;
double aumentoAntiguidade = anosTrabalhados / 100;
salario = salario * (1 + aumentoAntiguidade);

//se tem mais de 5 anos na empresa tem aumento de 10%
if(anosTrabalhados > 5){
    salario = salario * 1.1;
}

//tabela de IR de 2013
double aliquota = 0.0;
double parcelaDeduzir = 0.0;

// ifs estão de forma mais didática,
// na prática poderiam ser reduzidos
if(salario <= 1710.78){
    aliquota = 0.0;
    parcelaDeduzir = 0.0;
}
else if(salario > 1710.78 && salario <= 2563.91){
    aliquota = 7.5/100;
    parcelaDeduzir = 128.31;
}
else if(salario > 2563.91 && salario <= 3418.59){
```

```
        aliquota = 15.0/100;
        parcelaDeduzir = 320.60;
    }
    else if(salario > 3418.59 && salario <= 4271.59){
        aliquota = 22.5/100;
        parcelaDeduzir = 577.0;
    }
    else if(salario > 4271.59){
        aliquota = 27.5/100;
        parcelaDeduzir = 790.58;
    }

    double impostoSemDesconto = salario * aliquota;

    return impostoSemDesconto - parcelaDeduzir;
}
```

Em um cenário real, ao calcular o salário ainda poderíamos ter mais variáveis, como gratificações, incentivos ou um diferencial por um conhecimento específico, mas nos limitar ao exemplo apresentado. Você deve ter se sentido desanimado em ler esse código, pela sua extensão, diversos casos possíveis e complexidade. Imagine se fosse uma aplicação real.

Podemos perceber que o método `calculaImpostoDeRenda` sabe muita coisa, e quanto mais responsabilidades as classes possuem, mais difícil é mantê-las.

Agora consideremos que para um método desse tipo estar em produção, ele precisa estar testado. E mais do que isso, ele precisa de testes automatizados, pois testes manuais além de trabalhosos tendem a não ser tão metódicos e reproduzíveis. Mas como testar um método *sabichão*? Acredito que esse seja um bom candidato para uma “queima de arquivo”, pois ele sabe demais.

Mesmo antes de pensarmos em testar, precisamos organizar esse cálculo. Uma ótima ideia seria separar a calculadora de salários da calculadora de IR. Olhando o método completo, que é enorme, percebemos duas partes bem distintas dentro dele, claramente duas operações diferentes mas que estavam juntas no mesmo método: o cálculo do salário bruto, e o cálculo do IR.

Vamos então criar uma classe que tem como responsabilidade calcular só o salário bruto, assim caso essa regra mude, alteramos essa classe e não corremos o risco de danificar o cálculo do IR, pois eles ficarão independentes a partir de agora.

O código a seguir nós já conhecemos, só vamos separá-lo em uma classe:

```
public class CalculadoraDeSalarios {
    public double calculaSalario(Funcionario funcionario){
        double salario = funcionario.getCargo().getSalarioBase();
        Escolaridade escolaridadeFuncionario =
            funcionario.getEscolaridade();
        Escolaridade escolaridadeCargo =
            funcionario.getCargo().getEscolaridadeDesejada();

        // se o funcionário tem escolaridade inferior
        // à esperada para o cargo
        if(escolaridadeFuncionario.compareTo(escolaridadeCargo) < 0){
            salario = salario * 0.8;
        }
        // se o funcionário tem escolaridade superior
        // à esperada para o cargo
        else if(
            escolaridadeFuncionario.compareTo(escolaridadeCargo) > 0) {
            salario = salario * 1.2;
        }

        int anoAtual = getAnoAtual();
        int anoAdmissao = funcionario.getAnoAdmissao();

        //dá 1% de aumento para cada ano trabalhado na empresa
        double anosTrabalhados = anoAtual - anoAdmissao;
        double aumentoAntiguidade = anosTrabalhados / 100;
        salario = salario * (1 + aumentoAntiguidade);

        //se tem mais de 5 anos na empresa tem aumento de 10%
        if(anosTrabalhados > 5){
            salario = salario * 1.1;
        }

        return salario;
    }
}
```

Agora temos uma classe que só sabe calcular salários, e que ainda poderia ser melhorada, separando cada critério como escolaridade ou qualificação, funções ou cargos, e antiguidades em subcalculadoras. Estas encapsulariam cada lógica de cálculo específica e no final a calculadora geral combinaria os resultados. Mas faremos

isso em momento oportuno, o principal agora é vermos onde a CDI vai nos ajudar.

Com a separação, nossa calculadora de IR ficou bem mais simples:

```
public class CalculadoraDeImpostos {
    public double calculaImpostoDeRenda(Funcionario funcionario){
        CalculadoraDeSalarios calculadora = new CalculadoraDeSalarios();
        double salario = calculadora.calculaSalario(funcionario);

        //tabela de IR de 2013
        double aliquota = 0.0;
        double parcelaDeduzir = 0.0;

        // ifs estão de forma mais didática,
        // na prática poderiam ser reduzidos
        if(salario <= 1710.78){
            aliquota = 0.0;
            parcelaDeduzir = 0.0;
        }
        else if(salario > 1710.78 && salario <= 2563.91){
            aliquota = 7.5/100;
            parcelaDeduzir = 128.31;
        }
        else if(salario > 2563.91 && salario <= 3418.59){
            aliquota = 15.0/100;
            parcelaDeduzir = 320.60;
        }
        else if(salario > 3418.59 && salario <= 4271.59){
            aliquota = 22.5/100;
            parcelaDeduzir = 577.0;
        }
        else if(salario > 4271.59){
            aliquota = 27.5/100;
            parcelaDeduzir = 790.58;
        }

        double impostoSemDesconto = salario * aliquota;

        return impostoSemDesconto - parcelaDeduzir;
    }
}
```

REFATORAÇÃO E TESTABILIDADE

Nas aplicações que já temos, separar essas responsabilidades em classes distintas nem sempre é tão simples. Pode ocorrer de em uma aplicação similar o cálculo do salário e dos descontos irem ocorrendo de forma entrelaçada, uma vez parte de um, outra parte de outro.

Nesse caso pode ser interessante inverter a ordem que usamos aqui, e caso não existam, primeiro criar os testes que garantam o funcionamento do método que queremos refatorar. Depois de testado, aí sim faríamos uma refatoração completa, separando as responsabilidades, e então usaríamos os mesmos testes para garantir que a refatoração não danificou o funcionamento.

Somente depois que os testes estivessem todos verdes (executados com sucesso) é que aproveitaríamos os métodos de negócio mais bem escritos para melhorar também o código dos nossos testes.

ENTENDENDO MAIS DE TESTES AUTOMATIZADOS

Teste de código é uma matéria ampla, tanto é que temos excelentes livros sobre o assunto [?]. Assim sendo, o objetivo aqui não é cobrir totalmente esse assunto, e sim falar um pouco sobre como isso é importante no desenvolvimento de software.

A CDI não amarra nosso projeto de classes, nos dando liberdade para um projeto bem testável. No entanto, em alguns momentos precisamos mais do que testes de unidade, como testes de integração e de sistema. Quando passamos para os testes de integração, ferramentas como CDI costumam dar trabalho, mas no capítulo 9 veremos como isso pode ser feito sem muito trabalho.

Agora sim podemos partir para os testes. Vamos considerar um funcionário com salário de R\$ 3.000,00. Teremos que verificar se o imposto dele foi de R\$ 129,40, pois ele caiu na alíquota de 15% e sua parcela a deduzir é de R\$ 320,60. Não parece muito complicado, não é verdade? Seria algo parecido com o seguinte código.

```
public void testaCalculoIRNaSegundaFaixaDeContribuicao(){
    Funcionario funcionario = new Funcionario();
    funcionario.setSalario(3000.0); // esse método não existe

    CalculadoraDeImpostos calculadora =
        new CalculadoraDeImpostos();

    //a calculadora de IR usa outra classe para calcular o salário
    double impostoDeRenda =
        calculadora.calculaImpostoDeRenda(funcionario);

    Assert.assertEquals(129.4, impostoDeRenda, 0.0001);
}
```

O detalhe é que esse salário de R\$ 3.000,00 não pode ser atribuído assim, ele é calculado pela `CalculadoraDeSalarios` dentro do cálculo do IR. Isso quer dizer que temos dois cálculos acontecendo dentro do mesmo teste, e isso pode causar confusão.

No nosso caso, teríamos que configurar o `funcionario` com dados que saberíamos que ao final do cálculo do salário bruto, teríamos R\$ 3.000,00 como resultado. Mas pode ocorrer de configurarmos o funcionário corretamente e a `CalculadoraDeSalarios` de falhar, devolvendo um valor errado. Então, estaríamos testando o imposto achando que o salário bruto era `3000.0` mas na prática o valor seria outro, e o teste falharia.

O pior nesse exemplo é que o teste que estamos escrevendo chama-se `testaCalculoIRNaSegundaFaixaDeContribuicao`, então se ele falhar, a primeira coisa que virá às nossas mentes é que o cálculo do IR está com problemas, e não que o erro foi no cálculo do salário. Mas como lidar com esses casos?

Poderíamos nesse ponto ser um pouco impulsivos e sair alterando a assinatura do método que calcula o imposto de renda para este receber o salário já calculado, apenas um `double`, e então eliminaríamos a dependência com a `CalculadoraDeSalarios` nos nossos testes.

Essa pode ser, sim, uma das soluções, mas veremos que temos outras mais aplicáveis a qualquer modelo. Além do mais, se formos tentar mudar as assinaturas dos nossos métodos para receber dados mais independentes, já processados, em pouco tempo estaremos trabalhando somente com tipos simples de dados (primitivos e *wrappers*), e acabaremos com a OO, que tem como base a colaboração entre objetos.

A solução mais adequada para promover a testabilidade do nosso código sem

ferir a OO é o uso da **Injeção de Dependências**, ou *Dependency Injection (DI)* em inglês.

2.3 INTRODUÇÃO À INJEÇÃO DE DEPENDÊNCIAS

A injeção de dependências é muito importante para termos um sistema mais simples de manter. Apesar de parecer que sua principal vantagem é ajudar nos testes, isso na verdade é só uma forma de percebermos sua real vantagem, que é a facilidade em trocar peças do nosso sistema diminuindo muito a possibilidade de quebra. Vamos voltar ao nosso exemplo para entendermos melhor como isso funciona.

```
public class CalculadoraDeImpostos {  
  
    private CalculadoraDeSalarios calculadoraSalarios;  
  
    public CalculadoraDeImpostos(CalculadoraDeSalarios calculadora){  
        calculadoraSalarios = calculadora;  
    }  
  
    public double calculaImpostoDeRenda(Funcionario funcionario){  
        double salario = calculadoraSalarios.calculaSalario(funcionario);  
  
        //resto do método continua igual  
    }  
}
```

A principal mudança no desenho das classes é o que fizemos na `CalculadoraDeImpostos`. Na versão anterior de seu código, essa classe instanciava diretamente a classe `CalculadoraDeSalarios`, e assim não conseguíamos de uma forma simples interferir na relação entre uma classe e outra. Agora, apesar de sutil, a mudança é poderosa; pois a classe que antes ativamente resolvia a dependência agora espera pacientemente que alguém lhe supra essa dependência. E quando esse alguém (nós mesmos ou um *framework*) passa essa dependência para a classe que necessita, dizemos que ele está injetando a dependência em quem precisa.

Nesse primeiro momento nós mesmos estaremos passando para a `CalculadoraDeImpostos` a instância de `CalculadoraDeSalarios` que ela necessita, ou seja, a injeção de dependência será manual, mas é aqui que a CDI atuará, suprimindo “automaticamente” essas dependências. Veremos mais à frente

que, apesar das aspas, esse processo é tão simples que poderemos, sim, na grande maioria dos casos, considerá-lo automático.

Já fizemos as alterações na classe que possui a dependência, agora falta alterar nosso teste para injetar essa dependência da forma correta.

```
@Test
public void testaCalculoIRNaSegundaFaixaDeContribuicao(){
    Funcionario funcionario = new Funcionario();
    funcionario.setSalario(3000.0); //esse método não existe

    CalculadoraDeSalarios calculadoraSalarios =
        new CalculadoraDeSalarios();
    CalculadoraDeImpostos calculadoraImpostos =
        new CalculadoraDeImpostos(calculadoraSalarios);

    //a calculadora de IR usa outra classe para calcular o salário
    double imposto =
        calculadoraImpostos.calculaImpostoDeRenda(funcionario);

    Assert.assertEquals(129.4, imposto, 0.0001);
}
```

Esse passo foi apenas didático, pois não elimina nosso problema, apenas mostra como devemos montar a instância de `CalculadoraDeImpostos`. Para obtermos uma vantagem de verdade, precisaremos trocar a calculadora de salários. Criaremos uma nova classe para substituir a `CalculadoraDeSalarios`, e essa nova classe vai servir somente para nossos testes. Em vez de calcular de verdade, ela vai devolver o valor que especificarmos, assim não teremos mais a dependência entre as duas calculadoras dentro do mesmo teste.

```
public class CalculadoraDeSalariosMock extends CalculadoraDeSalarios {

    private double salarioFixo;

    public CalculadoraDeSalariosMock(double salarioFixo){
        this.salarioFixo = salarioFixo;
    }
    public double calculaSalario(Funcionario funcionario){
        return salarioFixo;
    }
}
```

Na classe `CalculadoraDeSalariosMock` nós estendemos a classe `CalculadoraDeSalarios` e sobrescrevemos o método `calculaSalario` para que ele funcione da forma que desejamos. É um serviço grande para um resultado simples, mas não se preocupe, existem frameworks que fazem isso sem a necessidade de criarmos uma nova classe, um *Mock*.

CRIAÇÃO SIMPLIFICADA DE MOCKS COM FRAMEWORKS

Mocks são objetos “burros”, programados para responder exatamente como precisarmos em cada situação. No nosso exemplo mandamos ele simplesmente retornar um valor, independentemente do que o método `calculaSalario` semanticamente representa.

Existem frameworks que facilitam bastante o trabalho como o Mockito, que será usado nos códigos de exemplo a partir do capítulo 10.

Apesar do trabalho a mais, fazer essa classe nos mostra que se tivéssemos definido a `CalculadoraDeSalarios` como uma interface, e não como uma classe, nosso *Mock* ficaria semanticamente melhor. O mock em si não é um problema pois no futuro vamos usar um framework para criá-lo sob demanda, mas caso precisemos em um determinado momento alterar a forma de calcular os salários, por exemplo depois de um novo plano de cargos e salários, poderemos cair no mesmo problema que vemos no exemplo.

Princípio da substituição de Liskov e o uso de interfaces

Quando precisamos trocar um objeto por outro, como no caso trocamos nossa calculadora real pelo mock, e no caso de um novo plano de cargos, aplicamos o Princípio da substituição de Liskov, segundo o qual o objeto mock consegue perfeitamente se passar pelo objeto real.

Perceba porém que para fazermos isso funcionar foi preciso que o mock seja subclasse da `CalculadoraDeSalarios`, o que do ponto de vista semântico não tem sentido algum. Novamente imaginemos isso aplicado no caso da criação de um novo plano de cargos, em que a implementação original será desprezada, mas para que conseguíssemos fazer a nova calculadora substituir a primeira, precisaríamos que esta fosse subclasse da primeira. Caso usássemos uma interface não teríamos esse problema.

Após entendermos um pouco mais sobre como um objeto substitui outro, vamos aplicar isso no nosso teste.

```
@Test
public void testaCalculoIRNaSegundaFaixaDeContribuicao(){
    Funcionario funcionario = new Funcionario();

    CalculadoraDeSalarios calculadoraSalarios =
        new CalculadoraDeSalariosMock(3000.0);

    CalculadoraDeImpostos calculadoraImpostos =
        new CalculadoraDeImpostos(calculadoraSalarios);

    //a calculadora de IR usa outra classe para calcular o salário
    double imposto =
        calculadoraImpostos.calculaImpostoDeRenda(funcionario);

    Assert.assertEquals(129.4, imposto, 0.0001);
}
```

Agora sim temos um código funcionando. Criamos manualmente um mock utilizando o princípio da substituição e injetamos esse mock como dependência da `CalculadoraDeImpostos`. Nada mal para um início. Acredito que foi possível perceber o quanto algo simples, como a injeção de dependências, pode fazer uma diferença na modularidade do nosso sistema, trocando uma classe por outra sem que as classes clientes percebam.

PRINCÍPIO DE LISKOV

Esse princípio descreve o que é necessário para fazer um objeto ser aceito no lugar de outro. Basicamente o novo objeto tem que ser um sub-tipo, ou em Java implementar a mesma interface que o primeiro. Além disso, os métodos da subclasse têm que aceitar **no mínimo** tudo que o método da superclasse aceita, e devolver **no máximo** o que o primeiro devolvia.

Como exemplo, podemos considerar um método que receba e devolva números. Se a classe original aceita apenas números positivos, a nova classe tem que no mínimo aceitar todos os números que a primeira aceitava, mas se ela aceitar também os números negativos não tem problema. Nesse caso, provavelmente num primeiro momento a opção de enviar números negativos não será usada pelo sistema, visto que ele já está condicionado à primeira classe que só aceita números positivos. Já se a nova classe aceitar menos que a original, por exemplo somente números maiores que mil, o sistema pode passar para essa nova classe o número 50, pois com a classe original era possível, mas na nova isso geraria um erro em tempo de execução.

A respeito do retorno, o princípio é o inverso: se a classe original retorna somente números positivos, podemos no máximo devolver os números positivos na nova classe. Isso porque o sistema já vai estar condicionado a esperar apenas números positivos, então ao retornar um número negativo poderíamos ter um erro em tempo de execução. Agora se retornarmos uma quantidade ainda mais restrita de resultados não teremos problema, como retornar apenas valores maiores que mil.

É possível percebermos que quanto à tipagem, as linguagens de programação dão conta do recado, já as restrições de entrada e saída dos métodos na maioria das vezes precisam ser tratadas pelo desenvolvedor, e por isso podemos considerar que a interface de uma classe não é só a assinatura de seus métodos, mas também as restrições que, quando existirem, devem estar especificadas via Javadoc ou outra forma efetiva de documentação.

2.4 A MOTIVAÇÃO DA CDI

Nosso exemplo até aqui foi simples, mas conforme ele for aumentando sentiremos cada vez mais a necessidade de algum gerenciador dessas dependências.

O mais importante nesse capítulo, é percebermos que conforme o sistema cresce, e vamos nos preocupando e mantê-lo organizado e testado, vai surgindo a necessidade de um auxílio.

Precisamos de uma ferramenta que nos dê a possibilidade de montar nossos objetos a partir de outros, como um “lego”. Porém essa ferramenta preferencialmente precisa nos livrar de configurações morosas, cheias de XML; e também de configurações simples, com anotações, mas que não conseguem analisar os tipos dos objetos, como vimos na seção 1.3. E é aqui que entra a CDI.

A CDI é uma especificação Java, cujo nome completo é “Context and Dependency Injection for Java EE” [?], mas como podemos perceber o “for Java EE” não entra na sigla. De fato, conforme formos nos aprofundando no assunto, vamos perceber que a CDI não é apenas para o Java EE, mas também para o Java SE.

Se você está iniciando no universo Java, pode ser que não esteja habituado com o termo “especificação”, mas fique tranquilo, trataremos disso logo a seguir. E para nos ajudar a lembrar sempre que estamos falando de uma especificação, procurarei sempre me referir a CDI no gênero feminino. Pois não falamos de um framework, e sim de uma especificação.

2.5 O QUE É E PARA QUE SERVE UMA ESPECIFICAÇÃO?

Apesar do convencimento da real necessidade de algo só vir quando nos deparamos com casos mais complexos, com os exemplos apresentados percebemos que há vantagem no uso da injeção de dependência, e no decorrer do livro teremos certeza disso. Tanto que um indicador da importância de uma matéria é a quantidade de pessoas tentando resolver seus problemas.

Quando o assunto é injeção de dependência, temos no ecossistema Java algumas opções de frameworks, sendo os mais conhecidos e modernos o Spring Framework e o JBoss Seam, já apresentados na seção 1.3.

Apesar de serem duas boas ferramentas, Spring e Seam são frameworks diferentes, e adotando qualquer um dos dois a mudança para outro framework que faça a mesma tarefa é muito custosa, não valendo a pena. Dessa forma caso o framework que estivermos utilizando comece a apresentar um nível elevado de bugs ou tenha

seu desenvolvimento descontinuado, nossa aplicação corre sérios riscos de ficar desatualizada ou ter falhas graves.

Por esse motivo surgem as especificações. Elas definem como uma determinada funcionalidade deve se comportar e os implementadores (empresas, fundações, grupos de usuários, indivíduos etc) têm liberdade de implementar da maneira que desejarem, desde que a especificação seja seguida.

Na prática, ela é um documento texto que especifica cada funcionamento esperado do software que a implemente. Com isso temos a facilidade de escolher a implementação que mais nos agrada, e caso a mesma seja descontinuada ou sua qualidade não nos satisfaça mais, podemos trocar de implementação com pouco esforço, visto que via de regra o comportamento de todas elas deve ser igual ou bem parecido. E para garantir que esse funcionamento seja padrão, existe uma bateria de testes que são aplicados a uma implementação que queira se certificar como aderente à especificação.

A CDI — Context and Dependency Injection for Java EE — é a especificação que rege como deve funcionar os frameworks de injeção de dependências para a plataforma Java. Dessa forma, agora temos uma maior liberdade de escolha na implementação, e mesmo frameworks que não a implementam acabam tendo um comportamento parecido com o da especificação, para que seja possível reutilizar o conhecimento e até mesmo código vindo de aplicações feitas com implementações de CDI.

Quando estivermos estudando especificamente a CDI, veremos que ela trouxe uma inovação em relação às outras especificações. É muito comum uma implementação trazer funcionalidades a mais do que a especificação determina, pois a velocidade da demanda da comunidade é superior à do comitê que evolui as especificações. Isso é muito bom pois temos novidades em uma velocidade maior, mas quando utilizamos essas funcionalidades não especificadas, acabamos nos prendendo a uma implementação específica, pois uma outra implementação pode não ter tal funcionalidade ou tê-la de forma diferente.

Por ser isso comum, a CDI já especificou como criar extensões portáveis entre implementações, com isso se estivermos utilizando o Weld (da JBoss), implementação de referência, e quisermos mudar para a OpenWebBean (da Apache), poderemos levar para essa segunda, as extensões desenvolvidas pela primeira, desde que tais extensões sejam implementadas seguindo as regras de portabilidade. É realmente algo muito interessante e que faz falta em outras especificações. No capítulo 9 veremos especificamente sobre essas extensões.

2.6 A CDI É SÓ PARA JAVA EE MESMO?

Apesar de seu nome, *Context and Dependency Injection for Java EE*, a CDI é uma especificação que pode ser utilizada em ambiente *Enterprise* ou em ambiente *Standard*, como dentro de um Tomcat, Jetty, ou até mesmo em uma aplicação Desktop. Esse nome não quer dizer onde a CDI roda, mas sim onde ela vem pronta para uso, que no caso é no ambiente Java EE, tanto no profile Full quanto no Web.

Até a versão 5, o Java EE era uma coleção enorme de especificações que, juntas, nos permitem desenvolver os mais variados tipos de aplicação, mas tinha muito mais funcionalidades do que a grande maioria das aplicações costuma usar. Por isso, a partir da versão 6, o Java EE passou a ter diferentes perfis: o Full e o Web. O Full, ou completo, é o equivalente à coleção do Java EE 5 e mais algumas especificações novas, dentre elas a CDI. Já o perfil Web é um subconjunto dessas especificações, contendo apenas as especificações mais úteis para as aplicações mais comuns.

Acontece no entanto de haver aplicações que usam ainda menos recursos do que os disponibilizados no perfil Web do Java EE, como uma aplicação que utilize JSF e JPA [?]. Esse tipo de aplicação precisa apenas de um *Servlet Container*, e apesar de ser uma aplicação web, nesse caso consideramos que ela está sobre a plataforma Java SE. Logo, não podemos confundir aplicações Java SE com aplicações Desktop, mas a CDI pode ser utilizada em ambos.

CAPÍTULO 3

Iniciando um projeto com CDI

Nos capítulos anteriores apenas iniciamos a conversa sobre CDI, que obviamente vai durar todo esse livro. Mas agora, nesses primeiro exemplos, já conseguiremos entender um pouco mais dos motivos que fazem da CDI uma ótima alternativa para utilizarmos nos nossos projetos.

Os exemplos desse livro serão baseados na IDE Netbeans, e no servidor de aplicação GlassFish 4, mas todos os conceitos são independentes de IDE e de Servidor. Utilizaremos também a implementação de referência, Weld, mas toda implementação certificada deve funcionar de forma semelhante.

Para facilitar o passo a passo, você pode clonar esse repositório no Github: <https://github.com/gscordeiro/livro-cdi>.

3.1 OLÁ CDI

Vamos aproveitar o exemplo da `CalculadoraDeImpostos` visto anteriormente e agora vamos colocá-la para funcionar com a CDI. Iniciaremos com uma *Servlet*, que

é uma classe Java que roda no servidor e é alcançável por uma *URL*. Esse primeiro código terá objetivo apenas didático, não vai fazer parte da versão final do sistema.

Para iniciar, faça o download (<https://netbeans.org/downloads>) da versão Java EE ou da versão Full (Tudo) do netbeans, pois assim já virá com o servidor GlassFish 4. Com a IDE aberta, crie uma aplicação Web e lhe atribua um nome. Quando for perguntado, informe que será uma aplicação Java EE 7.

Caso esteja usando Eclipse, você pode adicionar o servidor ao criar um novo projeto, indo em “New > Dynamic Web Project” e dê ao projeto o nome que preferir. Logo após especificar o nome do projeto, você poderá escolher o servidor (*Target runtime*).

Iremos iniciar nosso exemplo com uma Servlet. As IDEs possuem formas específicas para a criação de Servlets, mas nada que fuja muito de um clique com o botão direito, escolher a criação de uma nova Servlet e através de wizard especificar qual padrão de url e quais métodos pretendemos utilizar. No nosso exemplo especificamos respectivamente `/hello-cdi` e o método `doGet`.

```
@WebServlet("/hello-cdi")
public class IniciandoComCDI extends HttpServlet {

    @Inject
    private CalculadoraDeImpostos calculadoraImpostos;

    public IniciandoComCDI() {
        System.out.println("Instanciando a Servlet...");
    }

    @PostConstruct
    public void ok(){
        System.out.println("Servlet pronta.");
    }

    protected void doGet(HttpServletRequest req,
                          HttpServletResponse res)
        throws ServletException, IOException {

        double salarioBase =
            Double.parseDouble(req.getParameter("salario"));

        Funcionario funcionario = new FuncionarioBuilder()
```

```
        .comSalarioBaseDe(salarioBase)
        .build();

System.out.println("Efetuando o cálculo.");

//a calculadora de IR usa outra classe para calcular o salário
double imposto =
    calculadoraImpostos.calculaImpostoDeRenda(funcionario);

res.getOutputStream().print(
    String.format("Salario base: R$ %.2f\n" +
        "Imposto devido: R$ %.2f", salarioBase, imposto));
System.out.println("Fim.");
}

}
```

Não me importei em escrever diretamente no console (saída padrão) da aplicação. Além é claro de escrever na tela o resultado do cálculo.

Optei por uma servlet por esta ser a API mais básica do Java para a Web, dessa forma não precisamos de muito conhecimento acumulado para entender o exemplo.

Perceba que esse código faz uso das mesmas classes utilizadas no `testaCalculoIRNaSegundaFaixaDeContribuicao`, então não se preocupe em entender o sentido do cálculo, e sim a dinâmica de resolução das dependências.

Como já estamos utilizando a `CalculadoraDeImpostos` junto com a `CalculadoraDeSalarios`, sem *mocks*, precisamos criar um funcionário de verdade, e por isso utilizamos a classe `FuncionarioBuilder`, que facilita o processo de criação desse objeto.

```
public class FuncionarioBuilder {

    private int anoAdmissao;
    private Escolaridade escolaridadeFuncionario;
    private Escolaridade escolaridadeCargo;
    private double salarioBase;

    public FuncionarioBuilder() {
        escolaridadeCargo = Escolaridade.SUPERIOR;
        escolaridadeFuncionario = Escolaridade.SUPERIOR;
    }
}
```

```

        anoAdmissao = Calendar.getInstance().get(Calendar.YEAR);
    }

    public FuncionarioBuilder comSalarioBaseDe(double salarioBase){
        this.salarioBase = salarioBase;

        return this;
    }

    public Funcionario build(){
        Cargo cargo = new Cargo(salarioBase, escolaridadeCargo);
        return new Funcionario(cargo,
                                escolaridadeFuncionario,
                                anoAdmissao);
    }
}

```

Esse código é necessário pois a criação do Funcionário envolve a criação de Cargo, e como precisaremos criar um funcionário diversas vezes, como nos testes, a maneira de facilitar a criação dos objetos como queremos é via essa classe, que implementa o padrão *builder*.

Na *CalculadoraDeImpostos*, apenas adicionamos a anotação `@Inject` no construtor para que a CDI saiba que precisa injetar essa dependência para nós. Fora isso, assim como já vimos no código da servlet, utilizamos um método público e sem argumentos anotado com a anotação `@PostConstruct`, que é chamado pela CDI sempre que o objeto está pronto, ou seja, já teve todas suas dependências satisfeitas.

```

import javax.annotation.PostConstruct;
import javax.inject.Inject;

public class CalculadoraDeImpostos {

    private CalculadoraDeSalarios calculadoraSalarios;

    @Inject
    public CalculadoraDeImpostos(CalculadoraDeSalarios calculadora){
        System.out.println("Iniciando Calculadora de impostos...");
        calculadoraSalarios = calculadora;
    }
}

```

```
@PostConstruct
public void init(){
    System.out.println("Calculadora de impostos pronta!");
}

//resto do código permanece igual...

}
```

Também para facilitar a análise da sequência da resolução das dependências é que alteramos um pouco a `CalculadoraDeSalarios` para escrever no console quando ela começa a ser instanciada e quando ela estiver pronta.

```
import javax.annotation.PostConstruct;

public class CalculadoraDeSalarios {

    public CalculadoraDeSalarios() {
        System.out.println("Iniciando Calculadora de salários...");
    }

    @PostConstruct
    public void init(){
        System.out.println("Calculadora de salários pronta!");
    }

    //resto do código permanece igual...

}
```

Ainda existe um detalhe antes de executarmos nosso exemplo. Como criamos uma classe `CalculadoraDeSalariosMock` que estende a `CalculadoraDeSalarios`, quando a `CalculadoraDeImpostos` solicitar a dependência via seu construtor, a CDI ficará em dúvida de qual implementação utilizar para satisfazer a dependência, se a calculadora real ou a mock, pois ambas têm tipos compatíveis com a dependência. Por isso colocaremos a anotação `@Vetoed` na classe mock, indicando à CDI que essa classe não deve ser considerada uma candidata à resolução de dependências.

```
import javax.enterprise.inject.Vetoed;
```



```
@Vetoed
public class CalculadoraDeSalariosMock extends CalculadoraDeSalarios {
    //conteúdo permanece igual
}
```

Caso esteja utilizando a versão 1.0 da CDI, em que esta anotação não está disponível, você pode usar a anotação `@javax.enterprise.inject.Alternative` no lugar da `@Vetoed`. Essa anotação também marca o bean como não elegível em um primeiro momento, mas nos permite reativá-lo posteriormente via configuração. Mas não se preocupe com essa anotação por enquanto, veremos mais detalhes dela na seção 4.3.

Agora, analisando as classes `CalculadoraDeImpostos` e `CalculadoraDeSalarios` podemos perceber algo interessante, elas não precisam de nenhuma anotação para que a CDI as detecte e consiga utilizá-las. Tanto é que tivemos que colocar uma anotação no mock para que ela também não fosse considerada automaticamente. Fora essa, a outra anotação que temos é a da `Servlet`, que está ali porque precisamos mapear um padrão de url para ela, e não por causa da CDI.

Se executarmos o exemplo através da seguinte url: <http://localhost:8080/livro-cdi/hello-cdi?salario=3000>, teremos esse resultado.

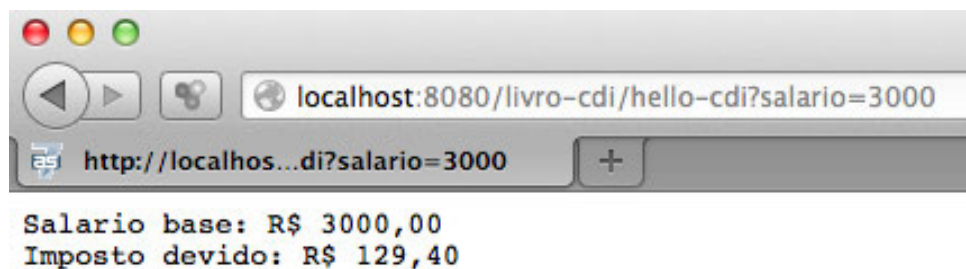


Figura 3.1: Hello world com CDI

Podemos ver que o resultado foi o mesmo dos testes vistos antes, mas o resultado não é o mais importante, o principal objetivo é entendermos como o processo todo funciona. Se analisarmos a saída no console da aplicação, teremos algo como isso:

```
Instanciando a Servlet...
Iniciando Calculadora de salários...
```

```
Calculadora de salários pronta!
Iniciando Calculadora de impostos...
Calculadora de impostos pronta!
Servlet pronta.
Efetuando o cálculo.
Fim.
```

Com esse exemplo bem simples já conseguimos perceber a dinâmica básica da resolução de dependências. Ao instanciar a Servlet `IniciandoComCDI`, a CDI percebeu que ela precisava de uma instância de `CalculadoraDeImpostos`, e por isso logo depois de instanciar o primeiro objeto, já começou a providenciar sua dependência. Porém, por mais que a Servlet não tenha uma dependência direta da `CalculadoraDeSalarios`, sem esta não é possível instanciar a `CalculadoraDeImpostos`.

Como utilizamos a injeção da `CalculadoraDeSalarios` no construtor da `CalculadoraDeImpostos`, não tem como a CDI iniciar a instanciação desta enquanto aquela não estiver pronta. Apenas didaticamente, se fizéssemos a seguinte alteração teríamos uma saída diferente:

```
import javax.annotation.PostConstruct;
import javax.inject.Inject;

public class CalculadoraDeImpostos {

    @Inject
    private CalculadoraDeSalarios calculadoraSalarios;

    public CalculadoraDeImpostos(){
        System.out.println("Iniciando Calculadora de impostos...");
    }

    @PostConstruct
    public void init(){
        System.out.println("Calculadora de impostos pronta!");
    }

    //resto do código permanece igual...

}
```

Após essa alteração, a saída no console seria a seguinte:

```
Instanciando a Servlet...
Iniciando Calculadora de impostos...
Iniciando Calculadora de salários...
Calculadora de salários pronta!
Calculadora de impostos pronta!
Servlet pronta.
Efetuando o cálculo.
Fim.
```

Podemos perceber que a CDI resolve a ordem em que os objetos precisam ser instanciados. Depois de instanciar a `CalculadoraDeSalarios`, ela é injetada na `CalculadoraDeImpostos`, e como essa era sua única dependência, a `CalculadoraDeImpostos` é avisada que está pronta, através de seu método anotado com `@PostConstruct`. Agora que está completa, ela é injetada na `Servlet`, que também é avisada que está pronta. Somente depois do ambiente pronto é que a requisição é atendida e o cálculo é efetuado.

Como observamos no exemplo, para solicitar a injeção de uma dependência basta utilizarmos a anotação `@javax.inject.Inject`. Dessa forma a CDI procura em seu contexto uma classe candidata a suprir essa dependência, e como por padrão toda classe dentro de um pacote CDI é candidata a ser injetada, não precisamos fazer nenhuma configuração nas classes. Veremos que em alguns casos é interessante especificarmos algumas características adicionais, mas isso é a exceção, pois o *default* já nos serve para a maioria dos casos. *Defaults* inteligentes são uma das marcas da CDI.

O USO DO `@PostConstruct`

Quando usamos CDI, ou mesmo outro framework que gerencia as dependências das nossas classes, precisamos ficar atentos sobre o fato de que o método que representa a instanciação em si não é a chamada do construtor, e sim, o método anotado com `@PostConstruct`, pois na invocação desse método sim temos certeza que o objeto está pronto.

Então, se pretendíamos fazer alguma programação no construtor, pode ser interessante fazer no `@PostConstruct`, pois nele, independentemente se a injeção de dependência foi via construtor, método inicializador ou diretamente nas propriedades, saberemos que o objeto está pronto. É uma espécie de construtor lógico do objeto.

Um exemplo disso é quando temos serialização de um objeto, como na passivação de um EJB. Pode ser que o container serialize o objeto quando ele se torna inativo e depois o retorne para a memória quando ele for usado novamente. Quando isso ocorre, uma nova instância é criada, e consequentemente seu construtor precisará ser invocado, pois é o *construtor físico*, já o método anotado com `@PostConstruct`, *construtor lógico*, não será invocado, pois trata-se do mesmo bean. Ele não está sendo construído, criado naquele momento, só está retornando ao estado original.

3.2 O QUE É UM PACOTE CDI?

Acabamos de ver que, por padrão, toda classe dentro de um pacote CDI é elegível para suprir uma dependência. Mas o que é um pacote CDI? E aquelas classes utilitárias que nós já temos prontas dentro de um jar, como podemos injetar instância delas dentro da nossa aplicação?

Como estamos trabalhando com uma aplicação Web (war), basta colocarmos um arquivo chamado `beans.xml` dentro da pasta `WEB-INF` da nossa aplicação. Agora se tiver um jar simples ou `ejb-jar`, colocamos esse arquivo dentro da pasta `META-INF` e pronto, a configuração da CDI está pronta.

Nesse ponto podemos nos perguntar: mas e as configuração da CDI, onde colocamos?

A resposta está novamente em *defaults* inteligentes. Como até aqui não precisa-

mos fazer configuração alguma, não precisamos sequer definir uma estrutura xml mínima dentro desse arquivo. Basta que ele exista com esse nome, mesmo estando vazio, que a CDI já começa a funcionar.

Somente quando precisarmos configurar algo específico, é que nos preocuparemos em por algo dentro do arquivo.

Até agora a estrutura da nossa aplicação está parecida com a da imagem a seguir.

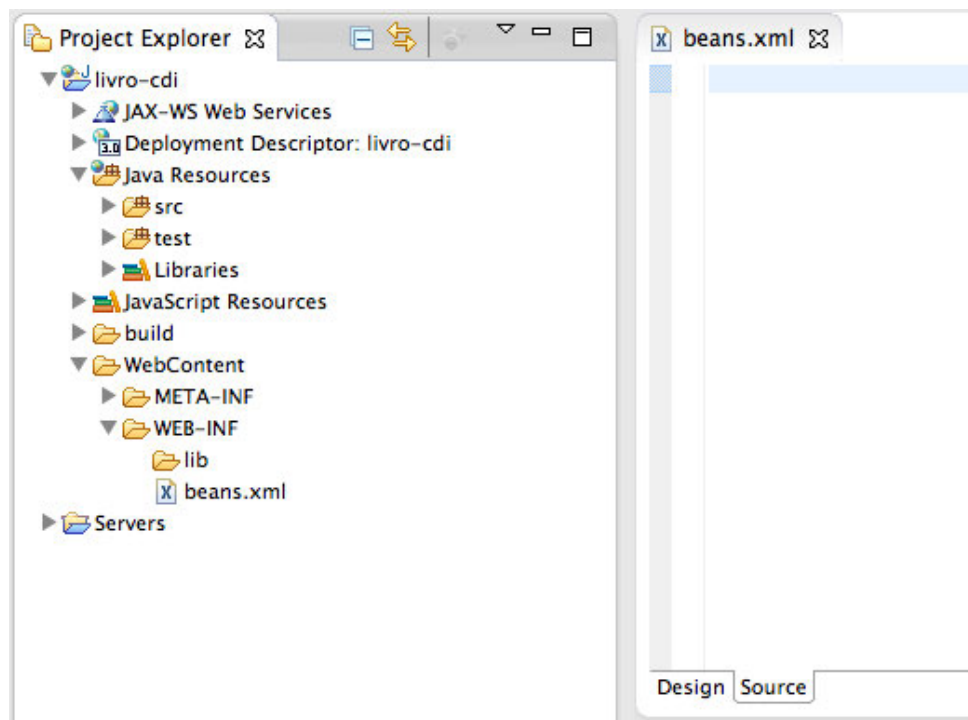


Figura 3.2: Estrutura da aplicação e arquivo beans.xml vazio

3.3 USANDO INJEÇÃO EM PROPRIEDADES, CONSTRUTORES E MÉTODOS INICIALIZADORES

Já sabemos que para injetar um recurso dentro da nossa classe é extramente simples, basta fazermos como no exemplo:

```
public class CalculadoraDeImpostos {
```

```
@Inject
private CalculadoraDeSalarios calculadoraSalarios;
//...
}
```

Apesar de ser bastante simples, se lembrarmos que uma das vantagens da injeção de dependência é a testabilidade, perceberemos que injetar recursos em uma propriedade privada não é nada testável. É só nos lembrar do teste visto na seção 2.3, em que nos deparamos com o problema de precisarmos substituir a `CalculadoraDeSalarios` dentro da `CalculadoraDeImpostos` para que a primeira devolvesse um valor fixo para a segunda. Mas isso só foi possível quando deixamos de injetar a dependência dentro de um atributo privado, e passamos a usar o construtor.

```
@Test
public void testaCalculoIRNaSegundaFaixaDeContribuicao(){
    Funcionario funcionario = new Funcionario();
    funcionario.setSalario(3000.0); // esse método não existe

    CalculadoraDeSalarios calcSalarios = new CalculadoraDeSalarios();

    CalculadoraDeImpostos calcImpostos = new CalculadoraDeImpostos();
    calcImpostos.calculadoraSalarios = calcSalarios; //não é possível!

    double impostoDeRenda =
        calcImpostos.calculaImpostoDeRenda(funcionario);

    Assert.assertEquals(129.4, impostoDeRenda, 0.0001);
}
```

Como nesse trecho utilizamos a injeção via propriedade privada, a CDI consegue injetar, mas nós não. Na verdade é possível, mas teríamos que utilizar recursos de programação que saem do que costumamos utilizar no dia a dia. Além disso, é totalmente desnecessário fazer essas “maracutaias” uma vez que a CDI consegue injetar a dependência de diversas maneiras.

3.4 AS FORMAS DE SE INJETAR DEPENDÊNCIAS

São três as formas básicas de injetar uma dependência, ou tipos de **pontos de injeção**, e os dois primeiros nós já vimos: via propriedade ou atributo, via construtor, ou via

método inicializador. A semântica dessas duas últimas opções é exatamente igual, como podemos perceber nos exemplos a seguir.

```
public class CalculadoraDeImpostos {

    private CalculadoraDeSalarios calculadoraSalarios;

    @Inject
    public void recebe(CalculadoraDeSalarios calcSalarios,
                      OutraClasse dependencia2, ...){

        this.calculadoraSalarios = calcSalarios;
        //obtem outras dependências e pode fazer alguma inicialização
    }
    //...
```

Ou então podemos usar o construtor de forma semelhante à do método inicializador (nesses exemplos a `dependencia2` do tipo `OutraClasse` tem o intuito apenas de mostrar a sintaxe, não fazendo parte do código real).

```
public class CalculadoraDeImpostos {

    private CalculadoraDeSalarios calculadoraSalarios;

    @Inject
    public Calculadora(CalculadoraDeSalarios calcSalarios,
                      OutraClasse dependencia2, ...){

        this.calculadoraSalarios = calcSalarios;
        //obtem outras dependências e pode fazer alguma inicialização
    }
    //...
```

A vantagem dessas duas abordagens é que podemos utilizar nos nossos testes o mesmo método que a CDI usa para injetar as dependências. A diferença das duas está apenas no uso ou não de um construtor para injetar as dependências. Se, por exemplo, utilizarmos algum framework que utiliza o construtor padrão (público e sem argumentos), e optarmos por injetar as dependências usando um construtor,

teremos que lembrar de criar manualmente o construtor padrão. Ainda assim, acredito que essa opção seja mais elegante, como podemos ver no novo código de teste a seguir.

```
@Test
public void testaCalculoIRNaSegundaFaixaDeContribuicao(){
    Funcionario funcionario = new Funcionario();

    CalculadoraDeSalarios calcSalariosMock =
        new CalculadoraDeSalariosMock(3000.0);

    //cria a CalculadoraDeImpostos passando dependência no construtor
    CalculadoraDeImpostos calcImpostos =
        new CalculadoraDeImpostos(calcSalariosMock);

    double impostoDeRenda =
        calcImpostos.calculaImpostoDeRenda(funcionario);

    Assert.assertEquals(129.4, impostoDeRenda, 0.0001);
}
```

Como se observa, voltamos a utilizar o *mock* criado na seção 2.3 para que o código seja executável. Agora vamos ver como fica o código se optarmos pelo método inicializador. Nesse caso, não precisaremos criar manualmente o construtor padrão, mas a utilização não fica tão limpa:

```
//cria a CalculadoraDeImpostos
CalculadoraDeImpostos calcImpostos = new CalculadoraDeImpostos();

//cria o mock
CalculadoraDeSalarios calcSalariosMock =
    new CalculadoraDeSalariosMock(3000.0);

//atribui o mock à calculadora de impostos =/
calcImpostos.recebe(mock);
```

Essa abordagem também nos permite passar as dependências dentro de um teste, mas a utilização é menos intuitiva.

3.5 RECAPITULANDO

Vimos as três formas de passar uma dependência: via atributo privado, método inicializador e via construtor. Vimos também que qualquer classe dentro de um pacote CDI é candidata para satisfazer uma dependência, ou seja, é injetável em outras sem precisar de nenhuma configuração ou anotação. Ao contrário, anotamos apenas quando queremos que a classe não seja injetável, por padrão todas são.

Como vimos até aqui, usar a CDI não é nada difícil, é praticamente Java puro. Para um projetinho simples, o que vimos até agora já bastaria, pois já sabemos injetar as dependências, que é uma das principais funcionalidades da CDI. Nos falta agora aprender como refinar um pouco mais esse mecanismo, além é claro das outras funcionalidades que a especificação nos traz.

No próximo capítulo veremos como especificar qualificadores às nossas classes, evitando casos de ambiguidade.

CAPÍTULO 4

Resolvendo dependências com tipagem forte

Talvez a principal funcionalidade da CDI seja a resolução de dependências de forma tipada. Enquanto as primeiras ferramentas de injeção de dependências eram baseadas em XML ou em anotações que usavam `Strings`, a CDI usa a mesma tipagem forte que estamos habituados a usar no Java.

Não injetamos uma dependência baseado no nome de um bean, e sim no seu tipo. A utilização de nomes é feita em casos excepcionais, por exemplo, quando queremos injetar um objeto de um tipo muito simples, como uma `String`.

Como temos muitos casos para analisar, em vez de escrever a teoria da resolução de dependências agora, vamos passar pelos exemplos que irão fundamentar o entendimento dessa teoria. Ao final do capítulo teremos uma revisão do que foi visto e a definição do funcionamento desse mecanismo tão importante.

4.1 LIDANDO COM A AMBIGUIDADE NAS DEPENDÊNCIAS

Em projetos médios ou grandes, costumamos ter uma diversidade de tipos de componentes, diferentes configurações de objetos. É comum termos diversos padrões de projetos coexistindo, ainda que não percebidos, algo que geralmente não ocorre em projetos pequenos. Esses diferentes arranjos de objetos acabam muitas vezes demandando uma configuração diferente ao usarmos CDI, mas nada que seja complicado.

Vamos adotar como exemplo a funcionalidade de cálculo da folha de pagamento da nossa empresa. Já vimos na seção 2.3 sobre o princípio da substituição de Liskov, e que é uma boa prática o uso de interfaces; logo iniciaremos essa funcionalidade pela definição da sua interface.

```
public interface CalculadoraFolhaPagamento {  
    Folha calculaFolha(List<Funcionario> funcionarios);  
}
```

Definimos uma interface bem simples para focarmos na CDI. Para nossa `CalculadoraFolhaPagamento` teremos duas implementações também simples, uma que efetua o cálculo efetivo da folha de pagamentos, salvando no banco de dados a informação da folha do mês corrente; e outra que faz apenas uma simulação, não produzindo efeitos permanentes, apenas apurando os valores.

Partiremos de implementações falsas, que apenas escrevem no console o que estariam fazendo:

```
public class CalculadoraFolhaPagamentoReal  
    implements CalculadoraFolhaPagamento {  
  
    public Folha calculaFolha(List<Funcionarios> funcionarios){  
  
        System.out.println("Efetua o cálculo "  
            + "real da folha de pagamentos");  
  
    }  
  
}
```

Apesar da implementação que acabamos de ver, conceitualmente essa implementação da `CalculadoraFolhaPagamento` além de calcular os valores, deixa

sempre no banco de dados o resultado do cálculo da folha do mês atual. É possível que no mundo real tivéssemos mais métodos nessa interface, possibilitando o cálculo retroativo, por exemplo, mas esse método já basta por enquanto.

Se for a primeira vez que executamos o cálculo dentro do mês, será criado um novo registro com os dados da folha desse mês, mas se já tiver um resultado calculado no banco, este deverá ser atualizado. Assim sendo, cada mês deve ter apenas um registro de cálculo da nossa folha.

Vamos agora à segunda implementação.

```
public class SimuladoraCalculoFolhaPagamento
    implements CalculadoraFolhaPagamento {

    public Folha calculaFolha(List<Funcionarios> funcionarios){

        System.out.println("Efetua o simulacao da folha de pagamentos");

    }

}
```

Já a `SimuladoraCalculoFolhaPagamento` faz os cálculos dos valores da folha sem considerar se há algum registro para o mês, e ao final não salva qualquer informação no banco de dados. Serve apenas para o setor financeiro projetar valores. Seria o equivalente a um simulador de financiamento de um banco. Os valores mostrados ali não ficam registrados no sistema do banco, servem apenas como uma pré-visualização.

Agora podemos criar outra servlet para testarmos nossas implementações. Vamos iniciar pela implementação “real”.

```
@WebServlet("/calcular-folha")
public class CalcularFolhaPagamento extends HttpServlet {

    @Inject
    private CalculadoraFolhaPagamento calculadoraFolha;

    public CalcularFolhaPagamento() {
        System.out.println("Instanciando a Servlet "
            + " CalcularFolhaPagamento...");
    }

}
```

```
@PostConstruct
public void ok(){
    System.out.println("CalcularFolhaPagamento pronta.");
}

protected void doGet(HttpServletRequest req,
                      HttpServletResponse res)
    throws ServletException, IOException {

    FuncionarioBuilder builder = new FuncionarioBuilder();

    Funcionario f1 = builder.comSalarioBaseDe(1000.0).build();
    Funcionario f2 = builder.comSalarioBaseDe(2000.0).build();
    Funcionario f3 = builder.comSalarioBaseDe(3000.0).build();

    List<Funcionario> funcionarios = Arrays.asList(f1, f2, f3);

    System.out.println("Efetuando o cálculo...");

    Folha folha = calculadoraFolha.calculaFolha(funcionarios);

    //mensagem para o usuário
    res.getOutputStream().print("Calculo da folha "
                                + "executado com sucesso");

    //saída no console
    System.out.println("Fim.");
}
}
```

O detalhe importante aqui é que não injetamos a implementação, senão não faria sentido definirmos a interface. Devemos sempre desenvolver pensando na abstração, e não no caso concreto. Sendo assim, solicitamos a injeção de uma implementação da interface `CalcularFolhaPagamento`. Com isso, ao criarmos uma nova implementação, não precisamos mudar nossa servlet.

A não observância dessas boas práticas de projeto de classes é que aumenta o efeito cascata quando temos uma mudança no nosso software. Usando a interface, se a forma de calcular a folha mudar, não precisamos ficar colando dúzias de *ifs* para

alterar a classe atual, podemos simplesmente criar uma nova implementação que substitui a antiga.

Podemos ainda rodar as duas implementações em paralelo, algo que não seria possível se alterássemos diretamente a implementação original. A menos é claro, que fosse feito um imenso *if* separando o cálculo novo do velho, o que certamente transformaria nosso código em algo enorme e cheio de duplicações; difícil de entender e manter.

Parece que está tudo certo, mas se executarmos nosso exemplo teremos uma surpresa (o nome totalmente qualificado da classe foi encurtado e a indentação alterada apenas para simplificar a análise do log):

```
org.jboss.weld.exceptions.DeploymentException:
WELD-001409 Ambiguous dependencies for type [CalculadoraFolhaPagamento]
with qualifiers [@Default] at injection point
[[field] @Inject private br...CalcularFolhaPagamento.calculadoraFolha].
Possible dependencies
[[Managed Bean
[class br...CalculadoraFolhaPagamentoReal]
    with qualifiers [@Any @Default],
Managed Bean
[class br...SimuladoraCalculoFolhaPagamento]
    with qualifiers [@Any @Default]]]
```

Ao iniciar a aplicação temos essa maravilha de mensagem de erro! Ela nos explica certinho o que ocorreu, e onde. Uma das premissas da CDI é tentar adiantar o máximo de erros possíveis ao iniciarmos seu contexto. Em vez de subirmos a aplicação com sucesso e encontrarmos diferentes erros a cada tela que acessamos, e pior, com mensagens nada explicativas, a CDI define que as validações devem ser feitas ao subir o contexto e as mensagens devem dizer o porquê do problema.

Lendo a mensagem percebemos que a CDI (no caso estamos usando a implementação de referência, o Weld) nos mostra exatamente qual o problema. Temos dois candidatos elegíveis para satisfazer a dependência no **ponto de injeção** `[[field] @Inject private br.com.casadocodigo.livrocdi.servlets.CalcularFolhaPagamento.calculadoraFolha]`.

Se voltarmos à seção 3.4, vamos relembrar os três tipos de ponto de injeção: atributo (aqui chamado de *field*), método inicializador e construtor. Na mensagem, aparece que a tentativa foi de injetar uma dependência em um atributo e ainda mostra qual foi: o atributo `calculadoraFolha` da classe

`br.com.casadocodigo.livrocdi.servlets.CalcularFolhaPagamento`. Assim, mesmo em uma aplicação grande, não teremos dúvida de onde está o problema relatado.

Continuando a análise da mensagem, veremos que o problema ocorreu porque encontramos dois candidatos elegíveis para a mesma dependência, e como não especificamos mais nenhum detalhe para ajudar a CDI a escolher, simplesmente foi lançada uma exceção.

Ainda na mensagem de erro vemos que além do tipo, `CalcularFolhaPagamento`, a CDI analisou também os *qualifiers*, ou qualificadores das dependências, mas como ambos os candidatos tinham os mesmos qualificadores, a ambiguidade permaneceu.

Nas seções seguintes veremos formas de eliminar essa ambiguidade.

4.2 OS QUALIFICADORES

A forma mais comum de eliminar a ambiguidade é através do uso de qualificadores. Esse nome, por sinal, é praticamente autoexplicativo. Com qualificadores conseguimos distinguir um candidato a suprir uma dependência de outro.

A definição de um qualificador é simples, criamos uma anotação que será o qualificador, e a anotamos para dizer que ela não é uma anotação qualquer.

```
import static java.lang.annotation.ElementType.*;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

import javax.inject.Qualifier;

@Target({TYPE, FIELD, METHOD, PARAMETER})
@Retention(RetentionPolicy.RUNTIME)
@Qualifier
public @interface Simulador {
}
```

Apesar das diversas anotações na nossa anotação, apenas a `@Qualifier` é da CDI, as outras duas são anotações normais do Java. Por isso no código deixei aparecendo os imports.

As duas primeiras anotações, `@Target` e `Retention`, servem respectivamente para dizer onde poderemos utilizar nosso qualificador, e para que nossa anotação

seja legível em tempo de execução, pois sem isso a CDI não conseguiria lê-la. Por fim, a anotação que interessa diz que nossa anotação é um qualificador. Agora basta qualificarmos nossas classes que não haverá mais dúvidas.

```
@Simulador
public class SimuladoraCalculoFolhaPagamento implements
    CalculadoraFolhaPagamento {

    //o resto do código permanece igual

}
```

Como nosso qualificador diz que o objeto marcado por ele é um simulador, deixamos o código real como está, e marcamos apenas o simulador.

Agora ao subir nossa aplicação não ocorre nenhum erro, e ao executar nossa servlet (<http://localhost:8080/livro-cdi/calcular-folha>), a saída no console é a seguinte.

```
Instanciando a Servlet CalcularFolhaPagamento...
CalcularFolhaPagamento pronta.
Efetuando o cálculo...
Efetua o cálculo real da folha de pagamentos
Fim.
```

Ou seja, a dependência injetada foi a implementação real da `CalculadoraFolhaPagamento`.

Uma coisa que devemos perceber: na nossa servlet pedimos para injetar uma `CalculadoraFolhaPagamento` sem especificar um qualificador, então na prática pedimos para que seja injetada uma instância com o qualificador `@Default`.

Se olharmos novamente o log do erro apresentado antes de resolvermos a ambiguidade, aparecia a mensagem de que foram encontrados dois candidatos com os tipos compatíveis, e com os mesmos qualificadores, `@Any` e `@Default`.

Logo, podemos concluir que assim como quando não especificamos nenhum construtor na nossa classe o Java automaticamente cria um construtor default, assim também a CDI qualifica nossos objetos com o qualificador default se não especificarmos outro.

Podemos concluir também que se anotarmos a classe `CalculadoraFolhaPagamentoReal` com o qualificador `@Simulador`, voltaremos a ter o mesmo problema de ambiguidade, mas dessa vez com os qualificadores `@Any` e `@Simulador`.

Qualificadores com atributos

Como sempre, agora que entendemos a solução, vamos evoluir um pouquinho o problema. Vamos supor que está sendo feito um estudo de viabilidade financeira de um novo plano de cargos para a empresa, e já que temos a funcionalidade de simular a folha de pagamento, o setor financeiro nos pediu para implementarmos um simulador que considerasse os parâmetros pretendidos no novo plano.

Dessa forma, a implementação real e sua servlet permanecem inalteradas, mas precisaremos de mais uma instância de `CalculadoraFolhaPagamento` para executar essa nova tarefa. O detalhe é que essa nova implementação também será um simulador, mas em vez de simular no plano de cargos atual, que é do ano de 2005, vai fazer os cálculos baseados na proposta de 2013.

Como resolvemos? Alteramos nosso qualificador para `@Simulador2005` e criamos um `@Simulador2013`? Já considerando essas situações — os qualificadores — assim como qualquer anotação Java pode ter atributos, no nosso vamos criar um que servirá para identificar o plano que está sendo considerado.

```
@Target({TYPE, FIELD, METHOD, PARAMETER})
@Retention(RetentionPolicy.RUNTIME)
@Qualifier
public @interface Simulador {
    PlanoDeCargos planoDeCargos() default PlanoDeCargos.VERSAO_2005;
}
```

Alteramos nosso qualificador para que ele aceite a indicação do plano de cargos que está sendo considerado. Para manter a compatibilidade, especificamos que caso não seja especificado um valor para essa propriedade, será considerado o plano de 2005, mas poderíamos deixar sem valor padrão e alterar nossa `SimuladoraCalculoFolhaPagamento` para especificar lá qual plano usar. Mas como em um caso real pode ser que tenhamos muito mais casos para mudar, dessa maneira minimizamos os impactos.

Outra coisa que podemos perceber é que estamos usando uma enum que representa os planos de cargos.

```
public enum PlanoDeCargos {
    VERSAO_2005, VERSAO_2013
}
```

É um código bem simples, que estamos usando apenas para não usarmos um inteiro puro, pois isso pode acarretar erro de digitação. E o `enum` tem ainda a vanta-

gem de trazer os valores possíveis prontos, enquanto se estivéssemos usando inteiro, além da possibilidade do erro de digitação, teríamos que saber de cabeça quais os valores válidos.

Agora sim, podemos criar o novo simulador especificando a versão do plano que estamos considerando.

```
import static
    br.com.casadocodigo.livrocdi.qualificadores.PlanoDeCargos.*;

@Simulador(planoDeCargos = VERSAO_2013)
public class SimuladoraCalculoFolhaPagamentoPlano2013 implements
    CalculadoraFolhaPagamento {

    @Override
    public Folha calculaFolha(List<Funcionario> funcionarios) {
        System.out.println("Simulação com plano de 2013");
        return null;
    }
}
```

A diferença está apenas no fato de especificarmos um valor diferente do padrão dentro do nosso qualificador. Agora quando formos injetar uma `CalculadoraFolhaPagamento` dentro de um bean, temos as seguintes opções:

```
@Inject
private CalculadoraFolhaPagamento calculadoraFolha;
```

Isso injeta a implementação real da interface pedida, que será um objeto da classe `CalculadoraFolhaPagamentoReal`. Na prática é como se estivéssemos especificando que queremos o qualificador `@Default`.

```
@Inject @Simulador
private CalculadoraFolhaPagamento calculadoraFolha;

//ou

@Inject @Simulador(planoDeCargos = VERSAO_2005)
private CalculadoraFolhaPagamento calculadoraFolha;
```

Como especificamos a `VERSAO_2005` como valor padrão do qualificador, essas duas opções são equivalentes, e a implementação injetada será a `SimuladoraCalculoFolhaPagamento`.

Por fim, temos a seguinte opção.

```
@Inject @Simulador(planoDeCargos = VERSAO_2013)
private CalculadoraFolhaPagamento calculadoraFolha;
```

Isso fará com que o bean injetado seja da classe `SimuladoraCalculoFolhaPagamentoPlano2013`.

A utilização de propriedades nos qualificadores é uma ferramenta interessante pois evita a criação de qualificadores esquisitos, que seriam na verdade uma especialização de outro qualificador. Em vez de termos dois qualificadores: `@Simulador2005` e `@Simulador2013`, temos apenas um qualificador no qual podemos especificar um atributo.

Podemos ter inclusive vários atributos dentro do nosso qualificador:

```
import javax.enterprise.util.Nonbinding;

@Target({TYPE, FIELD, METHOD, PARAMETER})
@Retention(RetentionPolicy.RUNTIME)
@Qualifier
public @interface Simulador {
    PlanoDeCargos planoDeCargos() default PlanoDeCargos.VERSAO_2005;
    boolean planoAprovado() default true;
    @Nonbinding boolean enviarNotificacao() default false;
}
```

Para mostrar como podemos utilizar mais de uma propriedade, alteramos nosso qualificador para que ele receba mais de uma propriedade. Nesse exemplo, adicionamos duas novas propriedades. A primeira para definir se o simulador é de um plano já aprovado ou não, e a segunda é apenas uma informação extra, que não influencia na qualificação propriamente dita.

Isso porque anotamos a propriedade `enviarNotificacao` com `@Nonbinding`, e dessa maneira informamos à CDI que a mesma não deve ser considerada quando um bean candidato estiver sendo escolhido. O valor aí presente serve apenas como uma propriedade informativa para o nosso código Java. Ou seja, em algum momento podemos ter um código analisando se o valor

da propriedade é verdadeiro para enviar, por exemplo, um e-mail avisando que o cálculo foi realizado e qual o resultado.

Apesar das propriedades nos qualificadores evitar a criação de novos qualificadores de maneira desnecessária, existirão casos em que teremos mais de um qualificador ao mesmo tempo. Cada qualificador desses pode ter atributos que especificam ainda mais o que queremos, que são os atributos que **não** estão anotados com `@Nonbinding`, e também atributos que possuam essa anotação e servem apenas como informação adicional.

É só lembrar que *Nonbinding* significa “não vinculante”, aquele que não cria vínculo. Fora essas propriedades não vinculantes, todas as outras refinam mais ainda a qualificação do bean — assim também como acontece quando temos mais de um qualificador.

Ao final do capítulo, na seção 4.9, teremos uma recapitulação de todo o processo de resolução de dependências envolvendo qualificadores e outras ferramentas que veremos até lá. Tente imaginar as possibilidades com as ferramentas que você já viu, e se permanecer alguma dúvida, com o resumo acredito que ela será sanada.

O qualificador `@Any`

Já entendemos como acabar com uma ambiguidade através de um qualificador, mas nos deparamos algumas vezes com esse qualificador `@Any`. Como o nome dele diz, esse qualificador está presente em *qualquer* objeto.

Para exemplificar, vamos mudar nossa servlet, que está funcionando sem qualquer ambiguidade, para que ela fique dessa forma:

```
@WebServlet("/calcular-folha")
public class CalcularFolhaPagamento extends HttpServlet {

    @Inject @Any
    private CalculadoraFolhaPagamento calculadoraFolha;

    //o restante permanece igual
}
```

Sabemos que temos três candidatos a satisfazer dependências do tipo `CalculadoraFolhaPagamento`, que são: `CalculadoraFolhaPagamentoReal`, que está com o qualificador `@Default`, já que não especificamos outro,

`SimuladoraCalculoFolhaPagamento`, o qual recebeu o qualificador `@Simulador`, e `SimuladoraCalculoFolhaPagamentoPlano2013`, que recebeu o qualificador `@Simulador(planoDeCargos = VERSAO_2013)`.

Se no ponto de injeção não colocarmos nenhum qualificador, estaremos implicitamente pedindo o default, e se colocarmos `@Simulador` sabemos qual instância será injetada também. Mas fazendo a alteração que acabamos de ver, estamos pedindo para que nesse caso seja considerado **qualquer** candidato disponível no contexto.

No nosso exemplo, isso fará com que voltemos a ter problemas de ambiguidade:

```
org.jboss.weld.exceptions.DeploymentException: WELD-001409
Ambiguous dependencies for type [CalculadoraFolhaPagamento]
with qualifiers [@Any] at injection point [[BackedAnnotatedField]
@Inject @Any private br...CalcularFolhaPagamento.calculadoraFolha].
Possible dependencies
[[Managed Bean [class br...SimuladoraCalculoFolhaPagamentoPlano2013]
with qualifiers [@Simulador @Any],
Managed Bean [class br...SimuladoraCalculoFolhaPagamento]
with qualifiers [@Simulador @Any],
Managed Bean [class br...CalculadoraFolhaPagamentoReal]
with qualifiers [@Any @Default]]].
Please see server.log for more details.
```

Para que serviria então esse qualificador? Ele, na verdade, seria o equivalente à classe `java.lang.Object`, que é mãe de qualquer objeto Java, mas para os objetos que estão no contexto CDI. Assim, se precisarmos, por algum motivo, buscar todos os candidatos de um determinado tipo, independente de qualificadores, usamos o qualificador especial `@Any`.

O uso ficará mais claro quando estivermos estudando a busca programática de dependências, na seção 4.8. Nesse cenário será interessante podermos buscar todos os candidatos a satisfazer uma dependência de um determinado tipo, e então tomar alguma decisão em tempo de execução, ou ainda usar todos esses objetos de uma vez.

Apenas para não ficar muito abstrato, vamos supor que tenhamos uma interface que define um mecanismo de notificação. Essas notificações seriam enviadas ao responsável pelo setor financeiro da empresa sempre que uma nova folha fosse gerada, podendo ser por e-mail, SMS ou qualquer meio que possamos imaginar.

Como não sabemos quais qualificadores as implementações dessa interface podem ter, podemos utilizar o qualificador `@Any` para buscar todas as implementações

disponíveis, colocá-las em uma `Collection`, e então passar por todas as instâncias invocando uma a uma para garantir que por algum meio a notificação chegaria a quem interessar.

4.3 ALTERNATIVES: ELIMINANDO AMBIGUIDADE E TORNANDO UM BEAN OPCIONAL

A anotação `@Alternative` serve para transformarmos um bean em uma alternativa a suprir uma dependência, ou seja, ele não será um candidato automaticamente disponível.

Todo bean alternativo só é injetável se for declarado no `beans.xml`, e nesse caso ele fica com uma prioridade maior do que o bean não alternativo.

No caso da aplicação que estamos desenvolvendo, vamos considerar que tivemos a criação de um novo plano de cargos e salários, que alterou totalmente a forma como nosso cálculo de salário é feito hoje. A primeira alternativa que podemos pensar é em alterar a classe `CalculadoraDeSalarios` para acomodar as novas regras, mas vamos analisar melhor esse caso.

Uma mudança grande assim pode não entrar em vigor imediatamente, então vamos considerar que a mudança entre em produção daqui três meses. Além disso, após entrar em produção pode ser que por descontentamento de alguns, a empresa volte atrás, para a regra que estava funcionando antes e parta para a elaboração de um novo plano.

Alterando diretamente o código da `CalculadoraDeSalarios`, não poderíamos colocar em produção imediatamente pois a regra nova só passaria a valer no futuro. E depois se tivermos que voltar a regra antiga teríamos que reverter as alterações. Apesar desses cenários serem bem cobertos por um bom uso de controle de versão de código, caso isso não seja feito corretamente, e não tenhamos um servidor de build, podemos ter situações em que não seria possível determinado desenvolvedor fazer um novo deploy pois a versão que está na sua máquina já está com o cálculo alterado. Claro que esse é um exemplo praticamente lúdico, mas não quer dizer que não ocorra.

Um dos usos de `@Alternative` pode ser justamente o desenvolvimento dessa nova versão da `CalculadoraDeSalarios`. Em vez de mudar a implementação da classe, extraímos uma interface dessa classe (geralmente as IDEs tem atalhos para fazer isso). Dessa forma teremos a interface `CalculadoraDeSalarios` e a implementação com o código atual, e todos os pontos de injeção devem continuar apon-

tando para o tipo `CalculadoraDeSalarios`, que antes era uma classe, e agora passa a ser uma interface.

```
public interface CalculadoraDeSalarios {

    double calculaSalario(Funcionario funcionario);

}

public class CalculadoraDeSalariosPlano2005
    implements CalculadoraDeSalarios {
    /*
        Código igual ao que tínhamos em
        CalculadoraDeSalarios quando era classe
    */
}
```

Agora como estamos criando um novo cálculo com base no novo plano de salários, basta criarmos uma nova implementação, que pode por exemplo se chamar `CalculadoraDeSalariosPlano2013`, para indicar que é o cálculo do plano criado nesse ano. Porém se colocarmos as duas implementações em produção ao mesmo tempo, teremos o já conhecido problema da ambiguidade. Por isso, na nova implementação utilizaremos a anotação `@Alternative`.

```
import javax.enterprise.inject.Alternative;

@Alternative
public class CalculadoraDeSalariosPlano2013
    implements CalculadoraDeSalarios {

    //o novo código, que nesse exemplo não influencia no entendimento
    @Override
    public double calculaSalario(Funcionario funcionario){
        return 0.0;
    }
}
```

Agora podemos deixar a implementação pronta, e inclusive fazer um deploy com ela sem o menor problema, pois os beans alternativos não são injetáveis em lugar nenhum até segunda ordem. Essa segunda ordem será a expressa habilitação desse bean no arquivo `beans.xml`, ou via anotação, momento em que ele passa a ser elegível e

inclusive “sobrescreve” o bean não alternativo. Em outras palavras, quando ligarmos a `CalculadoraDeSalariosPlano2013`, ela entra automaticamente no lugar da `CalculadoraDeSalariosPlano2005`. E caso seja necessário voltar atrás, basta comentar a linha do arquivo `beans.xml` onde habilitamos o bean.

Pela primeira vez estamos mexendo com um arquivo xml, e ainda assim seu código é mínimo. A grande maioria dele é o cabeçalho do arquivo, e eu encurtei o nome da classe para ficar bem em uma linha, mas sempre que usarmos o arquivo xml, deve usar o nome totalmente qualificado das classes.

```
<beans xmlns="http://xmlns.jcp.org/xml/ns/javaee"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
                           http://xmlns.jcp.org/xml/ns/javaee/beans_1_1.xsd">
  <alternatives>
    <class>
      br.com.casadocodigo...CalculadoraDeSalariosPlano2013
    </class>
  </alternatives>
</beans>
```

Agora, ao executarmos nosso sistema, a classe `CalculadoraDeSalariosPlano2013` é que será a injetável nos pontos que pedem pela interface `CalculadoraDeSalarios`.

Podemos perceber que o uso de `@Alternative` tem um propósito diferente do uso dos qualificadores vistos antes, é uma forma de desabilitar temporariamente um bean, e habilitá-lo em momento oportuno de uma forma que ele volte no lugar do bean equivalente.

Em um cenário mais complexo, podemos ter até mais de um bean marcado como alternativo. Por exemplo, quando utilizamos o padrão *Strategy*. É possível marcar todas as implementações como alternativas e habilitar somente aquela que queremos usar no momento. Porém, se habilitarmos mais de uma, voltaremos ao problema da ambiguidade, a menos que especifiquemos, via anotação, qual alternativa tem prioridade maior.

4.4 PRIORIDADE: NOVIDADE DA CDI 1.1, PARTE DO JAVA EE 7

A partir do Java EE 7, com a atualização da especificação de interceptor para a versão 1.2 (JSR 318 release de manutenção 2), foi adicionada a possibilidade de estipularmos ordenação dos interceptors; para que isso fosse possível, foi adicionada a anotação `@Priority` nas anotações comuns do Java (JSR 250). Quando estivermos estudando os interceptadores, veremos mais sobre isso. Por enquanto vamos analisar essa anotação do ponto de vista dos `@Alternatives`.

Como já vimos, usar alternativas é uma boa forma de criarmos diferentes opções para uma dependência, mas até o Java EE 6, não conseguíamos habilitar mais de uma alternativa ao mesmo tempo, pois teríamos o problema de ambiguidade. Um bean alternativo tem prioridade sobre um bean não alternativo, mas entre os alternativos não havia maneira de estipular prioridade.

Agora, além de não precisarmos mais utilizar o arquivo `beans.xml` para ativar uma alternativa, podemos estipular diferentes prioridades para cada uma, assim, será injetada a de prioridade maior.

A anotação `@Priority` requer um inteiro que especifica qual a prioridade, e será injetado o bean de prioridade maior. Não é obrigatório, mas é interessante utilizarmos o intervalo de valores que constam na especificação de interceptors:

- `Interceptor.Priority.PLATFORM_BEFORE = 0`
- `Interceptor.Priority.LIBRARY_BEFORE = 1000`
- `Interceptor.Priority.APPLICATION = 2000`
- `Interceptor.Priority.LIBRARY_AFTER = 3000`
- `Interceptor.Priority.PLATFORM_AFTER = 4000`

Os valores para plataforma e biblioteca servem para especificar respectivamente os intervalos que são usados pela plataforma Java, ou seja, as partes do próprio Java EE; e para serem usados pelas bibliotecas, frameworks, e outras coisas que executam em momento diferente da plataforma, e também da nossa aplicação.

Como estamos aqui tratando do desenvolvimento de aplicações, vamos trabalhar a partir do valor `Interceptor.Priority.APPLICATION`. Vamos então alterar nossas implementações de `CalculadoraDeSalarios` apenas para vermos essa possibilidade.

```
import javax.annotation.Priority;
import javax.enterprise.inject.Alternative;
import javax.interceptor.Interceptor;

@Alternative @Priority(Interceptor.Priority.APPLICATION + 1)
public class CalculadoraDeSalariosPlano2013
    implements CalculadoraDeSalarios {

    // código permanece igual

}
```

Vamos anotar nossa classe `CalculadoraDeSalariosPlano2013` com a anotação `@Priority` e informar a prioridade que queremos. Agora podemos comentar no `beans.xml` onde ativávamos esse bean alternativo, pois a anotação supre essa demanda.

```
<beans xmlns="http://xmlns.jcp.org/xml/ns/javaee"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
                           http://xmlns.jcp.org/xml/ns/javaee/beans_1_1.xsd">
  <alternatives>
    <!--
    <class>
      br.com.casadocodigo...CalculadoraDeSalariosPlano2013
    </class>
    -->
  </alternatives>
</beans>
```

Vamos agora deixar nossa implementação atual também como alternativa para podermos testar a priorização.

```
import javax.annotation.Priority;
import javax.enterprise.inject.Alternative;
import javax.interceptor.Interceptor;

@Alternative @Priority(Interceptor.Priority.APPLICATION)
public class CalculadoraDeSalariosPlano2005
    implements CalculadoraDeSalarios {
```

```
// código permanece igual  
}
```

Notem que a `CalculadoraDeSalariosPlano2013` ficou com uma prioridade maior para que ela seja a escolhida.

Para conseguirmos testar, podemos criar uma outra servlet que injeta essa dependência e simplesmente escrever a classe selecionada na resposta para o usuário. Em breve, estaremos melhorando essa forma de testar e a apresentação para o usuário, mas para darmos um passo de cada vez, ainda está mais simples fazer isso usando servlets auxiliares.

```
@WebServlet("/calcular-salario")  
public class CalculadoraSalario extends HttpServlet {  
  
    @Inject  
    private CalculadoraDeSalarios calculadoraDeSalarios;  
  
    @Override  
    protected void doGet(HttpServletRequest req, HttpServletResponse res)  
        throws ServletException, IOException {  
  
        res.getOutputStream()  
            .print("Calculadora: " + calculadoraDeSalarios.getClass());  
    }  
}
```

Aí basta alternarmos a prioridade de uma ou outra instância de `CalculadoraDeSalarios` e executar novamente para percebermos a mudança na seleção do bean a ser injetado.

4.5 BEANS NOMEADOS

Apesar da CDI trabalhar muito bem com a resolução de dependências através de tipos, há casos em que precisamos que um bean seja descoberto a partir de um nome. Isso pode ser necessário quando pedimos a injeção de um tipo muito simples, como uma `String` ou um `Integer` que signifique, por exemplo, o caminho padrão onde anexos serão salvos ou o número máximo de elementos a ser exibidos nas páginas de listagem do sistema, ou seja, nas paginações.

```
@Inject @Named("caminhoSalvarAnexos")
private String caminhoAnexos;

@Inject @Named("tamanhoPaginacao")
private Integer tamanhoPaginacao;
```

Nesse caso, como o tipo é muito comum, usamos o nome como um qualificador. Mas de onde virão esses objetos? Esses tipos são do próprio Java, logo, não são injetáveis diretamente, ainda mais com esses nomes. Para suprir essa necessidade, a CDI define os métodos produtores, que serão apresentados na seção 5.1, então não se preocupe com esse detalhe ainda.

Uma outra forma de recuperarmos essas informações que estamos pedindo via nome, seria através da definição de um tipo que represente as configurações do sistema. Nesse tipo perderíamos colocar essas e demais informações necessárias, como caminhos de *wSDL* para integração entre sistemas e o que mais for necessário. Poderíamos em um método produtor buscar essas informações no banco de dados e colocar o objeto com todas as informações no contexto CDI. Assim, em vez de buscar uma informação no banco cada vez que precisássemos, bastaria injetar o objeto que representa a configuração do sistema.

Mas se a funcionalidade de nomearmos um bean servisse somente para isso, acabaríamos de descobrir que ela na verdade não é muito útil. Porém temos um caso em que os nomes são necessários, e dessa vez não temos como fazer de outra forma. Esse caso é a utilização dos beans em *expression languages*, ou *ELs*. Por exemplo, em uma página JSP ou JSF onde temos `#{calculadoraFolhaBean.ultimoCalculo.valorFolha}`.

Quando trabalhamos com EL, temos apenas uma `String` que representa o nome do bean, no caso o `calculadoraFolhaBean`. Logo, a resolução desse objeto precisará ser feita via nome. Para que um bean seja alcançável por EL, precisamos “batizá-lo”, dar um nome a ele. Para isso utilizamos a anotação `javax.inject.Named`.

```
import javax.inject.Named;

@Named
public class CalculadoraFolhaBean {

    //precisa ter um método getter
    private FolhaPagamento ultimoCalculo;
```

```
...
}
```

Nessa anotação podemos especificar um nome, como `@Named("nomeDoBean")`, ou podemos deixar o nome padrão como no exemplo. Esse nome padrão nada mais é do que o nome simples da classe com a primeira letra passada para minúsculo. Por isso, na EL vista antes usávamos `calculadoraFolhaBean`.

Uma vez resolvido o objeto raiz da EL, que precisa ter um nome, os demais são alcançados via *reflection*, através dos respectivos *getters* em caso de leitura, e *setters* em caso de escrita. Esse funcionamento, porém, será visto em outro momento, pois o importante nessa seção é entendermos que além da resolução por tipo, a CDI suporta resolução por nome.

O mais comum é utilizarmos os beans nomeados como objeto raiz em uma EL, mas nada impede que o injetemos em outro bean; e nesse caso pode ser tanto por nome, como pelo tipo, como já estamos habituados.

4.6 TRABALHANDO COM HERANÇA ENTRE BEANS

Já vimos sobre princípio da substituição na seção 2.3; no ponto de vista do Java, se estivermos usando interfaces, conseguimos trocar suas implementações ou conseguimos trocar um tipo por seus subtipos. Usando CDI, no entanto, não basta usar `implements` ou `extends`.

Vamos tomar como exemplo a classe `CalculadoraDeSalariosPlano2013`, que já vimos nos exemplos anteriores, e uma nova classe que veremos agora, a `CalculadoraAssincronaDeSalariosPlano2013`. O código dessas classes não importam para nosso exemplo, o importante é entendermos como aplicar a substituição entre a calculadora e a calculadora assíncrona, seu subtipo.

```
@Calculadora
public class CalculadoraDeSalariosPlano2013
    implements CalculadoraDeSalarios {}
```

Para enriquecer nossa análise, adicionamos um qualificador novo, que usaremos apenas nesse exemplo: `@Calculadora`. Agora vejamos a calculadora assíncrona.

```
@Alternative @Priority(Interceptor.Priority.APPLICATION)
public class CalculadoraAssincronaDeSalariosPlano2013
    extends CalculadoraDeSalariosPlano2013 {}
```

Para que não tenhamos problemas de ambiguidade, vamos considerar, para o estudo de herança, que a `CalculadoraDeSalariosPlano2013` é uma classe simples (e não um `@Alternative` como vimos antes) e que a `CalculadoraAssincronaDeSalariosPlano2013` é um bean alternativo ativo, pois está anotado com `@Priority`. Esse ajuste é apenas para termos um cenário mais simples de analisar, porém com todos os ingredientes para uma boa análise desse tema.

Com base no que já sabemos, conseguimos prever como a CDI irá resolver a seguinte dependência.

```
@Inject
private CalculadoraDeSalarios calculadora;
```

Como ambas as classes que acabamos de ver são de tipo compatível com a interface solicitada, mas como a `CalculadoraAssincronaDeSalariosPlano2013` é alternativa, ou seja, tem prioridade, sabemos que ela será a implementação selecionada. Mas qual seria o resultado se o trecho que solicita a injeção da dependência fosse como o seguinte:

```
@Inject @Calculadora
private CalculadoraDeSalarios calculadora;
```

Nesse caso, seria injetada uma instância da classe `CalculadoraDeSalariosPlano2013`, pois sua subclasse não possui o qualificador `@Calculadora`. No entanto, por mais que não tenhamos analisado o código da classe `CalculadoraAssincronaDeSalariosPlano2013`, vamos considerar que ela é uma evolução de sua classe mãe, e que, ao criá-la, esperávamos que ela entrasse no lugar da classe mãe. Até porque, já estudamos o princípio da substituição e sabemos que é possível, mas como nos assegurar que a nova classe sempre será selecionada no lugar da antiga?

Uma forma de fazer isso seria, além de estender a classe original, colocar na subclasse todos os qualificadores da primeira. Com isso, no exemplo que acabamos de ver, em que além do tipo eram pedidos qualificadores (no caso um, mas poderiam ser diversos), o bean selecionado seria a calculadora assíncrona. Mesmo assim ainda teríamos dois casos em que a classe mãe continuaria sendo utilizada: métodos produtores e observadores de eventos. Esses dois assuntos ainda não foram vistos, mas o que importa por enquanto é que ambos não são herdados pela subclasse.

A anotação @Specializes

Para resolver essa questão, a CDI disponibiliza a anotação `@Specializes`, que marca a subclasse como uma substituta da classe mãe. Assim, mesmo que a nova implementação seja adicionada em um novo jar, muito tempo depois que o sistema está em produção, ela certamente entrará em todos os lugares onde a classe mãe era utilizada. Para tal, basta colocar essa anotação na classe filha.

```
@Specializes @Assincrono
public class CalculadoraAssincronaDeSalariosPlano2013
    extends CalculadoraDeSalariosPlano2013 {}
```

Agora sabemos que o novo bean substituirá totalmente o antigo, pois ao usar `@Specializes` a classe filha herda também todos os qualificadores da mãe. Isso não impede no entanto que a nova classe defina novos qualificadores, como o `@Assincrono` do exemplo.

Outra coisa herdada é o nome do bean original. Caso a classe mãe possui o qualificador `@Named`, a classe filha necessariamente herdará o mesmo nome. Se a classe mãe não possuir um nome, a filha fica livre para definir ou não um. Mas se a mãe for nomeada e a filha tentar definir um novo nome, será lançado um erro quando a configuração for lida enquanto a aplicação inicializa.

Porém, nem tudo é tão automático, como o mecanismo de especialização precisa garantir que realmente a classe mãe não será instanciada em momento algum pela CDI, e como os métodos produtores e observadores de eventos não são herdados, os mesmos ficarão inativos a menos que sejam reimplementados (sobrescritos) na subclasse.

4.7 RESTRINGINDO O TIPO DOS BEANS

Já vimos diversas formas de lidar com ambiguidade entre beans: qualificadores, alternativos, e até especializar um bean. Mas temos ainda outra forma de gerenciar nossas dependências, que é forçando o tipo do bean. Obviamente não podemos colocar qualquer tipo que queiramos, mas apenas limitar até que nível da hierarquia nosso bean estará apto a responder.

Vamos recapitular alguns dos exemplos que vimos até aqui, e verificar uma nova forma de lidar com a ambiguidade.

```
public interface CalculadoraDeSalarios { ... }
```

```
public class CalculadoraDeSalariosPlano2013
    implements CalculadoraDeSalarios { ... }

public class CalculadoraAssincronaDeSalariosPlano2013
    extends CalculadoraDeSalariosPlano2013 { ... }
```

Como já vimos até agora, a resolução de dependências da CDI é fortemente baseada em tipos, então esse é o fator determinante na escolha de um desses candidatos. Vamos ver os próximos exemplos.

```
//problema de ambiguidade
@Inject private CalculadoraDeSalarios calculadora;

//problema de ambiguidade
@Inject private CalculadoraDeSalariosPlano2013 calculadora;

//sem problema de ambiguidade
@Inject private CalculadoraAssincronaDeSalariosPlano2013 calculadora;
```

Os problemas de ambiguidade que acabamos de ver envolvem `CalculadoraDeSalariosPlano2013` e `CalculadoraAssincronaDeSalariosPlano2013`. Isso porque naturalmente cada bean é candidato a suprir dependências de qualquer um dos tipos do bean. No caso, `CalculadoraAssincronaDeSalariosPlano2013` responde pelo próprio tipo, por `CalculadoraDeSalariosPlano2013`, por `CalculadoraDeSalarios` e por `java.lang.Object`. Mas podemos restringir por quais tipos nosso bean está disposto a responder, para isso basta usarmos a anotação `@javax.enterprise.inject.Typed`.

```
//responde apenas por CalculadoraDeSalarios e Object
@Inject @Typed(CalculadoraDeSalarios.class)
public class CalculadoraAssincronaDeSalariosPlano2013
    extends CalculadoraDeSalariosPlano2013 { ... }

/*
    responde por CalculadoraDeSalarios,
    CalculadoraDeSalariosPlano2013 e Object
*/
```



```
@Typed({CalculadoraDeSalarios.class,
        CalculadoraDeSalariosPlano2013.class})
public class CalculadoraAssincronaDeSalariosPlano2013
    extends CalculadoraDeSalariosPlano2013 { ... }
```

Restringir o tipo de um bean pode, em determinada situação, resolver um problema de ambiguidade. Por exemplo, se injetarmos o tipo `CalculadoraDeSalariosPlano2013` e restringirmos o tipo de `CalculadoraAssincronaDeSalariosPlano2013` para o tipo `CalculadoraDeSalarios`. Nesse caso apenas o próprio bean do tipo `CalculadoraDeSalariosPlano2013` poderá se candidatar.

Essa talvez não seja a forma mais habitual de lidar com ambiguidade, mas é mais uma ferramenta que a CDI nos oferece.

4.8 RESOLUÇÃO DE DEPENDÊNCIA SOB DEMANDA E LOOKUP PROGRAMÁTICO

Há situações em que a injeção de dependência comum não é o bastante. Podemos ter problemas de dependência cíclica, ou problemas de lentidão quando um bean tem muitas dependências, que por sua vez pode ter também várias outras dependências. Para solucionar esse problema podemos lançar mão da injeção de dependência *lazy*, ou sob demanda.

Para exemplificar, vamos tomar como exemplo a injeção de uma instância de `CalculadoraFolhaPagamento`,

```
//forma tradicional
@Inject
private CalculadoraFolhaPagamento calculadoraFolhaPagamentoPadrao;

//injeção sob demanda
@Inject
private Instance<CalculadoraFolhaPagamento>
    calculadoraFolhaPagamentoLazy;
```

Nesses exemplos, conforme já vimos, teremos a injeção de uma instância da implementação `CalculadoraFolhaPagamentoReal`. Isso porque usamos o qualificador `@Default`, e as demais implementações, `SimuladoraCalculoFolhaPagamento` e

`SimuladoraCalculoFolhaPagamentoPlano2013`, possuem o qualificador `@Simulador`.

A diferença é que ao injetarmos da forma tradicional, a instância é imediatamente criada - ou recuperada, dependendo do escopo -, enquanto a forma *lazy* é apenas um link para o tipo pretendido. A instância real é disponibilizada através do método `get()`, como vemos a seguir.

```
@Inject
private Instance<CalculadoraFolhaPagamento>
    calculadoraFolhaPagamentoLazy;
...
```

```
CalculadoraFolhaPagamento calculadora =
    calculadoraFolhaPagamentoLazy.get();
```

Somente nesse momento a instanciação acontece. Essa abordagem ajuda em casos de injeção lenta. Caso tenhamos muitas dependências que são utilizadas somente em determinados casos, por exemplo dependendo de alguma informação externa, pedir para todas as dependências serem resolvidas desde o início poderá provocar uma certa lentidão no sistema.

Em cenários assim, é bem possível que esse bean esteja pouco coeso, fazendo coisas diferentes, coisas demais. Porém quando trabalhamos com instanciação manual das dependências, acabamos instanciando cada dependência só quando elas são demandadas, o que acaba maquiando o que talvez seja um problema de projeto das classes.

Porém sabemos que nem sempre é simples fazer alguns ajustes na aplicação, como por exemplo separar beans muito grandes em outros mais coesos, se esse for o caso, ainda será possível usar a instanciação sob demanda para melhorar a performance. Usando a interface `javax.enterprise.inject.Instance`, criamos um ponteiro que não cria a instância imediatamente, mas no permite fazer no momento apropriado, assim como faríamos se tivéssemos instanciando todas as classes manualmente. O mesmo pode ser usado para evitar os problemas de dependência cíclica, que também pode significar um problema de projeto que só aparece quando passamos a utilizar uma gestão de dependência automática.

Lookup programático

Como falamos quando vimos o qualificador `@Any`, ele será muito útil agora quando vamos, programaticamente, buscar as dependências da nossa aplicação.

Através desse qualificador conseguiremos recuperar todas as instâncias candidatas, como podemos ver a seguir.

```
//forma tradicional lança exceção
@Inject @Any
private CalculadoraFolhaPagamento todasCalculadorasFolhaPagamento;

//injeção sob demanda
@Inject @Any
private Instance<CalculadoraFolhaPagamento> todasCalculadorasFolhaLazy;
```

Enquanto na forma tradicional nós teríamos uma exceção ao tentar injetar `CalculadoraFolhaPagamento` usando o qualificador `@Any`, na injeção sob demanda só teríamos a exceção se chamássemos o método `get()`. Mas aqui vai entrar o dinamismo de descobrirmos a instância correta para o momento.

```
@Inject @Any
private Instance<CalculadoraFolhaPagamento> todasCalculadoras;
...
if(simulacao){

    Instance<CalculadoraFolhaPagamento> simuladoras =
        todasCalculadoras.select(new SimuladorQualifier());
}
```

O método `select` da interface `Instance` filtra as possibilidades, agora em vez de três, temos apenas um candidato: `SimuladoraCalculoFolhaPagamento`. Mas vamos analisar melhor alguns pontos.

Há uma sobrecarga do método `select`, nos permitindo filtrar as possibilidade através de qualificadores ou pelo tipo. Utilizamos aqui o filtro pelo qualificador `@Simulador`, mas para fazer isso criamos uma classe chamada `SimuladorQualifier`, como podemos ver a seguir.

```
public class SimuladorQualifier
    extends AnnotationLiteral<Simulador> implements Simulador {

    private PlanoDeCargos planoDeCargos;
    private boolean planoAprovado;
    private boolean enviarNotificacao;

    public SimuladorQualifier(){
```

```

        this(PlanoDeCargos.VERSAO_2005, true, true);
    }

    public SimuladorQualifier(PlanoDeCargos planoDeCargos,
                             boolean planoAprovado, boolean enviarNotificacao){
        ...
    }

    //os métodos da anotação simplesmente devolvem as propriedades
}

```

Como é possível ver, essa classe por padrão especificará o `PlanoDeCargos.VERSAO_2005` no processo de seleção, ou seja, é como se estivéssemos definindo um ponto de injeção como este:

```

@Inject @Simulador(planoDeCargos = VERSAO_2013)
private CalculadoraFolhaPagamento calculadoraFolhaPagamento;

```

Porém como estamos usando um qualificador com propriedades, temos esse grande trabalho para reproduzir essa anotação programaticamente. Se fosse um qualificador simples, sem propriedades, bastaria utilizarmos a classe utilitária `javax.enterprise.util.AnnotationLiteral`. Não precisaríamos dessa classe `SimuladorQualifier`, poderíamos reescrever o exemplo anterior da seguinte maneira.

```

@Inject @Any
private Instance<CalculadoraFolhaPagamento> todasCalculadoras;
...
if(simulacao){

    Instance<CalculadoraFolhaPagamento> simuladoras =
        todasCalculadoras.select(new AnnotationLiteral<Simulador>()){});
}

```

A interface `Instance` nos oferece uma *interface fluente*, permitindo invocações seguidas do método `select`, onde podemos ir a cada invocação refinando mais o filtro dos candidatos a suprir a dependência. Esse mesmo tipo de lookup nós voltaremos a ver na seção 7.4, porém em vez de injetar dependências, estaremos utilizando o lookup para especificar dinamicamente o evento que será disparado.

4.9 RESUMO DO FUNCIONAMENTO DA RESOLUÇÃO DE DEPENDÊNCIAS

Como dito no início do capítulo, essa definição ficou para o final pois para chegar aqui passamos por diversos exemplos, e sem eles esse resumo poderia ficar muito abstrato. Essa será uma seção que talvez valha a pena voltar para relemburar, seja durante a leitura deste livro, ou quando tiver dúvidas nos seus projetos.

Já vimos como funcionam os qualificadores e os beans alternativos. Agora podemos formalizar como a CDI resolve as dependências de uma forma resumida.

Avaliando o tipo

O primeiro elemento a ser considerado é o tipo do objeto. O tipo que está no ponto de injeção, ou seja, aquele que está pedindo a dependência, é o que chamamos de tipo requerido. Serão considerados beans candidatos aqueles que têm tipos compatíveis com o tipo requerido. Se esta for uma interface serão aceitos beans de classes que a implementa e subclasses dessas. Se o tipo requerido for uma classe, serão aceitos beans do mesmo tipo ou de subtipos do requerido. É só imaginar que o teste será feito usando `instanceof`, se a resposta for verdadeira, o bean será elegível.

Quando se tratar de um tipo primitivo, será considerado o seu tipo *wrapper*, ou seja, o tipo de um `int` será `java.lang.Integer` e vice-versa.

Se o tipo requerido for um tipo genérico, como `Dao<X>`, serão aceitos somente beans sem tipagem e tipo base exatamente igual (`Dao`), ou com tipo parametrizado compatível. Por exemplo, `Dao<Y>`, onde `Y` for compatível com (não é o mesmo que subtipo de) `X`. Nos exemplos a frente veremos com mais detalhes. Caso o tipo requerido seja genérico e não esteja tipado, como `Dao`, o bean injetado deverá estar sem tipagem ou tipado com `Object`.

Por exemplo, se tivermos a dependência a seguir:

```
@Inject Dao<Pessoa> dao;
```

Nesse caso serão aceitos beans dos tipos `Dao`, sem tipagem alguma, ou `Dao<Pessoa>`.

Agora vamos considerar que temos também as seguintes classes:

```
public class PessoaFisica extends Pessoa { ... }  
  
public class PessoaDao extends Dao<Pessoa> { ... }
```

Com esse cenário, só seria possível injetar `Dao<PessoaFisica>` se o ponto de injeção definisse `@Inject Dao<? extends Pessoa>`, pois `Dao<PessoaFisica>` não é um tipo compatível com `Dao<Pessoa>`. Agora, no caso em que o coringa foi usado, temos um ponto de injeção compatível pois não limitamos o tipo parametrizado a `Pessoa`, em vez disso incluímos seus subtipos.

Voltando ao ponto de injeção `@Inject Dao<Pessoa> dao`, conseguiríamos também injetar nele uma instância de `PessoaDao`, pois é um tipo compatível.

Já analisando os arrays, só serão considerados se os candidatos forem do mesmo tipo do requerido; não são considerados arrays de subtipos do esperado. Por exemplo, vamos olhar o ponto de injeção a seguir:

```
@Inject Pessoa[] pessoas;
```

Aqui é possível injetar um array do tipo `Pessoa[]`, mas não é possível injetar um array do tipo `PessoaFisica[]`, pois, apesar de `PessoaFisica` ser um tipo compatível com `Pessoa`, os respectivos arrays não são.

Avaliando os qualificadores

Além da tipagem, temos os qualificadores e alternativos como já vimos. Caso o ponto de injeção especifique um qualificador, serão considerados apenas os beans que tenham todos os qualificadores solicitados. Caso não exista um bean com todos os qualificadores solicitados, a CDI lançará um exceção, pois a dependência não pode ser resolvida.

Para todos os efeitos da resolução de dependências, quando uma dependência requer alguns qualificadores, não adianta ter quase todos desses qualificadores, se o bean não possuir ao menos um dos requeridos, ele não será escolhido. Já se ele possuir os qualificadores requeridos e ainda mais, não haverá problema, o que não pode é faltar um qualificador.

Ainda sobre qualificadores, precisamos lembrar de seus atributos, como já vimos. Assim sendo, quando um qualificador é especificado, só serão considerados aceitáveis beans com os mesmos qualificadores, e cada qualificador deve ter **todos** os atributos com os mesmos valores do qualificador requerido. A exceção é quanto aos atributos anotados com `@Nonbinding`, pois esses não são vinculantes.

Avaliando os alternativos

Sempre que tivermos vários beans compatíveis com a dependência requerida, ou seja, tipo e qualificadores aceitáveis, a CDI verifica se dentre eles há algum bean

alternativo. Caso exista, os beans não alternativos são descartados, e somente os alternativos, que naturalmente tem prioridade, serão avaliados.

Dentre os beans alternativos restantes, será escolhido aquele de maior prioridade, como visto na seção 4.4. Caso não haja um bean com prioridade superior, sejam alternativos com mesma prioridade ou quando tivermos apenas beans não alternativos, a CDI lançará uma exceção, pois não terá critérios para escolher entre os elegíveis; é o caso da ambiguidade que já vimos nesse capítulo.

CAPÍTULO 5

O ciclo de vida dos objetos gerenciados pela CDI

Assim como toda ferramenta de alto nível, como JSF e JPA, quando trabalhamos com CDI temos que lidar com o ciclo de vida dos objetos. A partir do momento em que não somos nós que instanciamos os objetos, temos que ter uma forma de interagir com esse processo, ou no mínimo entender como ele funciona.

Desde o capítulo 3 temos visto que para termos um objeto injetável, um bean CDI, basta que ele esteja dentro de um pacote CDI. Não precisamos sequer anotar a classe com algo da CDI. Isso não quer dizer, porém, que não tenhamos uma forma de construir objetos que não estão automaticamente disponíveis no seu contexto. Para isso usaremos os métodos produtores.

Temos também uma forma de liberar alguns recursos quando objetos são destruídos, são como métodos destrutores. E entre a criação e a destruição do objeto temos seu tempo de vida, que é definido pelo escopo, mais um elemento que veremos nesse capítulo.

5.1 MÉTODOS PRODUTORES

Em algumas situações, precisaremos injetar uma classe que não é um bean CDI, como a instância de uma classe de alguma biblioteca pré-existente. Um exemplo disso é o uso de uma biblioteca de log, como a SLF4J (www.slf4j.org). Para tal, vamos voltar ao exemplo da nossa primeira servlet, `IniciandoComCDI`, e trocar nossas escritas no console pelo uso do SLF4J.

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

@WebServlet("/hello-cdi")
public class IniciandoComCDI extends HttpServlet {

    private Logger logger =
        LoggerFactory.getLogger(IniciandoComCDI.class);

    public IniciandoComCDI() {
        //System.out.println("Instanciando a Servlet...");
        logger.info("Instanciando a Servlet...");
    }
    ...
}
```

O uso da API de log é bem simples, como acabamos de ver, mas não seria mais interessante se pudéssemos injetar o `Logger`?

```
@Inject private Logger logger;
```

Nesses casos é que vamos usar os métodos produtores. Novamente temos sorte, pois a implementação do produtor é algo extramente simples.

```
import javax.enterprise.inject.Produces;
public class Produtor {

    @Produces
    public Logger criaLogger(){
        return LoggerFactory.getLogger(IniciandoComCDI.class);
    }
}
```

Bem simples. Basta criarmos um método anotado com `@Produces` que retorne o tipo que queremos produzir. Dessa maneira, quando o ponto de injeção solicitar a dependência, a CDI irá chamar o método produtor para gerar o objeto esperado.

Quando o contexto CDI sobe, as dependências são todas checadas para ver se tem beans que as satisfaçam. É nesse momento que a CDI lança exceções caso a dependência não possa ser satisfeita ou em caso de ambiguidade, que é quando encontra mais de um candidato compatível. No caso do `Logger`, como estamos diante de um tipo que não é parte de um bean CDI, o normal seria ser lançada uma exceção dizendo que a dependência não poderia ser satisfeita, mas como temos o método produtor, a CDI sabe que conseguirá produzir o bean quando for demandada.

Como o método produtor ensina uma nova forma de instanciar um bean, caso criemos um produtor de uma classe que já está dentro do nosso projeto ou de outro pacote CDI, estaremos criando uma ambiguidade, mas no nosso exemplo esse não será um problema.

Porém, se repararmos bem, temos um problema no nosso produtor: ele só produz `Loggers` da classe `IniciandoComCDI`. Serviu para mostrar o mecanismo básico dos produtores, mas obviamente não vai nos servir mais. Para que conseguíssemos resolver esse problema precisaríamos ter uma forma de saber em que ponto estará sendo injetado o bean que estamos produzindo. Para isso temos a interface `javax.enterprise.inject.spi.InjectionPoint`, que nos dá exatamente essa informação.

Entre os métodos dessa interface temos um que devolve onde o objeto está sendo injetado.

```
public interface InjectionPoint {  
    java.lang.reflect.Member getMember();  
    ...  
}
```

O tipo `java.lang.reflect.Member` é uma interface, abstração de um construtor, de uma propriedade ou de um método de uma classe. Como já vimos na seção 3.3, esses são os tipos de ponto de injeção possíveis.

No nosso exemplo, estamos injetando a dependência em uma propriedade, que na API de reflection é do tipo `java.lang.reflect.Field`, uma implementação de `Member`. A partir desse tipo acessamos a classe onde a propriedade está declarada através do método `getDeclaringClass()`, que nos devolve o `java.lang.Class` que possui nossa propriedade, ou seja, o tipo `IniciandoComCDI`.

```
import javax.enterprise.inject.Produces;
public class Produtor {

    @Produces
    public Logger criaLogger(InjectionPoint ip) {
        return Logger.getLogger(ip.getMember()
            .getDeclaringClass().getName());
    }
}
```

Agora temos nosso produtor genérico para poder produzir `Logger` em qualquer lugar do nosso sistema. Porém, pode ser que precisemos de mais informações para criar o bean, por exemplo o usuário logado, para isso basta injetarmos esse objeto no nosso produtor, afinal ele é uma classe normal.

```
public class Produtor {

    private Logger logger = LoggerFactory.getLogger(Produtor.class);

    @Inject
    private UsuarioLogado usuario;

    @Produces
    public Logger criaLogger(InjectionPoint ip) {

        Class classe = ip.getMember().getDeclaringClass();

        logger.info("Criando log para a classe " + classe +
            " e para o usuário " + usuario);

        return Logger.getLogger(classe.getName());
    }
}
```

Nesse caso usamos a dependência extra apenas para gerar um log, mas vimos como injetá-la. Entretanto, temos uma forma ainda mais simples de fazer isso. Apesar de podermos ter métodos produtores em qualquer classe, não é incomum termos uma classe no nosso sistema com a responsabilidade de produzir diversas dependências como o `Logger` que acabamos de ver. Se nessa classe tivermos vários métodos produtores, e alguns deles precisarem de uma ou duas informações extras para gerar o bean, pode ser que a classe no final tenha várias dependências. E isso não é o maior

problema, o que ocorre é que não fica claro qual dependência está sendo usada para produzir qual bean.

Para resolver esse problema, podemos injetar as dependências no próprio método produtor. Se olharmos de novo, veremos que o próprio `InjectionPoint` foi passado para nosso método como uma dependência. Basta seguirmos a mesma lógica para as demais dependências.

```
public class Produtor {  
  
    @Produces  
    public Logger criaLogger(InjectionPoint ip, UsuarioLogado usuario) {  
        ...  
    }  
}
```

Agora, sabemos exatamente quais dependências são necessárias para produzir nosso bean. Além disso, a classe fica mais limpa, e o código mais elegante. E simplicidade e elegância nunca fez mal a código algum.

Como o produtor é uma forma de colocar uma classe comum (por exemplo de biblioteca não CDI) dentro do contexto CDI, podemos especificar no produtor as mesmas coisas que podemos anotar na própria classe, como qualificadores e escopos. O primeiro já é nosso conhecido, mas o segundo será visto ainda nesse capítulo.

```
public class Produtor {  
  
    @Produces @QualificadorX @QualificadorY(VALOR_Z)  
    public Logger criaLogger(InjectionPoint ip, UsuarioLogado usuario) {  
        ...  
    }  
}
```

Propriedade produtora

Já vimos que os métodos produtores podem estar presentes em qualquer bean, mas além de um método, podemos ter uma propriedade anotada com `@Produces`. As regras são basicamente as mesmas dos métodos, a diferença é que caso a propriedade produtora esteja definida dentro de um EJB do tipo Session Bean, esta deverá obrigatoriamente ser estática.

Um exemplo do uso dessa funcionalidade é colocarmos no contexto CDI objetos do Java EE.

```
public class ProdutorJava EE {

    @PersistenceContext(unitName="organizacionalDatabase")
    @Produces @OrganizacionalDatabase EntityManager organizacionalEM;

    ...
}
```

Nesse exemplo, nós usamos a forma Java EE de injetar o `EntityManager` do banco de dados que chamamos de “organizacional”. Ao mesmo tempo produzimos esse mesmo objeto com o qualificador `@OrganizacionalDatabase`. Com isso, as demais classes do sistema podem simplesmente injetar dessa forma:

```
@Inject @OrganizacionalDatabase
EntityManager organizacionalEntityManager;
```

A vantagem dessa tradução — da forma Java EE para a CDI — é que podemos usar apenas a forma CDI de injetar dependências, e com isso rodar nossa aplicação também em ambientes não Java EE, como o Tomcat. Nesse último ambiente poderíamos ter um método produtor como o que segue.

```
public class ProdutorJavaSE {
    @Produces @OrganizacionalDatabase
    public EntityManager criaEntityManager(){
        EntityManagerFactory emf = Persistence
            .createEntityManagerFactory("organizacionalDatabase");

        return emf.createEntityManager();
    }
    ...
}
```

Agora, com base no que já aprendemos sobre alternativas, poderíamos marcar ambos produtores: `ProdutorJava EE` e `ProdutorJavaSE` como `@Alternatives` e habilitar no `beans.xml` apenas o correspondente ao ambiente que estivermos instalando nossa aplicação.

Restringindo o tipo do bean produzido

Tanto nos métodos quanto nas propriedades produtoras, podemos utilizar a mesma anotação `@Typed` vista na seção 4.7. Dessa maneira o bean produzido não será candidato a suprir dependências de todos os tipos possíveis, mas somente dos especificados na anotação e `java.lang.Object`.

```
@Produces @Typed(CalculadoraDeSalarios.class)
public CalculadoraDeSalariosPlano2013 criaCalculadora(){
    return new CalculadoraAssincronaDeSalariosPlano2013();
}
```

5.2 ESCOPOS DA CDI

Como estamos falando do ciclo de vida, chegou a hora de falarmos dos escopos, meio pelo qual definimos o tempo de vida de um bean. Até agora, nossos exemplos têm sido baseado em servlets, que serviram até aqui por serem extremamente simples, mas para falarmos de escopos, precisaremos de algo mais flexível, pois as servlets são objetos criados uma única vez e vivem enquanto a aplicação estiver rodando. Como acompanham o ciclo de vida da aplicação, é como se sempre fossem do escopo *application*. Digo como se fossem pois não especificamos escopos em servlets.

Pra explorar todos os escopos possíveis, pararemos de usar as servlets e usaremos JSF. Não é objetivo deste livro ensinar JSF, mas o usaremos para exemplificar os escopos. Você pode fazer uso de CDI com outros frameworks, mas a escolha do JSF é por ser parte do Java EE, assim como o CDI. Para saber mais sobre JSF, e também sobre JPA, recomendo o livro “Aplicações Java para a web com JSF e JPA” da editora Casa do Código [?].

Antes de entrar em um escopo específico, vamos fazer uma tradução básica de uma de nossas servlets para um bean JSF. Iniciaremos pela servlet `IniciandoComCDI`, e agora que já passamos pelo *hello world*, poderemos dar um significado melhor para essa classe.

O código atual dela é o seguinte:

```
@WebServlet("/hello-cdi")
public class IniciandoComCDI extends HttpServlet {

    @Inject
    private CalculadoraDeImpostos calculadoraImpostos;

    public IniciandoComCDI() {
        System.out.println("Instanciando a Servlet...");
    }

    @PostConstruct
    public void ok(){
```

```

        System.out.println("Servlet pronta.");
    }

    protected void doGet(HttpServletRequest req,
                        HttpServletResponse res)
                        throws ServletException, IOException {

        double salarioBase =
            Double.parseDouble(req.getParameter("salario"));

        Funcionario funcionario = new FuncionarioBuilder()
            .comSalarioBaseDe(salarioBase)
            .build();

        System.out.println("Efetuando o cálculo.");

        //a calculadora de IR usa outra classe para calcular o salário
        double imposto = calculadoraImpostos
            .calculaImpostoDeRenda(funcionario);

        res.getOutputStream().print(
            String.format("Salario base: R$ %.2f\n" +
                "Imposto devido: R$ %.2f", salarioBase, imposto));
        System.out.println("Fim.");
    }
}

```

O primeiro passo para mudarmos nossa servlet, é transformá-la em uma classe Java simples, também conhecida como POJO (Plain Old Java Object), ou seja, a boa e velha classe Java, que não precisa implementar interfaces ou estender classes de algum framework. Agora, como nossa classe não é mais uma servlet, o método `doGet` vai virar um método simples, que renomearemos para algo significativo, como `calculaImposto`. E aproveitando a mudança de nome, vamos dar também um nome significativo para a classe:

```

public class CalculadoraImpostosBean {

    @Inject
    private CalculadoraDeImpostos calculadoraImpostos;
}

```

```
public CalculadoraImpostosBean() {
    System.out.println("Instanciando a CalculadoraImpostosBean...");
}

@PostConstruct
public void ok(){
    System.out.println("CalculadoraImpostosBean pronta.");
}

public void calculaImposto(HttpServletRequest req,
                           HttpServletResponse res)
    throws ServletException, IOException {

    double salarioBase =
        Double.parseDouble(req.getParameter("salario"));

    Funcionario funcionario = new FuncionarioBuilder()
        .comSalarioBaseDe(salarioBase)
        .build();

    System.out.println("Efetuando o cálculo.");

    //a calculadora de IR usa outra classe para calcular o salário
    double imposto = calculadoraImpostos
        .calculaImpostoDeRenda(funcionario);

    res.getOutputStream().print(
        String.format("Salario base: R$ %.2f\n" +
            "Imposto devido: R$ %.2f", salarioBase, imposto));
    System.out.println("Fim.");
}
}
```

Estamos mudando a classe aos poucos: por enquanto mudamos basicamente o nome da classe, inclusive nas mensagens que saiam no console. Agora vamos arrumar a assinatura do método `calculaImposto`, que ainda está com cara de servlet. Colocaremos uma propriedade na nossa `CalculadoraImpostosBean` que armazenará a informações que antes vinha do objeto `req`, do tipo `HttpServletRequest`.


```

public class CalculadoraImpostosBean {

    //o resto permanece igual ...

    private double salarioBase; //getters e setters
    private double imposto; //getters e setters

    public void calculaImposto() {

        Funcionario funcionario = new FuncionarioBuilder()
                                .comSalarioBaseDe(salarioBase)
                                .build();

        System.out.println("Efetuando o cálculo.");

        imposto = calculadoraImpostos.calculaImpostoDeRenda(funcionario);

        System.out.println("Fim.");
    }
}

```

No nosso novo código o imposto é calculado e armazenado em uma propriedade do bean, pois no mundo real nossas classes de negócio apenas efetuam a lógica da aplicação, elas não são responsáveis pela apresentação dos resultados ao usuário. Para isso criaremos uma página JSF simples, que apenas terá um campo para a entrada do salário e no final apresentará o valor do imposto.

```

<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://xmlns.jcp.org/jsf/html">
    <h:form>
        <h:panelGrid columns="2">
            Salário base:
            <h:inputText value="#{calculadoraImpostosBean.salarioBase}"/>

            Imposto Calculado:
            <h:outputText value="#{calculadoraImpostosBean.imposto}"/>

            <h:commandButton value="Calcular Imposto"
                            action="#{calculadoraImpostosBean.calculaImposto()}/>
        </h:panelGrid>

```

```
</h:form>
</html>
```

Mas antes de executar essa página, precisamos dar um nome ao nosso bean, como vimos na seção 4.5. Para isso temos que fazer uma última alteração no nosso bean:

```
@Named
public class CalculadoraImpostosBean { ... }
```

Na página que criamos temos o componente `h:panelGrid` que serve apenas para fazer a formatação da página. Com ele, especificamos a quantidade de colunas que desejamos ter. Como especificamos duas colunas, significa que a cada dois componentes — no exemplo a label e o componente JSF — será criada uma nova linha. Assim não nos preocupamos com o alinhamento, e a saída já ficará igual à seguinte:

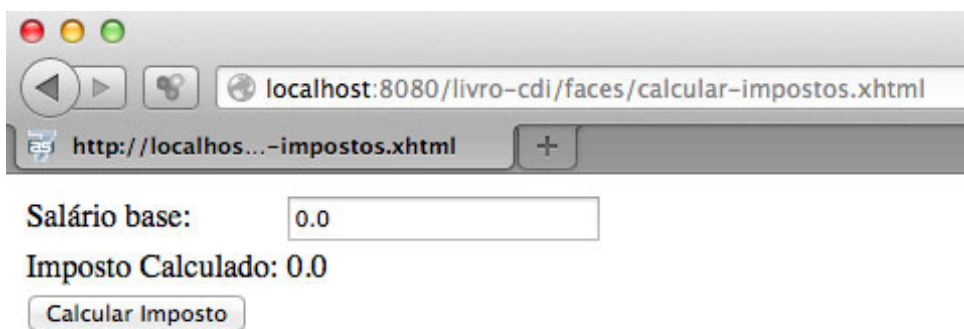


Figura 5.1: Formulário da calculadora de impostos feita com JSF

Os demais componentes são de input e output simples, além do botão que executa a ação, chamando o método ligado à propriedade `action`.

Se executarmos nossa aplicação como está, veremos que o valor apresentado no imposto estará zerado, mas isso porque não especificamos nenhum escopo. Em vários dos nossos beans realmente não especificaremos nenhum escopo e isso não será um problema, porém sem passar pelos outros será mais difícil entender esse escopo *default*. Por isso, voltaremos a falar dele na seção 5.7.

5.3 ESCOPO DE REQUISIÇÃO COM @REQUESTSCOPED

O escopo *request* é o mais comum em uma aplicação web, pois toda interação entre o browser e o servidor é um request. Esse escopo define um ciclo de vida que se inicia no momento em que a requisição do usuário chega no servidor, e dura todo o processamento e também todo o processo de geração da resposta para o usuário. Somente quando a resposta para o usuário é terminada que o request termina.

Parece simples, mas é preciso ficarmos atentos principalmente a essa última parte: a geração da resposta para o usuário. Não adianta acompanharmos a execução passo a passo de toda a lógica da aplicação e achar que está tudo certo, pois se algo ocorrer depois do processamento, mas antes da geração da resposta terminar, ainda podemos ter problema, pois o ciclo como um todo é que importa.

Uma forma de tentarmos ilustrar o funcionamento desse escopo é através da figura a seguir:

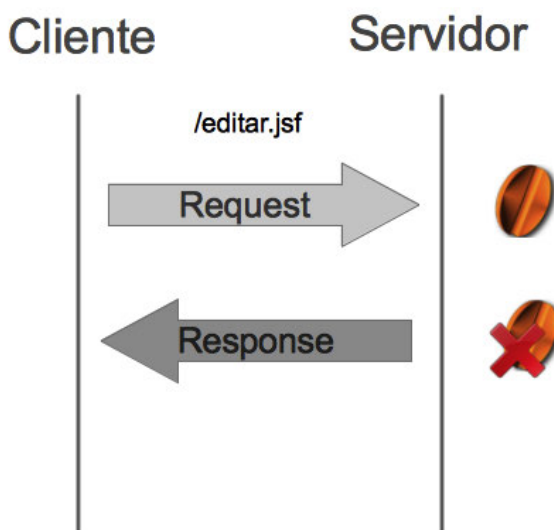


Figura 5.2: Ilustração do escopo request

Na figura exemplificamos a chamada da URL `/editar.jsf`. Como já foi dito, o bean é criado no momento em que a requisição chega no servidor, e termina no

momento em que sai dele. Novamente precisamos nos atentar, pois a resposta é gerada toda no servidor. Às vezes, em uma olhada rápida, podemos pensar que o request terminaria na chegada da resposta ao browser, mas essa volta nada mais é do que uma transferência pela rede de um HTML pronto.

Retornando ao nosso exemplo, basta colocarmos a anotação do escopo request que as coisas funcionarão:

```
import javax.enterprise.context.RequestScoped;
import javax.inject.Named;

@Named @RequestScoped
public class CalculadoraImpostosBean { ... }
```

A cada clique no botão "Calcular Imposto" estaremos disparando uma nova requisição. A cada requisição, será criada uma nova instância de `CalculadoraImpostosBean`, que receberá suas dependências e calculará o imposto. Depois o JSF, assim como faria uma página JSP, acessa o bean para buscar os valores que precisa para montar a tela de resposta. Após terminar a geração da resposta, o request termina, e como todos esses beans estão nesse escopo, nesse momento eles são retirados da memória.

Como este é o primeiro escopo que estamos vendo, vamos ver mais uma ilustração que nos auxilia a compreender esse ciclo:

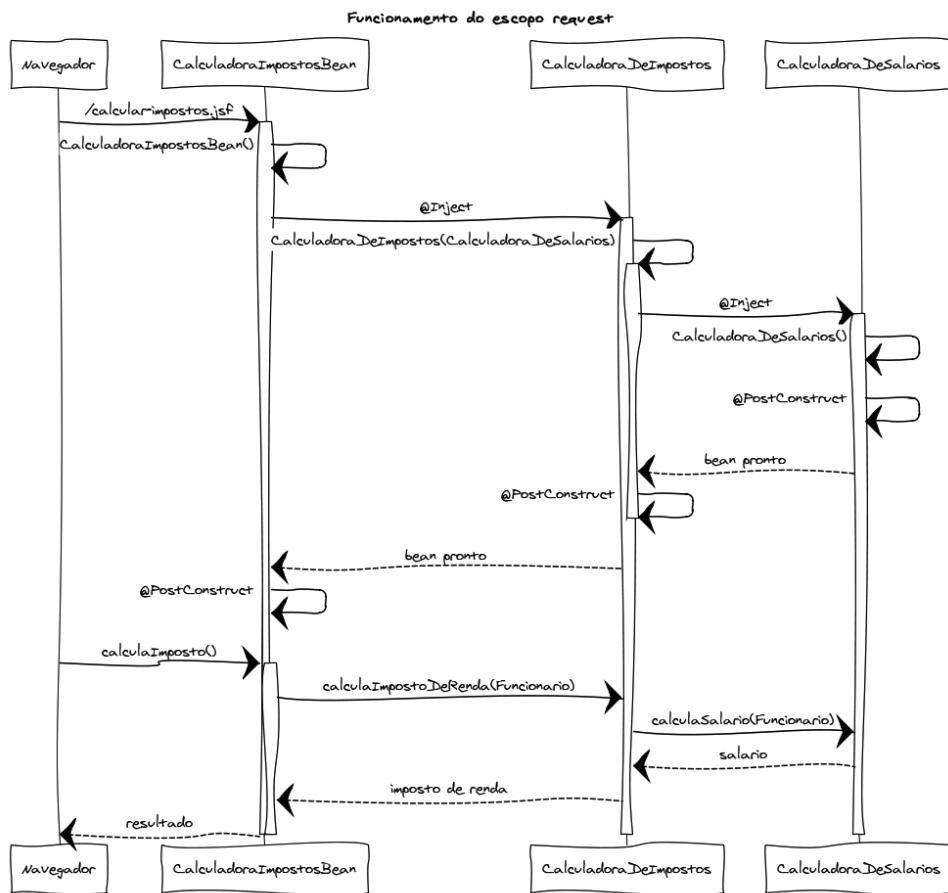


Figura 5.3: Ciclo da requisição do cálculo de imposto

Podemos dividir essa imagem em duas partes, a primeira mostra o processo de criação do bean, parecido com o que fizemos na seção 3.1, mas naquele momento usamos a análise de console, agora estamos vendo graficamente. No nosso caso temos três beans envolvidos `CalculadoraImpostosBean` -> `CalculadoraDeImpostos` -> `CalculadoraDeSalarios`, e nossa figura ilustra essa dependência.

Na segunda metade, temos a chamada do método `calculadoraImpostosBean.calculaImposto()` pelo botão da nossa tela. E depois da resposta entregue ao navegador, os beans são removidos da

memória pois o escopo — request — termina.

Se desconsiderarmos a chamada do método e analisarmos apenas o processo de criação dos beans, e ao final a seu descarte, estaremos analisando exatamente o mesmo funcionamento independentemente do escopo. Conforme formos analisando os demais escopos isso ficará mais claro.

5.4 ESCOPO DE SESSÃO COM O @SESSIONSCOPED

O processo de criação e descarte de um bean não muda se o escopo é request ou session, o que muda é quando o processo ocorre. Enquanto o request compreende cada invocação ao servidor feita pelo cliente, a sessão engloba todas as requisições de um mesmo cliente. Para facilitar, é só imaginar o processo de autenticação em uma aplicação web. Enquanto tivermos com o browser aberto, estaremos logados na aplicação, isso porque as informações de autenticação costumam ficar no escopo de sessão.

Cada vez que acessamos uma aplicação pela primeira vez, com um determinado navegador, é criada uma nova sessão. Mas isso não cria automaticamente os beans de escopo sessão, estes geralmente são criados quando utilizados pela primeira vez, e duram enquanto a sessão existir. Enquanto conseguimos perceber pelo console que, no escopo request, os objetos são recriados a cada solicitação; com o escopo sessão, só perceberemos essa criação uma vez para cada browser utilizado.

Geralmente a sessão é mantida através de um cookie ou então algum parâmetro na url que é gerado pelo servidor, assim conseguimos abrir mais de uma sessão se usarmos browsers diferentes ou a chamada sessão anônima suportada pelos navegadores mais modernos.

Para testar a criação sessões, é possível fechar o browser e abrir novamente pra criar uma nova sessão, mas isso não significa que a antiga será fechada. Geralmente o que ocorre quando um usuário fecha o browser sem usar alguma opção “sair” da aplicação, é que a sessão ficará inativa no servidor, ocupando recursos desnecessariamente, até que o *timeout* seja alcançado. Isso geralmente ocorre após trinta minutos de inatividade.

A seguir uma imagem para ilustrar esse funcionamento.

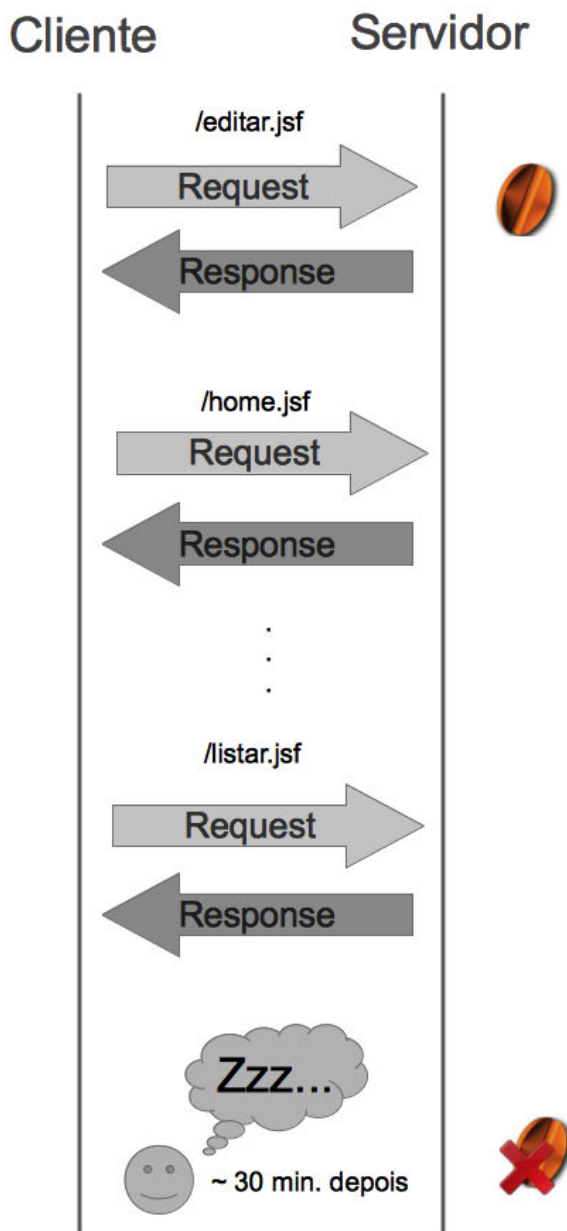


Figura 5.4: Ilustração do escopo de sessão

No desenvolvimento de aplicações reais, muitas vezes surgem funcionalidades cujos estados são mais difíceis de gerenciar do que uma simples requisição, e o primeiro impulso pode ser mudar o escopo do bean para sessão. Esse procedimento resolve o problema em tempo de desenvolvimento, mas cria um novo problema em tempo de execução.

Simplesmente passar um bean para escopo de sessão faz com que a aplicação passe a consumir recursos de forma exagerada. Recursos que deveriam ser mantidos por poucas requisições acabam ficando na memória enquanto o usuário estiver logado, um desperdício. A melhor prática na programação como um todo, é sempre usar o menor escopo que resolva o problema, e quando falamos de escopo isso é ainda mais importante. Antes de mudar o escopo de um bean de request para session por este primeiro parecer pequeno demais para a funcionalidade, olhe o escopo de conversação na seção 5.6.

Como o exemplo das informações de autenticação do usuário, o escopo de sessão serve para armazenar informações que devem estar em memória durante toda a navegação do usuário na aplicação. Outros exemplos podem ser as permissões do usuário, informações de perfil de acesso, nome, hora do login, entre outras que possam ser interessantes dependendo da aplicação.

5.5 @APPLICATIONSCOPED: O MAIOR ESCOPO POSSÍVEL

O escopo de aplicação não foge do funcionamento padrão, a dinâmica é a mesma, a diferença está em sua amplitude. Enquanto o escopo request dura uma única requisição de um determinado usuário, e sessão engloba toda a utilização do sistema por um usuário, a aplicação envolve todos usuário da aplicação. Assim, se colocarmos algo nesse escopo, estará disponível para qualquer usuário que acessar a aplicação.

Um exemplo seria a funcionalidade de listar todos os usuários que estão online, e quem sabe até um chat entre eles. É um escopo que cruza todas as sessões, então se temos que tomar cuidado ao por algo no escopo sessão, temos que redobrar isso no escopo aplicação.

Ainda como exemplo de aplicações para esse escopo, poderíamos pensar em listagens fixas, como estados e municípios. Esse pode ser um uso legítimo, pois realmente é algo comum a qualquer usuário, mas existem outras opções. Nesse caso, podemos utilizar um cache de segundo nível na camada de persistência, algo da JPA, e não da CDI. Uma boa maneira de desenvolver aplicações é procurar sempre usar a solução correta para cada problema. Muitas vezes colocar algo no escopo aplicação

pode resolver, mas se puder utilizar uma solução de cache, terá nas mãos uma opção mais robusta.

Para ilustrar o funcionamento desse escopo temos a seguinte imagem:

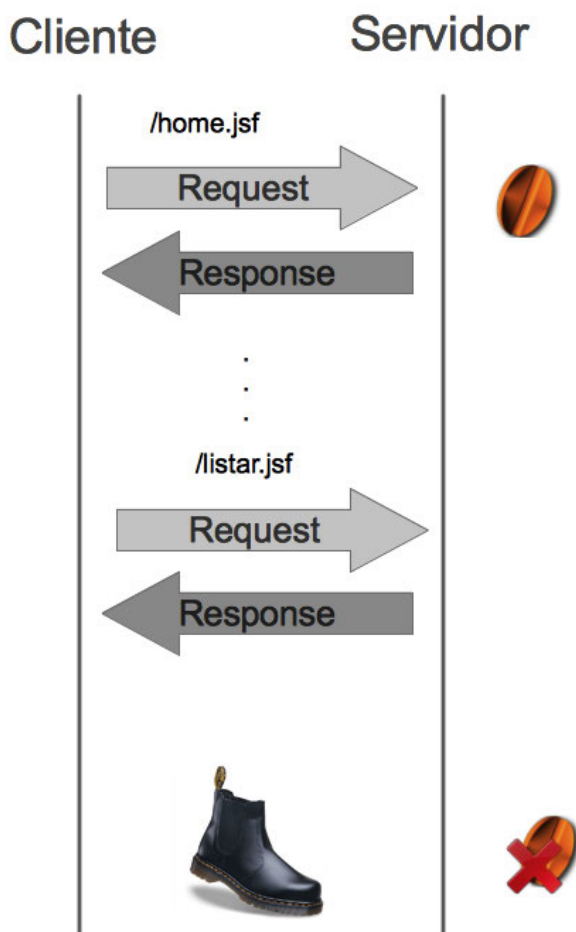


Figura 5.5: Ilustração do escopo de aplicação

Como vimos, o escopo só termina quando baixamos a aplicação, então colo-

cando algo aqui, estaremos adicionando algo na memória que ficará lá por muito tempo.

5.6 @CONVERSATIONSCOPED: VOCÊ CRIANDO SEU PRÓPRIO ESCOPO

O escopo de conversação, é um pouco diferente dos anteriores porque ele possui dois comportamentos distintos. Não basta anotar o bean com esse escopo, temos que escolher quando passar a conversação de um estado para o outro.

Uma conversação é um escopo inteligente, que por padrão se comporta como o escopo request, perdendo seu estado a cada requisição. O interessante é que podemos escolher um dado momento para que o escopo se torne persistente, como se fosse o escopo de sessão. Porém, enquanto este permanece na memória por muito tempo, podemos dizer à conversação que ela volte a se comportar como o escopo request, então os dados nela armazenados são descartados ao final da próxima requisição, e ela volta a se comportar como o request.

O estado padrão, parecido com request é chamado de *transient*, que é quando os estados não são mantidos. Já invocando o método `Conversation.begin()` nossa conversação passa para o estado *long-running*, e este mantém os dados até que o método `Conversation.end()` seja chamado, quando volta a ser *transient* e os dados deixam de ser armazenados. A imagem a seguir ilustra esse funcionamento.

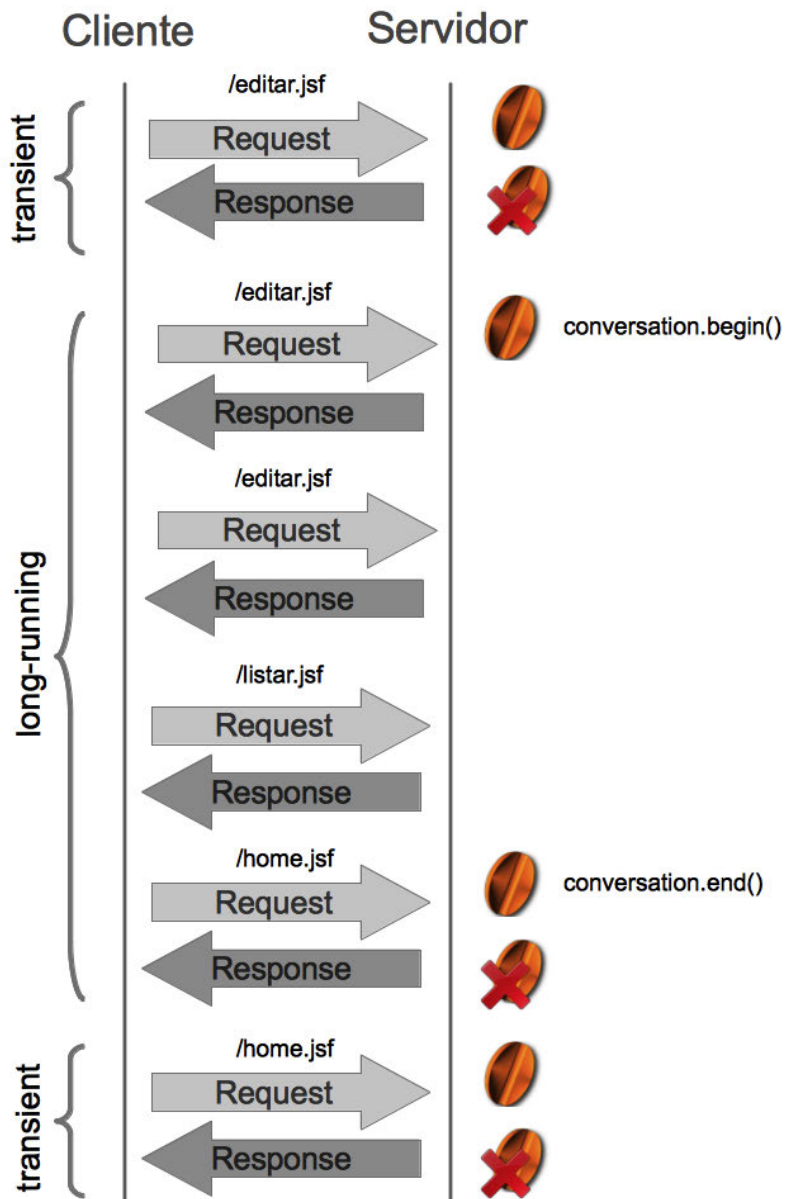


Figura 5.6: Ilustração do escopo de conversação

Vamos analisar como implementamos isso no nosso código. Como já foi dito, no caso da conversação não basta anotar o bean, temos que definir quando ela passar de um estado para outro. Criaremos no nosso sistema a funcionalidade de calcular uma folha de pagamento baseado em uma lista de funcionários que vamos adicionando um a um. Nosso exemplo será mais simples do que seria um exemplo real, e para simplificar, em vez de usarmos banco de dados, usaremos a classe `FuncionarioBuilder`.

```
import javax.enterprise.context.Conversation;
import javax.enterprise.context.ConversationScoped;

@ConversationScoped @Named
public classe CalculadoraFolhaBean implements Serializable {

    @Inject
    private Conversation conversation;

    @Inject
    private CalculadoraFolhaPagamento calculadoraFolha;

    public void iniciaConversacao(){
        if(conversation.isTransient()){
            conversation.begin();
        }
    }

    public void terminaConversacao(){
        if(!conversation.isTransient()){
            conversation.end();
        }
    }

    ...
}
```

O primeiro passo é criarmos um bean nomeado para que seja acessível da nossa página JSE. Como acabamos de ver, não basta anotar o bean com o escopo, precisamos injetar uma instância da interface `javax.enterprise.context.Conversation` e utilizá-la para gerenciar os estados da conversação. Além de iniciar e finalizar a conversação (alternar entre

transient e long-running), temos o método `setTimeout(long milliseconds)`, que usamos para mudar o timeout dessa conversação específica. Injetamos também uma instância de `CalculadoraFolhaPagamento` que é quem sabe fazer o cálculo que precisamos.

É importante também que nosso bean, e todos os beans injetados, como `CalculadoraFolhaPagamento`, implementem a interface `Serializable`. Sem isso será lançada uma exceção durante a inicialização da aplicação.

Criaremos uma tela simples, onde podemos informar o salário de um funcionário e adicioná-lo na lista através de um botão. Teremos um outro botão para efetuar o cálculo e outros dois para iniciar e terminar a conversação. Além desses componentes, escreveremos a quantidade de funcionário que já estão na lista, e, assim, vamos percebendo a alteração de estado do nosso bean antes de cálculo. O código da tela pode ser visto a seguir.

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://xmlns.jcp.org/jsf/html">
  <h:form>
    <h:panelGrid columns="2">
      Salário base: <h:inputText
        value="#{calculadoraFolhaBean.salarioFuncionario}"/>
      Funcionário adicionados: <h:outputText
        value="#{calculadoraFolhaBean.funcionarios.size()}/>
      Valor da Folha: <h:outputText
        value="#{calculadoraFolhaBean.folha != null ?
          calculadoraFolhaBean.folha.valor : 'Não calculado'}/>

      <h:commandButton value="Inicia Conversação"
        action="#{calculadoraFolhaBean.iniciaConversacao()}/>
      <h:commandButton value="Termina Conversação"
        action="#{calculadoraFolhaBean.terminaConversacao()}/>
      <h:commandButton value="Adicionar Funcionário"
        action="#{calculadoraFolhaBean.adicionaFuncionario()}/>
      <h:commandButton value="Calcular Folha!"
        action="#{calculadoraFolhaBean.calcularFolha()}/>
    </h:panelGrid>
  </h:form>
</html>
```

Nossa tela, quando executando, se parecerá com essa:

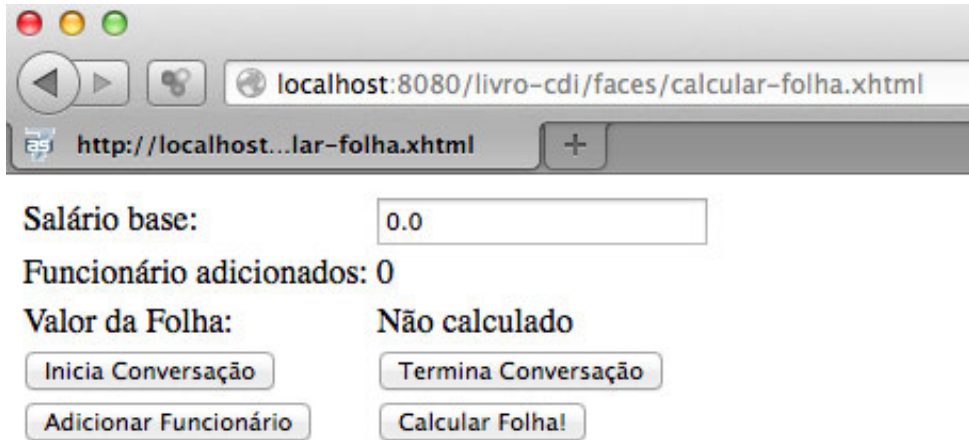


Figura 5.7: Tela de cálculo de folha

Porém ainda precisamos do restante da implementação da classe `CalculadoraFolhaBean`:

```
@Named @ConversationScoped
public class CalculadoraFolhaBean implements Serializable{

    ...

    private FuncionarioBuilder builder;
    private Folha folha; //getter
    private List<Funcionario> funcionarios; //getter
    private double salarioFuncionario; //getter e setter

    @PostConstruct
    public void init(){
        builder = new FuncionarioBuilder();
        funcionarios = new ArrayList<>();
    }

    public void adicionaFuncionario(){
        Funcionario funcionario = builder
            .comSalarioBaseDe(salarioFuncionario).build();
        getFuncionarios().add(funcionario);
    }
}
```

```

    }

    public void calcularFolha(){
        folha = calculadoraFolha.calculaFolha(getFuncionarios());
    }
}

```

E como até agora não tínhamos visto a implementação da `CalculadoraFolhaPagamentoReal`, veremos agora:

```

public class CalculadoraFolhaPagamentoReal implements
    CalculadoraFolhaPagamento, Serializable {

    @Inject
    private CalculadoraDeSalarios calculadoraDeSalarios;

    @Override
    public Folha calculaFolha(List<Funcionario> funcionarios) {
        double valor = 0.0;
        for (Funcionario funcionario : funcionarios) {
            valor += calculadoraDeSalarios.calculaSalario(funcionario);
        }
        return new Folha(new Date(), valor, funcionarios);
    }
}

```

Nossa calculadora está pronta para ser testada. Experimente digitar o salário do funcionário no campo correspondente e acione o botão "Adicionar Funcionário". Altere o valor do salário e acione o botão novamente. Você perceberá que sem iniciar a conversação, nunca conseguirá sair de um único funcionário adicionado.

Se observar a imagem 5.6, perceberá que antes de iniciar a conversação, o bean morre logo após o término da requisição. Assim, por mais que a tela apresente a informação de que a lista já contém um `Funcionario`, no servidor o bean `CalculadoraFolhaBean` já foi destruído, e quando acionar o botão "Adicionar Funcionário" mais uma vez, estaremos na verdade recriando todos os objetos, e adicionando neles o primeiro `Funcionario` novamente.

A partir do momento em que clicamos no botão "Inicia Conversação", transformamos a conversação em *long-running conversation*, e com isso a

`CalculadoraFolhaBean` não morre mais após o término da requisição. Nos próximos acionamentos desse botão perceberemos que a contagem de funcionários vai aumentando, e podemos finalmente usar o botão `"Calcular Folha!"`.

Experimente também utilizar o botão `"Termina Conversação"`, e perceba que na próxima ação nossa aplicação se comportará novamente como se fosse a primeira execução. Isso porque, ao transformarmos uma conversação *long-running* em *transient*, fazemos com que o bean `CalculadoraFolhaBean` volte a ser descartado ao final de cada requisição.

Nesse exemplo utilizamos um único bean e uma única tela para facilitar o entendimento, mas uma conversação pode durar por várias telas e diferentes bean dentro da nossa aplicação. Realmente esse é um escopo bastante versátil.

Além desse escopo ter o modo “liga-desliga” de armazenamento de dados, ainda que esqueçamos de finalizar a conversação, ela não será como a sessão que guarda os valores por um longo tempo, por exemplo, trinta minutos se nada for feito. No caso da conversação, mesmo que esteja no estado *long-running*, os dados são descartados quando a conversação atinge seu *timeout*. A vantagem é que esse timeout não coincide com o da sessão, por padrão é de apenas dois minutos, economizando recursos mais do que a sessão, ainda que esqueçamos de finalizar a conversação.

Pode parecer uma boa opção diminuir o tempo da sessão para economizar recursos, mas isso nos leva a efeitos indesejados. Se deixarmos um tempo muito pequeno, podemos derrubar a sessão de usuários que estão lendo algum texto na tela do sistema, e isso pode causar certo desconforto. Mesmo que você tenha outros meios, como cookies, para manter a autenticação do usuário, certamente perderá outras informações que o usuário possar ter preenchido pouco antes iniciar a leitura do texto.

Novamente devemos buscar a ferramenta certa para cada problema, e na maioria das situações veremos que a conversação é um escopo mais apropriado para as funções da nossa aplicação do que a sessão.

Quando estivermos vendo um pouco sobre CDI juntamente com EJBs, veremos que esse escopo traz outras vantagens, como a capacidade de manter aberto o contexto de persistência, `EntityManager`, durante toda a conversação. Com isso temos a vantagem do cache de primeiro nível sempre à mão, mantendo os objetos persistentes sempre gerenciados. Mesmo EJBs não sendo objeto deste livro, veremos um pouco mais sobre isso quando estivermos apreciando a integração da CDI com os demais elementos do Java EE.

5.7 @DEPENDENT: O ESCOPO PADRÃO DO CDI

O escopo dependente foi o primeiro que utilizamos nos nossos exemplos, e continuamos a utilizá-lo o tempo todo. Esse é o padrão, aquele atribuído ao bean quando não especificamos outro. Para nossa análise, vamos utilizar o mesmo exemplo do escopo de conversação, onde temos o bean `CalculadoraFolhaBean` e a `CalculadoraFolhaPagamento`. Enquanto o primeiro tem o de conversação, o segundo, que não especifica nenhum explicitamente, tem escopo dependente.

Agora que temos um exemplo, o entendimento desse escopo é simples. Vimos que a `CalculadoraFolhaPagamento` tem escopo dependente, mas dependente de quem? Do `CalculadoraFolhaBean`, que foi quem solicitou a injeção do bean dependente. Assim, se nosso bean não especificar um escopo, ele seguirá o de quem o solicitar.

Antes de iniciarmos a conversação longa, tanto o `CalculadoraFolhaBean` quanto o `CalculadoraFolhaPagamento` estavam funcionando com escopo de conversação em modo transiente. Após iniciarmos a conversação, ambos passaram para *long-running conversation*. Quando um é criado, o outro também, e o mesmo ocorre no caso da destruição do bean. Na imagem a seguir, temos uma ilustração de como o escopo dependente se comporta quando estiver dependendo de um escopo request.

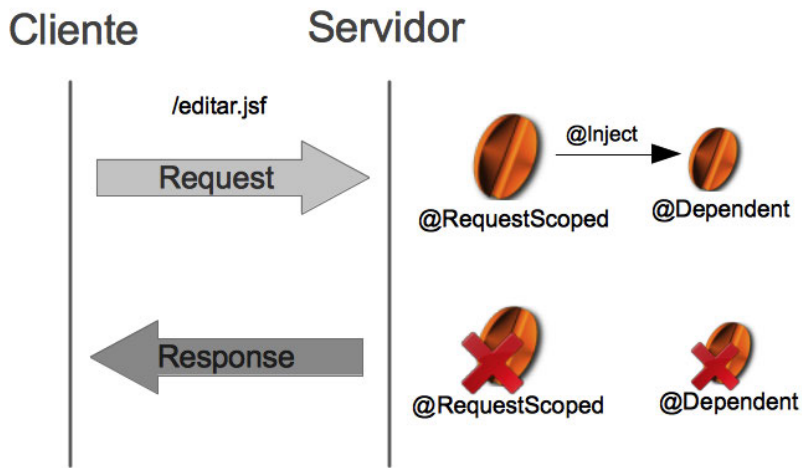


Figura 5.8: Ilustração do escopo dependente junto ao escopo request

E agora uma imagem que mostra o escopo dependente sendo utilizado por um bean com escopo de sessão. A dinâmica é a mesma de quando é utilizado por um bean com escopo request. O dependente sempre segue o ciclo de vida do bean principal.

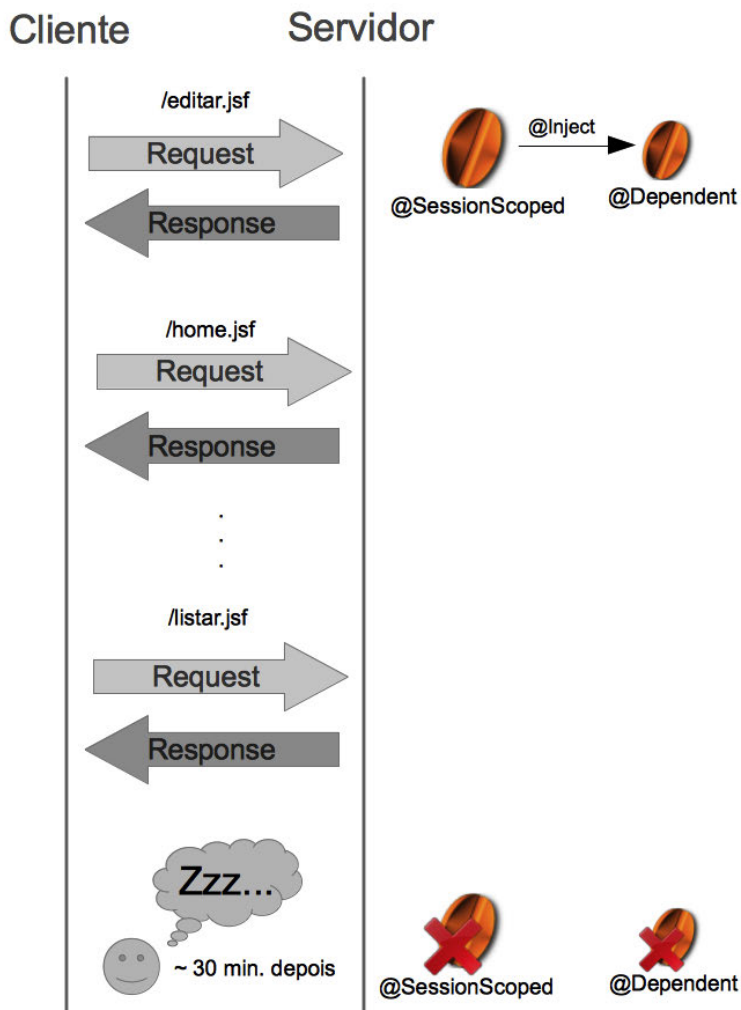


Figura 5.9: Ilustração do escopo dependente junto ao escopo de sessão

Onde definir os escopos e quais as restrições?

Não existe uma regra sobre onde definir um escopo, mas o mais comum é definirmos nos beans nomeados, que serão acessados pelas nossas páginas JSF ou JSP, e estes usam os beans com escopo dependente, e são buscados via tipo e não por nome.

Nada impede no entanto que beans não nomeados tenham escopos definidos.

Uma restrição que temos quanto a escopos, é que beans só acessam outros com escopo igual ou maior que o seu. Por exemplo, não tem como um bean `@ApplicationScoped` injetar um bean `@RequestScoped`, porém o contrário é possível.

Também não há restrição que impeça deixarmos um bean nomeado com escopo dependente, mas se olharmos a primeira versão do nosso bean `CalculadoraImpostosBean` definido na seção 5.2 veremos que ela não funcionava. Isso porque o bean estava sendo acessado diretamente pela página JSF mas não tinha um escopo próprio, era dependente. Assim, cada vez que a página tentava acessar o bean, ou a cada EL que chamava seu nome, era criado um novo bean. Então, é obvio que que não funcionaria, pois o valor informado pelo usuário acabaria descartado logo em seguida.

5.8 MÉTODOS FINALIZADORES

Quando um objeto é criado pelo próprio container, por exemplo o servidor de aplicações JBoss ou Glassfish, este tem a responsabilidade de gerenciar também o descarte desse objeto. Porém quando nós mesmos o criamos e o inserimos no contexto CDI através de um método ou propriedade produtora, nós temos que cuidar para que o objeto seja retirado da memória de forma correta.

Um exemplo é a produção de um `EntityManager` como vimos na seção 5.1. Quando injetamos uma instância usando a anotação `@PersistenceContext`, é porque o servidor de aplicações gerencia esse objeto para nós. Assim sendo, não precisamos chamar o método `EntityManager.close()` pois o servidor faz isso. Agora se o criamos manualmente usando `Persistence.createEntityManagerFactory.criaEntityManager`, nós é que temos a responsabilidade de fechá-lo.

Para fazer isso, basta definirmos um método e anotar a propriedade que queremos fechar com `@Disposes`:

```
import javax.enterprise.inject.Disposes;

public class ProdutorJavaSE {

    @Produces @OrganizacionalDatabase
    public EntityManager criaEntityManager(){
```

```

        EntityManagerFactory emf = Persistence
            .createEntityManagerFactory("organizacionalDatabase");

        return emf.createEntityManager();
    }

    public void fechaEntityManager(
        @Disposes @OrganizacionalDatabase EntityManager em){

        em.close();
    }
}

```

O processo de vinculação do método *disposer* ao objeto que será descartado segue a mesma política da resolução de dependência para os pontos de injeção. Perceba que o método finalizador parece um método inicializador, que é anotado com `@Inject` e em cada parâmetro declara uma dependência. A diferença é que, em vez de anotar o método, anotamos a propriedade que queremos destruir com `@Disposes`, e as demais anotações e o tipo declarado servem para especificar os objetos que podem ser finalizados através do nosso método.

Assim como no método inicializador, podemos declarar mais parâmetros no método finalizador, a diferença é que somente um deles será referente ao objeto que será destruído, os demais serão pontos de injeção.

```

...
public void fechaEntityManager(
    @Disposes @OrganizacionalDatabase EntityManager em,
    Logger logger){

    logger.info("Fechando EntityManager");
    em.close();
}
...

```

Nesse exemplo o parâmetro `Logger logger` é um ponto de injeção assim como podemos ter no método produtor, e serve para auxiliar no processo de finalização do objeto. Poderia inclusive ter qualificadores como qualquer ponto de injeção. No entanto se mais de um parâmetro estiver anotado com `@Disposes`, ou se a CDI encontrar mais de um método finalizador para o mesmo bean, será lançado um erro durante as validações feitas quando a aplicação é iniciada.

CAPÍTULO 6

Interceptors e Decorators

Neste capítulo veremos duas funcionalidades muito interessantes: *interceptors* e *decorators*. São mecanismos tecnicamente parecidos, mas semanticamente bem diferentes. Ambos nos permitem, cada um com sua aplicabilidade, estender nossa aplicação de forma elegante, sem mexer, ou mexendo muito pouco no nosso código.

6.1 IMPLEMENTANDO REQUISITOS TRANSVERSAIS COM INTERCEPTORS

Um interceptor, ou interceptador, é uma espécie de *web filter* da API de servlets, mas em vez de agir nas requisições web, funciona nas invocações de métodos dos componentes CDI. Os interceptadores têm esse nome porque eles interceptam a chamada de método e podem executar ações específicas antes, depois ou no lugar da requisição original.

Se compararmos o funcionamento do filtro e do interceptador, veremos muitas semelhanças.

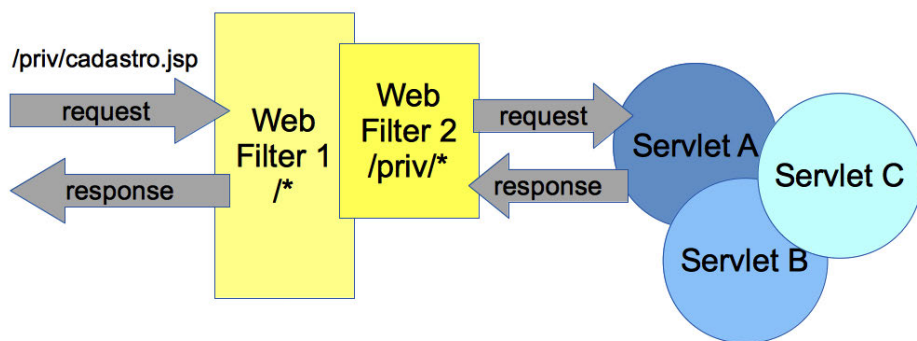


Figura 6.1: Funcionamento dos filtros web

Basicamente os filtros web são objetos que interceptam a chamada para uma servlet baseados em um padrão de url, que no exemplo foram `/*` e `/priv/*`.

Já o interceptor, como veremos na imagem a seguir, tem o mesmo funcionamento mas, em vez de padrão de url, baseia-se em anotações específicas para interceptar as chamadas de métodos. Além disso, em vez de servlets, com os interceptadores trabalhamos no nível de beans CDI e seus métodos.

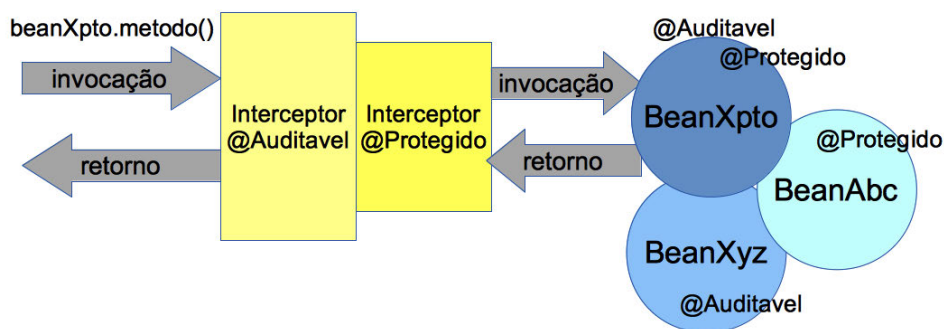


Figura 6.2: Funcionamento dos interceptadores

E qual a vantagem disso? Talvez se você nunca criou um *filter*, não imagina por que usaria um *interceptor*. Esse tipo de funcionalidade é muito importante para im-

plementarmos requisitos transversais do sistema. Por exemplo, requisitos de segurança.

Enquanto num filtro web poderíamos especificar que todas as requisições do sistema só podem ser realizadas se o usuário já tiver feito login, podemos especificar que um método específico, ou todos os métodos de uma determinada classe, só pode ser executado por uma pessoa com determinado perfil. Ou ainda, que após a execução de qualquer método de uma classe deve ser gerado um log, assim como faríamos ao gerar um log de qualquer requisição web.

O funcionamento é simples, mas um exemplo facilita ainda mais:

```
@WebFilter("/*")
public class FiltroAutenticacao implements Filter {

    @Inject
    private Logger logger;

    @Override
    public void doFilter(ServletRequest request,
                        ServletResponse response, FilterChain chain)
        throws IOException, ServletException {

        //faz um processamento prévio
        HttpServletRequest req = (HttpServletRequest) request;
        HttpSession session = req.getSession();
        Object usuario = session.getAttribute("usuarioLogado");

        String paginaDeLogin = "login.jsf";
        String paginaAtual = req.getRequestURI();

        if(usuario == null && !paginaAtual.endsWith(paginaDeLogin)){
            //envia para tela de autenticação
            logger.info("usuário não autenticado");
            ((HttpServletRequest)response).sendRedirect(paginaDeLogin);
        }
        else {

            //chama a ação original
            chain.doFilter(request, response);

            //faz processamento posterior
```



```

        logger.info("terminando requisição para " + paginaAtual);
    }

    }
    ...
}

```

Como acabamos de ver, criamos um filtro web que força o usuário a se autenticar. Poderíamos fazer qualquer tipo de lógica antes e depois da execução do que o usuário pretendia fazer, que é representada pela chamada do método `%chain.doFilter(request, response)`. E o mais interessante: como a chamada desse método é o que dispara a execução alvo do usuário, é possível, baseado em algum critério, optarmos simplesmente por não fazer essa chamada, ou fazer uma outra coisa em seu lugar. É o que ocorre quando o usuário não se autenticou no sistema. No lugar de processarmos a requisição para a página que ele tentava acessar, enviamos-lo para a tela de login. Trocamos a ação original do usuário por outra mais apropriada no momento.

De forma bastante semelhante podemos definir um interceptor:

```

@Interceptor @Priority(Interceptor.Priority.APPLICATION)
public class Auditor {

    @Inject
    private Logger logger;

    @AroundInvoke
    public Object auditar(InvocationContext context) throws Exception {

        //faz processamento antes
        logger.info("faz processamento anterior");

        Method method = context.getMethod();
        Object target = context.getTarget();
        Object[] params = context.getParameters();

        logger.info(String.format("auditando o método: '%s' "
            + "do objeto: '%s' "
            + "com os parâmetros: '%s'", method, target, params));
    }
}

```

```
//chama o método original
logger.info("chama método original");
Object retorno = context.proceed();

//faz processamento posterior
logger.info("faz processamento posterior");

return retorno;
}
}
```

A diferença é que o interceptor está trabalhando dentro do contexto CDI, mas assim como no filtro, podemos fazer operações antes, depois, ou mesmo substituir a execução original por outra coisa, que pode inclusive não fazer nada. Para isso, bastaria colocar a invocação do método `context.proceed()` dentro de um `if` que checasse nossos critérios. É uma forma mais simples, mas que atende à maioria dos casos do que anteriormente fazíamos com programação orientada a aspectos, ou AOP.

Em comum também a todas essas formas de implementar requisitos transversais está o forte apelo semântico de não utilizá-las para implementar requisitos funcionais. Isso quer dizer que não devemos codificar requisitos de negócio ali dentro, como algum tipo de cálculo. Para essa aplicação temos os *decorators*, pense em interceptadores apenas para validações de acesso, auditoria e outras regras gerais.

Por mais que usando a interface `InvocationContext` seja possível, conceitualmente um interceptador não deve saber qual classe está interceptando. Obviamente, saber quem está sendo interceptado não é o problema, podemos usar essa informação para gerar um log, por exemplo, mas não para tomar uma decisão de negócio. Se você acha que tem essa necessidade, talvez o melhor seja criar um outro interceptador com uma função específica, em vez de fazer *ifs* dentro de um só, ou então criar um *decorator*.

Mas se o interceptador não sabe quem está interceptando, como fazemos a ligação entre ambos? Fazemos através dos *interceptor bindings*. Isso nada mais é do que uma anotação que é colocada no interceptador e nas classes ou métodos que desejamos interceptar.

```
import javax.interceptor.InterceptorBinding;
```

```

@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE, ElementType.METHOD})
@InterceptorBinding
public @interface Auditavel {}

```

Assim como um qualificador, que vimos na seção 4.2, podemos ter atributos dentro dos *interceptor bindings*, e a menos que a propriedade seja anotada com `@javax.enterprise.util.Nonbinding`, só serão ligados beans e interceptors que especificarem os mesmos valores para cada propriedade na anotação que faz a relação.

Criada a anotação vinculadora, anotamos a classe ou método que desejamos auditar.

```

@Auditavel
public class GestoraDePermissoes { ... }

public class GestoraDePermissoes {

    @Auditavel
    public void mudaPermissoes(Funcionario f, Permissao[] permissoes) {
        ...
    }
}

```

Podemos também adicionar a meta-anotação `@java.lang.annotation.Inherited` no *interceptor binding*, dessa forma nossa anotação será herdável. No nosso exemplo, significa que classes filhas da `GestoraDePermissoes` também serão auditáveis.

```

@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE, ElementType.METHOD})
@Inherited
@InterceptorBinding
public @interface Auditavel {}

@Auditavel
public class GestoraDePermissoes { ... }

public class GestoraAssincronaDePermissoes
    extends GestoraDePermissoes { ... }

```

Além disso, como veremos na seção 7.1, podemos colocar um ou mais *interceptor bindings* dentro de estereótipos. Assim, um bean com esse estereótipo passa a ser interceptado automaticamente. A restrição nesse caso é que só podemos colocar em estereótipos *interceptor bindings* com `@Target(ElementType.TYPE)`.

HERDANDO ANOTAÇÕES

Apesar de não termos comentado dessa possibilidade, podemos utilizar a anotação `@java.lang.annotation.Inherited` também nos qualificadores, mas essa possibilidade é desestimulada pela especificação. Enquanto a classe filha de um tipo interceptado provavelmente mereça ser interceptado também, os qualificadores preferencialmente devem ser usado com moderação.

No exemplo que vimos, faz sentido a `GestoraAssincronaDePermissoes` ser auditável, afinal, ela também é uma `GestoraDePermissoes`, e é auditada.

Já quando trabalhamos com qualificadores, temos o intuito de individualizar ao máximo cada bean, e se seus subtipos herdarem seus qualificadores teremos mais chances de acontecerem problemas de ambiguidade. Se quisermos que um subtipo tenha os mesmos qualificadores, pode ser o caso de usarmos `@Specializes`, como vimos na seção 4.6.

Ainda falta algo para que nosso interceptor funcione como pretendido: adicionar nele a anotação vinculante. Podemos inclusive colocar mais de uma anotação de *binding* no mesmo interceptor, mas no nosso exemplo usaremos apenas a recém criada `@Auditavel`.

```
@Interceptor @Auditavel @Priority(Interceptor.Priority.APPLICATION)
public class Auditor {
    ...
}
```

Agora sim temos nosso interceptador funcionando como esperávamos, pois além de especificar qual *interceptor binding* ele usará, já o ativamos com a anotação `@Priority`. Assim como fizemos com os beans *alternatives*, utilizamos essa anotação não só para ativar, mas também para definir a prioridade, ou ordem de

execução do nosso interceptor. Isso, porém, só é possível se estivermos utilizando CDI 1.1.

Se estivermos utilizando a versão 1.0, precisaremos registrar nossos interceptadores via arquivo `beans.xml`, e não conseguimos definir a ordem de execução dos mesmos; pelo menos não sem utilizar alguma configuração específica de alguma implementação ou biblioteca auxiliar. Para ativar nosso interceptador sem utilizar a anotação `@Priority`, precisaremos configurar da seguinte maneira.

```
<beans xmlns="http://xmlns.jcp.org/xml/ns/javaee"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
        http://xmlns.jcp.org/xml/ns/javaee/beans_1_1.xsd">
  <interceptors>
    <class>
      br.com.casadocodigo.livrocdi.interceptor.Auditor
    </class>
    <class>...</class> <!-- um interceptor em cada tag class -->
  </interceptors>
</beans>
```

Assim como não conseguimos definir a ordem de execução apenas com o arquivo `beans.xml`, não temos como definir a ordem de execução dos interceptadores se estes possuírem a mesma prioridade (`@Priority`). Porém, se tivermos interceptadores com prioridades diferentes, será executado primeiro aquele que tiver um número de prioridade menor. Ou seja, interceptor com prioridade 1 (um) executa antes do que tiver prioridade 10 (dez).

Apesar de termos visto um funcionamento parecido com o do filtro web, utilizando a anotação `@AroundInvoke`, é possível definirmos mais alguns modos de interceptação com CDI. Todos os tipos que veremos a seguir utilizam a mesma forma de fazer *binding* entre o bean interceptado e o interceptor, a diferença fica apenas no momento em que o método do interceptor será chamado.

Interceptor `@AroundTimeout`

Este tipo de interceptor é utilizado quando desejamos interceptar a execução de métodos invocados pelo serviço de *timer* do Java EE. São aqueles métodos geralmente anotados com `@Timeout` ou `@Schedule`:

```
...
@Schedule(hour = "*", minute = "*/10")
```

```
public void importaDadosDoFTP(){
    ...
}
...
```

Nesse exemplo, consideramos que nosso sistema a cada 10 (dez) minutos busca algum tipo de dado no FTP. Porém, esse serviço de timer só funciona se nosso bean foi um EJB, e nossos exemplos não são EJBs. Como estamos falando de interceptadores, é importante citarmos esse tipo. Para interceptarmos as invocações dos métodos disparados pelo serviço de timer, utilizamos a anotação `@javax.interceptor.AroundTimeout`, como podemos ver a seguir.

```
...
@AroundTimeout
public Object timer(InvocationContext context) throws Exception {

    //faz algo antes
    Object target = context.getTarget();

    Object retorno = context.proceed();

    //faz algo depois
    return retorno;
}
...
```

É interessante notar, ainda, que o método `importaDadosDoFTP()` pode ser invocado tanto pelo serviço de *timer* quanto diretamente por um cliente do EJB no qual ele está definido. Porém como temos duas formas de chamar o mesmo método, podemos ter tanto um interceptador `@AroundTimeout` quanto um `@AroundInvoke` aptos a interceptar esse método. Como sabemos qual será chamado? A CDI cuida disso corretamente. Se o método for chamado diretamente pelo cliente do EJB, o interceptador utilizado será o `@AroundInvoke`, mas se for chamado via *timer*, será utilizado o `AroundTimeout`.

6.2 INTERCEPTADORES DE CICLO DE VIDA

Além dos tipos de interceptadores que vimos, existem os interceptadores de ciclo de vida. Dois deles utilizam anotações que já usamos há muito tempo:

`@PostConstruct` e `@PreDestroy`, e servem para interceptar o mesmo momento que ativa os métodos anotados por elas. Já o terceiro, nos permite interceptar o construtor dos beans, e será por ele que iniciaremos.

Interceptador `@AroundConstruct`

Ao utilizar a anotação `@javax.interceptor.AroundConstruct` definimos um interceptador que, em vez de interceptar a chamada de um método qualquer do nosso bean, intercepta a chamada do seu construtor.

```
...
@AroundConstruct
public void interceptaConstrutor(InvocationContext context)
                                throws Exception {

    context.proceed(); //chama o construtor interceptado
    Object objetoRecemCriado = context.getTarget();
}
...
```

Como nesse caso estamos interceptando o construtor do bean, a chamada de `context.proceed()` faz com que o objeto cujo construtor foi interrompido seja criado. Se não chamarmos esse método, o objeto não será criado. Apenas após a chamada do método `proceed()` conseguimos recuperar o objeto recém criado através do método `getTarget()` do `InvocationContext`. Esse objeto, porém, não está com suas dependências povoadas, então podemos ter problemas como `NullPointerException` se tentarmos executar algum de seus métodos.

Obviamente, se nosso objeto alvo definir que suas dependências deverão ser todas satisfeitas via seu construtor, teremos o objeto recém criado já com essas dependências, mas como alguns objetos podem receber dependências de outra forma, fazer uso do objeto novo é arriscado. Até porque, voltando à premissa de que o interceptador não deve conhecer qual objeto está sendo interceptado, é impreciso fazer muitas suposições sobre como este objeto recebe suas dependências.

Só teremos certeza que o objeto está totalmente montado quando o método anotado com `@PostConstruct` for invocado. Porém até agora só utilizamos essa anotação para definir métodos dentro do próprio bean que foi criado. Mas temos também a opção de interagir com esse momento via interceptadores.

Interceptadores @PostConstruct e @PreDestroy

Desde o início dos nossos exemplos temos utilizado `@PostConstruct` para que nossos beans saibam quando estão prontos, com todas as dependências satisfeitas, e assim possam fazer algo. De forma parecida, podemos utilizar `@PreDestroy` para executar alguma operação antes do nosso bean ser removido da memória.

Até aqui, nossa utilização tem se parecido com a seguinte.

```
@Named @RequestScoped @Auditavel
public class CalculadoraImpostosBean {

    @Inject private CalculadoraDeImpostos calculadoraImpostos;

    @PostConstruct
    public void init(){
        System.out.println("CalculadoraImpostosBean pronta.");
    }

    @PreDestroy
    public void destroy(){
        System.out.println("CalculadoraImpostosBean indo embora.");
    }
}
```

Essa é a forma de utilizar essas anotações dentro do próprio bean, mas podemos utilizá-las também em um interceptor. Como em todos os demais tipos, a vinculação entre o interceptor e o bean é feita via *interceptor bindings*. No nosso exemplo, esse binding é feito pela anotação `@Auditavel`, e nosso interceptador é o `Auditor`.

```
@Interceptor @Auditavel @Priority(Interceptor.Priority.APPLICATION)
public class Auditor {

    @PostConstruct
    public void postConstruct(InvocationContext context)
        throws Exception{
        Object target = context.getTarget();

        //chama método anotado com @PostConstruct, se houver
        context.proceed();
    }
}
```



```
@PreDestroy
public void preDestroy(InvocationContext context) throws Exception{
    Object target = context.getTarget();

    //chama método anotado com @PreDestroy, se houver
    context.proceed();
}
}
```

Basicamente, a diferença é que ao remover os métodos de *callback* de dentro do próprio bean, conseguimos dar um comportamento mais uniforme dentro do nosso sistema. Não precisamos repetir várias vezes o mesmo código que cria um log mostrando o ciclo de vida de cada um dos beans do sistema. Basta fazer isso no interceptor e vinculá-lo ao bean através da anotação de vinculação — no exemplo, a `@Auditavel`.

Note também que por estarmos interceptando o método original, sempre temos a opção de chamá-lo ou não via método `proceed()`. Nesse exemplo acima, o bean pode até não ter um método de *callback* dentro dele, mesmo assim o interceptador consegue capturar o momento apropriado. Agora, se o bean tiver esses métodos dentro dele, os mesmos serão executados no momento em que chamamos o método `proceed()`.

6.3 ESTENDENDO UM BEAN EXISTENTE COM DECORATORS

Decorators em CDI nada mais são do que uma forma de implementarmos o padrão de projeto de mesmo nome. Logo, a definição de um se aplica ao outro.

Um decorator, ou decorador, é um objeto que estende as funcionalidades de um objeto já existente, para isso ele precisa ter a mesma tipagem do objeto decorado. A grande sacada desse padrão é que em vez de estender o objeto original e sobrescrever os métodos que desejamos alterar ou criar novos métodos, fazemos isso de forma dinâmica, em tempo de execução. Vamos ver isso de forma gráfica para facilitar o entendimento.

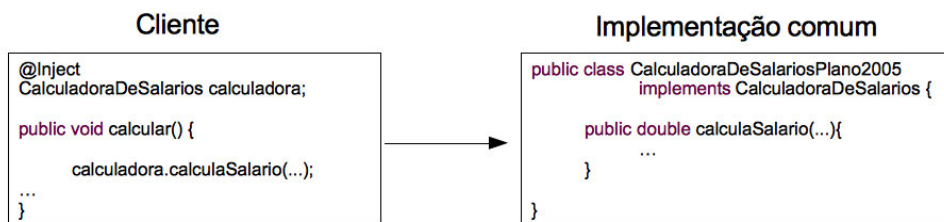


Figura 6.3: Implementação simples de uma interface

Iniciamos vendo uma classe cliente utilizando uma simples implementação de uma interface, que no caso foi a `CalculadoraDeSalarios`. Até aqui nada de novo, mas vai nos ajudar a perceber a evolução do exemplo. A seguir veremos uma forma de estender a implementação que acabamos de ver.

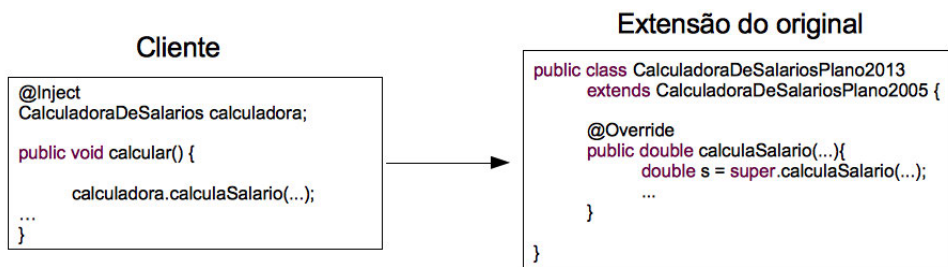


Figura 6.4: Estendendo a implementação original

Quando estendemos uma implementação existente, como no exemplo que acabamos de ver, podemos sobrescrever os métodos existentes e chamar a implementação original através do `super`. Como a classe `CalculadoraDeSalariosPlano2013` estende a `CalculadoraDeSalariosPlano2005`, e esta, por sua vez, implementa a interface `CalculadoraDeSalarios`, logo, a implementação do plano de 2013 também implementa essa interface, e por isso pode ser usada pela nossa classe cliente.

Essa é uma forma legítima de estender um comportamento, tanto é que a própria linguagem nos dá meios de fazê-lo, mas novamente temos que tomar cuidado com a semântica. Será que a implementação do plano de 2013 é a implementação do

plano de 2005, ou simplesmente a usa? Quando usamos extensão ou implementação, lemos que uma classe *é* do tipo estendido ou implementado. Ou seja, a `CalculadoraDeSalariosPlano2005` *é* uma `CalculadoraDeSalarios`. Até aí tudo bem. Mas será que a `CalculadoraDeSalariosPlano2013` *é* uma `CalculadoraDeSalariosPlano2005`? Ou seria apenas uma `CalculadoraDeSalarios`?

Essa questão semântica é um dos aspectos que podemos tratar melhor com um decorador, mas não ficamos limitados a isso. Porém, como veremos ainda bastante sobre decoradores, nesse momento o aspecto semântico já nos vale a análise. Por isso, vamos olhar como resolvê-lo usando composição em vez de herança.

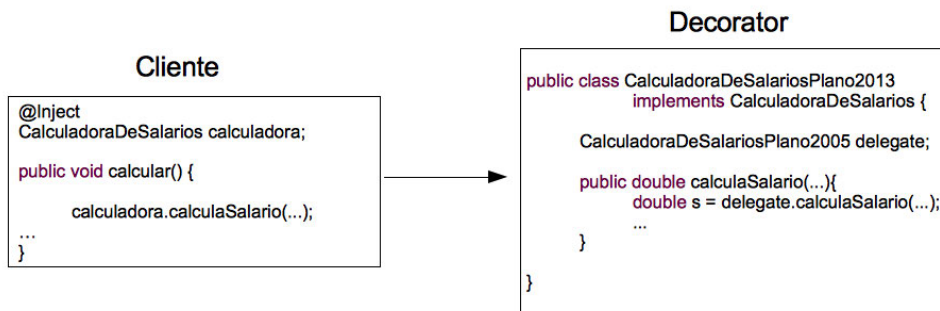


Figura 6.5: Usando composição em vez de herança

Agora com esse arranjo, temos uma leitura correta das classes envolvidas: a `CalculadoraDeSalariosPlano2013` *é* uma `CalculadoraDeSalarios` e para executar a sua tarefa *usa* a `CalculadoraDeSalariosPlano2005`. Ótimo, problema semântico resolvido, mas será que um decorador é só um nome bonitinho para a máxima “quando possível, use composição em vez de herança”? Obviamente que não, e agora que entendemos a estrutura do decorador, veremos em que situações ele nos será muito útil.

6.4 VINCULAÇÃO DINÂMICA ENTRE O OBJETO DECORADO E SEU DECORADOR

Enquanto no exemplo que acabamos de ver a ligação entre os objetos foi definida em tempo de desenvolvimento, utilizando a linguagem de programação, a funcionalidade que veremos a partir da CDI vincula os objetos em tempo de execução.

Na verdade, essa é uma característica do próprio padrão de projeto *decorator*, não é apenas usar composição, é fazer isso com a aplicação rodando.

O funcionamento do decorador lembra muito o do interceptador. A diferença é que, neste último, a interceptação é feita de forma genérica, para adicionar à requisição requisitos não funcionais, como segurança; já no decorador adicionamos funcionalidades de negócio, tanto é que o decorador precisa ter o mesmo tipo do objeto decorado.

Quando a CDI vai injetar um determinado objeto na classe cliente, ela verifica se não existe um decorador com tipo compatível, se tiver, esse decorador é passado no lugar do objeto original, e o original é injetado dentro do decorador, e dentro deste passa a ser chamado de *delegate*. Vejamos um exemplo prático.

```
@Inject
private CalculadoraDeSalarios calculadoraDeSalarios;
```

Aqui temos um código já conhecido, que nada mais é do que um cliente da interface `CalculadoraDeSalarios` recebendo-a do contexto CDI. Agora vamos analisar a implementação da `CalculadoraDeSalariosPlano2013`, que é um decorador.

```
@Decorator
public class CalculadoraDeSalariosPlano2013
    implements CalculadoraDeSalarios {

    @Inject @Delegate
    private CalculadoraDeSalarios delegate;

    @Inject
    private Logger logger;

    @Override
    public double calculaSalario(Funcionario funcionario){

        logger.info("Delegate plano 2013");

        double salario = delegate.calculaSalario(funcionario);

        return salario * 1.1;
    }
}
```

Nesse exemplo, vamos considerar que depois de muita negociação com os funcionários, a empresa onde trabalhamos decidiu que o plano de salários de 2013 apenas irá conceder um aumento de 10% (dez por cento) sobre o plano anterior. Em vez de criarmos uma nova implementação da interface `CalculadoraDeSalarios`, utilizamos a existente, que já está testada, e adicionamos ao retorno do seu cálculo o percentual de aumento estipulado.

O interessante de analisarmos em um decorador é o ponto de injeção anotado com `@Delegate`. Perceba que, fora essa anotação, este é um ponto de injeção comum, onde definimos o tipo que desejamos e anotamos com `@Inject`. Podemos adicionar aí também qualquer qualificador que desejarmos, e ao fazer isso estaremos especificando mais em quais casos desejamos que o decorador seja utilizado.

O funcionamento é simples, o decorador será utilizado para substituir o objeto original toda vez que seu *delegate* combinar com o que a classe cliente espera que seja injetado. Em outras palavras, se a classe cliente espera uma `CalculadoraDeSalarios` com qualificador `@Default` e nosso decorador sabe decorar um objeto com o mesmo tipo, então ele será passado à classe cliente, e o objeto original será seu *delegate*.

Se quisermos que nosso decorador atue sobre qualquer `CalculadoraDeSalarios`, e não apenas o `@Default`, podemos fazer a seguinte alteração.

```
@Decorator
public class CalculadoraDeSalariosPlano2013
    implements CalculadoraDeSalarios {

    @Inject @Delegate @Any
    private CalculadoraDeSalarios delegate;
    ...
}
```

Agora qualquer `CalculadoraDeSalarios` passará a ser decorada pelo `CalculadoraDeSalariosPlano2013`. Como vimos na seção 4.2, o qualificador `@Any` é praticamente um desqualificador, fazendo com que o ponto de injeção marcado com ele aceite objetos com quaisquer qualificadores, e não somente o `@Default`, que sempre está implícito quando não especificamos outro.

Uma vez que entendemos o mecanismo de “interceptação” que nosso decorador acaba usando para entrar no lugar dos objetos originais, podemos voltar a analisar o exemplo anterior, em que tínhamos a injeção de um `Logger` além do *delegate*. Isso

porque nosso decorador é um bean como qualquer outro, e assim pode ter também suas próprias dependências. Essas dependências serão resolvidas como qualquer outra, podendo ter os mesmos problemas de ambiguidade ou não tendo quem a supra; podem ser interceptadas, e podem inclusive serem decoradas por um outro decorador. Só não podemos ter uma dependência cíclica, na qual pedimos para injetar, em uma propriedade que não o *delegate*, um objeto que está sendo decorado pela nossa classe. Cada decorador pode ter um único delegate, mas pode ter diversos pontos de injeção normais. Também como um bean normal, nosso decorador pode receber as dependências via propriedade, como no exemplo, ou com as outras vias possíveis: construtor e método inicializador.

Ótimo, agora temos um decorador funcionando, mas e se quisermos mais informações sobre o bean que estamos decorando em tempo de execução? Por exemplo, se quisermos saber exatamente qual o seu tipo, seus pontos de injeção etc; fazemos isso via reflection em cima do delegate? Existe uma forma mais simples, adicionada na versão 1.1 da CDI, que é adicionar uma instância de `javax.enterprise.inject.spi.Bean`, representando o bean decorado.

A interface `Bean` nos permite acessar essas metainformações que queremos saber, e para termos acesso a ela basta adicioná-la como dependência anotando-a com `@Decorated`.

```
@Decorator
public class CalculadoraDeSalariosPlano2013
    implements CalculadoraDeSalarios {

    @Inject @Delegate @Any
    private CalculadoraDeSalarios delegate;

    @Inject @Decorated
    private Bean<CalculadoraDeSalarios> bean;
    ...
}
```

Como a própria anotação sugere, esse é o `Bean` que está sendo decorado, e através dessa interface respondemos as perguntas anteriores, e podemos inclusive utilizá-las para adicionar mais informações ao nosso log.

```
...
@Inject @Decorated
private Bean<CalculadoraDeSalarios> bean;
```

```

@Override
public double calculaSalario(Funcionario funcionario){

    logger.info("Delegate plano 2013");
    Class beanClass = bean.getBeanClass();
    Set<InjectionPoint> injectionPoints = bean.getInjectionPoints();
    logger.info(String.format("Bean original: '%s' "
        + "com injection points: '%s'", beanClass,
        injectionPoints));

    double salario = delegate.calculaSalario(funcionario);

    return salario * 1.1;
}
...

```

Com o nosso decorador completo, falta apenas habilitá-lo, pois assim como os interceptadores e os beans alternativos, um decorador precisa ser ativado, seja via `beans.xml` ou via anotação.

O arquivo `beans.xml` que habilita nosso decorador fica da seguinte maneira.

```

<beans xmlns="http://xmlns.jcp.org/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
        http://xmlns.jcp.org/xml/ns/javaee/beans_1_1.xsd">
    <interceptors>
        <class>
            br.com.casadocodigo...CalculadoraDeSalariosPlano2013
        </class>
        <class>...</class> <!-- um interceptador em cada tag class -->
    </interceptors>
</beans>

```

Encurtei o nome da classe apenas para facilitar a leitura, mas se trata do nome completo da classe, assim como em todos os outros exemplos apresentados. Também como nos outros exemplos onde usamos o arquivo XML, a partir da versão 1.1 da CDI podemos utilizar a anotação `@Priority` para ativar esse recurso. Com a alteração a seguir não precisamos alterar o `beans.xml`.

```

@Decorator @Priority(Interceptor.Priority.APPLICATION)
public class CalculadoraDeSalariosPlano2013

```

```
... implements CalculadoraDeSalarios {  
...  
}
```

Assim como os interceptors, podemos também ordenar nossos decoradores. Da mesma forma que podemos ter mais de um interceptador executando em uma mesma requisição, podemos ter mais de um decorador. Com decoradores não parece tão óbvio, mas se tivermos uma hierarquia de beans um pouco mais complexa, podemos ter um decorador em um nível mais abstrato e outro em um nível mais específico dessa hierarquia. Prioridades menores executam primeiro, e beans ativados com `@Priority` executam antes dos ativados via arquivo XML. Além disso, os decoradores são chamados sempre depois dos interceptadores.

CAPÍTULO 7

Definindo Estereótipos e diminuindo o acoplamento utilizando Eventos

Agora que já sabemos sobre qualificadores, mais especificamente beans nomeados, escopos e interceptadores, podemos ver uma ferramenta muito útil da CDI que é a criação de estereótipos.

Além disso, veremos também neste capítulo como utilizar Eventos, uma funcionalidade que nos dará a possibilidade de estender nossa aplicação com baixíssimo acoplamento. Provavelmente durante a leitura você lembrará de situações nos sistemas em que trabalha em que a utilização de eventos diminuiria o acoplamento na comunicação entre classes, módulos, e até mesmo sistemas diferentes.

7.1 DEFININDO ESTEREÓTIPOS

Quando alguém fala que uma determinada pessoa é um nerd, que imagem vem à sua cabeça? Talvez uma pessoa pequena, de óculos, quieta, inteligente, e provavelmente com um notebook por perto. Se você pensou em algo parecido, sabe o que é um estereótipo. Podemos pensar em diversos exemplos diferentes, mas o significado é sempre o mesmo: uma generalização de características comuns, uma imagem preconcebida de algo.

Até aqui vimos diversas configurações que podemos fazer em nossos beans, tais como definir um nome, um escopo, qualificadores e interceptadores. À primeira vista não parece um trabalho muito complicado, e de fato não é, mas podemos simplificar essas configurações criando um estereótipo para os beans. Para servir de base, vamos relembra o exemplo da classe `CalculadoraImpostosBean` que vimos na seção 5.2.

```
@Named @RequestScoped
public class CalculadoraImpostosBean { ... }
```

Agora vamos pensar no nosso sistema de uma forma mais abstrata. Temos um sistema que trata o RH e folha de pagamentos da nossa empresa, e como já vimos pelos diversos exemplos nele existem diversas calculadoras diferentes, uma de impostos, outra de salário e uma outra de folha de pagamento. Além disso, podemos ter ainda calculadoras simuladas para, por exemplo, serem utilizadas em um estudos de impacto financeiro quando um novo plano de cargos e salários for proposto.

Podemos perceber que o termo “calculadora” é recorrente no nosso sistema, assim como CRUDs, ou cadastros de tabelas básicas pode ser comum no projeto que você trabalha hoje. Quando pensamos em um termo significativo, uma espécie de abstração comum a vários locais do sistema, geralmente pensamos em interfaces. É possível até definirmos uma interface para as calculadoras também, mas esse não será o foco aqui. Nosso objetivo é ver de forma parecida os beans que manipulam as calculadoras, os controladores que injetam as calculadoras propriamente ditas. Isso, no entanto, não significa que não possamos aplicar o mesmo raciocínio às implementações das calculadoras, mas trabalhando com os controladores teremos mais mecanismos a analisar.

Olhando especificamente para nosso `CalculadoraImpostosBean`, mas enxergando a abstração de todos os `CalculadoraXYZBean`, percebemos que todos eles possuem nome e escopo. É um começo, mas podemos ir além. Poderíamos, por exemplo, definir que esses beans devem ser auditados, usando a anotação

`@Auditavel` que vimos na seção 6.1, e assim ficaria registrado no log da aplicação quem anda executando o quê. O resultado seria algo como o código a seguir.

```
@Named @RequestScoped @Auditavel
public class CalculadoraImpostosBean { ... }
```

Agora pensando um pouco mais em questões de infraestrutura, poderíamos ter um interceptador geral que, diferente do ligado à anotação `@Auditavel`, que trata mais de auditoria de que coisas mais restritas, lidasse com questões gerais genéricas da aplicação, como qual método está sendo chamado, qual a URL atual, e outras informações não relacionadas ao negócio. Esse interceptador estaria vinculado a uma anotação que definimos agora: `@Rastreavel`. Tanto seu código quanto o do interceptador não importam, pois estamos interessados na semântica apenas. Feito isso, se olharmos nosso bean novamente, veremos que a quantidade de anotações já incomoda.

```
@Named @RequestScoped
@Auditavel @Rastreavel
public class CalculadoraImpostosBean { ... }
```

O pior desse cenário não é nem o tanto de anotações em cima da classe, e sim a possibilidade de esquecermos de definir o mesmo leque de anotações quando estivermos definindo o `CalculadoraXYZBean`. Ainda mais agora que estamos definindo interceptadores genéricos que servem a questões de infraestrutura, como o `@Rastreavel`. Essas geralmente são as primeiras coisas que são esquecidas conforme o sistema cresce. Um desenvolvedor lembra de colocar essas anotações, o outro não, e o quando temos um problema para analisar o log está inconsistente.

Para evitar esses problemas, podemos definir uma anotação que resume tudo isso. Podemos, por exemplo, chamá-la de `@CalculadoraBean`, e ela terá o seguinte código.

```
import javax.enterprise.inject.Stereotype;

@Stereotype
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
public @interface CalculadoraBean {
}
```

Aqui, ainda, temos a definição mínima do nosso estereótipo. Estamos usando `@Target (TYPE)`, mas também podemos combinar com os *targets* `METHOD` e

FIELD. Para que o estereótipo `CalculadoraBean` seja um resumo de todas aquelas anotações, temos que colocá-las nele.

```
import javax.enterprise.inject.Stereotype;

@Stereotype
@Named @RequestScoped
@Auditavel @Rastreavel
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
public @interface CalculadoraBean {
}
```

É muita coisa junta, mas agora temos um condensado de todas as configurações que desejamos colocar nos beans desse tipo. Agora é só anotar nosso alvo com esse estereótipo.

```
@CalculadoraBean
public class CalculadoraImpostosBean { ... }
```

De agora em diante, se quisermos adicionar um novo interceptor em todos os beans dessa “família”, basta adicionar no estereótipo. Ok, muito interessante, mas e se o bean `CalculadoraFolhaDePagamentoBean` estiver em uma tela mais complexa que a responsável por calcular impostos, e por isso precisarmos de um escopo mais apropriado como `@ConversationScoped`? Não tem problema, pois o estereótipo define o escopo padrão, podendo ser sobrescrito pelo bean que o utilizar.

```
@CalculadoraBean
@ConversationScoped
@Named("calculadoraFolhaBean")
@UmNovoQualificadorQualquer
@MaisUmInterceptorBinding
public class CalculadoraFolhaDePagamentoBean { ... }
```

Como podemos ver, as configurações do estereótipo não são limitadoras, são padronizadoras, podendo ser estendidas pelo bean cliente. Sempre que possível, tente padronizar suas configurações para evitar inconsistências. Apesar de termos analisado um estereótipo para a camada “controladora” do MVC, como estamos falando de CDI, podemos definir um estereótipo para qualquer lugar da nossa aplicação, como na implementação das calculadoras.

Podemos, por exemplo, definir um estereótipo como `@Alternative` para que seja usado nas simuladoras de cálculo.

```
@Stereotype
@Alternative
@Rastreavel
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
public @interface Mock {
}
```

E podemos ainda combinar estereótipos simplesmente anotando um estereótipo com outro.

```
@Stereotype
@CalculadoraBean @Mock
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
public @interface CalculadoraMockBean {
}
```

Assim qualquer bean anotado com `@CalculadoraMockBean` terá um nome padrão, será `@RequestScoped`, `@Auditavel` e `@Rastreavel`, e também `@Alternative`. Certamente é uma forma muito flexível de organizar os beans com características semelhantes.

@Model: estereótipo de série

Aprendemos a criar nossos próprios estereótipos, mas caso queiramos apenas um que defina o escopo request como padrão e que o bean que o utilizar será nomeado, podemos utilizar o estereótipo `@javax.enterprise.inject.Model`. Seu objetivo é disponibilizar, sem a necessidade de codificação alguma, um estereótipo que sirva para os beans da camada “model” do MVC.

7.2 UTILIZANDO EVENTOS PARA OBTER UM BAIXÍSSIMO ACOPLAMENTO

Quando pensamos em CDI, a primeira funcionalidade que nos vem à cabeça é a resolução de dependências. Este é o assunto mais buscado sobre a especificação,

mas nesta seção veremos como o mecanismo de eventos pode ser de grande valia para nossas aplicações.

A utilização é extremamente simples, mas precisamos pensar nas funcionalidades dos nossos sistemas de uma forma diferente. Para nosso estudo, vamos considerar a gestão de funcionários da nossa empresa. Imagine que na empresa temos diversos sistemas, e que para acessar esses sistemas os funcionários precisam, além de seu login e senha, estarem com sua situação funcional ativa dentro da empresa. Ou seja, quando o funcionário for desligado, seu acesso deve ser cortado nos sistemas. Semelhantemente, podemos criar o acesso aos sistemas quando um novo funcionário for contratado, então logo que ele sair do RH da empresa, já poderá começar a trabalhar.

Quais serão os sistemas que os funcionários da empresa utilizam vai depender de cada ramo de atuação. Por exemplo, em uma empresa de desenvolvimento de software, os funcionários terão acesso ao repositório de código e à ferramenta de controle de issues. Em outras empresas talvez tenhamos que dar acesso ao ERP, ou qualquer sistema que você possa imaginar. Agora pense como criar esses acessos logo depois do cadastro do funcionário, e como removê-los no desligamento. Será que algo parecido com o pseudocódigo abaixo é uma boa alternativa?

```
public void cadastrarFuncionario(Funcionario funcionario){
    //lógica de ativação do funcionário
    funcionario.setStatus(ATIVO);
    entityManager.persist(funcionario);

    criaUsuarioGit(funcionario);
    criaUsuarioIssueTracker(funcionario);
    criaUsuarioERP(funcionario);
}
```

Como podemos perceber, acabamos criando um acoplamento entre a gestão de funcionários os diversos sistemas que eles utilizarão no seu dia a dia. Pode não parecer algo tão absurdo, mas não é o desejado, visto que esses sistemas podem mudar. Podemos trocar o *issue tracker* utilizado, podemos passar a usar algum novo sistema ou deixar de criar usuários no ERP para todos os funcionários, deixando apenas para aqueles que de fato precisarão operar esse sistema. A cada mudança desse tipo, teremos que alterar o método de cadastro do funcionário, mesmo sem qualquer alteração na lógica de ativação de funcionário do ponto de vista do RH. Isso nos dá um forte indício de que devemos mudar algo.

Uma evolução nesse cenário seria definirmos algum tipo de interface que abstraia essa criação de usuários.

```
public interface GerenciadorDeAcesso {  
    void criaUsuario(Funcionario funcionario)  
}  
  
public class GerenciadorDeAcessoGit implements GerenciadorDeAcesso {...}  
public class GerenciadorDeAcessoERP implements GerenciadorDeAcesso {...}  
//outras implementações
```

E agora podemos alterar nossa gerenciadora de funcionários para utilizar essa abstração

```
public class GerenciadorFuncionarioBean {  
    private List<GerenciadorDeAcesso> gerenciadoresAcesso =  
        Arrays.asList(  
            new GerenciadorDeAcessoGit(),  
            new GerenciadorDeAcessoERP(),  
            new GerenciadorDeAcessoIssueTracker());  
  
    public void cadastrarFuncionario(Funcionario funcionario){  
        //lógica de ativação do funcionário  
        funcionario.setStatus(ATIVO);  
        entityManager.persist(funcionario);  
  
        for(GerenciadorDeAcesso gerenciador : gerenciadoresAcesso){  
            gerenciador.criaUsuario(funcionario);  
        }  
    }  
}
```

Dessa maneira é possível alterar muita coisa apenas criando novas implementações de `GerenciadorDeAcesso` e adicionando ou removendo instâncias da lista `gerenciadoresAcesso`. Ainda podíamos evoluir nossa solução criando uma espécie de federação de gestores de usuários, tratando essa coleção como se fosse um único objeto, e deixando que esse objeto delegasse cada chamada aos elementos da lista. Isso com certeza melhoraria ainda mais a solução, mas ainda podemos ter novas situações fugindo do nosso controle.

Vamos supor que, por questões legais, nossa empresa tenha que informar ao INSS quando realizar uma nova contratação. Seria uma situação muito parecida com

a criação de usuários feita através das implementações de `GerenciadorDeAcesso`, mas criar uma nova implementação dessa interface para informar ao INSS seria uma quebra na semântica da interface. Até porque teremos outros métodos de concessão de permissões que a implementação do INSS não necessitaria. E agora? Como resolver esse problema?

Poderíamos criar uma interface ancestral da `GerenciadorDeAcesso`, com apenas métodos como `criarUsuario` e `removerUsuario` e então criar a implementação do INSS baseado nela. Mas será que precisamos mesmo fazer essas coisas dentro do método de negócio? A resposta é não, e usaremos eventos para resolver esses problemas. A vantagem é que provavelmente também estaremos preparados para qualquer outra nova operação que tenhamos de fazer ao contratar um funcionário, sem correr o risco de quebrar a semântica das interfaces que temos, e sem ter que alterá-las para isso.

```
import javax.enterprise.event.Event;

public class GerenciadorFuncionarioBean {

    @Inject
    private Event<Funcionario> eventoFuncionarioCadastrado;

    public void cadastrarFuncionario(Funcionario funcionario){
        //lógica de ativação do funcionário
        funcionario.setStatus(ATIVO);
        entityManager.persist(funcionario);

        eventoFuncionarioCadastrado.fire(funcionario);
    }
}
```

Criamos um objeto que lança eventos relacionados ao tipo `Funcionario`, e o chamamos de `eventoFuncionarioCadastrado`, que é do tipo `javax.enterprise.event.Event`. Com isso, agora nosso método de negócio não depende de qualquer `GerenciadorXYZ`, ele simplesmente lança um evento escutável por qualquer objeto do contexto CDI. Assim, se algum objeto quiser fazer alguma operação no cadastramento de um novo funcionário, basta escutar esse evento. O acoplamento entre quem lança o evento e quem o escuta é praticamente zero.

7.3 OBSERVADORES SÍNCRONOS DE EVENTOS

Para um objeto receber a notificação de que um novo `Funcionario` foi admitido, basta escutar o evento lançado pelo `GerenciadorFuncionarioBean`.

```
import javax.enterprise.event.Observes;

public class GerenciadorDeAcessoGit {

    public void criaUsuarioGit(@Observes Funcionario funcionario){
        ...
    }
}
```

A implementação, como acabamos de ver, é bem simples. O melhor é que nem precisamos definir qualquer interface, uma vez que não existe qualquer vínculo entre `GerenciadorFuncionarioBean` e `GerenciadorDeAcessoGit`. Se existe algum vínculo, é totalmente indireto, pois por padrão os observadores são síncronos. Assim, caso um observador demore demais para processar algo, ou lance uma exceção, quem lançou o evento será impactado, e o acoplamento entre os dois se restringe a isso.

Não podemos no entanto diminuir demais essa característica, visto que em um cenário Java EE, onde a transação é gerenciada pelo container, o método que lança o evento e o que o observa executam na mesma transação. Assim, a exceção lançada por um observador pode causar *rollback* em toda a transação, que envolve o método de negócio e todos os observadores. Essa pode ser uma característica desejável, nos permitindo validar algo via observador, mas pode não ser. Nesse caso a melhor opção seria utilizar um evento assíncrono.

7.4 EVENTOS E QUALIFICADORES

O exemplo de notificador e observador que vimos foi extramente simples: envolve apenas um evento do tipo `Funcionario`. Mas para que esse mecanismo seja eficaz, precisamos conseguir diferenciar, por exemplo, um evento de contratação de funcionário de um evento de desligamento, e para fazer isso, utilizamos qualificadores, da mesma forma que fazíamos para acabar com a ambiguidade na resolução de dependências.

Para demonstrar, vamos alterar nosso exemplo para distinguirmos eventos de contratação e eventos de demissão de funcionários. Os qualificadores utilizados são

criados exatamente da mesma forma que os qualificadores vistos na seção 4.2, usados na resolução de dependências, por esse motivo não é necessário vermos o código aqui.

```
public class GerenciadorFuncionarioBean {

    @Inject @Contratacao
    private Event<Funcionario> eventoContratacao;

    @Inject @Demissao
    private Event<Funcionario> eventoDemissao;

    public void cadastrarFuncionario(Funcionario funcionario){
        ...
        eventoContratacao.fire(funcionario);
    }

    public void demitirFuncionario(Funcionario funcionario){
        ...
        eventoDemissao.fire(funcionario);
    }
}
```

Várias classes podem observar o mesmo evento, mas para facilitar vamos colocar alguns observadores, um abaixo do outro.

```
public void escutaContratacao(@Observes @Contratacao Funcionario f){
    ...
}
public void escutaDemissao(@Observes @Demissao Funcionario f){
    ...
}
public void escutaQualquerEventoDeFuncionario(@Observes Funcionario f){
    ...
}
```

Enquanto os dois primeiros observadores são específicos para cada tipo de evento de `Funcionario`, o terceiro observará qualquer evento do tipo `Funcionario`, seja `@Contratacao`, `@Demissao` ou outra qualidade de evento de funcionário que venha a ser criado. Mas nosso exemplo ainda pode ser incrementado. Iremos adicionar um outro tipo de desligamento de funcionário, a aposentadoria. Agora ao desligar um funcionário precisaremos definir o tipo de desligamento.

```
public class GerenciadorFuncionarioBean {

    private TipoDesligamento tipoDesligamento;

    @Inject @Aposentadoria
    private Event<Funcionario> eventoAposentadoria;

    @Inject @Demissao
    private Event<Funcionario> eventoDemissao;

    ...

    public void desligarFuncionario(Funcionario funcionario){
        ...
        if(tipoDesligamento == TipoDesligamento.APOSENTADORIA){
            eventoAposentadoria.fire(funcionario);
        }
        else if(tipoDesligamento == TipoDesligamento.DEMISSAO){
            eventoDemissao.fire(funcionario);
        }
    }
}
```

Esse exemplo funciona, mas poderíamos trocar os dois pontos de injeção de eventos por apenas um, e qualificá-lo dinamicamente, parecido com o que fizemos na seção 4.8 quando vimos o lookup programático. Pode parecer que fica mais difícil, mas isso nos dá muita possibilidade de tomara de decisão em tempo de execução.

```
public class GerenciadorFuncionarioBean {

    private TipoDesligamento tipoDesligamento;

    @Inject
    private Event<Funcionario> eventoDesligamento;

    ...

    public void desligarFuncionario(Funcionario funcionario){
        ...
        Annotation qualificador = null;
```

```

        if(tipoDesligamento == TipoDesligamento.APOSENTADORIA){
            qualificador = new AnnotationLiteral<Aposentadoria>(){};
        }
        else if(tipoDesligamento == TipoDesligamento.DEMISSAO){
            qualificador = new AnnotationLiteral<Demissao>(){};
        }

        Event<Funcionario> eventoQualificado = eventoDesligamento
                                                    .select(qualificador);
        eventoQualificado.fire(funcionario);
    }
}

```

Fizemos uso da classe utilitária `javax.enterprise.util.AnnotationLiteral`, que deve ser tipada com a anotação que desejamos representar. Logo depois, de posse do objeto `qualificador`, que é do tipo `java.lang.annotation.Annotation`, usamos o método `select(Annotation)` do objeto `Event`. É como se estivéssemos saindo de um *select* sem *where* para um com *where qualificador = q*. Obviamente nesse exemplo `q` representa o qualificador definido dinamicamente, dentro do nosso método. Feita essa alteração, os observadores não mudam absolutamente nada, pois os eventos continuarão sendo qualificados.

Qualificadores de eventos com atributos

Novamente igual vimos na seção 4.2, podemos definir atributos nos qualificadores utilizados. Esses atributos também servem para relacionar o notificador e o observador do evento, a menos que a propriedade do qualificador esteja anotada com `@Nonbinding`; exatamente como acontece quando usamos um qualificador para injetar dependências. Por exemplo, a partir de agora passaremos a usar o qualificador `@Desligamento`.

```

@Target({TYPE, FIELD, METHOD, PARAMETER})
@Retention(RetentionPolicy.RUNTIME)
@Qualifier
public @interface Desligamento {
    TipoDesligamento value() default TipoDesligamento.APOSENTADORIA;
}

```

Vamos alterar mais uma vez nosso exemplo para usarmos apenas um qualificador, e fazer a distinção via atributo. Utilizaremos a mesma enum que usamos nos

ifs anteriores. Para não termos códigos muito extensos vamos olhar apenas como injetar os produtores de eventos na nossa classe.

```
@Inject @Desligamento(TipoDesligamento.APOSENTADORIA)
private Event<Funcionario> eventoAposentadoria;

@Inject @Desligamento(TipoDesligamento.DEMISSAO)
private Event<Funcionario> eventoDemissao;

//e os observadores ficam como abaixo

public void observaAposentadoriaFuncionario(@Observes
        @Desligamento(APOSENTADORIA) Funcionario funcionario){
    ...
}
public void observaDemissaoFuncionario(@Observes
        @Desligamento(DEMISSAO) Funcionario funcionario){
    ...
}
```

Feito isso, poderíamos usar cada instância dessa como já vimos nos exemplos anteriores. Porém o mais interessante seria vermos como qualificar dinamicamente nosso produtor de eventos, assim como acabamos de fazer quando tínhamos dois qualificadores. O problema é que não conseguimos fazer isso de uma maneira simples, pois agora temos que devolver a anotação no método `value()` do nosso objeto qualificador. Parte disso pode ser feita com classes aninhadas, mas usaremos uma classe servirá para substituir a `AnnotationLiteral` pois dessa maneira o código fica mais organizado.

```
public abstract class DesligamentoQualifier
        extends AnnotationLiteral<Desligamento>
        implements Desligamento {

    private TipoDesligamento tipoDesligamento;

    public DesligamentoQualifier(TipoDesligamento tipoDesligamento){
        this.tipoDesligamento = tipoDesligamento;
    }

    public TipoDesligamento value() {
        return tipoDesligamento;
    }
}
```

```
    }
}
```

E agora voltamos mais uma vez à versão em que temos apenas uma instância de `Event`, e qualificaremos isso de forma dinâmica, mas agora usando qualificadores com propriedades.

```
public class GerenciadorFuncionarioBean {

    private TipoDesligamento tipoDesligamento;

    @Inject
    private Event<Funcionario> eventoDesligamento;

    ...

    public void desligarFuncionario(Funcionario funcionario){
        ...
        Annotation qualificador = null;
        if(tipoDesligamento == TipoDesligamento.APOSENTADORIA){
            qualificador = new DesligamentoQualifier(APOSENTADORIA){};
        }
        else if(tipoDesligamento == TipoDesligamento.DEMISSAO){
            qualificador = new DesligamentoQualifier(DEMISSAO){};
        }

        Event<Funcionario> eventoQualificado = eventoDesligamento
                                                .select(qualificador);
        eventoQualificado.fire(funcionario);
    }
}
```

7.5 OBSERVADORES ASSÍNCRONOS DE EVENTOS

Já vimos como funciona o mecanismo de eventos, e vimos também que, por padrão, os observadores são síncronos. Porém, como foi dito, a execução lenta de um observador pode impactar na execução do método de negócio, e em alguns casos não há o que ser feito, precisamos realizar uma tarefa que é lenta, mas certamente não queremos degradar o funcionamento do método notificador. A saída para isso é a utilização de observadores assíncronos.

Como ainda vamos analisar no capítulo 8, a CDI não sobreposição funcionalidades de outras especificações do Java EE. E o suporte a métodos assíncronos já é tratado pela especificação de EJBs, assim sendo, utilizaremos esse suporte para criarmos observadores assíncronos.

Logo, somente com CDI, não conseguimos executar observadores assíncronos em um *servlet container* como Tomcat ou Jetty. Isso, é claro, sem alguma extensão que adicione essa funcionalidade fora de um servidor de aplicação como JBossAS (Wildfly) ou Glassfish. Sobre extensões, nos aprofundaremos no capítulo 9. Por enquanto apenas utilizaremos a funcionalidade disponível no Java EE.

Agora que sabemos que essa funcionalidade é na verdade um empréstimo da especificação de EJBs, vamos ver como implementamos um observador assíncrono.

```
@Stateless
public class ObservadorDeFuncionarios {

    @Asynchronous
    public void observaDemissaoFuncionario(@Observes
                                           @Desligamento(DEMISSAO) Funcionario funcionario){
        ...
    }
}
```

Basta que nosso beans seja um EJB e anotar nosso observador com `@javax.ejb.Asynchronous` e o método passa a executar em um contexto diferente do método notificador. Considerando o contexto transacional do método de negócio, o observador assíncrono funciona como se estivesse anotado com `@javax.ejb.TransactionAttribute(REQUIRES_NEW)`. Em outras palavras, o observador não provoca *rollback* no método notificador.

Sabendo dessa funcionalidade, pode parecer que encontramos uma bala de prata, uma vez que o observador pode demorar sem impactar o notificador. Já sabemos também que uma exceção no observador também não vai atrapalhar o método de negócio ou os demais observadores. Mas temos que nos lembrar que em alguns casos, o observador estar na mesma transação que o notificador é o desejável.

Vamos imaginar o cenário de um sistema de vendas, onde ao vender um item é lançado um evento, que é capturado pelo módulo de estoque para que este decresça a quantidade do produto. Agora, imaginemos que ocorra um problema no módulo de estoque. Pode ser interessante que a exceção chegue à operação de venda

para que o estoque não fique inconsistente. Então a operação da venda pode ser repetida, e se não houver uma nova exceção, a transação ocorre de forma consistente.

Em um cenário como o descrito, caso estejamos usando um observador assíncrono, ocorrendo uma exceção no módulo de estoque, teríamos que voltar ao módulo de compra para desfazer a venda. Nesse caso, se o cliente já foi embora, não será possível refazer a operação. Teríamos que notificar o problema de alguma forma para poder tratá-lo de forma manual. Em resumo, em algumas situações desejamos que o observador seja síncrono e em outras assíncrono, não existe um único remédio para todos os males.

7.6 INJETANDO DEPENDÊNCIAS NOS OBSERVADORES

Um observador de evento é um bean comum, dessa maneira, é possível injetar qualquer dependência nele. Porém, temos a opção de fazer algo bem parecido com o que vimos nos métodos produtores, que é receber uma dependência como parâmetro do método.

Podemos criar o bean `ObservadorDeFuncionarios` e injetar nele um `Logger` como poderíamos fazer em qualquer outra parte do sistema.

```
public class ObservadorDeFuncionarios {
    @Inject
    private Logger logger;

    public void observaFuncionario(@Observes Funcionario funcionario){
        logger.info("Recebendo evento do funcionario {} ", funcionario);
    }
}
```

Mas podemos também receber o `Logger` como dependência no próprio método observador. As vantagens de se fazer isso são as mesmas comentadas na seção 5.1 quando vimos os métodos produtores. Basicamente quando colocamos as dependências no método, fica claro o que é necessário para o observador funcionar. É como uma documentação a nível de assinatura de método.

```
public class ObservadorDeFuncionarios {

    public void observaFuncionario(@Observes Funcionario funcionario,
                                   Logger logger){
        logger.info("Recebendo evento do funcionario {} ", funcionario);
    }
}
```

```
    }  
}
```

Cada método observador só pode ter um parâmetro anotado com `@Observes`, os demais parâmetros serão pontos de injeção.

Usando a interface `EventMetadata`

Utilizando a injeção de dependências nos parâmetros dos métodos observadores, podemos utilizar mais uma funcionalidade da CDI 1.1, que é a interface `javax.enterprise.inject.spi.EventMetadata`. Essa interface possui apenas três métodos, como podemos ver a seguir.

```
public interface EventMetadata {  
    javax.enterprise.inject.spi.InjectionPoint    getInjectionPoint();  
    java.util.Set<Annotation>    getQualifiers();  
    java.lang.reflect.Type    getType();  
}
```

O primeiro método devolve o `InjectionPoint` referente ao objeto `Event<T>` que lançou o evento, ou pode retornar `null` se o evento foi disparado diretamente via `BeanManager.fireEvent(Object objetoDoEvento, Annotation... qualificadores)`.

Os outros dois métodos retornam respectivamente os qualificadores e o tipo do evento disparado. Isso porque não necessariamente esses valores serão exatamente iguais aos que estão configurados no método observador. Como já vimos, nosso observador pode ter menos qualificadores que o evento original; assim também como o tipo pode ser compatível, sem ser o mesmo.

Diferente de quando injetamos o `Logger` tanto no bean quanto no método, o `EventMetadata` só pode ser injetado no método observador.

```
public void observaFuncionario(@Observes Funcionario funcionario,  
                               Logger logger, EventMetadata eventMetadata){  
  
    logger.info("Recebendo evento do funcionario {} ", funcionario);  
    InjectionPoint injectionPoint = eventMetadata.getInjectionPoint();  
    if(injectionPoint != null){  
        logger.info("Evento disparado por {} ",  
                    injectionPoint.getMember());  
    }  
}
```

```
}
```

Apesar de estarmos finalizando o capítulo, ainda veremos um pouco mais sobre eventos, e um exemplo real do uso de `BeanManager.fireEvent` na [seção 8.2](#), onde aprenderemos sobre CDI e JSF.

CAPÍTULO 8

A relação da CDI com as demais especificações Java EE

O Java EE é uma coleção de diversas especificações, como EJB, JPA, JSF, Beans Validation, JMS, JTA, e mais uma dúzia de siglas que às vezes assusta quem está começando a trabalhar com Java EE. Mas a boa notícia é que são especificações simples de usar, e a cada versão do Java EE elas ficam mais simples ainda. No meio de todas essas especificações, temos a CDI, que acaba funcionando como um sistema nervoso, ou mais precisamente o cérebro do Java EE, pois está presente em praticamente qualquer outra especificação moderna.

Algumas especificações são praticamente parte da CDI, sendo difícil utilizar esta sem usar os recursos das demais. Um exemplo é a JSR 318, Interceptors 1.2, do qual até já vimos parte de seus funcionamento na seção [6.1](#).

Existe também a especificação “Dependency Injection for Java” (JSR 330), que tem o nome bem parecido com a própria CDI (“Contexts and Dependency Injection for the Java EE platform”). Mas essa, porém, não passa de um especificação-anã,

que apenas padroniza as anotações do pacote `javax.inject` como `@Inject` e `@Qualifier`. Ela serve para outros provedores de injeção de dependência, como Spring, utilizem as mesmas anotações que as utilizadas pelas implementações de CDI, como o Weld, mesmo não implementando a mesma especificação.

A CDI foi introduzida no Java EE 6, e como estava sendo desenvolvida em paralelo com diversas outras especificações, algumas delas, como o JSF 2.0, não eram totalmente compatível com a nova especificação. Porém, agora no Java EE 7, essas integrações foram amadurecidas, fechando pequenas lacunas que existiam nas versões anteriores.

Em todas as integrações, a CDI funciona basicamente como um injetor de recursos da outra especificação, por isso vamos nos ater neste capítulo a algumas integrações que são mais perceptíveis no dia a dia.

8.1 RELAÇÃO ENTRE CDI E EJB/JPA

Quando utilizamos CDI e JPA dentro de um servidor de aplicações, que também tem disponível a implementação de EJB, nem nos damos conta de estar usando essa última quando vamos salvar um objeto no banco de dados. Apesar de termos passado por JPA apenas de passagem em alguns momentos, por este não ser o foco deste livro, sabemos que só conseguimos escrever no banco de dados utilizando JPA se existir uma transação ativa. Sem transações é possível apenas ler dados.

Acontece que quando temos EJBs disponíveis, podemos fazer algo tão simples quanto o seguinte para salvar um objeto.

```
@Stateless
public class FuncionarioBean {
    @Inject
    private EntityManager em;

    public void salvarFuncionario(Funcionario funcionario){
        em.persist(funcionario);
    }
}
```

Basta anotar nosso bean com `@Stateless` ou `@Stateful` para que todos os métodos dentro do bean sejam automaticamente transacionais. Ou seja, no exemplo que acabamos de ver, o método `salvarFuncionario` não precisa lidar com transações, pois isso é feito automaticamente pelo servidor. Ótimo, mas e se estivermos

utilizando CDI e JPA em um servlet container como o Tomcat ou Jetty? Nesse caso teremos que lidar com as transações manualmente, como no exemplo a seguir.

```
public class FuncionarioBean {  
    @Inject  
    private EntityManager em;  
  
    public void salvarFuncionario(Funcionario funcionario){  
        EntityTransaction tx = em.getTransaction();  
        tx.begin();  
        em.persist(funcionario);  
        tx.commit();  
    }  
}
```

Nos dois exemplos, consideramos que existe um produtor CDI que disponibiliza o `EntityManager` para nosso bean, assim como visto na seção 5.1. Principalmente para quem está acostumado à praticidade da gestão automática de transações, e viu como é simples fazer um interceptador com CDI (como na seção 6.1), fica a dúvida: por que a CDI não um interceptador pronto para gerenciar transações? A resposta é que, por mais que seja simples fazer isso, a CDI não oferece nada que sobreponha algo já disponível no Java EE. Se já temos essa funcionalidade nos EJBs, a CDI não oferece uma alternativa.

Isso, no entanto, não significa que não pudéssemos criar esse interceptador quando formos rodar CDI e JPA fora de um servidor de aplicação, ou que não tenhamos extensões prontas que nos oferecem essa funcionalidade. Ainda veremos sobre extensões no capítulo 9, mas elas são basicamente jars que adicionamos à nossa aplicação e assim ganhamos funcionalidades que vão além da CDI padrão.

Para nossa sorte, essa funcionalidade é tão simples de implementar que sequer precisamos lançar mão da funcionalidade de extensão, basta pra isso implementar um interceptador conforme já vimos. Um exemplo de implementação que nos daria essa funcionalidade é o código a seguir.

```
@Interceptor @Transacional  
public class InterceptorGerenciadorDeTransacoes {  
  
    @Inject  
    private EntityManager entityManager;
```

```

@AroundInvoke
public Object gerenciaTransacao(InvocationContext context)
    throws Exception {

    EntityTransaction tx = null;
    try {
        //em ambiente JTA apenas o getTransaction já lança exceção
        tx = entityManager.getTransaction();
        tx.begin();
        Object retorno = context.proceed();
        tx.commit();
        return retorno;
    } catch (RuntimeException e) {
        if(tx != null) tx.rollback();
        throw e;
    }
}
}

```

No nosso exemplo, apenas injetamos no interceptador o mesmo `EntityManager` que está disponível no bean, e então fizemos a gestão manual da transação. Conforme o comentário no código, temos que tomar cuidado para não colocar para executar esse interceptador dentro do servidor de aplicação, pois lá dentro, quem cuida das transações é o *container*. Por isso, é interessante ativar esse interceptador apenas em *servlet containers*.

Para nosso exemplo funcionar, precisamos que nosso `EntityManager` tenha o escopo de requisição, que é o que mais se assemelha ao escopo de transação dos EJBs.

```

public class ProdutorEntityManager {

    @Produces @ApplicationScoped
    private EntityManagerFactory factory = Persistence
        .createEntityManagerFactory("default");

    @Produces @RequestScoped
    public EntityManager criaEntityManager(
        EntityManagerFactory factory){
        return factory.createEntityManager();
    }
}

```

```
public void fechaEntityManager(  
    @Disposes EntityManager entityManager){  
    entityManager.close();  
}  
}
```

Esse escopo é muito importante, pois se não o especificarmos, a CDI atribui o escopo *dependent*, e uma nova instância será criada toda vez que for solicitado. No nosso caso, o problema seria a existência de uma instância de `EntityManager` dentro da classe `FuncionarioBean`, e outra instância dentro do interceptor, fazendo nosso mecanismo não funcionar como o esperado. Depois de tudo, precisamos apenas usar o interceptor.

```
@Transacional  
public class FuncionarioBean {  
    @Inject private EntityManager em;  
  
    public void salvarFuncionario(Funcionario funcionario){  
        em.persist(funcionario);  
    }  
}
```

OU

```
public class FuncionarioBean {  
    @Inject private EntityManager em;  
  
    @Transacional  
    public void salvarFuncionario(Funcionario funcionario){  
        em.persist(funcionario);  
    }  
}
```

O código do *interceptor binding* `@Transacional` não está nesse exemplo, mas já vimos como criá-los quando estudamos interceptadores.

Agora se nossa aplicação trabalhar com mais de um banco de dados, teremos mais de um `EntityManager` disponível. Nesse caso, precisaremos fazer uma injeção mais elaborada no nosso interceptor para conseguirmos utilizar o objeto correto. Como ainda não vimos como fazer isso, vamos retomar esse assunto na seção 4.8, onde veremos como fazer injeções programáticas.

Se quisermos disponibilizar nosso interceptador para qualquer aplicação, basta colocar o arquivo `beans.xml` dentro da pasta `META-INF` do jar que irá conter nosso interceptador. Com isso, quando o contexto CDI iniciar na aplicação que usa nossos jar, por termos esse arquivo nesse lugar, o jar será considerado um pacote CDI, e o interceptor e qualquer outro bean que estiver dentro do jar ficará disponível na aplicação.

Porém como já vimos na seção 6.1, não basta definir o interceptador, precisamos ativá-lo. Isso pode ser feito no arquivo `beans.xml` ou simplesmente adicionando a anotação `@Priority` ao interceptador. Mas se tivermos tanto aplicações que rodam em *servlet container* quando em servidores de aplicação, o mais indicado seria ativarmos via XML somente no ambiente correto; uma vez que a anotação já ativa o interceptador por padrão.

8.2 RELAÇÃO ENTRE CDI E JSF

O JSF 2.0 e o CDI 1.0 tiveram alguns problemas de integração, pois foram desenvolvidos paralelamente na época do lançamento do Java EE 6. Agora no Java EE 7, com JSF 2.2 e CDI 1.1, esses problemas foram superados. O único ponto que ainda temos algum atrito é com as anotações de escopo do JSF, que se encontram no pacote `javax.faces.bean`, pois elas têm a mesma função dos escopos definidos no pacote `javax.enterprise.context`. Tanto é que a documentação do JSF 2.2 já informa que as anotações desse pacote devem ser depreciadas em novas versões da especificação, e nos sugere usar as anotações do pacote CDI sempre que possível. Até porque, por conta dessa sobreposição, a CDI não reconhece as anotações de escopo do pacote do JSF.

Utilizando as últimas versões disponíveis até o momento — JSF 2.2 e CDI 1.1 — não teremos os problemas de incompatibilidade, que são principalmente a incapacidade de injetarmos recursos através da CDI dentro de conversores e validadores JSF. De qualquer forma, mesmo utilizando as versões anteriores conseguimos contornar essa deficiência utilizando extensões da CDI como DeltaSpike da fundação apache (<http://deltaspike.apache.org/jsf.html>). Porém, veremos mais de extensões no capítulo 9, por enquanto basta sabermos que essas limitações não precisam impactar nossas aplicações.

Agora que já falamos do histórico, das antigas limitações e que as novas versões as resolveram, vamos explorar o que podemos fazer com a integração entre essas duas especificações. Para iniciarmos de uma forma leve, podemos criar mé-

todo produtores para objetos do contexto JSF que muitas vezes precisamos buscar via API, como por exemplo `FacesContext`, `ExternalContext`, ambos do pacote `javax.faces.context`, e `javax.faces.application.Application`. Esses, é claro, são apenas alguns exemplos, podemos fazer isso para qualquer outro bean.

```
public class ProdutorFaces {

    @Produces
    private FacesContext context = FacesContext.getCurrentInstance();

    @Produces
    private ExternalContext externalContext = FacesContext
        .getCurrentInstance().getExternalContext();

    @Produces
    private Application application = FacesContext
        .getCurrentInstance().getApplication();
}
```

Podemos ainda criar qualificadores representando os escopos que podemos acessar via `ExternalContext` e injetar diretamente mapas dentro dos nossos beans.

```
@Inject @ApplicationMap
private Map<String, Object> applicationMap;

@Inject @SessionMap
private Map<String, Object> sessionMap;

@Inject @RequestMap
private Map<String, Object> requestMap;

@Inject @RequestParamMap
private Map<String, String> requestParameterMap;
```

E para disponibilizarmos esses objetos, basta incrementarmos mais nosso `ProdutorFaces` com os seguintes métodos.

```
@Produces @ApplicationMap
public Map<String, Object>
```

```

        disponibilizaApplicationMap(ExternalContext ec){
            return ec.getApplicationMap();
        }

@Produces @SessionMap
public Map<String, Object> disponibilizaSessionMap(ExternalContext ec){
    return ec.getSessionMap();
}

@Produces @RequestMap
public Map<String, Object> disponibilizaRequestMap(ExternalContext ec){
    return ec.getRequestMap();
}

@Produces @RequestParameterMap
public Map<String, String>
    disponibilizaParameterMap(ExternalContext ec){
    return ec.getRequestParameterMap();
}

```

Como acabamos de ver, é bem simples e útil fazer esse tipo de integração, e a única coisa que usamos foi a funcionalidade da CDI de criar produtores. Esses exemplos deixam bem claro como a CDI é ao mesmo tempo abrangente — pois usamos com qualquer coisa dentro do Java EE —, poderosa, e o mais importante, simples.

Observadores para eventos do JSF

Criar produtores para os objetos mais acessados do JSF é bastante simples, mas agora vamos dar um passo além nessa integração. Trabalhando com JSF, em alguns momentos precisamos interagir com o ciclo de vida de uma requisição, fazendo alguma operação depois da fase *Restore View* ou antes da *Render Response*, apenas para exemplificar. Porém, para fazer isso não podemos usar, pelo menos não de forma natural, nossos beans “de negócio”, que são POJOs; e sim implementar a interface `PhaseListener`.

```

public class SegurancaPhaseListener implements PhaseListener {

    @Override
    public void afterPhase(PhaseEvent event) {
        //verifica se o usuário está logado
        //se não estiver, envia para tela de login
    }
}

```

```
    }

    @Override
    public void beforePhase(PhaseEvent event) {
    }

    @Override
    public PhaseId getPhaseId() {
        return PhaseId.RESTORE_VIEW;
    }
}
```

Nesse exemplo, criamos um listener que vai verificar se o usuário está logado ou não e, caso não esteja, enviá-lo à tela de login. O código que faz isso não importa, pois varia em cada organização. O interessante aqui é analisarmos como temos que fazer, e como seria interessante fazermos, como no exemplo abaixo.

```
public class SegurancaListener {

    public void verificaLogin(
        @Observes @After @RestoreView PhaseEvent e){
        //executa a lógica de checagem do login
    }
}
```

A vantagem do segundo exemplo, é que podemos lidar com os eventos do ciclo de vida do JSF da mesma forma que fazemos com qualquer outro evento, ou seja, dentro dos nossos beans. Uma outra limitação que contornamos com essa mudança, é que antes da versão 2.2 do JSF, não era possível termos pontos de injeção dentro dos `PhaseListeners`, o que nos limitava bastante. Já usando eventos CDI, mesmo usando JSF anterior, não tínhamos esse problema pois quem escuta o evento já é um bean CDI. Essa limitação no entanto é facilmente contornada utilizando o módulo JSF da já citada extensão `DeltaSpike`.

Mas como podemos, sem a necessidade de qualquer extensão disparar um evento CDI equivalente a um `PhaseEvent` do JSF? A implementação não é difícil, e usaremos os mesmos mecanismos de eventos que já vimos até aqui. Para avançarmos aos poucos, vamos iniciar apenas com a implementação que resolve exatamente o cenário que queremos observar, que é o momento logo após o término da fase *Restore View*.

```

public class EventPhaseListener implements PhaseListener {

    @Inject
    @After @RestoreView
    private Event<PhaseEvent> afterRestoreView;

    @Override
    public void afterPhase(PhaseEvent event) {
        afterRestoreView.fire(event);
    }

    @Override
    public void beforePhase(PhaseEvent event) {
    }

    @Override
    public PhaseId getPhaseId() {
        return PhaseId.RESTORE_VIEW;
    }
}

```

Somente isso é suficiente para resolver nosso problema. Contudo, olhando o exemplo, vemos que usamos injeção de dependências dentro do `PhaseListener`, e caso você precise fazer isso dentro de uma aplicação que não utilize a versão 2.2 do JSF, você terá que lançar esses eventos de outra maneira.

Para isso precisaremos do objeto `BeanManager`, que pode ser obtido via injeção (que não está disponível no nosso cenário), ou via lookup com base em seu nome JNDI. A obtenção dessa instância será feita através do método `CdiUtils.getBeanManager()`, que por enquanto irá abstrair as diferenças desse processo quando estamos em ambiente Java EE ou não. Não se preocupe, pois veremos esses detalhes quando abordarmos como usar CDI em diferentes ambientes, no capítulo 10.

```

public class EventPhaseListener implements PhaseListener {

    @Override
    public void afterPhase(PhaseEvent event) {
        BeanManager beanManager = new CdiUtils().getBeanManager();

        beanManager.fireEvent(event, new AnnotationLiteral<After>(){}),
    }
}

```

```

        new AnnotationLiteral<RestoreView>(){});
    }
    ...
}

```

Agora nosso código funciona em ambientes pré JSF 2.2, e apesar de mais trabalhoso, o processo para enviar o evento se baseia em algo que já vimos na seção 7.4, que são os literais de qualificadores. Para usar o método `BeanManager.fireEvent` basta passarmos o objeto que representa o evento, e seus qualificadores, que independente de termos ou não usado injeção de dependência, são o `After` e o `RestoreView`.

Como já sabemos lançar eventos dentro do `PhaseListener` em qualquer versão do JSF, voltaremos a incrementar a `EventPhaseListener` para que agora ela dispare para o contexto CDI evento referentes a todas as fases do JSF. Como esse livro é baseado em Java EE 7, usaremos o exemplo do JSF 2.2 ou superior, que suporta injeção de dependências no `PhaseListener`, mas com o que vimos seremos capazes de adaptar o exemplo para outros cenários.

Quando estudamos sobre eventos, e também agora quando vimos como usar uma versão mais antiga de JSF, usamos `AnnotationLiterals` para diminuir a quantidade de injeções que precisamos usar para implementar nossa solução. Em vez de usarmos algo parecido com o primeiro exemplo a seguir, faremos algo mais parecido com o segundo.

```

@Inject
@Before @RestoreView
private Event<PhaseEvent> beforeRestoreView;

@Inject
@After @RestoreView
private Event<PhaseEvent> afterRestoreView;

@Inject
@Before @ApplyRequestValues
private Event<PhaseEvent> beforeApplyRequestValues;

...

```

Seguindo essa linha, teríamos pelo menos 12 (doze) pontos de injeção na nossa classe, pois teríamos o *before* e o *after* para cada uma das 6 (seis) fases do JSF. No lugar dessa abordagem, utilizaremos a forma programática como podemos ver a seguir.

```
public class EventPhaseListener implements PhaseListener {

    @Inject
    private Event<PhaseEvent> event;

    @Override
    public PhaseId getPhaseId() {
        return PhaseId.ANY_PHASE;
    }

    @Override
    public void afterPhase(PhaseEvent event) {
        disparaEvento(event, new AnnotationLiteral<After>(){});
    }

    @Override
    public void beforePhase(PhaseEvent event) {
        disparaEvento(event, new AnnotationLiteral<Before>(){});
    }

    public void disparaEvento(PhaseEvent phaseEvent,
                              Annotation qualificadorAntesOuDepois){

        Annotation qualificadorFase = null;

        if(RESTORE_VIEW.equals(phaseEvent.getPhaseId())){
            qualificadorFase = new AnnotationLiteral<RestoreView>(){};
        }
        else if(APPLY_REQUEST_VALUES.equals(phaseEvent.getPhaseId())){
            qualificadorFase =
                new AnnotationLiteral<ApplyRequestValues>(){};
        }
        else if(PROCESS_VALIDATIONS.equals(phaseEvent.getPhaseId())){
            qualificadorFase =
                new AnnotationLiteral<ProcessValidations>(){};
        }
        else if(UPDATE_MODEL_VALUES.equals(phaseEvent.getPhaseId())){
            qualificadorFase =
                new AnnotationLiteral<UpdateModelValues>(){};
        }
        else if(INVOKE_APPLICATION.equals(phaseEvent.getPhaseId())){
```

```
        qualificadorFase =  
            new AnnotationLiteral<InvokeApplication>(){};  
    }  
    else if (RENDER_RESPONSE.equals(phaseEvent.getPhaseId())){  
        qualificadorFase =  
            new AnnotationLiteral<RenderResponse>(){};  
    }  
  
    event.select(qualificadorAntesOuDepois)  
        .select(qualificadorFase)  
        .fire(phaseEvent);  
}  
}
```

Ficou um trecho de código um pouco longo, mas apesar de não ser muito estético, com muitos `if else`, é simples de entender. Primeiramente alteramos o retorno do método `getPhaseId()` para informar que nosso *listener* escuta todas as fases do JSF. Depois apenas repetimos algo que já vimos quando estudamos eventos: injetamos um evento genérico, sem qualificadores, e depois utilizamos o método `select(Annotation qualificador)` para adicionar os qualificadores no nosso evento antes de dispará-lo. Perceba como a interface fluente facilita nosso trabalho — basta chamar duas vezes o método, sem precisar criar um *array* ou lista para passar os qualificadores de uma vez.

É possível explorar ainda muito mais possibilidades de integração entre CDI, JSF e JPA, mas o intuito dessas sessões foi mostrar o caminho, e deixar que seus conhecimentos de CDI adquiridos até aqui sugiram outros cenários. Além do mais, se entrarmos em muitos detalhes de JSF e JPA, desviaremos o foco do livro, que é CDI. Mas caso também queira aprofundar seus conhecimentos nessas duas ferramentas, dê uma olhada no meu livro da Casa do Código sobre esse assunto [?].

8.3 RELAÇÃO ENTRE CDI E JMS

No Java EE 7, a especificação de JMS — serviço de mensageria Java — ganhou uma nova API, bem mais fácil de usar, e bem integrada com CDI. Nessa seção iremos ver como utilizar essa poderosa ferramenta junto com a funcionalidade de eventos da CDI.

A nova API de JMS, que veio na versão 2.0 da especificação, trouxe para ela a

facilidade que os *Session Beans* passaram a ter a partir da sua versão 3.0. Isso por si já pode ser um fator que ajude a aumentar a utilização de mensageria dentro das aplicações Java corporativas. É verdade que somente o uso de métodos de negócio assíncronos já facilita bastante nosso trabalho em execuções mais custosas, mas JMS vai além.

O serviço de mensageria Java dá a nossa aplicação uma capacidade de processamento assíncrono muito grande. Para exemplificar, vamos imaginar uma aplicação de processamento de pagamentos de uma outra aplicação de comércio eletrônico. Apesar das interfaces da atualidade nos dar a impressão de ser tudo síncrono, vamos pensar mais especificamente em boletos, que naturalmente são assíncronos, geralmente sendo processados com um dia de atraso.

Imagine que sua aplicação tem um alto poder de processamento, porém devido a uma super promoção de um grande site de vendas, as requisições começaram a chegar em um volume muito maior do que sua aplicação consegue processar. O cenário poderia ser catastrófico, sua aplicação poderia parar de receber requisições enquanto tenta processar a demanda, e o site de vendas poderia perder inúmeros negócios pois não conseguiria terminar a venda, que depende do sistema de pagamentos. Dependendo do caso, caberia até processo.

O JMS, assim como qualquer outro serviço, não é a Panaceia que deixará tudo em ordem em um piscar de olhos, mas ajudará e muito. No nosso exemplo hipotético, a aplicação de vendas, não solicitaria para nossa aplicação processar o pagamento e ficaria esperando. Ela registraria a venda na fila de processamento e responderia que em instantes o usuário receberia a notificação ou então faria uma tela de progresso que faz o usuário aguardar alguns instantes para dar a impressão que o processo é “online”.

Essa fila geralmente é persistente, ou seja, se tivermos mil vendas para serem processadas e o servidor cair, quando este voltar ainda teremos os mil itens na fila. Além disso, a fila é gerenciada automaticamente pelo servidor de aplicação. Enquanto existirem elementos na fila, o bean responsável por processar as vendas será chamado. E o que dá poder a essa ferramenta é que há, na verdade, um *pool* de beans de um mesmo tipo, paralelizando o processamento o máximo possível para sua aplicação dar conta da demanda. Esses beans que processam elementos de uma fila, ou tópico, são os MDBs, ou *Message-Driven Beans*.

A principal diferença de um *Message-Driven Bean* para um *Session Bean* é que a aplicação de vendas não acessa diretamente o MDB como faria com o EJB. Ela coloca a solicitação na fila, e o container é que chama o MDB para processar o próximo

elemento da fila. Assim como os EJBs *Stateless* os MDBs também não mantêm estado, e por isso em ambos temos o uso do *pool*, que é um dos responsáveis pela boa responsividade.

Esse livro, porém, não tem como finalidade mostrar como utilizar JMS para aumentar a escalabilidade das nossas aplicações, por hora veremos como criar uma ponte entre os eventos CDI e as mensagens JMS.

Uma das principais diferenças entre os eventos CDI e as mensagens JMS é que os eventos só existem dentro do contexto de uma aplicação, enquanto as mensagens podem ser consumidas por diversas aplicações. Isso porque a o tópico ou a fila não são privativos de uma aplicação, são recursos que ficam disponíveis para qualquer aplicação que conseguir se comunicar com o servidor onde elas estão. Em cenários maiores teremos inclusive servidores dedicados exclusivamente a processar filas e tópicos.

Nessa seção, usaremos a facilidade dos eventos e a disponibilidade das mensagens para fazer com que o evento lançado por uma aplicação seja escutado por outra.

Comunicando duas aplicações com uma ponte CDI - JMS

Dentro de um mesmo sistema podemos utilizar eventos para integrar módulos diferentes com um baixo acoplamento, mas quando pensamos em aplicações diferentes temos ainda mais opções. Por exemplo, é possível, ao cadastrar um novo funcionário, disparar em outros sistemas a tarefa de criação de conta no controlador de domínio e liberar seu acesso na cancela do estacionamento. E no momento do desligamento do funcionário cancelar todos os acessos anteriormente concedidos. Pode até ser possível ter todos esses controles no mesmo sistema, mas o mais comum não é esse.

No exemplo que vamos analisar agora, vamos considerar a integração com uma agência de empregos. Quando um funcionário se desligar da empresa, e não for por aposentadoria, nós notificaremos a agência para que ela já inicie a busca pela recolocação do profissional no mercado de trabalho. Como nossa aplicação só está tratando aposentadoria e demissão, vamos fazer em cima do exemplo de demissão, mas seria possível olhar qualquer evento de desligamento e usar o `EventMetadata` para ignorarmos os desligamentos que tivessem qualificador de aposentadoria.

Conceitualmente, o que desejamos é algo assim.

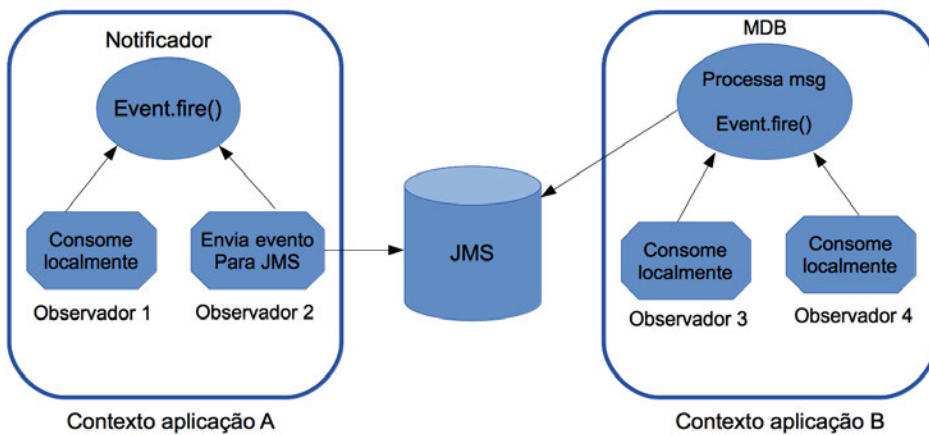


Figura 8.1: Evento da aplicação A sendo consumido pela aplicação B via JMS

Pela imagem vemos um evento produzido na aplicação A, sendo consumido por 4 (quatro) observadores, dois deles da aplicação B. Via código o esperado seria um código semelhante ao seguinte.

```
public class GerenciadorFuncionarioBean {

    @Inject @Desligamento(DEMISSAO)
    private Event<Funcionario> eventoDemissao;

    public void demitirFuncionario(Funcionario funcionario){
        eventoDemissao.fire(funcionario);
    }
}
```

E o seguinte código já no sistema da agência de empregos:

```
public class AgenciaDeEmpregos {

    public void ajudarFuncionarioDemitido(
        @Observes @Desligamento(DEMISSAO) Funcionario funcionario,
        Logger logger){

        logger.info("Ajudando {} a conseguir um novo emprego",
            funcionario);
    }
}
```

```
}  
}
```

Para que isso funcione, no entanto, vamos precisar colocar na nossa aplicação um observador que escute esse evento e cria uma mensagem na fila JMS, e na aplicação da agência de empregos colocar um MDB que leia a mensagem e crie novamente o evento nos mesmos moldes do original. Para simplificar, criaremos nossa integração de forma específica para esse cenário, mas com um pouco de abstração e reflexão é possível criar implementações genéricas que atendem bem a maioria dos casos.

Iniciaremos pelo código que cria a mensagem JMS. Nosso exemplo será baseado na API simplificada do JMS 2.0, mas é possível portar o código para a API tradicional caso esteja executando em ambiente pré Java EE 7.

```
@Stateless  
public class ObservadorDeFuncionarios {  
  
    @Inject  
    private JMSContext jmsContext;  
  
    @Resource(mappedName = "jms/demissaoTopic")  
    private Topic topicoDemissao;  
  
    @Asynchronous  
    public void notificaAgenciaDeEmprego(  
        @Observes @Desligamento(DEMISSAO) Funcionario funcionario,  
        Logger logger) throws Exception{  
  
        logger.info("Enviando o(a) {} para agencia de empregos",  
            funcionario);  
        jmsContext.createProducer().send(topicoDemissao, funcionario);  
    }  
}
```

Estamos usando um tópico JMS que está no mesmo servidor que nossa aplicação, e recuperamos esse tópico via `@Resource(mappedName = "jms/demissaoTopic")`. Tópicos e filas devem ser criados no servidor de aplicação que estivermos utilizando, não é algo da aplicação. No glassfish, só precisamos tomar o cuidado de não dar o mesmo nome JNDI e nome físico para o tópico ou fila, senão o nome não será encontrado. O valor que passamos para o atributo `mappedName` da anotação `@Resource` é o nome JNDI, e não o físico.

Perceba que fizemos nosso observador ser assíncrono, pois assim, caso passemos a utilizar um servidor JMS remoto, uma possível degradação no tempo de resposta ao colocar a mensagem na fila não impactará no método de negócio.

A implementação como um todo ficou bem simples. Usamos o `JMSContext` para criar um *producer* de mensagens e em seguida enviamos o objeto funcionário, que deve ser serializável, para o tópico de demissões. Agora no lado da agência de empregos teremos um MDB que escuta esse tópico e recria a mensagem no contexto daquela aplicação.

```
@MessageDriven(mappedName = "jms/demissaoTopic")
public class AgenciaDeEmpregosMessageBean implements MessageListener {

    @Inject @Demissao
    private Event<Funcionario> eventoDemissao;

    @Inject
    private Logger logger;

    @Override
    public void onMessage(Message message) {
        try {
            Funcionario funcionario =
                message.getBody(Funcionario.class);

            logger.info("Recebida a demissão do funcionario {} via JMS",
                funcionario);

            eventoDemissao.fire(funcionario);

        } catch (JMSException ex) {
            logger.error("Erro ao ler a mensagem via JMS", ex);
        }
    }
}
```

Um MDB, como acabamos de ver, é simplesmente um bean anotado com `@javax.ejb.MessageDriven` e que implementa a interface `javax.jms.MessageListener`. No corpo do método `onMessage(javax.jms.Message)` precisamos apenas recuperar o con-

teúdo da mensagem, que nesse caso é um objeto do tipo `Funcionario`. Para isso, utilizamos o método `getBody` que foi adicionado à versão 2.0 de JMS para facilitar o trabalho. Com o funcionário em mãos, e sabendo que essa mensagem é referente a uma demissão, basta lançar um novo evento CDI.

Para abstrairmos essa funcionalidade, poderíamos criar um observador global do lado da nossa aplicação, e via `EventMetadata` criar um objeto que represente o evento. Em vez de termos o objeto funcionário, teríamos também os qualificadores utilizados. Na outra aplicação, no exemplo a agência de empregos, leríamos esse objeto que representa o evento e, antes de lançá-lo no contexto CDI, especificaríamos os qualificadores de forma programática como já fizemos algumas vezes. Além dessas alterações, criaríamos também um tópico com título mais genérico, como por exemplo `"jms/cdiEventsTopic"` ou algo parecido.

Perceba também que para diferenciar mais uma aplicação da outra, utilizamos o qualificador `@Desligamento (DEMISSAO)`, e na agência de empregos utilizamos `@Demissao`. Para facilitar a criação de algo genérico, que possa ser compartilhado pelas diversas aplicações da nossa organização, o melhor é criarmos um jar com o conjunto de qualificadores e classes que serão compartilhadas entre as aplicações. Geralmente esses jars não possuem qualquer classe inteligente, apenas tipos compartilhados, e são chamados de `<nomeDaAplicacaoDeOrigem>-client.jar`. Obviamente essa é apenas uma sugestão, mas fica simples perceber que todas as aplicações que precisarão se comunicar com a `nossaApp` precisam do `nossaApp-client.jar`.

8.4 EMPACOTANDO AS MELHORIAS

Todos os exemplos que vimos nesse capítulo são passíveis de empacotamento, ou seja, podemos colocar essas pequenas melhorias em um projeto separado, gerar um jar, e colocar nas nossas aplicações. Qualquer aplicação que utilizar esse jar terá automaticamente suporte à injeção de recursos da JPA e JSF, e fazendo as generalizações comentadas na última seção, poderíamos ter eventos CDI passando de uma aplicação para a outra via JMS.

Implementar isso não é nada complicado — basta colocar os arquivos de configuração no local correto. Enquanto nas aplicações web os arquivos `faces-config.xml` e `beans.xml` ficam geralmente na pasta `WEB-INF`, ao criar uma biblioteca precisamos colocar esses arquivos dentro da pasta `META-INF` que fica na raiz do jar.

Assim, mesmo ainda sem termos visto o mecanismo de extensões propriamente dito, já conseguiremos criar bibliotecas de apoio com muita funcionalidade interessante.

CAPÍTULO 9

Extensões CDI

Uma especificação nada mais é do que uma padronização de como fazer determinada tarefa. A JPA padroniza o mapeamento objeto-relacional, JSF padroniza como desenvolver clientes web com um sofisticado gerenciamento de estado e com CDI temos a padronização de como injetar dependências e gerenciar escopos dentro do Java EE. Porém, sempre que algo é padronizado, a tendência é limitar a inovação. Como, por exemplo, inovar no mapeamento objeto-relacional se a JPA define as funcionalidades?

A forma encontrada para que a inovação continue apesar das especificações é através da adição de funcionalidades não especificadas. Estas adicionam funcionalidades importantes às implementações das especificações. O exemplo mais notório é o da JPA, onde temos o EclipseLink (<http://www.eclipse.org/eclipselink>) como implementação de referência, e o Hibernate (<http://hibernate.org>) como a implementação mais utilizada.

Prosseguindo na análise da JPA, temos funcionalidades não especificadas nessas, e em outras implementações, que adicionam características muito importantes em

cada implementação. No Hibernate temos o cache de consulta, que nos permite evitar diversas consultas desnecessárias no banco de dados, além do cache de segundo nível de entidades que foi adicionado à especificação depois que já estava disponível no Hibernate. Já no EclipseLink temos o suporte a *multitenant*, que nos permite escolher diferentes estratégias para fazer nossa aplicação funcionar com diferentes clientes. Isso permite que uma mesma instância de aplicação de vendas seja utilizada por diferentes sites de *e-commerce*, por exemplo.

Qual o problema de uma implementação adicionar funcionalidades não especificadas? Praticamente nenhum, desde que você não troque de implementação. O “detalhe” é que a possibilidade de trocar de implementação é uma das vantagens da especificação, e acabamos perdendo isso. Se por exemplo utilizarmos o suporte a cache de consulta do Hibernate, caso mudemos para o EclipseLink teremos que fazer alguma alteração na nossa aplicação para usar o suporte equivalente do EclipseLink. Há casos também em que uma implementação não tem um equivalente à funcionalidade da outra. Por exemplo, até o momento o Hibernate não tem um suporte a *multitenant* tão elaborado quanto o do EclipseLink.

Qual a diferença do suporte a extensões da CDI?

Como a CDI é uma especificação mais recente, inteligentemente adicionou o suporte a extensões portáteis, de forma que implementadores da especificação, ou mesmo qualquer um de nós, pudessem desenvolver funcionalidades além das especificadas que não dependam de uma implementação, e sim da própria especificação.

Voltando ao paralelo com a JPA, seria como se pudéssemos adicionar a funcionalidade de *multitenant* do EclipseLink enquanto usamos o Hibernate como implementação da JPA. Isso nos permite ao mesmo tempo desfrutar da inovação constante que as implementações das especificações nos oferece, sem, no entanto, ficarmos presos a essas implementações. Usamos a implementação e as funcionalidades adicionais, mas na hora de trocar de implementação, continuamos com os adicionais. Essa sofisticação foi resultado da evolução do framework Seam, desenvolvido pela JBoss, e que acabou servindo de base para a CDI.

9.1 DO SEAM À CDI

O JBoss Seam foi um framework criado com o objetivo de aparar as arestas existentes entre as diversas especificações do Java EE 5. Essa versão do Java EE foi focada na facilidade de implementação, e foi um divisor de águas do Java para aplicações cor-

porativas, pois saiu de algo extremamente verboso, cheio de XML, para algo simples de implementar utilizando as anotações, que haviam sido introduzidas pouco antes no Java 5.

A utilização do Java EE ficou bem mais simples, com JPA 1.0, JSF 1.2 e EJB 3.0. Porém, essas especificações não tinham um trabalho de integração muito elaborado, e aí entrava o Seam, que podemos traduzir para junção, integração.

O Seam fez bastante sucesso, e além de simplesmente integrar as diversas especificações do Java EE, passou a oferecer uma coleção de utilidades, como geração facilitada de PDFs, segurança, integração com BPM (workflow) e engine de regras (Drools), dentre outras. Enquanto algumas dessas utilidades são integrações com outros frameworks, que podem ser utilizados separadamente, como o *Drools* (<http://www.jboss.org/drools>) e o *jBPM* (<http://www.jboss.org/jbpm>), outras coisas como o suporte a escrita de PDFs através de tags, como se fosse um HTML, era algo totalmente integrado ao Seam.

Ocorre que, conceitualmente, muitas dessas utilidades eram acessórias, e seria interessante poder utilizá-las com qualquer framework. Enquanto isso, o Seam possuía um núcleo que era o responsável pela injeção de dependências; núcleo este que “concorria” com o núcleo de outros frameworks como o Spring (<http://www.springsource.org/spring-framework>).

Como o Java EE não possuía uma especificação que padronizasse esse mecanismo, em vez de desenvolver uma nova versão do Seam evoluindo tanto o *core* quanto os utilitários, a JBoss liderou a especificação da CDI como substituta do núcleo do Seam. E como era desejável que os utilitários do Seam não se perdessem, mas também adicionar tudo aquilo na especificação desviaria seu foco, a solução foi tratá-los como extensões. Dessa forma, a CDI já nasceu com a ideia de ser extensível. A seguir é possível ver uma imagem, que ficava disponível no site do Seam, que ilustra essa evolução do Seam 2, para o Seam 3, que passou a ser apenas uma coleção de extensões.

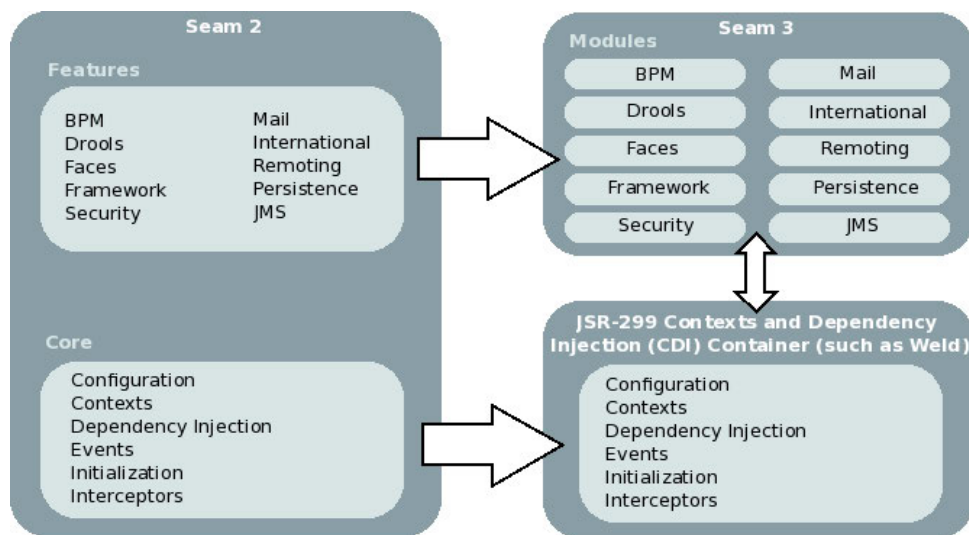


Figura 9.1: Evolução do Seam 2 até a CDI e o Seam 3

A JBoss no entanto descontinuou o Seam 3, e passou o projeto para a Fundação Apache, que o chamou de DeltaSpike (<http://deltaspike.apache.org>). Alguns módulos, entretanto, tornaram-se bibliotecas independentes, como pode ser visto na página do Seam (<http://www.seamframework.org>).

9.2 EXTENSÕES PORTÁVEIS DISPONÍVEIS

O projeto DeltaSpike ainda está bem recente, e apesar de ter se baseado em versões completas dos módulos do Seam 3, ainda não tem uma versão estável disponível. Ainda é possível acessar os módulos do Seam 3, mas eles não são mais atualizados, então bugs não serão corrigidos nem novas funcionalidades introduzidas. Mesmo assim pode ser uma opção caso o DeltaSpike não ofereça a funcionalidade que procuramos.

Há ainda o MyFaces CODI (<https://bitly.com/myfaces-codi>), também da apache, que tem diversas funcionalidades mais voltadas para JSF, como podemos ver na wiki do projeto (<https://bitly.com/codi-wiki>). Uma das principais vantagens do CODI, são os diferentes escopos oferecidos, como uma conversação diferente da padrão do CDI, com suporte a *restart*, subconversações, grupo de conversações dentre outras peculiaridades. Temos também escopos *window* e *view access*. A documen-

tação dessas funcionalidades está disponível na wiki do projeto CODI, no módulo JSF. Certamente esta é uma página que vale a pena ser visitada.

Há também esta lista com diversas extensões CDI: <https://groups.diigo.com/group/cdi-extensions>. Aqui é possível encontrar link para as extensões do Seam 3, que já não estão mais listadas na página do projeto, além de outros que apontam para posts de blogs e repositórios (github ou outros) que possuem as mais diversas implementações de extensões CDI.

Mas o que podemos esperar dessas extensões? Além de algumas inovações, o mais comum é trazer para o ambiente não Java EE, funcionalidades que geralmente já estão disponíveis nesse ambiente. Nesse caso não temos algo novo, apenas conseguimos, por exemplo, desfrutar de gerenciamento automático de transações dentro de um servlet container como Tomcat ou Jetty.

Temos também, no exemplo do JSF, um adiantamento de funcionalidades que viriam somente em versões seguintes das especificações. Por exemplo, a maioria das extensões integradas com JSF permitem a injeção de dependências dentro de conversores, validadores e `PhaseListeners` em versões anteriores ao JSF 2.2, quando isso ainda não era possível nativamente.

9.3 CRIANDO NOSSAS EXTENSÕES

Criar uma extensão CDI é simples, e faremos dois exemplos que provarão isso. Antes vamos entender o que é preciso para criar uma extensão. A CDI faz uso de uma funcionalidade adicionada no Java 6, que é uma forma simplificada, e padronizada, de carregar a implementação de um serviço qualquer. É bem parecido com o que vimos lá no início, na seção 1.3, onde carregávamos a implementação de uma interface de um arquivo properties.

Essa funcionalidade é oferecida através da classe `java.util.ServiceLoader`, e seu funcionamento é simples, basta criarmos um arquivo texto com o nome da interface (ou classe abstrata) que estamos implementando, e colocá-lo dentro da pasta `META-INF/services`. No nosso caso, estaremos implementando a interface `javax.enterprise.inject.spi.Extension`, então todas as extensões CDI estarão no arquivo `META-INF/services/javax.enterprise.inject.spi.Extension`. Dentro desse arquivo criaremos uma linha com o nome completo de cada classe que implementa a interface em questão.

A interface `Extension` da CDI não possui qualquer método para implemen-

tarmos, é apenas o que chamamos de interface de marcação. Em vez de usarmos método fixos, a CDI definiu diversos eventos que podemos observar durante a subida da aplicação, e então fazer as modificações que desejarmos. Existem diversos eventos que podemos observar, tais como:

- `BeforeBeanDiscovery`
- `AfterBeanDiscovery`
- `ProcessAnnotatedType`
- `ProcessBeanAttributes`
- `ProcessProducer`
- `ProcessObserverMethod`
- `ProcessInjectionPoint`

Além desses, temos muitos outros que podem ser encontrados no pacote `javax.enterprise.inject.spi`. Aliás, agora que estaremos desenvolvendo extensões, passaremos a olhar bastante para os pacotes `spi` que existem dentro da CDI. SPI significa *Service Provider Interface*, e como a partir de agora estaremos desenvolvendo extensões que proveem serviços dentro da CDI, é essa interface que usaremos. Interface aqui no sentido mais amplo, como um ponto de vista da API de CDI.

O funcionamento básico das extensões é observar determinado evento, e modificar o objeto detectado pela CDI. Como o mecanismo de extensão é projetado para nos permitir alterar o contexto original, nossas extensões executam logo na subida da aplicação, antes da injeção de dependência estar disponível, e temos acesso a objetos modificáveis que representam o conteúdo até então. Ou seja, podemos modificar as anotações de um bean que acabou de ser lido, ou mesmo acessar a listagem dos beans e remover, ou adicionar, um elemento lá dentro do contexto.

Por mais que este mecanismo de extensão seja muito bem projetado, e simples de usar, ainda existem detalhes que não estão disponíveis na *SPI*, nos obrigando a utilizar classes específicas da implementação, que no nosso caso é a implementação de referência, o **Weld**. Quando precisamos utilizar desses mecanismos, temos as chamadas extensões não portáveis, pois estamos nos prendendo à implementação. Isso muitas vezes não é um problema, até porque já é comum nos prender a uma implementação quando usamos JPA, por exemplo. Mas como extensões portáveis são

uma evolução, a tendência é esse mecanismos continuar evoluindo para precisarmos cada vez menos desenvolver extensões não portáveis.

Nos exemplos que iremos desenvolver, veremos uma extensão portátil, que pode ser integrada em um projeto que usa por exemplo *OpenWebBeans* (implementação de CDI da Apache); e outra não portátil, que faz uso de classes do *Weld*. Por sorte, os dois principais servidores de aplicação open-source, JBoss/Wildfly e Glassfish, utilizam o *Weld* como implementação.

9.4 EXTENSÃO PORTÁVEL: USANDO CONVENÇÃO EM VEZ DE CONFIGURAÇÃO

Provavelmente você conhece um conceito muito utilizado hoje em dia que é a utilização de convenção em vez de configuração. Sabemos que configurar algo com anotação é bem simples, ainda mais quando temos estereótipos que podem combinar diversas configurações em uma única anotação.

Mesmo tendo meios bem simples de configurar, vamos criar uma convenção: qualquer classe dentro de um pacote `controllers` ou que tenha nome terminado em `Controller` deve receber o estereótipo `@Controller`. Esse estereótipo apenas define o escopo `request` e adiciona um nome ao bean. Na prática é igual ao estereótipo `@Model` que já vem na CDI, mas como já vimos na seção 7.1, podemos usar nosso estereótipo próprio para adicionar interceptadores e até mesmo outros estereótipos. Por esse motivo vamos usar essa anotação, em vez de separadamente adicionar `@Named` e `@RequestScoped` ao nosso bean. Então vamos ao exemplo.

```
public class ControllerPorConvensaoExtension implements Extension {

    private Logger logger = LoggerFactory
        .getLogger(ControllerPorConvensaoExtension.class);

    void configuraControllers(@Observes ProcessAnnotatedType pat) {
        AnnotatedType at = pat.getAnnotatedType();

        logger.info("Passando pelo AnnotatedType {}", at)
    }
}
```

A estrutura da extensão é somente essa, bem simples. Estamos utilizando o evento `ProcessAnnotatedType`, que observa cada `AnnotatedType`,

que representa cada objeto dentro do contexto CDI. Temos também o evento `ProcessBeanAttributes`, que observa cada `BeanAttributes`, que é uma outra visão, ainda mais simples, de cada `AnnotatedType`. Agora para que nossa extensão possa ser carregada, precisamos do arquivo `META-INF/services/javax.enterprise.inject.spi.Extension` como já foi dito. Dentro desse arquivo temos apenas o nome da classe da nossa extensão.

```
br.com.casadocodigo.livrocdi.extension.ControllerPorConvensaoExtension
```

Vista a estrutura, vamos implementar o conteúdo do método `configuraControllers`.

```
void configuraControllers(@Observes ProcessAnnotatedType pat) {

    AnnotatedType at = pat.getAnnotatedType();
    Class classe = at.getJavaClass();

    if (classe.getPackage().getName().endsWith("controllers")
        || classe.getSimpleName().endsWith("Controller")) {

        pat.setAnnotatedType(new AnnotatedTypeControllerWrapper(at));
        logger.info("Controlador encontrado: {}", classe);
    }
}
```

Acabamos de implementar a lógica da nossa convenção: pacote `controllers` ou classe com nome terminado em `Controller`. O trabalho de adicionar o estereótipo `@Controller` vai ser feito dentro da classe `AnnotatedTypeControllerWrapper`, mas antes disso, vamos analisar onde ela está sendo usada.

A grande facilidade do mecanismo de extensão da CDI é que cada evento observado, como o `ProcessAnnotatedType` nos devolve um objeto relacionado. Nesse caso estamos executando um código nosso a cada `AnnotatedType` descoberto pelo contexto. Esse objeto pode ser recuperado pelo método `getAnnotatedType`, mas pode ser alterado pelo método `setAnnotatedType`. Para todos os outros eventos que podemos observar para construir nossas extensões o funcionamento é muito parecido. Recuperamos o objeto recém analisado através do método `get`, e podemos usar o método `set` para trocar esse objeto por outro com as características que

desejamos. Outro método que geralmente está disponível nos objetos `ProcessXYZ` é o `veto()`, que pode ser utilizado para remover esse objeto do contexto, como se ele estivesse anotado com `@Vetoed`.

O código da classe `AnnotatedTypeControllerWrapper` tem um funcionamento muito simples, como podemos ver a seguir.

```
class AnnotatedTypeControllerWrapper implements AnnotatedType {

    private AnnotatedType wrapped;

    AnnotatedTypeControllerWrapper(AnnotatedType original) {
        this.wrapped = original;
    }

    @Override
    public Set<Annotation> getAnnotations() {
        Set<Annotation> annotations =
            new HashSet<>(wrapped.getAnnotations());
        annotations.add(new AnnotationLiteral<Controller>());
        return annotations;
    }

    @Override
    public Class getJavaClass() {
        return wrapped.getJavaClass();
    }

    ...
}
```

Existem vários outros métodos que precisam ser implementados, mas todos seguem a mesma filosofia do `getJavaClass()`, apenas repassando a invocação para o objeto `wrapped`. Agora analisemos o método que nos interessa: `getAnnotations()`. Vejamos como ele é simples, consistindo apenas em adicionar a anotação `Controller` na lista de anotações da nossa classe. Assim, não fazemos as demais anotações pararem de funcionar, apenas adicionamos um estereótipo. Se a classe já tiver um escopo ou um nome, os dados do nosso estereótipo será sobrescrito por essas anotações específicas. Bem simples e funcional.

9.5 EXTENSÃO NÃO PORTÁVEL: EVENTOS ASSÍNCRONOS FORA DO JAVA EE

Apesar de muito simples de usar, o mecanismo de extensão não nos deixa manipular todos os objetos que gostaríamos. Nesse exemplo vamos manipular os métodos observadores, queremos trocar o observador normal por um que execute em uma `Thread` separada para permitir a execução assíncrona.

Na CDI podemos acompanhar os métodos observadores presentes na aplicação da seguinte forma.

```
public class EventoAssincronoExtension implements Extension {

    private Logger logger = LoggerFactory
        .getLogger(EventoAssincronoExtension.class);

    public void analisaObservers(@Observes ProcessObserverMethod pom) {
        ObserverMethod observerMethod = pom.getObserverMethod();

        jogger.info("Método {} é um observer" + observerMethod);
    }
}
```

Como vimos ao construir a extensão `ControllerPorConvensaoExtension`, o comum é termos um método `get` e um `set` em cada `ProcessXYZ` dentro da SPI da CDI. Porém a interface `ProcessObserverMethod` não possui um método `setObserverMethod`, nos obrigando a utilizar classes específicas do Weld, deixando de ter uma extensão portátil. Em versões futuras podemos ter essa limitação removida; por hora veremos como sair dessa situação, e de situações que podem ocorrer no nosso dia a dia e que a SPI não nos ofereça uma saída.

A primeira coisa que faremos é observar se a classe que implementa a interface `ProcessObserverMethod` nos oferece esse método. A classe em questão é a `org.jboss.weld.bootstrap.events.ProcessObserverMethodImpl`, mas ela também não oferece um `setObserverMethod`. A saída será utilizar um evento diferente, o `AfterBeanDiscovery`.

Esse evento é lançado depois que todos os beans, e também os observadores foram descobertos e não há problema em suas definições. Então depois de tudo pronto, vamos tentar mudar algo que não tivemos abertura para mudar no momento apropriado. A estrutura da nossa extensão ficará então da seguinte forma.

```
public class EventoAssincronoExtension implements Extension {

    private Logger logger = LoggerFactory
        .getLogger(EventoAssincronoExtension.class);

    void afterBeanDiscovery(@Observes AfterBeanDiscovery abd,
        BeanManager bm) {
        logger.info("Todos os beans já foram lidos");
    }
}
```

Apenas trocamos o evento que iremos observar. Nesse trecho, adicionamos também o `javax.enterprise.inject.spi.BeanManager`. Enquanto em um observador de eventos comum podemos receber qualquer objeto como dependência do método, bastando para isso colocar mais parâmetros além no anotado pelo `@Observes`; quando implementamos uma extensão, o único objeto injetável é o `BeanManager`. Isso porque o método será chamado antes do contexto estar terminado, assim não seria possível encontrar as dependências. Já o `BeanManager` é um objeto único e central na CDI, logo ele existe desde o início da inicialização da aplicação.

Nesse momento, podemos utilizar os métodos do `BeanManager` para acessar suas estruturas internas e modificá-las. Podemos fazer isso com os beans e diversos outros elementos do contexto, mas infelizmente não temos uma maneira de alterar a definição de observadores de eventos através dessa interface.

A saída será utilizar a implementação de `BeanManager`, e então recuperar a lista de observadores. Faremos isso da seguinte maneira.

```
void afterBeanDiscovery(@Observes AfterBeanDiscovery abd,
    BeanManager bm) {

    BeanManagerImpl manager = ((BeanManagerProxy) bm).delegate();

    List<ObserverMethod<?>> observers = manager.getObservers();

}
```

Através da classe `org.jboss.weld.manager.BeanManagerImpl` conseguimos acessar o método `getObservers()`, e aí sim poderemos prosseguir na implementação da nossa extensão. Porém antes de avançar, vamos entender melhor como essa extensão irá funcionar.

Novamente os observadores assíncronos

Já vimos sobre observadores assíncronos de eventos na seção 7.5. Naquela ocasião, vimos que para usar a funcionalidade precisávamos que o bean fosse um EJB, e que o método estivesse anotado com `@Asynchronous`. Também já vimos que a CDI não oferece uma forma nativa de termos observadores assíncronos, pois o intuito não é competir com outras especificações. Logo, se quisermos executar CDI fora do servidor de aplicações, sem EJBs, precisaremos de uma extensão que nos ofereça isso. Podemos buscar uma pronta nas referências apresentadas neste capítulo, ou então desenvolver a nossa.

O intuito da nossa implementação não é competir com qualquer extensão desenvolvida por organizações como Apache ou JBoss, e sim mostrar como podemos implementar coisas muito úteis quando não encontrarmos uma alternativa pronta.

No nosso exemplo, teremos apenas uma anotação Java simples, `@Assincrono`, que será colocada sobre os métodos observadores que desejamos que sejam executados de forma assíncrona. A diferença para a anotação `@Asynchronous`, é que a nossa pode ser colocada em um bean comum, e não apenas em EJBs.

Como a SPI da CDI não prevê esse tipo de manipulação de observadores, não há uma forma simples de manipularmos esses métodos. Precisaremos de uma tarefa um pouco mais trabalhosa - mas nem tanto - e que envolve mais algumas classes específicas do Weld. Como estamos usando algo específico da implementação, podemos ter alterações dessas classes no futuro, por esse motivo o melhor é seria usarmos só classes da especificação. Mas já vimos que no nosso exemplo não será possível.

Vejamos então como trocar o `ObserverMethod` padrão, por um *wrapper* que faça-o executar em uma `Thread` separada.

```
void afterBeanDiscovery(@Observes AfterBeanDiscovery abd,
                        BeanManager bm) {

    BeanManagerImpl manager = ((BeanManagerProxy) bm).delegate();

    List<ObserverMethod<?>> observers = manager.getObservers();

    for (ObserverMethod<?> observerMethod : observers) {
        ObserverMethodImpl observerMethodImpl =
            (ObserverMethodImpl) observerMethod;

        MethodInjectionPoint methodInjectionPoint =
            observerMethodImpl.getMethod();
```

```
    AnnotatedMethod annotatedMethod =
        methodInjectionPoint.getAnnotated();
    Method method = annotatedMethod.getJavaMember();

    if (method.isAnnotationPresent(Assincrono.class)) {

        //aqui chegamos ao ponto que desejávamos!
    }
}
```

Precisamos usar diversas classes para poder chegar até o `java.lang.reflect.Method` que representa o método observador, para então verificar se ele está anotado com `@Assincrono`, através de `if (method.isAnnotationPresent (Assincrono.class))`.

Depois de tudo isso, conseguimos identificar o `ObserverMethod` que desejamos trocar dentro do contexto CDI. Para isso criaremos uma classe *wrapper* assim como fizemos na extensão `ControllerPorConvensaoExtension`. E então realizaremos, dentro do `if` anterior, a troca de um objeto pelo outro.

```
if (method.isAnnotationPresent(Assincrono.class)) {

    ObserverMethod original = observerMethodImpl;
    ObserverMethod observerMethodWrapper =
        new ObserverMethodWrapper(original);

    observers.remove(original);
    abd.addObserverMethod(observerMethodWrapper);
}
```

Após instanciarmos nossa classe `ObserverMethodWrapper`, removemos a original da lista de `observers` e adicionamos o observador novo em seu lugar. Adicionamos o novo usando o método `AfterBeanDiscovery.addObserverMethod(ObserverMethod)`. Podemos perceber que o método de adicionar o observador existe, o que não existe é uma forma padrão de trocar o objeto antigo pelo novo. Optamos pelo método porque ele faz parte da SPI, então preferimos utilizá-lo, senão adicionaríamos ele direto na lista `observers`, que é uma referência de dentro do contexto CDI.

Falta agora o código da classe `ObserverMethodWrapper`, que trocará a chamada original por uma que lança uma nova `Thread`.

```
import java.util.concurrent.ExecutorService;
...

public class ObserverMethodWrapper<T> implements ObserverMethod<T> {

    private Logger logger = LoggerFactory
        .getLogger(ObserverMethodWrapper.class);

    private ObserverMethod delegate;

    private ExecutorService executor;

    public ObserverMethodWrapper(ObserverMethod delegate) {
        this.delegate = delegate;
    }

    ...
}
```

Ainda não vimos todos os métodos da nossa classe, mas a estrutura básica já nos fornece alguma informação. Perceba a utilização da classe `ExecutorService`, que serve para dispararmos execuções de `Runnable`s em novas `Threads`. Vamos assumir que conseguiremos uma instância dessa interface e prosseguir. Depois voltaremos e analisaremos um pouco sobre como conseguimos essa instância.

A grande maioria dos métodos são apenas delegações de chamadas de método para o objeto `delegate`.

```
@Override
public Class getBeanClass() {
    return delegate.getBeanClass();
}

@Override
public Type getObservedType() {
    return delegate.getObservedType();
}

//delegação dos demais métodos para o delegate

@Override
```

```
public void notify(Object event) {  
    getExecutor().execute(new ExecuteEventRunnable(delegate, event));  
}
```

O método mais interessante nesse exemplo é o `notify(Object)`. Esse é o método que é responsável por disparar a notificação para o observador. O que fazemos aqui é, em vez de somente delegar a chamada para o objeto `delegate`, o que acabaria não surtindo nenhuma alteração no comportamento padrão do observador, chamar o `delegate.notify(event)` dentro de uma nova `Thread`.

O método `getExecutor()` nos devolve uma instância de `ExecutorService`, e sabendo que esse objeto consegue disparar a execução de um `Runnable` em uma nova `Thread`, agora precisamos ver a implementação da classe `ExecuteEventRunnable`.

```
class ExecuteEventRunnable implements Runnable {  
  
    private ObserverMethod delegate;  
    private Object event;  
  
    public ExecuteEventRunnable(ObserverMethod delegate, Object event) {  
        this.delegate = delegate;  
        this.event = event;  
    }  
  
    @Override  
    public void run() {  
        delegate.notify(event);  
    }  
}
```

Simplemente chamamos o observador original, representado pelo objeto `delegate`, dentro de um `Runnable`. Sabendo um pouco do funcionamento de `Threads` dentro do Java, sabemos que basta fazer algo como `new Thread(Runnable).start()` para executar o `Runnable` em uma nova `Thread`, realmente é bem simples, mas como estamos rodando nosso código dentro de um servidor, essa não é uma abordagem recomendada.

Tradicionalmente, a utilização de múltiplas threads dentro do Java EE é bem diferente do seu uso no JavaSE. Enquanto em uma aplicação desktop podemos usar `new Thread` como citado, no Java EE a forma de fazer isso era disparar uma mensagem JMS para que um *Message-Driven Bean* a processasse de forma assíncrona.

No Java EE 6 isso foi simplificado com a adição da anotação `@Asynchronous` que já vimos. Isso facilitou muito o uso, mas como estamos construindo uma alternativa justamente a esse mecanismo, precisamos de outra alternativa.

Felizmente, o Java EE 7 adicionou o uma nova especificação, a *"Concurrency Utilities for Java EE"* (JSR-236) [?]. Essa especificação simplesmente nos oferece uma API Java EE compatível com a API JavaSE disponível desde o Java 5. Enquanto no SE essas funcionalidades estão no pacote `java.util.concurrent`, no EE 7, essa nova especificação definiu o pacote `javax.enterprise.concurrent`. De forma parecida, no JSE temos principalmente as interfaces `ExecutorService` e `ThreadFactory`, já no JEE temos as respectivas interfaces `ManagedExecutorService` e `ManagedThreadFactory`.

Após esse breve entendimento do significado dessas interfaces, vejamos a implementação do método `getExecutor()`, que é onde buscaremos as instâncias corretas.

```
private ExecutorService getExecutor() {
    if (executor != null) {
        return executor;
    }

    try {
        Context ctx = new InitialContext();
        executor = (ManagedExecutorService)
            ctx.lookup("java:comp/env/concurrent/ManagedExecutorService");
    } catch (Exception e) {
        logger.error("Erro ao buscar o "
            + " ManagedExecutorService (JSR 236)", e);

        logger.warn("Usando implementação simples " +
            " de newCachedThreadPool()");

        executor = Executors.newCachedThreadPool();
    }

    return executor;
}
```

Sempre buscaremos no contexto da aplicação uma instância gerenciada de `ExecutorService`, pois ela certamente estará integrada ao *pool* de `Threads` que

o servidor já possui, e não correremos o risco de interferir negativamente no ciclo de vida da aplicação e do servidor, evitando qualquer vazamento de memória ou coisas do gênero. Para conseguirmos recuperar esse objeto, adicionamos no `web.xml` da aplicação o seguinte conteúdo.

```
<resource-env-ref>
  <resource-env-ref-name>
    concurrent/ManagedExecutorService
  </resource-env-ref-name>
  <resource-env-ref-type>
    javax.enterprise.concurrent.ManagedExecutorService
  </resource-env-ref-type>
</resource-env-ref>
```

Precisamos dessa configuração porque não existe um caminho padrão definido para recuperarmos o `ManagedExecutorService`. O mesmo vale para o `ManagedThreadFactory`.

Para terminar nossa análise, precisamos ter bastante atenção no conteúdo do `catch` no método `getExecutor()`. Caso não consigamos recuperar o `ManagedExecutorService`, utilizaremos o método `Executors.newCachedThreadPool()`, porém registraremos um aviso no log para que isso seja averiguado. Pode ser que a configuração do nome JNDI não esteja correta, mas pode ser que não esteja disponível uma implementação de `ExecutorService`. Apesar dessa implementação ser parte do Java EE, é bem possível que em breve conseguiremos utilizá-la dentro de servlet containers.

Espero que neste capítulo tenha ficado claro a vantagem de termos extensões portáteis, mas que também tenha servidor para indicar um caminho quando essa não for uma possibilidade, no caso das extensões não portáteis. Como acabamos de falar de servlet containers, passaremos agora para o último capítulo, onde veremos a utilização de CDI em ambientes não Java EE, dentre eles o servlet container.

CAPÍTULO 10

Executando CDI em diferentes ambientes

A CDI é uma especificação pensada para o Java EE, mas como já adiantamos em diversos momentos, é possível executá-lo fora desse ambiente. Nesse capítulo veremos como executar CDI em *servlet containers*, a partir de uma classe *main* e também em testes de integração.

Nosso foco nesses exemplos vai deixar de ser a funcionalidade da CDI, pois já vimos bastante sobre isso durante todo o livro. Agora iremos focar na configuração necessária para executar nesse ambientes. Essas são configurações específicas para o Weld, implementação de referência da CDI, pois a especificação não aborda esses ambientes.

10.1 EXECUTANDO CDI EM SERVLET CONTAINERS

Ao executar uma aplicação dentro de um servlet container, temos que lembrar das diversas funcionalidades que não estarão disponíveis, como controle automático de transações, eventos assíncronos, mensageria, dentre outras. Porém isso muitas vezes não é um problema, uma vez que já sabemos como implementar, com pouco esforço, essas funcionalidades, seja via extensão ou simplesmente adicionando um jar com as funcionalidades implementadas.

Nessa seção iremos utilizar como referência o tomcat, mas na documentação do Weld encontramos como proceder a configuração para executar no jetty também. Como esses servidores não tem a obrigação de prover os serviços da CDI, teremos que, pela primeira vez, fazer o download do Weld. Obviamente é possível utilizar gerenciadores de dependência como maven ou ivy, mas por termos que adicionar basicamente dois jars, procederemos da forma tradicional mesmo.

O download do Weld pode ser feito no seguinte endereço: <http://www.seamframework.org/Weld/Downloads>. Após descompactar o arquivo baixado, precisamos de apenas dois jars que podem ser encontrados dentro da pasta `artifacts`, são eles `cdi-api.jar` e `weld-servlet.jar`. Basta criarmos um projeto web na IDE da nossa preferência e colocar esses jars dentro da pasta `WEB-INF/lib`. Precisamos lembrar também, assim como já fazíamos nos projetos Java EE, é de colocar o arquivo `beans.xml`, mesmo vazio, dentro da pasta `WEB-INF`.

A partir desse momento já estamos com o *setup* básico feito, temos então que partir para as configurações propriamente ditas. Não se preocupe com a possibilidade de esquecer algum detalhe, pois você pode verificar se seu projeto possui todas as configurações comparando-o com o projeto disponível no github: <https://github.com/gscordeiro/cdi-tomcat>.

Um objeto muito importante para a CDI é o `BeanManager`. Esse objeto é sempre necessário e alcançável via JNDI em ambiente Java EE. Porém como estamos usando um servlet container, vamos precisar fazer uma configuração para adicionar esse objeto dentro do contexto JNDI da aplicação. No tomcat, essa configuração é feita dentro do arquivo `META-INF/context.xml`, cujo conteúdo vemos a seguir.

```
<Context>
  <Resource name="BeanManager" auth="Container"
    type="javax.enterprise.inject.spi.BeanManager"
    factory="org.jboss.weld.resources.ManagerObjectFactory" />
</Context>
```

Para completar a configuração do `BeanManager`, precisamos adicionar no `web.xml` uma entrada que cria uma referência com o nome "BeanManager" para o tipo `javax.enterprise.inject.spi.BeanManager`.

```
<resource-env-ref>
  <resource-env-ref-name>BeanManager</resource-env-ref-name>
  <resource-env-ref-type>
    javax.enterprise.inject.spi.BeanManager
  </resource-env-ref-type>
</resource-env-ref>
```

Ao atribuímos esse nome ao `BeanManager`, ele fica disponível no contexto JNDI com o nome "java:comp/env/BeanManager". Nós não precisamos usar diretamente esse nome, mas o módulo *servlet* do Weld vai precisar.

Agora que o JNDI está pronto, precisamos colocar mais uma configuração no `web.xml`, que é o *Listener* do Weld que irá subir o contexto CDI e também interagir com as requisições.

```
<listener>
  <listener-class>
    org.jboss.weld.environment.servlet.Listener
  </listener-class>
</listener>
```

Agora sim terminamos, nossa aplicação está pronta para utilizar as funcionalidades da CDI. Apenas para testarmos, vamos criar uma *servlet* que faz a injeção de uma `CalculadoraSalarios`. Apesar de continuarmos no mesmo exemplo da aplicação Java EE, aqui iremos ter implementações bem mais simples, pois o intuito é apenas testar o *setup* do ambiente.

```
@WebServlet("/calculadoraSalario")
public class CalculadoraSalarioServlet extends HttpServlet {

    @Inject
    private CalculadoraSalarios calculadoraSalarios;

    @Override
    protected void doPost(HttpServletRequest req,
                          HttpServletResponse resp)
        throws ServletException, IOException {
```

```

Funcionario funcionario = new Funcionario(
    req.getParameter("funcionario"),
    Double.parseDouble(req.getParameter("salario")));

double salario =
    calculadoraSalarios.calculaSalario(funcionario);

System.out.printf("Salário do(a) %s é %.2f",
    funcionario.getNome(),
    salario);
}
}

```

O uso do `System.out` é apenas para simplificar o exemplo, no projeto do github temos um simples formulário de entrada de dados e uma página que exibe o resultado.

A implementação da `CalculadoraSalarios` aqui nesse projeto é bem mais simples, apenas adiciona 10% (dez por cento) ao salário base do funcionário.

```

public class CalculadoraSalarios {

    @Inject
    private Event<Funcionario> calculoSalarioEvent;

    public double calculaSalario(Funcionario funcionario){
        calculoSalarioEvent.fire(funcionario);
        return funcionario.getSalarioBase() * 1.1;
    }
}

```

Temos também o lançamento de um evento simples, que serve para indicar que o salário do funcionário foi calculado. Como já vimos esse conteúdo, aqui não utilizamos qualquer qualificador; queremos apenas ver que os eventos também funcionam no servlet container como no application server.

Por fim temos o observador do evento, nos mesmos moldes que já vimos antes.

```

public class ObservadorCalculadora {

    public void escutaCalculoSalario(@Observes Funcionario funcionario){
        System.out.printf("O salário do(a) %s "
            + " acabou de ser calculado\n", funcionario.getNome());
    }
}

```

```

    }
}

```

Essa seção foi relativamente curta pois já vimos todo o conteúdo, e também porque a configuração em si também não é extensa. Então se em nossa organização utilizamos servlet containers em vez de servidores de aplicação Java EE, não precisamos abrir mão das grandes vantagens da CDI. Tomcat e Jetty diferem bem pouco em sua configuração. Como o arquivo `web.xml` faz parte da especificação de servlets, a única mudança será no arquivo `META-INF/context.xml` do tomcat, que no jetty se chama `WEB-INF/jetty-env.xml` e tem o seguinte conteúdo.

```

<!DOCTYPE Configure PUBLIC "-//Mort Bay Consulting//DTD Configure//EN"
    "http://www.eclipse.org/jetty/configure.dtd">

<Configure id="webAppCtx"
    class="org.eclipse.jetty.webapp.WebAppContext">
    <New id="BeanManager" class="org.eclipse.jetty.plus.jndi.Resource">
        <Arg> <Ref id="webAppCtx"/> </Arg>
        <Arg>BeanManager</Arg>
        <Arg>
            <New class="javax.naming.Reference">
                <Arg>javax.enterprise.inject.spi.BeanManager</Arg>
                <Arg>org.jboss.weld.resources.ManagerObjectFactory</Arg>
                <Arg/>
            </New>
        </Arg>
    </New>
</Configure>

```

10.2 EXECUTANDO CDI A PARTIR DE UMA CLASSE MAIN

Assim como precisamos fazer na seção anterior, precisamos baixar o Weld na página do projeto: <http://www.seamframework.org/Weld/Downloads>. Já tendo os jars baixados, precisamos criar um novo projeto Java simples e adicionar os jars `cdi-api.jar` e `weld-se.jar` no seu classpath. Você também pode verificar seu projeto comparando com o disponível no github: <https://github.com/gscordeiro/cdi-desktop>.

Não será necessário criar configurações para usar Weld dentro da nossa *main*, basta fazermos a inicialização do contexto manualmente.

```
public class CalculadoraMain {

    public static void main(String[] args) {

        Weld weld = new Weld();
        WeldContainer container = weld.initialize();

        CalculadoraSalarios calculadoraSalarios = container.instance()
            .select(CalculadoraSalarios.class).get();

        Funcionario funcionario = new Funcionario("Fulano", 5000.0);

        double salario =
            calculadoraSalarios.calculaSalario(funcionario);

        System.out.printf("Salário calculado do %s é %.2f\n",
            funcionario.getNome(), salario);

        weld.shutdown();

    }
}
```

Uma limitação do Weld em aplicação JSE está nos escopos disponíveis: aplicação, dependente e singleton, já que não existe requisição e sessão fora de um servidor.

Apesar de na *main* não termos usado `@Inject`, utilizamos uma forma de lookup programático. Mas a injeção continua funcionando, basta vermos o código da *CalculadoraSalarios*.

```
public class CalculadoraSalarios {

    @Inject
    private Event<Funcionario> calculoSalarioEvent;

    public double calculaSalario(Funcionario funcionario){
        calculoSalarioEvent.fire(funcionario);
        return funcionario.getSalarioBase() * 1.1;
    }
}
```

É exatamente o mesmo código que utilizamos na aplicação *cdi-tomcat*.

Podemos também fugir da subida manual do contexto do Weld, bastando para isso usar uma classe `main` disponível dentro do `weld-se.jar`: `org.jboss.weld.environment.se.StartMain`. Ao subir o contexto, o Weld lança o evento `ContainerInitialized`, e nossa “falsa `main`” observa esse evento.

```
import org.jboss.weld.environment.se.bindings.Parameters;
import org.jboss.weld.environment.se.events.ContainerInitialized;

@Singleton
public class CalculadoraFalsaMain {

    @Inject
    private CalculadoraSalarios calculadoraSalarios;

    public void printHello(@Observes ContainerInitialized event,
                           @Parameters List<String> parameters) {

        Funcionario funcionario = new Funcionario("Fulano", 6000.0);

        double salario =
            calculadoraSalarios.calculaSalario(funcionario);

        System.out.printf("Salário calculado do %s é %.2f\n",
                           funcionario.getNome(), salario);

    }
}
```

Agora sim, nossa “falsa `main`” consegue trabalhar com injeção da `CalculadoraSalarios`. Além disso, usamos o qualificador `@Parameters` para solicitar ao container a injeção dos parâmetros da `main` original; o famoso `String[] args`.

Assim como no projeto `cdi-tomcat`, os eventos continuam funcionando, assim como interceptadores, decoradores e as outras funcionalidades da CDI.

10.3 É COMO TESTAR UMA APLICAÇÃO CDI?

Já no início do livro mostramos como a CDI nos deixa livres para construir projetos de classes testáveis. Isso para testes de unidade resolve a questão, mas precisamos também lidar com testes de integração e de sistema. Nosso foco nessa seção estará

nos testes de integração, pois é onde a CDI atua, fazendo a injeção das dependências, decorando e interceptando as invocações, além é claro de eventos e outros tópicos que já vimos.

Atualmente, a ferramenta de automação de testes de integração mais utilizada no Java EE é o Arquillian (<http://arquillian.org/>) . Ele nos permite, de dentro do nosso caso de teste, subir um contexto de CDI para que consigamos executar nosso teste junto com injeção de dependência, controle de transações, e tudo mais que se fizer necessário para nossos testes reproduzirem situações reais da aplicação.

Um pouco de Arquillian

Para construir nossos testes de integração com Arquillian, precisamos entender como eles funcionarão com a ferramenta. Para isso vamos fazer um paralelo com o processo manual de construção da aplicação, deploy e testes.

Quando estamos desenvolvendo nossa aplicação, colocamos todas classes e arquivos de configuração necessários nos locais corretos, em seguida fazemos o empacotamento da aplicação. Esse empacotamento pode ser um arquivo jar, war ou ear. Muitas vezes a IDE esconde esse processo de nós, fazendo o empacotamento e o deploy e a inicialização do servidor de forma automática quando executamos a ação “Executar”. Porém temos que ter em mente que todos esses passos precisam acontecer para nossa aplicação possa ser testada.

Para fazer o empacotamento da aplicação a ser testada, utilizamos o Shrink-Wrap (<http://www.jboss.org/shrinkwrap>) , que é uma ferramenta de criação programática de pacotes java. Veremos na prática como utilizar essa ferramenta, que nos oferece uma API fluente muito simples de usar.

Após o empacotamento, precisamos fazer o deploy desse pacote. Aqui entra um outro aspecto muito interessante do Arquillian, que são os *container adapters*. Temos basicamente três tipos de *adapters*: Embedded, Managed e Remote. Além disso temos adapters para diferentes containers de cada um desses tipos.

Por exemplo, analisando GlassFish e JBoss, vemos que temos os três tipos de adaptador de GlassFish 3.1 (mas que executa também o GlassFish 4); já para o JBoss 7.1 temos apenas os tipos Managed e Remote.

A diferenciação entre esses tipos é simples:

- **Embedded:** Nesse tipo de adaptador, o Arquillian sobe uma instância do container, seja ele GlassFish, JBoss, outro servidor ou até mesmo um container só com o Weld em execução, tudo isso de dentro do próprio teste. Aqui não

é necessário baixar o servidor e instalar (descompactar) na nossa máquina, podemos deixar o Arquillian fazer todo o trabalho. Por isso esse tipo de adaptador é chamado “embarcado”, pois ele sobe de dentro do próprio teste.

- **Managed:** Nesse tipo de adaptador, o Arquillian usa uma instalação do container existente externamente, mas é ele quem gerencia a subida e a descida do container.
- **Remote:** Já neste último tipo de adaptador, o Arquillian faz uso de uma instância já existente e em execução de container. Ou seja, o servidor já deve estar rodando para que o deploy do nosso pacote seja realizado.

Em todos esses tipos de adaptadores, o Arquillian faz o deploy, executa os testes, e depois faz o undeploy do pacote. Nos dois primeiros tipos, no `before class` dos nossos testes o Arquillian também sobe o container e no `after class` baixa esse container de forma automática. Já o tipo **Remote** é o mais comum quando temos um servidor voltado para os testes, ou quando temos muitos testes e gostaríamos de subir o servidor uma única vez. Nesse caso podemos fazer um script que suba e desça o servidor antes e depois dos testes. No nosso exemplo faremos uso do adaptador GlassFish Embedded.

Configurando as dependências

A montagem do ambiente Arquillian não é muito simples, envolve muitas dependências, assim, é difícil configurá-lo sem o auxílio de ferramentas como o maven ou o ivy. Por ser a forma mais comum, e consequentemente mais fácil de encontrar exemplos, utilizaremos o maven. Como o arquivo `pom.xml` costuma ser extenso, veremos apenas partes dele aqui, mas seu conteúdo completo está disponível no github: <https://github.com/gscordeiro/cdi-arquillian>.

Criamos um novo projeto com suporte ao maven para colocar nossos testes. Essa separação tem somente o intuito de deixar o projeto inicial mais leve, senão enquanto estivéssemos adicionando pequenas funcionalidades e executando o projeto no servidor, teríamos o maven rodando toda a bateria de testes, subindo e descendo o contexto do glassfish toda hora, deixando o processo todo mais lento. Obviamente isso é configurável, mas além de simplificar o processo como um todo, o projeto apartado serve como uma referência rápida para você que já tem seus projetos CDI e deseja adicionar somente os testes. Ou seja, a separação é apenas didática, em um projeto real os testes ficam junto com seu projeto.

Para mantermos o foco, vamos ignorar, por hora, configurações comuns do projeto no maven, como dependências de Java EE e JUnit.

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.jboss.arquillian</groupId>
      <artifactId>arquillian-bom</artifactId>
      <version>1.1.1.Final</version>
      <scope>import</scope>
      <type>pom</type>
    </dependency>
  </dependencies>
</dependencyManagement>

<dependencies>
  <dependency>
    <groupId>org.glassfish.main.extras</groupId>
    <artifactId>glassfish-embedded-all</artifactId>
    <version>4.0</version>
    <scope>provided</scope>
  </dependency>
  <dependency>
    <groupId>org.jboss.arquillian.container</groupId>
    <artifactId>arquillian-glassfish-embedded-3.1</artifactId>
    <version>1.0.0.CR4</version>
    <scope>test</scope>
  </dependency>
  <dependency>
    <groupId>org.jboss.arquillian.junit</groupId>
    <artifactId>arquillian-junit-container</artifactId>
    <version>1.1.0.Final</version>
    <scope>test</scope>
  </dependency>
</dependencies>
```

Além das dependências básicas, como Java EE e JUnit, também deixaremos para logo mais a configuração de persistência do arquillian. Agora vamos ver como realizar um teste simples usando o arquillian.

```
@RunWith(Arquillian.class)
public class CalculadoraSalariosTest {
```

```
@Deployment
public static JavaArchive createDeployment() {

    return ShrinkWrap.create(JavaArchive.class)
        .addClass(Funcionario.class)
        .addClass(CalculadoraSalarios.class)
        .addClass(CalculadoraSalariosDezPorcento.class)
        .addAsManifestResource(EmptyAsset.INSTANCE, "beans.xml");
}
...
}
```

Aqui podemos ver a estrutura básica da classe de testes. Usamos a anotação `@org.junit.runner.RunWith` do JUnit e passamos ao Arquillian o gerenciamento do ciclo de vida do teste, permitindo à ferramenta subir e descer o container quando necessário.

O empacotamento do que será testado é feito através de um método estático que devolve o pacote montado programaticamente. No nosso exemplo é um `org.jboss.shrinkwrap.api.spec.JavaArchive`. Por mais que nossa aplicação seja web, como vamos testar apenas a calculadora, não havendo qualquer tela, podemos fazer nesse momento um jar em vez de um war.

Para montar nosso pacote vamos adicionando as classes, e também os recursos que necessitarmos. Perceba como criamos um arquivo chamado `"beans.xml"`, em branco (`EmptyAsset.INSTANCE`), e dentro do “manifest resource” (`addAsManifestResource`). Esse último método é muito interessante pois sabe exatamente o local onde o arquivo deve ser colocado. Se estivéssemos criando um `WarArchive` teríamos ainda um método `addAsWebInfResource` para colocar o recurso também no local correto. Muito mais simples do que ficarmos nos preocupando com as pastas e arquivos `"build.xml"` do Ant.

Mais um fator importante a notarmos é que criamos sempre pacotes somente com o necessário para executar o teste atual. Então se nossa aplicação é grande, cheia de EJBs, páginas web, e diversas bibliotecas, a aplicação demora mais para subir. Mas criando um pacote como esse do teste que tem apenas três classes, certamente o tempo de subir e descer o pacote será incomparavelmente menor.

Vamos agora retornar à classe `CalculadoraSalariosTest`, onde vamos testar nossa calculadora.

```

@Deployment
public static JavaArchive createDeployment() { ... }

@Inject
private CalculadoraSalarios calculadoraSalarios;

@Test
public void deveAumentarSalarioBaseEmDezPorcento(){

    Funcionario funcionario =
        new Funcionario("Fulano", 4000.0, Escolaridade.MEDIO);

    double salario = calculadoraSalarios.calculaSalario(funcionario);

    assertEquals(4400.0, salario, 0.001);
}

```

Essa classe `Funcionario` e a classe `CalculadoraSalarios` são versões simplificadas das classes que já vimos. Isso novamente apenas para deixar o projeto `cdi-arquillian` mais simples. A implementação `CalculadoraSalariosDezPorcento`, por exemplo, apenas adiciona 10% (dez por cento) ao valor do salário base do `Funcionario`.

Testando ambiguidade

Podemos fazer mais um teste interessante para verificarmos a ambiguidade de dependências.

```

@RunWith(Arquillian.class)
public class AmbiguidadeTest {

    @Deployment
    public static JavaArchive createDeployment() {
        return ShrinkWrap.create(JavaArchive.class)
            .addClass(CalculadoraSalarios.class)
            .addClass(CalculadoraSalariosDezPorcento.class)
            .addClass(CalculadoraSalariosVintePorcento.class)
            .addAsManifestResource(EmptyAsset.INSTANCE, "beans.xml");
    }
}

```

Nesse exemplo, montamos um pacote com a interface `CalculadoraSalarios`

e suas duas implementações. Agora vamos injetar uma instância dessa interface utilizando `javax.enterprise.inject.Instance`, e em seguida vamos verificar a ambiguidade e depois resolvê-la.

```
@Deployment
public static JavaArchive createDeployment() { ... }

@Inject
private Instance<CalculadoraSalarios> calculadoraSalarios;

@Test
public void deveApresentarAmbiguidade(){
    assertTrue(calculadoraSalarios.isAmbiguous());
}

@Test
public void naoDeveApresentarAmbiguidade(){
    Instance<? extends CalculadoraSalarios> calculadoraEspecificas =
        calculadoraSalarios.select(
            CalculadoraSalariosVintePorcento.class);

    assertFalse(calculadoraEspecificas.isAmbiguous());
}
```

Até aqui, poderíamos ter utilizado apenas o adaptador do próprio Weld no lugar de termos utilizado o do GlassFish. Porém agora iremos explorar exemplos envolvendo persistência, e nesse caso precisamos de uma solução mais completa.

10.4 DESENVOLVENDO TESTES DE INTEGRAÇÃO COM PERSISTÊNCIA

Como resultado da seção anterior, já temos o ambiente de testes funcionando com o arquillian. Agora vamos expandir nosso ambiente para termos suporte à extensão de persistência. Precisamos para isso adicionar o seguinte trecho de código na seção `<dependencies>` do `pom.xml`.

```
<dependencies>
    <dependency>
        <groupId>org.jboss.arquillian.extension</groupId>
        <artifactId>arquillian-persistence-impl</artifactId>
```

```

    <version>1.0.0.Alpha5</version>
    <scope>test</scope>
  </dependency>
</dependencies>

```

Agora conseguimos facilmente controlar transação dentro dos testes, e conseguimos também utilizar o DBUnit (<http://dbunit.sourceforge.net>). É muita possibilidade junta, mas vejamos como a utilização é simples. Para iniciar, faremos um teste bem simples de inserção de funcionários no banco de dados. Novamente veremos o código em partes para irmos analisando cada parte.

```

@RunWith(Arquillian.class)
public class FuncionarioPersistenceTest {

    @Deployment
    public static JavaArchive createDeployment() {

        return ShrinkWrap.create(JavaArchive.class)
            .addClass(Funcionario.class)
            .addAsManifestResource(
                new File("src/test/resources/META-INF/persistence-teste.xml"),
                    "persistence.xml")
            .addAsManifestResource(EmptyAsset.INSTANCE, "beans.xml");
    }
    ...
}

```

Agora nosso pacote tem somente uma classe, a entidade `Funcionario`, e temos mais dois recursos: o primeiro é o arquivo `persistence.xml` e o outro é o `beans.xml`. Já vimos que este último será um arquivo vazio, mas o `persistence` é bastante interessante, pois criamos um arquivo chamado `persistence-teste.xml`, com uma configuração de banco de dados específica para o ambiente de testes, e ao adicioná-lo no pacote mudamos seu nome para `persistence.xml`, que é o esperado pelo container.

Nesse arquivo, configuramos o mecanismo de persistência para construir o banco quando o contexto subir e para apagar o banco quando os testes terminarem. Dessa forma, a cada teste o banco será reconstruído, então não precisamos nos preocupar em apagar os dados manualmente entre uma execução e outra do teste.

Nesse momento temos um pacote CDI que possui a funcionalidade de persistência ativada. E agora podemos novamente encarar o próprio teste como se fosse

um bean gerenciado pela CDI, injetando inclusive o `EntityManager`.

```
@Deployment
public static JavaArchive createDeployment() { ... }

@PersistenceContext(unitName = "livroCdiPU")
EntityManager em;

@Test
@Transactional
public void deveInserir2Funcionarios() {

    TypedQuery<Funcionario> query =
        em.createQuery("select f from Funcionario f", Funcionario.class);

    assertEquals(0, query.getResultList().size());

    Funcionario funcionario1 =
        new Funcionario("Xico", 2000.0, Escolaridade.FUNDAMENTAL);
    Funcionario funcionario2 =
        new Funcionario("Maria", 4000.0, Escolaridade.MEDIO);

    assertEquals(null, funcionario1.getId());
    assertEquals(null, funcionario2.getId());

    em.persist(funcionario1);
    em.persist(funcionario2);

    assertEquals(Integer.valueOf(1), funcionario1.getId());
    assertEquals(Integer.valueOf(2), funcionario2.getId());

    assertEquals(2, query.getResultList().size());
}
```

Esse teste nos mostra como proceder no teste de persistência. Com a extensão de persistência do arquillian, temos acesso à anotação `@org.jboss.arquillian.persistence.Transactional`. Anotando o teste com ela, este fica transacional, nos permitindo salvar objetos no banco como se estivéssemos dentro de um EJB. Uma outra prática é anotar o método de testes, ou toda a classe, com `@Transactional(TransactionMode.ROLLBACK)`, para que os dados não sejam persistidos, gerando sempre um *rollback* na transação atual.

Assim os diversos métodos não influenciam um no outro.

O uso do DBUnit de forma muito simples

Outra ferramenta que nos auxilia a criar testes envolvendo a camada de persistência, é o DBUnit. Como já adicionamos a extensão de persistência do arquillian, o DBUnit já está disponível.

O DBUnit trabalha basicamente montando uma massa de dados antes de cada teste, e ao final verifica se os dados ficaram como o esperado. Ele trabalha no nível de banco de dados, e não nos objetos, o que nos permite inclusive testar possíveis procedures existentes no projeto.

```
@RunWith(Arquillian.class)
public class FuncionarioDBUnitTest {

    @Deployment
    public static JavaArchive createDeployment() {...}

    @PersistenceContext(unitName = "livroCdiPU")
    EntityManager em;

    @Test
    @UsingDataSet("funcionarios.xls")
    @ShouldMatchDataSet("funcionarios-expected.xls")
    public void deveDar10PorcentoAumentoAosFuncionariosNivelSuperior() {

        List<Funcionario> funcionarios =
            em.createQuery("select f from Funcionario f "
+ "where f.escolaridade = :escolaridade", Funcionario.class)
                .setParameter("escolaridade", Escolaridade.SUPERIOR)
                .getResultList();

        for(Funcionario funcionario : funcionarios){
            double salarioAtual = funcionario.getSalarioBase();
            funcionario.setSalarioBase(salarioAtual * 1.1);
        }

    }
}
```

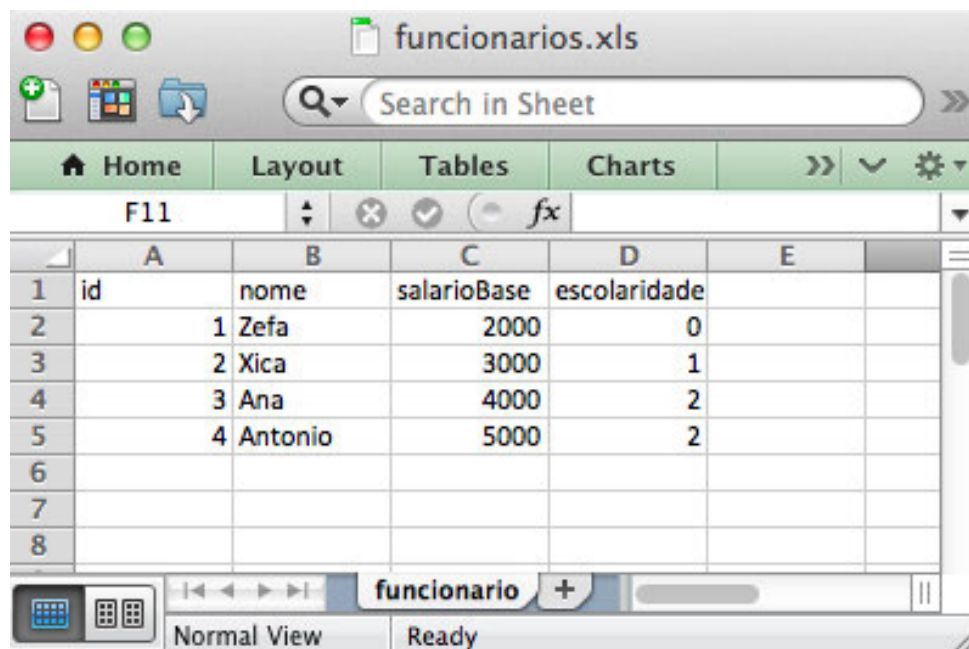
Nesse outro teste, o método `createDeployment()` é exatamente igual ao do exemplo anterior. O que podemos notar de diferença é a presença das duas anotações que faz uso referência ao DBUnit: `@UsingDataSet` e `@ShouldMatchDataSet`.

No DBUnit, os *datasets* podem ser vários tipos de arquivos, como XML, YML, e no nosso exemplo utilizamos planilhas XLS, pois elas se parecem bem com as tabelas que desejamos representar. Para utilizar uma planilha, basta que cada planilha dentro do arquivo tenha o mesmo nome de cada tabela, e cada coluna da planilha tem que ter o mesmo nome das colunas do banco.

Como essa ferramenta trabalha no nível de tabelas, não temos que analisar o modelo OO, e sim o relacional. Precisamos reproduzir nas planilhas o modelo relacional, não o OO, isso é importante. Para ficar bem claro, vamos analisar a estrutura dessa classe `Funcionario`, que é mais simples do que a tínhamos visto até antes de iniciarmos esses testes.

```
public enum Escolaridade {  
    FUNDAMENTAL, MEDIO, SUPERIOR  
}  
  
@Entity  
public class Funcionario implements Serializable {  
  
    @Id @GeneratedValue  
    private Integer id;  
    private String nome;  
    private double salarioBase;  
    private Escolaridade escolaridade;  
  
    //getters e setters  
}
```

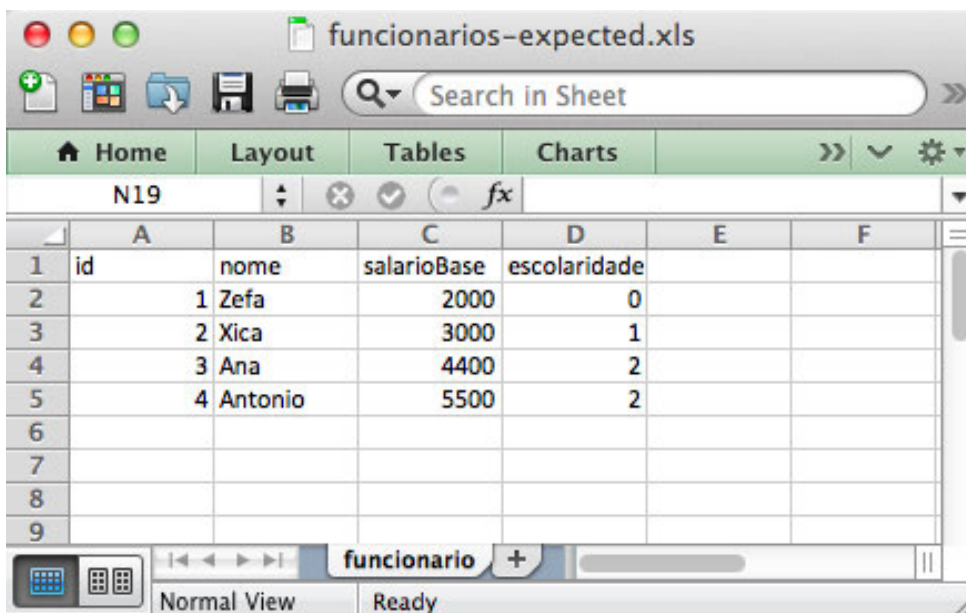
Agora vejamos as duas planilhas, a que possui os dados iniciais, e a que possui os dados esperados.



	A	B	C	D	E
1	id	nome	salarioBase	escolaridade	
2	1	Zefa	2000	0	
3	2	Xica	3000	1	
4	3	Ana	4000	2	
5	4	Antonio	5000	2	
6					
7					
8					

Figura 10.1: Dados iniciais do teste

O próprio DBUnit apaga todos os dados da tabela e deixa exatamente com os dados esperados.



	A	B	C	D	E	F
1	id	nome	salarioBase	escolaridade		
2		1 Zefa	2000	0		
3		2 Xica	3000	1		
4		3 Ana	4400	2		
5		4 Antonio	5500	2		
6						
7						
8						
9						

Figura 10.2: Dados esperados ao final do teste

Após o teste é feita a verificação se os dados ficaram de acordo com o esperado. Se não estiver o teste falha, não é necessário *assert*.

Como nosso teste é feito com base na escolaridade do funcionário, e esse dado é um Enum, na tabela, por padrão, esse dado vira um inteiro. Como podemos ver, temos dois funcionários com nível superior: Ana, com salário base de R\$ 4.000,00; e Antônio com salário de R\$ 5.000,00. Na planilha com os dados esperados alteramos apenas os valores dos salários desses funcionários, acrescentando 10% (dez por cento) como pede o teste.

A partir desse ponto, basta implementar os demais testes de negócio da nossa aplicação, pois os mecanismos já estão funcionando. Espero que esse último capítulo tenha servido também como um pequeno tutorial de Arquillian, pois além de testar as funcionalidades da CDI, essa ferramenta auxilia muito nos testes com Java EE de modo geral.

Assim como este último capítulo, espero que este livro tenha servido para auxiliar no seu desenvolvimento profissional como um todo. A CDI é uma especificação muito interessante, e espero que seus estudos tanto em CDI como em Java EE continuem evoluindo. O Java EE tem mudado bastante nas últimas versões, se tornando

cada vez mais simples de utilizar, mesmo assim acredito que este livro tenha contribuído para deixar seu entendimento ainda mais simples.