Curso

Spring Framework

Phillip Calçado “Shoes”
Fragmental TI LTDA.
http://www.fragmental.com.br

Sobre a Fragmental TI LTDA

A Fragmental Tecnologia da Informação LTDA é uma empresa prestadora de serviços nas áreas de consultoria e treinamento em tecnologia. Sua especialidade é o uso e divulgação de práticas consagradas e modernas para auxiliar o desenvolvimento de software.

Esta apostila é parte integrante dos treinamentos da Fragmental.

Sobre o Autor

Phillip Calçado "Shoes" é arquiteto de software com 10 anos de experiência. É membro do conselho nacional da *International Association of Software Architects* (IASA), Coordenador do RioJUG (Grupo de Usuários Java do Rio de Janeiro), do GUJ, articulista da revista Mundo Java e palestrante nos principais eventos nacionais de desenvolvimento.

Licença desta Obra

Atribuição-Uso Não-Comercial-Compatilhamento pela mesma licença 2.5 Brasil

Você pode:

- copiar, distribuir, exibir e executar a obra
- criar obras derivadas

Sob as seguintes condições:

Atribuição. Você deve dar crédito ao autor original, da forma especificada pelo autor ou licenciante.

Uso Não-Comercial. Você não pode utilizar esta obra com finalidades comerciais.

Compartilhamento pela mesma Licença. Se você alterar, transformar, ou criar outra obra com base nesta, você somente poderá distribuir a obra resultante sob uma licença idêntica a esta.

- Para cada novo uso ou distribuição, você deve deixar claro para outros os termos da licença desta obra.
- Qualquer uma destas condições podem ser renunciadas, desde que Você obtenha permissão do autor.

Qualquer direito de uso legítimo (ou "fair use") concedido por lei, ou qualquer outro direito protegido pela legislação local, não são em hipótese alguma afetados pelo disposto acima.

Este é um sumário para leigos da Licença Jurídica (na íntegra).

1. Introdução

Java é a plataforma tecnológica de maior sucesso na história da computação. De uma simples linguagem para a criação de pequenos aplicativos para *set-top boxes* (como as utilizada pela TV digital interativa), se tornou uma plataforma completa, abrangendo desde programas para pequenos dispositivos até as maiores aplicações coorporativas do mundo.

A maioria das aplicações desenvolvidas em Java hoje são focadas no mercado coorporativo. Estas aplicações geralmente possuem características básicas em comum: interface web, persistência em banco de dados relacional, comunicação com outros sistemas, necessidade de autenticação e autorização de usuários, etc.

Para padronizar o desenvolvimento e os produtos disponíveis, o consórcio que rege a plataforma Java, *Java Community Process* (JCP), criou a macro-especificação *Java 2 Enterprise Edition* (J2EE). Esta reúne diversas especificações menores que ditam sobre como as características de aplicações coorporativas são implementadas em Java, por exemplo como uma aplicação web em Java deve ser ou como acessar o banco de dados nesta linguagem.

Como a plataforma Java é utilizada universalmente, todo tipo de aplicação é desenvolvida nela. J2EE foi criada visando suportar aplicações de grande porte e exerce muito bem seu papel neste cenário, mas acaba trazendo muita complexidade para o desenvolvimento de aplicações não tão grandes. Ocorre que a grande maioria das aplicações existentes hoje não é de grande porte e não requer todos estes recursos, tendo ainda assim que pagar o preço da complexidade de uma plataforma "Tamanho Único”**.**

Para amenizar e reduzir este problema, surgiram alternativas vindas da comunidade de desenvolvedores Java. Soluções simples criadas por pequenos grupos se espalharam e tornaram-se um padrão *de facto.* Uma destas soluções alternativas é o Spring Framework, criado por Rod Johnson para amenizar os problemas que este encontrava ao utilizar a tecnologia de *Enterprise Java Beans* (EJB), parte principal do J2EE.

A prova da força das soluções alternativas veio com a versão 5.0 de J2EE, agora chamado apenas *Java Enterprise Edition* (Java EE). Esta traz recursos que claramente foram inspirados tanto no Spring como em outras soluções alternativas. É a padronização de técnicas e tecnologias que a comunidade de desenvolvedores consagrou nos últimos anos.

Esta apostila é parte integrante de um curso sobre o Spring Framework, em sua versão 1.2. Seu conteúdo visa explicar e treinar o leitor no uso desta ferramenta.

Apesar do lançamento da solução padronizada trazida pelo Java EE 5.0, espera-se que a adoção desta plataforma pelas empresas demore um bom tempo. Além da necessidade de treinar todos os recursos humanos nesta nova plataforma, existe a necessidade de atualizar os servidores de aplicação e máquinas virtuais (Java EE 5.0 só funciona em *Java Virtual Machines* 5.0). Utilizando o Spring Framework e outras soluções alternativas, podemos ter os benefícios da nova versão do Java EE em servidores de aplicação e JVMs antigas, a partir da versão 1.3 de Java.

2. Introduzindo Dependency Injection

Até mesmo o mais trivial dos sistemas em Java possui classes e componentes que se relacionam entre si. A forma como implementamos este relacionamento é muito importante para a flexibilidade e qualidade geral do sistema.

Imagine uma classe `GerenciadorUsuarios`, que para funcionar precisa ter acesso a uma classe `UsuarioDao`, que conecta ao banco de dados:

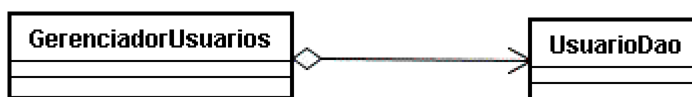


Figura 2.1 – Dependência entre Classes do Exemplo

```
public class GerenciadorUsuarios {
      private UsuarioDao usuarioDao=null;

      //... continuação da classe
}
```

Existem diversas maneiras de se implementar esta relação, vamos analisar algumas

Instanciação Direta

A forma mais natural de se obter uma referência para o `UsuarioDao` de dentro de `GerenciadorUsuarios` seria instanciando-a diretamente, como no exemplo abaixo:

```
public class GerenciadorUsuarios {

      private UsuarioDao usuarioDao= new UsuarioDao();

      //... continuação da classe
}
```

Instanciação Direta é quando uma classe cria o objeto de que precisa. Em Java isso geralmente é feito com o operador **`new`**.

Acontece que utilizando esta forma para componentes de aplicações corporativas é criada uma dependência muito grande entre os objetos. A classe `GerenciadorUsuarios` vai precisar saber como iniciar a `UsuarioDao` para que esta possa operar. Vamos supor que a `UsuarioDao` necessite de alguma configuração para ser utilizada, esta configuração terá que ser feita no construtor da classe `GerenciadorUsuarios`:

```
public class GerenciadorUsuarios {

      private UsuarioDao usuarioDao= null;

      public GerenciadorUsuario(){
            //configura o DAO
```

```
            usuarioDao = new UsuarioDao();
            usuarioDao.setServidorBancoDeDados("mysql.fragmental.com.br:3306");
            usuarioDao.setUsuario("pcalcado");
            usuarioDao.setSenha("jap123");
            usuarioDao.setSchema("curso-spring");
            usuarioDao.setDriverJdbc("com.mysql.jdbc.Driver");
            //... e mais configurações
      }

      //... continuação da classe
}
```

Se qualquer uma destas configurações mudar (mudar o nome do servidor, a porta de conexão, o driver utilizado, senha do banco de dados, etc.) esta classe terá que ser alterada. Estamos violando um princípio básico de engenharia de software gerando um alto *acoplamento* entre estas classes.

Uso de Registry

Existem diversas soluções possíveis para este problema, até a versão 1.4, J2EE utiliza a solução de registrar os recursos num diretório de componentes chamado JNDI (*Java Naming and Directory Interface*). Esta solução é baseada no Padrão Arquitetural *Registry*. Ao utilizar o Registry, temos um lugar comum onde as classes podem obter referências a outras classes.

Registry é um Padrão Arquitetural onde componentes são obtidos através da consulta a um lugar comum. Em J2EE geralmente utiliza-se a árvore JNDI.

No exemplo, poderíamos fazer `GerenciadorUsuarios` obter uma referência ao `UsuarioDao` por meio do Registry JNDI. Primeiro, a classe `UsuarioDao` precisa ser registrada (o que é chamado de *bind*), geralmente isto é feito pelo container utilizando arquivos de configuração (*deployment descriptors*) em um código parecido com este:

```
//configura o DAO
UsuarioDao usuarioDao = new UsuarioDao();
usuarioDao.setServidorBancoDeDados("mysql.fragmental.com.br:3306");
usuarioDao.setUsuario("pcalcado");
usuarioDao.setSenha("jap123");
usuarioDao.setSchema("curso-spring");
usuarioDao.setDriverJdbc("com.mysql.jdbc.Driver");
//... e mais configurações

//Registra o DAO
Context ctx = new InitialContext();
ctx.bind("usuarioDao", usuarioDao);
```

Logo depois, a nossa classe pode obter uma referência do Registry, processo chamado de *lookup*:

```
public class GerenciadorUsuarios {

	private UsuarioDao usuarioDao= null;

	public GerenciadorUsuarios(){
		try{
		Context ctx = new InitialContext();
		usuarioDao = (UsuarioDao) ctx.lookup("usuarioDao");
		}catch(NamingException ex){
			//tratar exceção
		}
	}

	//... continuação da classe
}
```

Desta forma a inicialização do componente fica fora da classe cliente deste, e geralmente pode ser feita através de arquivos de configuração.

Esta solução por vezes é implementada utilizando um objeto que segue o Design Pattern *Singleton*, fazendo uso de métodos estáticos. Esta opção é altamente problemática porque o uso de Singletons e métodos estáticos tende a criar um sistema altamente procedural e com problemas diversos de concorrência.

Apesar de ser uma solução muito boa para diversos casos, existem problemas nesta abordagem que tornam o desenvolvimento em Java EE mais complexo. O primeiro deles é a dependência gerada entre a classe e o ambiente.

Não é possível utilizar a classe `GerenciadorUsuarios` fora de um ambiente onde JNDI esteja disponível, e isto geralmente indica um Servidor de Aplicações. Ao utilizar JNDI desta maneira nós perdemos a portabilidade da classe e seu reuso, já que não poderíamos a reaproveitar em outros sistemas diferentes do original (por exemplo um sistema feito em Java sem J2EE).

Outro problema muito importante está na modelagem do sistema. Uma boa modelagem de negócios vai sempre representar o mundo real em objetos o mais próximo possível da realidade e no mundo das regras de negócio provavelmente não existe o conceito de *bind* ou *lookup*. Estes conceitos são *complexidade acidental*, limitações impostas pela plataforma.

Finalmente, este tipo de alternativa consegue reunir boa parte dos problemas encontrados com o uso de variáveis globais em toda a história da Ciência da Computação.

Dependency Injection

A pedra fundamental do Spring Framework é o conceito de *Dependency Injection*.

> Dependency Injection *(Injeção de Dependências - DI) é uma forma de* Inversion Of Control *(Inversão de Controle – IoC)* na qual um componente não instancia suas dependências mas o ambiente (container) automaticamente as fornece.

Este é o ***Princípio de Hollywood****: "Não me ligue, eu te ligo"*, ou seja: o objeto não chama o container, o container chama o objeto. No exemplo, a classe GerenciadorUsuarios poderia ser implementada exatamente desta forma:

```
public class GerenciadorUsuarios {

      private UsuarioDao usuarioDao= null;

      public void setUsuarioDao(UsuarioDao usuarioDao) {
            this.usuarioDao = usuarioDao;
      }

      //... continuação da classe
}
```

E o container se encarregaria de instanciar o `UsuarioDao` e chamar o método `setUsuarioDao()`, como no trecho de código abaixo:

```
// configura o DAO
UsuarioDao usuarioDao = new UsuarioDao();
usuarioDao.setServidorBancoDeDados("mysql.fragmental.com.br:3306");
usuarioDao.setUsuario("pcalcado");
usuarioDao.setSenha("jap123");
usuarioDao.setSchema("curso-spring");
usuarioDao.setDriverJdbc("com.mysql.jdbc.Driver");
// ... e mais configurações
//coloca o usuarioDao dentro do GerenciadorUsuarios
gerenciadorUsuarios.setUsuarioDao(usuarioDao);
```

No caso do Spring, é preciso declarar as dependências em um arquivo de configuração. Toda a parte de instanciar o objeto e injetar esta instância é feito de maneira automática.

Ao utilizar DI, a classe não depende mais do ambiente. Se utilizado fora de um container, basta que alguém o forneça suas dependências utilizando um trecho de código como o descrito acima. Também não precisa saber anda sobre a infraestrutura, se usa JNDI ou outro recurso, já que não precisa invoca-la.

Usos Indicados

Cada uma das formas de implementação de dependências mostradas acima possui cenários onde sua aplicação é ótima.

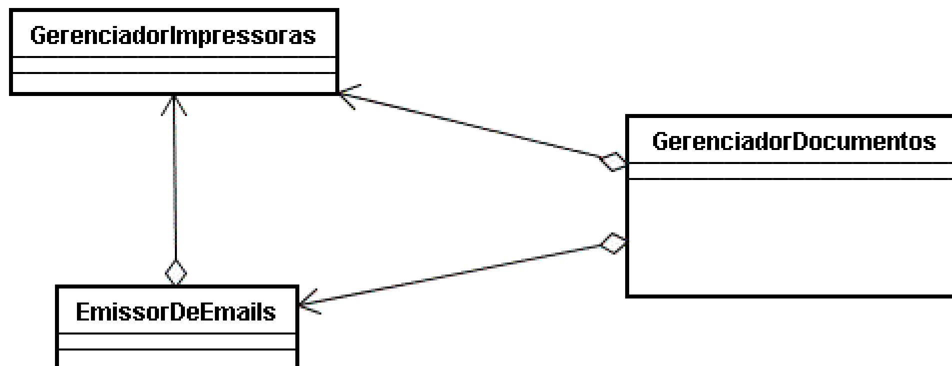
Para componentes ou *Services*, o uso de instanciação direta é altamente desaconselhável. Componentes geralmente são classes de granularidade grossa que fornecem serviços a outros componentes e como visto o uso de instanciação direta faria com que um componente criasse um acoplamento muito grande com outro.

O uso de Registry para componentes é útil para lidar com o legado de aplicações J2EE. Como veremos, o Spring também pode trabalhar com JNDI o que facilita muito esta integração. Alguns componentes de infraestrutura também podem tirar proveito desta técnica. De uma maneira geral, componentes de aplicação, que não fazem parte da infraestrutura, têm maior benefícios quando implementadas utilizando Dependency Injection.

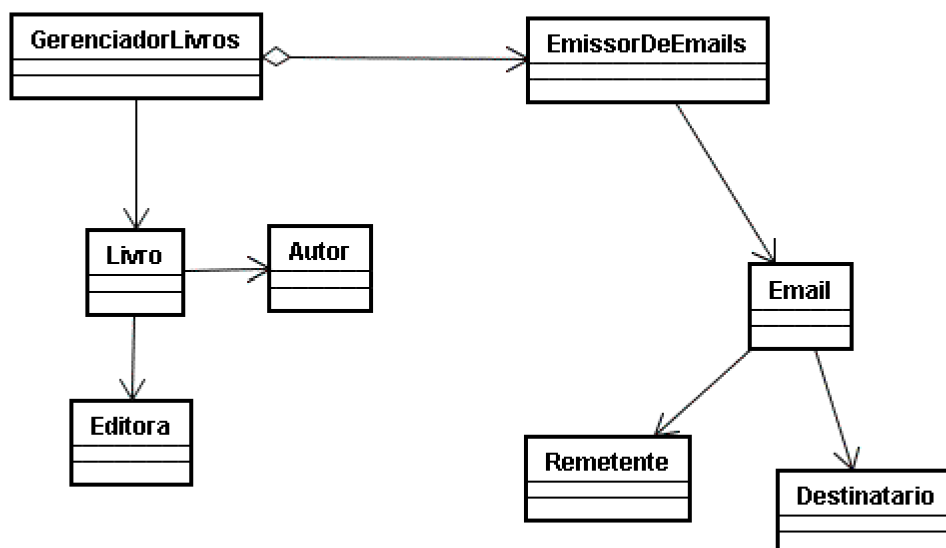
Da mesma forma, o uso de Registry ou Dependency Injection para implementar a relação entre duas classes simples é um erro na maioria das vezes. Classes simples, principalmente se persistentes, devem utilizar instanciação direta.

Exercícios

1 – Dado o diagrama de classes abaixo, implemente a dependência utilizando as três formas descritas neste capítulo.



2 – Dado o diagrama abaixo, identifique a melhor estratégia de implementação de cada uma das relações mostradas.



3. Dependency Injection no Spring Framework

Um bom entendimento do conceito de DI é fundamental para o bom uso do Spring. Toda a plataforma desta ferramenta é baseada neste conceito.

Implementar DI no Spring é muito simples.Vamos a um exemplo ilustrativo:

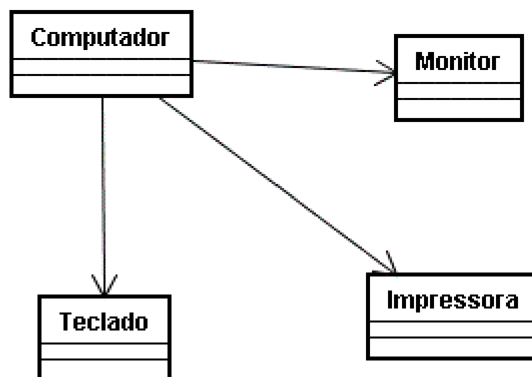


Figura 3.1 – Domínio do Exemplo

Temos uma classe `Computador` que possui referências para diversas outras classes que compõem o sistema. Vamos utilizar o Spring para automaticamente instanciar estas dependências do `Computador` e os injetar no objeto. Primeiro, definimos as classes envolvidas:

```
public class Computador {
	private Impressora impressora = null;

	private Monitor monitor = null;

	private Teclado teclado = null;

	public void setImpressora(Impressora impressora) {
		this.impressora = impressora;
	}

	public void setMonitor(Monitor monitor) {
		this.monitor = monitor;
	}

	public void setTeclado(Teclado teclado) {
		this.teclado = teclado;
	}

	public void ligar(){
		monitor.exibeMensagem("Digite texto para impressão");
```

```
            String texto = teclado.ler();
            impressora.imprimir(texto);
            monitor.exibeMensagem("Texto Impresso!");
      }
}
```

```
public class Impressora {

      public void imprimir(String texto) {
            // nossa impressora virtual apenasescreve na tela
            System.out.println("[IMPRESSORA] " + texto);

      }

}
```

```
public class Monitor {

      /**
       * Exibe uma mensagem de texto.
       *
       * @param mensagem
       */
      public void exibeMensagem(String mensagem) {
            // No nosso exemplo, vamos exibir a mensagem na saída principal
            System.out.println("[MONITOR] "+mensagem);
      }

}
```

```
public class Teclado {

      /**
       * Lê um tesxto do usuário.
       *
       * @return
       */
      public String ler() {
```

```
            String texto = null;

            //imprime um 'prompt'
            System.out.print("[TECLADO]>");

            try {
                  texto = new BufferedReader(new InputStreamReader(System.in))
                              .readLine();
            } catch (IOException e) {
                  System.out.println("Erro lendo teclado!");
                  e.printStackTrace();
            }
            return texto;
      }

}
```

Note que não existe código de inicialização das outras classes na classe `Computador`. Se instanciarmos esta classe manualmente e executarmos o método `ligar()` teremos uma `NullPointerException` já que os componentes que ele necessita não foram fornecidos.

Para testarmos nossa implementação, podemos criar uma classe que inicializa e supre todas as dependências, como a descrita abaixo:

```
public class IniciaComputador {
      public static void main(String[] args) {
            Computador computador = new Computador();
            computador.setImpressora(new Impressora());
            computador.setTeclado(new Teclado());
            computador.setMonitor(new Monitor());
            computador.ligar();
      }
}
```

Esta classe é útil para testes unitários (um conceito que veremos adiante), mas para o uso na aplicação real vamos deixar este trabalho para o Spring.

Para fazer o Spring entender a dependência entre nossas classes, precisamos criar um arquivo de configuração. Este geralmente recebe o nome de `applicationContext.xml`. Vejamos o arquivo que define nosso exemplo:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
      <bean id="computadorBean"
      class="br.com.fragmental.cursos.spring.apostila.capitulo3.Computador">
```

```
            <property name="impressora" ref="impressoraBean"/>
            <property name="teclado" ref="tecladoBean"/>
            <property name="monitor" ref="monitorBean"/>
      </bean>

      <bean id="impressoraBean"
      class="br.com.fragmental.cursos.spring.apostila.capitulo3.Impressora"/>

      <bean id="tecladoBean"
      class="br.com.fragmental.cursos.spring.apostila.capitulo3.Teclado"/>

      <bean id="monitorBean"
      class="br.com.fragmental.cursos.spring.apostila.capitulo3.Monitor"/>
</beans>
```

E para testar nosso exemplo utilizando Spring podemos utilizar a classe abaixo:

```
public class IniciaUsandoSpring {

      public static void main(String[] args) {
            ApplicationContext applicationContext = new
ClassPathXmlApplicationContext(
      "classpath:br/com/fragmental/cursos/spring/apostila/capitulo3/applicationC
ontext-Capitulo3.xml");
            Computador computador = (Computador)
applicationContext.getBean("computadorBean");
            computador.ligar();
      }
}
```

Para entender o que o Spring requer como configuração, vamos analisar o arquivo de configuração.

O Arquivo ApplicationContext.xml

O `applicationContext.xml` é onde são declarados os *beans* do Spring. O framework chama de beans todas as classes que gerencia. As classes precisam ser declaradas utilizando o elemento `<bean>`, seguindo o formato:

```
<bean id="identificador do bean" class="FQN da classe que implementa o bean" >
      <property name="nome do atributo" ref="id do bean que satisfaz a
dependência" />
</bean>
```

***FQN** significa Fully-Qualified Name, ou nome completo. O FQN de uma classe é o nome da classe com o seu pacote completo. Por exemplo, o FQN da classe* `String` *é* `java.lang.String`*, o FQN da classe* `List` *é* `java.util.List`.

No exemplo, declaramos os beans `computadorBean`, `impressoraBean`, `monitorBean` e `tecladoBean`. O bean `computadorBean` possui como atributos referências para os outros beans e estas referências são

declaradas utilizando elementos `property` dentro da declaração do bean.

Ao utilizar o elemento `property`, o Spring irá tentar utilizar um método `set`**`NomeDoAtributo`**`()` para preencher o bean com a dependência configurada.

Muitas vezes, para não quebrar a invariante (veja *[ContratosNulos]*) de um objeto é necessário que suas dependências sejam supridas durante a inicialização deste. Para estes casos, o Spring oferece a possibilidade de injetar as dependências utilizando o construtor do objeto ao invés de seus métodos mutadores (*setters*).

Imagine que tenhamos um objeto `Carro` que para funcionar sempre precisa de um objeto `Motor`. Como a presença de um `Motor` faz parte da invariante do `Carro`, ele é recebido no construtor:

```
public class Carro {

	private Motor motor = null;

	public Carro(Motor motor) {
		super();
		this.motor = motor;
	}

	public void rodar() {
		motor.iniciar();
	}
}
```

```
public class Motor {
	public void iniciar() {
		System.out.println("Motor iniciado");
	}
}
```

Para definir a mensagem, utilizamos o arquivo de configuração abaixo:

```
<bean id="carro"
class="br.com.fragmental.cursos.spring.apostila.capitulo3.Carro">
	<constructor-arg ref="motor" />
</bean>
<bean id="motor"
class="br.com.fragmental.cursos.spring.apostila.capitulo3.Motor" />
```

Literais, Listas e Mapas

As dependências podem ser outros beans ou valores. Quando a dependência é de um bean, o atributo `ref` do elemento `property` deve ser utilizado para indicar o bean desejado, como no nosso exemplo.

Podemos também definir valores simples (texto ou numerais) diretamente no XML, como no exemplo abaixo:

```
public class AloMundo {

      private String mensagem=null;

      public void setMensagem(String mensagem) {
            this.mensagem = mensagem;
      }

      public void executar(){
            System.out.println("Mensagem: "+mensagem);
      }
}
```

Podemos definir o conteúdo do atributo `mensagem` diretamente no `applicationContext.xml`, como abaixo:

```
<bean id="aloMundo"
class="br.com.fragmental.cursos.spring.apostila.capitulo3.AloMundo">
      <property name="mensagem"><value>Testando</value></property>
</bean>
```

O arquivo de configuração também possui suporte para listas de elementos. Suponha, por exemplo, que temos uma classe como a descrita abaixo.

```
public class SalaDeAula {
      private List alunos = null;
      public void setAlunos(List alunos) {
            this.alunos = alunos;
      }
      public void chamada(){
            for (Iterator it = alunos.iterator(); it.hasNext();) {
                  Aluno aluno = (Aluno) it.next();
                  aluno.presente();
            }
      }

}
```

Que recebe uma lista de `Aluno`s como argumento em seu construtor. A classe `Aluno` é bem simples:

```
public class Aluno {
      private String nome = null;

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

```
	}

	public void presente() {
		System.out.println(nome + " presente!");
	}
}
```

Para criar e injetar a lista de alunos, podemos utilizar a sintaxe abaixo:

```
<bean id="sala"
class="br.com.fragmental.cursos.spring.apostila.capitulo3.SalaDeAula">
	<property name="alunos">
		<list>
			<ref bean="bernarda" />
			<ref bean="phillip" />
			<ref bean="valentina" />
		</list>
	</property>
</bean>

<bean id="bernarda"
class="br.com.fragmental.cursos.spring.apostila.capitulo3.Aluno">
	<property name="nome"><value>Bernarda</value></property>
</bean>

<bean id="phillip"
class="br.com.fragmental.cursos.spring.apostila.capitulo3.Aluno">
	<property name="nome"><value>Phillip</value></property>
</bean>

<bean id="valentina"
class="br.com.fragmental.cursos.spring.apostila.capitulo3.Aluno">
	<property name="nome"><value>Valentina</value></property>
</bean>
```

Simplesmente definimos um elemento `<list>` contendo os beans desejados na ordem desejada.

Caso o objeto receba um mapa (`java.util.Map`), também existe uma sintaxe facilitando sua configuração:

```
public class Dicionario {

	private Map definicoes = null;

	public void setDefinicoes(Map definicoes) {
```

```
            this.definicoes = definicoes;
      }

      public void imprimir() {
            for (Iterator it = definicoes.keySet().iterator(); it.hasNext();) {
                  String palavra = (String) it.next();
                  System.out.println(palavra + " SIGNIFICA "
                              + definicoes.get(palavra));
            }
      }
}
```

```
<bean id="dicionario"
class="br.com.fragmental.cursos.spring.apostila.capitulo3.Dicionario">
      <property name="definicoes">
            <map>
                  <entry key="mother"><value>Mãe</value></entry>
                  <entry key="father"><value>Pai</value></entry>
            </map>
      </property>
</bean>
```

Existe ainda uma terceira facilidade, muito útil quando utilizamos o Spring em conjunto com outros frameworks, que é a definição de *Properties* (`java.util.Properties`). Properties são o mecanismo padrão na plataforma Java para a criação de arquivos de configuração simples, o Spring permite que você crie esta configuração diretamente no `applicationContext.xml`, sem necessidade de arquivos externos. Por exemplo, dada esta classe configurada via um arquivo Properties:

```
public class ConexaoBancoDeDados {

      private String servidor = null;

      private int porta = 0;

      private String usuario = null;

      private String senha = null;

      public void setConfiguracao(Properties properties) {
            this.servidor = properties.getProperty("banco.servidor");
            this.porta =
Integer.parseInt(properties.getProperty("banco.porta"));
```

```
            this.usuario = properties.getProperty("banco.usuario");
            this.senha = properties.getProperty("banco.senha");


      }


      public void conectar() {
            System.out.println("[" + usuario + "(" + senha + ")" + "@" +
servidor
                        + ":" + porta + "]");
      }
}
```

Podemos utilizar a seguinte sintaxe para a configurarmos via `applicationContext.xml`:

```
<bean id="conexao"
class="br.com.fragmental.cursos.spring.apostila.capitulo3.ConexaoBancoDeDados">
      <property name="configuracao">
            <props>
                  <prop key="banco.servidor">fragmental.com.br</prop>
                  <prop key="banco.porta">3306</prop>
                  <prop key="banco.usuario">vmaia</prop>
                  <prop key="banco.senha">vc123al</prop>
            </props>
      </property>
</bean>
```

Em termos práticos, a diferença entre um mapa e um Properties é que o último só aceita Strings, enquanto o anterior pode contêm referências a outros objetos. O uso de arquivos Properties, como mencionado, é muito útil para a integração com outros frameworks que serão vistos no decorrer do curso.

Nota: *O uso do formato "`xyzBean`" para o id dos beans declarados (como em "`computadorBean`" ao invés de simplesmente "`computador`") não é uma recomendação. Este formato foi utilizado apenas para deixar o exemplo mais claro.*

Exercícios

1 – Crie arquivos de configuração do Spring para os cenários descritos nos exercícios do capítulo anterior.

2 – Escreva classes correspondentes ao arquivo de configuração descrito abaixo:

```
<bean id="A"
class="br.com.fragmental.cursos.spring.apostila.capitulo3.exercícios.ClasseA" />

<bean id="B"
class="br.com.fragmental.cursos.spring.apostila.capitulo3.exercícios.ClasseB">

      <property name="teste" ref="A" />

</bean>

<bean id="C"
class="br.com.fragmental.cursos.spring.apostila.capitulo3.exercícios.ClasseC">

<property name="lista"><list>

                  <ref bean="A" />

                  <ref bean="B" /></list>

</bean>
```

4. Introduzindo Aspect-Oriented Programming

A programação Orientada a Objetos (*Object-Oriented Programming* - OOP) é utilizada para dar mais modularidade ao código desenvolvido e para deixar o software mais parecido com o mundo real. Ao utilizar corretamente objetos, podemos mapear quase que diretamente os problemas do mundo real para software.

Por exemplo, ao modelarmos um sistema que cuida da aprovação de empréstimos, podemos trazer os conceitos envolvido (empréstimo, crédito, cliente...) diretamente para nosso software, como no diagrama abaixo.

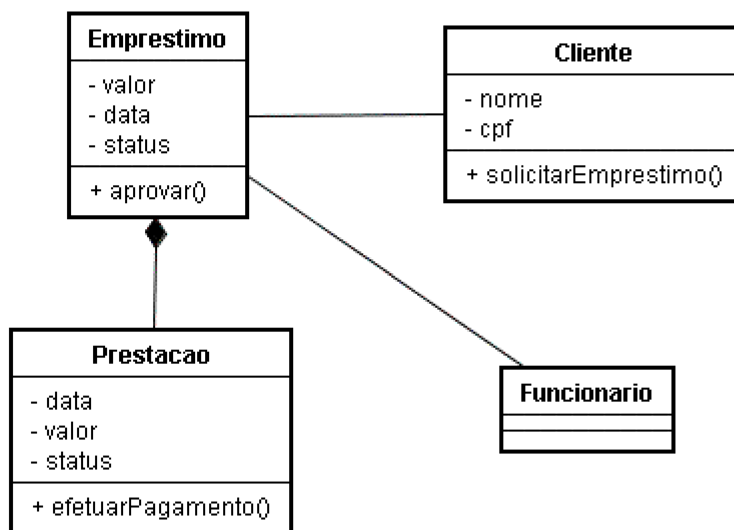


Figura 4.1 – Domínio de um Sistema

Entretanto, ainda existem e existirão por muito tempo limitações técnicas que fazem com que os conceitos de negócio que modelamos sofram "invasão" de conceitos técnicos, como autenticação de usuários, transações com bancos de dados e arquivos de *log*. Estes conceitos não fazem parte do domínio do problema, existem apenas para satisfazer a necessidade do software.

Normalmente, o que se faz com OOP é simplesmente misturar estes conceitos, criando código que mistura regras de negócio com limitações tecnológicas. Por exemplo, o método abaixo traz o código de negócios necessário para aprovar um empréstimo no exemplo acima.

```
public void aprovar() throws MaximoPrestacoesExcedidoException,
CreditoExcedidoException{
	//Checa se o empréstimo excede o limite do cliente
	if(!cliente.podeEmprestar(this.valor)){
		this.status=Status.NEGADO;
		throw new CreditoExcedidoException("Emprestimo muito alto");
	}

	//Checa se o numero de prestações é maior que o limite
	if(prestacoes.size()>Prestacao.MAXIMO_PRESTACOES){
		this.status=Status.NEGADO;
		throw new MaximoPrestacoesExcedidoException("O empréstimo deve ser
pago em até "+Prestacao.MAXIMO_PRESTACOES+" vezes!");
```

```
            }
        //se passou nas checagens:
        //empréstimo está aprovado
        this.status=Status.APROVADO;
}
```

Para que este código tire proveito dos recursos de aplicações modernas, por exemplo para que possa participar de transações no banco de dados e criar registros em um arquivo de log, ele precisa ser modificado para incluir código que lida exclusivamente com isso, como abaixo.

```
public void aprovar() throws MaximoPrestacoesExcedidoException,
                    CreditoExcedidoException, SQLException, NamingException {

        Connection conexao = criaTransacao();

        // Checa se o empréstimo excede o limite do cliente
        if (!cliente.podeEmprestar(this.valor)) {

                logger.log(Level.INFO, "Empréstimo [" + this.getCodigo()
                        + "] negado por exceder crédito em ["
                        + this.getValor().subtrair(cliente.getCredito()) + "]");

                this.status = Status.NEGADO;
                throw new CreditoExcedidoException("Emprestimo muito alto");
        }

                // Checa se o numero de prestações é maior que o limite
        if (prestacoes.size() > Prestacao.MAXIMO_PRESTACOES) {

                logger.log(Level.INFO, "Empréstimo [" + this.getCodigo()
                        + "] negado por exceder número máximo de prestações ["
                        + this.getPrestacoes().size() + "]");

                this.status = Status.NEGADO;
                throw new MaximoPrestacoesExcedidoException(
                        "O empréstimo deve ser pago em até "
                              + Prestacao.MAXIMO_PRESTACOES + " vezes!");
        }

        // se passou nas checagens:
        logger.log(Level.INFO, "Empréstimo [" + this.getCodigo()
```

```
                        + "] aprovado");
        // empréstimo está aprovado
        this.status = Status.APROVADO;

        conexao.commit();
}
```

A parte destacada em amarelo é o código adicionado exclusivamente para que o trecho de código tenha acesso aos recursos tecnológicos. Como visto, o código de negócios, que é o que realmente importa ao criar um software, fica soterrado no meio do código de infra-estrutura, que existe apenas para satisfazer necessidades técnicas.

Mesmo neste exemplo extremamente simples o código de infra-estrutura já representa boa parte do código da aplicação. Este problema faz com que mudanças em regras de negócio toquem o código de infra-estrutura, e vice-versa. Não é tão fácil separar mudanças no aspecto não-funcional das funcionais, tudo está no mesmo lugar!

Coesão é a métrica de quanto as linhas de código em um método (ou os métodos em uma classe) são relacionados entre si. Um método que faz conexão com banco de dados **e** processa regras de negócio tem baixa coesão.

Também existe o problema da repetição de código. Mesmo lidando apenas com código de negócios podemos acabar tendo que repetir uma mesma rotina várias vezes, linha por linha. No exemplo abaixo, vemos que ao ser criado um novo `Emprestimo` um `Funcionario` precisa ser notificado do fato.

```
public Emprestimo solicitarEmprestimo() {
        Emprestimo novo = new Emprestimo(this);
        notificadorFuncionarios.notificar(novo);
        return novo;
}
```

Acontece que por uma mudança nos requisitos agora o funcionário deve ser avisado em qualquer alteração no `Status` do `Emprestimo`, então ao invés de fazer simplesmente `this.status=` o código deve invocar este método:

```
private void setStatus(Status novoStatus) {
        this.status = novoStatus;

        notificadorFuncionarios.notificar(novoStatus);
}
```

Os dois métodos, em classes diferentes, possuem rotinas para notificar o `Funcionario`. Não se trata apenas de encapsular esta rotina num método, isso já foi feito (como o método `notificar()`). O fato deste método ser invocado sempre e da mesma maneira por diversas classes é que pode se tornar um problema quando a implementação mudar. O ideal seria conseguir “avisar” ao sistema de que toda vez que os métodos `solicitarEmprestimo()` e `setStatus()` (e qualquer outro que precise desta funcionalidade) forem chamados deve ser disparada uma notificação.

Conceitos Ortogonais e Aspectos

Como visto, temos uma mistura de código que lida com vários interesses distintos ao escrever código. Estes interesses são chamados de *conceitos ortogonais*.

Conceitos Ortogonais (*Orthogonal Concerns*) são os diversos conceitos com que um código de aplicação tem que lidar, como regras de negócio, segurança e conexões com bancos de dados.

A idéia por trás de *Aspect-Oriented Programming* é tratar estes conceitos em lugares separados (entenda-se "classes separadas" se estamos falando de Java) enquanto estamos desenvolvendo e uni-las em tempo de execução. Estes conceitos que se integram ao código de negócios são chamados de *Aspectos*.

Aspectos são a separação do trecho de código que implementa conceitos ortogonais em classes distintas que são unidas em tempo de execução (*runtime*).

Ao invés de um grande bloco de código que cuida de transações, segurança, conexão com o banco de dados e ainda das regras de negócio nós podemos definir cada um destes aspectos em uma ou mais classes diferentes e a ferramenta de AOP irá fazer esta mistura enquanto o programa é executado. O diagrama abaixo ilustra os diversos aspectos separados enquanto estamos desenvolvendo (esquerda) e o produto final em tempo de execução, a classe que mistura todos estes aspectos (direita).

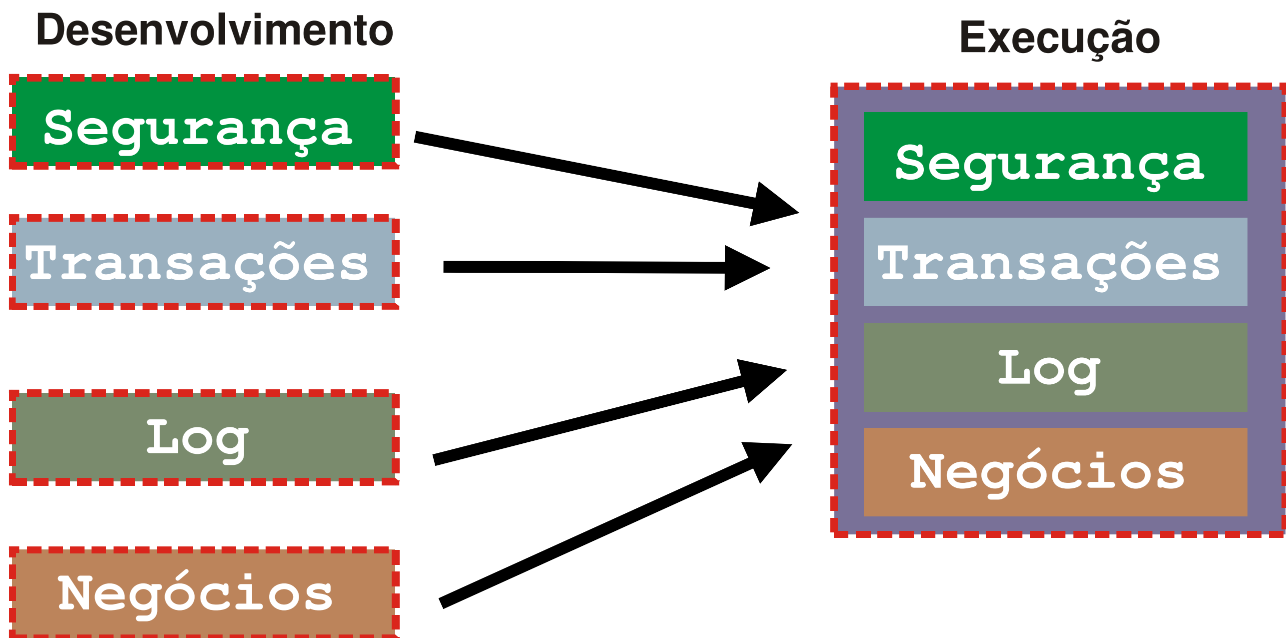


Figura 4.2 – Aspectos em Desenvolvimento e Execução

Claro que desta forma podemos também reaproveitar os aspectos em diversas classes. Geralmente você vai precisar iniciar ou terminar transações, verificar se o usuário tem os privilégios necessários e demais aspectos em diversas classes de negócio, utilizando AOP você pode criar aspectos apenas uma vez e ligá-los a diversas classes de negócio.

Aspectos são formados por:

- ***Advices*** - Código a ser executado para cumprir uma tarefa (abrir ou fechar uma conexão, por exemplo)
- ***Pointcuts*** – Lugares na classe-alvo (a classe com regras de negócio) onde o *Advice* deve ser invocado (por exemplo sempre que um método da classe `UsuarioDao` for chamado)

Ou seja, podemos ter um aspecto descrito como: *"Toda vez que se invocar um método cujo nome comece com 'fechar' (por exemplo 'fecharVenda') execute o código que inicia uma transação no banco de dados"*.

Usos Indicados

AOP é uma técnica utilizada principalmente para duas coisas: prevenir repetição de código e remover código de infra-estrutura do código da aplicação. Ao contrário do mito popular, não existe uma competição entre AOP e OOP, uma complementa a outra. Também não há necessidade de utilizar OOP com técnicas

de AOP, teoricamente podemos ter os mesmos benefícios em outros paradigmas.

Containeres como o Spring e EJB 3.0 utilizam AOP para atuar de forma transparente. Além de interceptarem o fluxo de execução do código para controlar a forma com que as classes se comportam, eles oferecem recursos para que o programador defina seus próprios aspectos.

É importante notar que AOP é uma técnica útil para fazer o que foi pensada: separar conceitos misturados em um único artefato de código (como uma classe) em outros, reutilizáveis. A tecnologia em si é poderosa e possibilita usos muito mais amplos mas normalmente isso não é uma boa idéia, causando complexidade excessiva que não se faz necessária.

Exercícios

1 - O trecho de código abaixo foi marcado em áreas com conceitos ortogonais distintos. Identifique o conceito implementado por cada área.

```
public class Noticia {
	private Usuario usuario = null;
	private SecaoDoSite secao = null;
	private String titulo=null;
	private String texto = null;

	public void publicar() throws UsuarioNaoAutorizadoException,
NamingException, SQLException {
		// verifica se o usuário é jornalista e se pode postar nesta seção
		if (!(usuario.isJornalista() &&
usuario.temDireitoPublicacao(secao))) {
			throw new UsuarioNaoAutorizadoException("Não pode publicar");
		}

		Context initContext = new InitialContext();
		Context envContext = (Context) initContext.lookup("java:/comp/env");
		DataSource ds = (DataSource) envContext.lookup("jdbc/mysql");
		Connection conn = ds.getConnection();

		PreparedStatement ps = conn.prepareStatement("INSERT INTO MATERIAS
(titulo, texto) VALUES (?,?)");
		ps.setString(0, titulo);
		ps.setString(1, texto);
		ps.execute();
		conn.commit();
	}

}
```

5. Aspect-Oriented Programming no Spring Framework

O Spring utiliza AOP para implementar o controle que realiza sobre seus beans, por exemplo ao gerenciar transações configuradas no `applicationContext.xml`. A estrutura de AOP provida pelo framework não oferece todos os recursos de outras implementações mas os recursos oferecidos costumam ser mais que suficientes para a criação de aplicativos.

Nota: A Interface21, empresa que emprega a maioria dos desenvolvedores do Spring Framework, contratou há algum tempo o criado do AspectJ, framework Java mais famoso e poderoso para AOP. A integração entre estes frameworks já pode ser vista na versão 2.0 do Spring Framework, trazendo a maioria dos recursos de AOP não presentes em versões anteriores.

Para entender como podemos implementar AOP no Spring vamos a um exemplo simples. Existe uma classe responsável por salvar arquivos no disco rígido (como é um exemplo a parte que efetivamente escreveria o arquivo em disco não está presente).

```
public class GerenciadorArquivos {
	public void salvarArquivo(Diretorio diretorioASalvar, Arquivo 
arquivoSalvo){
		System.out.println("GerenciadorArquivos: SALVO 
["+arquivoSalvo.getNome()+"] em ["+diretorioASalvar.getNome()+"]");
	}
}
```

Esta classe é utilizada por outra responsável por publicar notícias, cada notícia viraria um arquivo.

```
public class PublicadorNoticias {
	private GerenciadorArquivos gerenciadorArquivos = null;
	public void setGerenciadorArquivos(GerenciadorArquivos 
gerenciadorArquivos) {
		this.gerenciadorArquivos = gerenciadorArquivos;
	}
	public void publicar(String titulo){
		Arquivo arquivo = new Arquivo(titulo);
		Diretorio diretorio = new Diretorio("/usr/local/reportagens");
		gerenciadorArquivos.salvarArquivo(diretorio, arquivo);
	}
}
```

Isso tudo é configurado no Spring como vimos em capítulos anteriores:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
        "http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
	<bean id="gerenciadorArquivo" 
class="br.com.fragmental.cursos.spring.apostila.capitulo5.GerenciadorArquivos"
/>
	<bean id="publicadorNoticias"
```

```
        class="br.com.fragmental.cursos.spring.apostila.capitulo5.PublicadorNotici
as">
                <property name="gerenciadorArquivos" ref="gerenciadorArquivo"/>
        </bean>
</beans>
```

Para testar, utilizamos este trecho de código:

```
ApplicationContext applicationContext = new ClassPathXmlApplicationContext(

        "classpath:br/com/fragmental/cursos/spring/apostila/capitulo3/applicationC
ontext-Capitulo3.xml");
                Computador computador = (Computador)
applicationContext.getBean("computadorBean");
                computador.ligar();
```

E temos como resultado impresso na saída do programa:

```
GerenciadorArquivos: SALVO [Lançado Spring Framework 2.0!] em
[/usr/local/reportagens]
```

Este é o fluxo padrão de negócios.

Utilizando Advices Before e After

Para evitar a escrita de arquivos desnecessários em disco, foi determinado que uma mensagem deve ser imprimida num arquivo de log sempre que um arquivo for salvo. O meio mais simples seria fazer a classe `GerenciadorArquivos` escrever este log, mas já vimos que misturar código de negócios com código de infra-estrutura é problemático. Podemos então utilizar AOP para criar um aspecto que faça o log.

O primeiro passo é criar o advice. No caso usaremos um advice do tipo *Before*, chamado antes da invocação real do método.

```
public class LogAdvice implements MethodBeforeAdvice {
        public void before(Method metodo, Object[] argumentos, Object alvo) {
                // lista de parâmetros
                StringBuffer lista = new StringBuffer();
                for (int i = 0; i < argumentos.length; i++) {
                        lista.append("\n" + argumentos[i]);
                }
                System.out.println("LOG --- " + new Date() + " --- Executado método
'"
                        + metodo.getName() + "' utilizando como parâmetros " +
lista);
        }
}
```

Depois, modificamos o `applicationContext.xml`. Note que alteramos o `id` do GerenciadorArquivos para `gerenciadorArquivosTarget`. Esta mudança se faz necessária porque agora o gerenciadorArquivos não é mais apenas uma instância da nossa classe `GerenciadorArquivos`, o que o `PublicadorNoticias` recebe é uma mistura entre a classe `GerenciadorArquivos` e o `LogAdvice`, criada em tempo de execução pelo Spring.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
        "http://www.springframework.org/dtd/spring-beans.dtd">
<beans>

      <bean id="gerenciadorArquivoTarget"

      class="br.com.fragmental.cursos.spring.apostila.capitulo5.GerenciadorArquivos"/>

      <bean id="gerenciadorArquivo"
            class="org.springframework.aop.framework.ProxyFactoryBean">
            <property name="interceptorNames"><value>logAdvice</value></property>
            <property name="target" ref="gerenciadorArquivoTarget" />
      </bean>

      <bean id="logAdvice" class="br.com.fragmental.cursos.spring.apostila.capitulo5.LogAdvice" />

      <bean id="publicadorNoticias"

      class="br.com.fragmental.cursos.spring.apostila.capitulo5.PublicadorNoticias">
            <property name="gerenciadorArquivos" ref="gerenciadorArquivo"/>
      </bean>

</beans>
```

Esta mistura é feita quando nós declaramos que o bean `gerenciadorArquivos` é agora uma instância de `ProxyFactoryBean`. Esta classe fornecida pelo Spring Framework é configurada com o target que definimos antes e a lista de advices que devem ser aplicados.

Target é a classe 'crua' que vai ser misturada com os advices para gerar uma classe modificada.

A saída após estas modificações é:

```
LOG --- Sat Aug 05 21:29:25 GMT-03:00 2006 --- Executado método 'salvarArquivo' utilizando como parâmetros
br.com.fragmental.cursos.spring.apostila.capitulo5.Diretorio@ed0338
br.com.fragmental.cursos.spring.apostila.capitulo5.Arquivo@6e70c7
GerenciadorArquivos: SALVO [Lançado Spring Framework 2.0!] em [/usr/local/reportagens]
```

Ou seja: adicionamos a funcionalidade de registrar a execução do método sem alterar qualquer linha de código da classe `GerenciadorArquivos`.

Nota: Para utilizar este exemplo você precisa da biblioteca CGLIB2 no Classpath. Esta biblioteca permite que o Spring mude o código de classes e não trabalhe apenas com interfaces.

Existe também o *After* advice, que como o nome diz é executado após o método. Por exemplo, vamos definir um advice do tipo After que notifica (escrevendo na tela neste exemplo) sobre a publicação de uma nova notícia. Este advice será invocado após o método `publicar()` do `PublicadorNoticias`.

Primeiro, criamos o advice.

```
public class NotificacaoAdvice implements AfterReturningAdvice {
	Notificador notificador = null;
	public void afterReturning(Object valorDeRetorno, Method metodo,
			Object[] argumentos, Object alvo) {
		Arquivo arquivo = (Arquivo)argumentos[1];
		notificador.notificarSobreNoticiaNova(arquivo.getNome());
	}

	public void setNotificador(Notificador notificador) {
		this.notificador = notificador;
	}
}
```

O método `afterReturning()`, executado logo após o método na classe de negócios (nosso target) apenas usa a informação recebida como parâmetro e a envia ao notificador.

Para que este advice seja ativado, basta modificar o `applicationContext.xml`, acrescentando o advice no `ProxyFactory` como fizemos com o advice Before anteriormente.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
        "http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
	<bean id="gerenciadorArquivoTarget"

	class="br.com.fragmental.cursos.spring.apostila.capitulo5.GerenciadorArqui
vos"/>


	<bean id="gerenciadorArquivo"
		class="org.springframework.aop.framework.ProxyFactoryBean">
		<property
name="interceptorNames"><list><value>notificacaoAdvice</value>
			<value>logAdvice</value></list></property>
		<property name="target" ref="gerenciadorArquivoTarget" />
	</bean>


	<bean id="logAdvice"
class="br.com.fragmental.cursos.spring.apostila.capitulo5.LogAdvice" />
```

```
	<bean id="publicadorNoticias"

	class="br.com.fragmental.cursos.spring.apostila.capitulo5.PublicadorNotici
as">
		<property name="gerenciadorArquivos" ref="gerenciadorArquivo"/>
	</bean>


	<bean id="notificador"
class="br.com.fragmental.cursos.spring.apostila.capitulo5.Notificador" />


	<bean id="notificacaoAdvice"
class="br.com.fragmental.cursos.spring.apostila.capitulo5.NotificacaoAdvice">
		<property name="notificador" ref="notificador" />
	</bean>


</beans>
```

Como pode ser percebido nós utilizamos um elemento `<list>` para definir dois valores, os nomes dos advices. Podem haver mais de um advice de cada tipo sem problemas.

Outro ponto interessante é que o advice em si é um bean gerenciador pelo Spring e pode receber injeção de dependências: o `notificacaoAdvice` do exemplo é injetado com um `notificador`.

A saída do código de exemplo fica:

```
LOG --- Sat Aug 05 13:12:00 GMT-03:00 2006 --- Executádo método 'salvarArquivo'
utilizando como parâmetros
br.com.fragmental.cursos.spring.apostila.capitulo5.Diretorio@12a1e44
br.com.fragmental.cursos.spring.apostila.capitulo5.Arquivo@29428e
GerenciadorArquivos: SALVO [Lançado Spring Framework 2.0!] em
[/usr/local/reportagens]
Notificador: EXTRA EXTRA! 'Lançado Spring Framework 2.0!' !!!
```

Outro advice com comportamento muito parecido é o *Throws*, que é executado quando uma exceção é lançada. Para utilizar este advice deve-se implementar a interface `ThrowsAdvice`.

Advice do Tipo Around

Advices Before e After já são muito poderosos em si, mas eles possuem grandes limitações: advices After não podem alterar o retorno e a única maneira de um advice Before não permitir a execução do método do target, por exemplo porque quebra uma restrição de segurança, é lançando uma exceção.

Para operações que realmente interfiram na execução de um método devemos utilizar o *Around* advice. Este advice é muito utilizado nos mecanismos internos do Spring, para demonstrar seu uso vamos a outro exemplo.

Temos uma classe `UsuarioDao` que recebe uma conexão, um objeto usuário e o salva no banco de dados (este código é meramente ilustrativo e não deve ser tomado como exemplo de DAO).

```
public class UsuarioDao {
	public void salvar(Usuario usuario, Conexao conexao) {
```

```
            String sql = "INSERT INTO USUARIOS VALUES(" + usuario.getLogin() +
")";
            System.out.println("DAO: Executando SQL=[" + sql + "]");
            conexao.executar(sql);
      }
}
```

Acontece que após algum tempo percebeu-se que não há como garantir que a conexão esteja aberta quando o DAO a receber. Ao invés de modificar todos os DAOs do sistema é mais simples criar um advice que cuide deste processo de abertura e fechamento de conexão. Abaixo temos um advice tipo Around que cumpre este papel.

```
public class VerificaConexaoDisponivelAdvice implements MethodInterceptor {
      public Object invoke(MethodInvocation invocacao) {
            Object[] argumentos = invocacao.getArguments();
            // pega a conexão passada como parâmetro
            Conexao conexao = (Conexao) argumentos[1];
            // se não existe conexão, retorne
            if (conexao == null) {
                  System.out.println("Nenhuma conexão passada");
                  return null;
            } else {
                  // se a conexão existe mas não está aberta, abra-a
                  if (!conexao.isAberta()) {
                        System.out.println("Conexão fechada, abrindo...");
                        conexao.abrir();
                  }
                  // execute o método
                  try {
                        System.out.println("Invocando...");
                        invocacao.proceed();
                  } catch (Throwable e) {
                        // problemas na execução do método!
                        e.printStackTrace();
                  }
                  finally{
                        //Fechando conexão usada
                        System.out.println("Fechando Conexão...");
                        conexao.fechar();
                  }
                  // este método não tem retorno (void)
                  return null;
```

```
            }
      }
}
```

Para termos este advice funcionando no nosso sistema basta configurarmos o `applicationContext.xml`:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
        "http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
      <bean id="usuarioDaoTarget"
class="br.com.fragmental.cursos.spring.apostila.capitulo5.UsuarioDao" />


      <bean id="usuarioDao"
            class="org.springframework.aop.framework.ProxyFactoryBean">
            <property name="interceptorNames">
<value>verificaConexaoDisponivelAdvice</value>
      </property>
            <property name="target" ref="usuarioDaoTarget" />
      </bean>


      <bean id="verificaConexaoDisponivelAdvice"
class="br.com.fragmental.cursos.spring.apostila.capitulo5.VerificaConexaoDisponi
velAdvice" />
</beans>
```

Testando com o exemplo abaixo:

```
ApplicationContext applicationContext = new ClassPathXmlApplicationContext(
      "classpath:br/com/fragmental/cursos/spring/apostila/capitulo5/applicationC
ontext-Capitulo5.xml");
UsuarioDao dao = (UsuarioDao) applicationContext.getBean("usuarioDao");
Usuario usuario = new Usuario("pcalcado");


// conexão fechada...
System.out.println("Primeiro exemplo");
Conexao conexao = new Conexao();
dao.salvar(usuario, conexao);
System.out.println("-------------------------");
// sem conexão...
System.out.println("Segundo exemplo");
dao.salvar(usuario, null);
System.out.println("-------------------------");
```

```
// conexão aberta...
System.out.println("Terceiro exemplo");
Conexao conexao2 = new Conexao();
conexao2.abrir();
dao.salvar(usuario, conexao2);
```

Temos a saída:

```
Primeiro exemplo
Conexão fechada, abrindo...
Invocando...
DAO: Executando SQL=[INSERT INTO USUARIOS VALUES(pcalcado)]
Fechando Conexão...
-------------------------
Segundo exemplo
Nenhuma conexão passada
-------------------------
Terceiro exemplo
Invocando...
DAO: Executando SQL=[INSERT INTO USUARIOS VALUES(pcalcado)]
Fechando Conexão...
```

Exercícios

1 – Modifique o exemplo tirando o log da classe `GerenciadorArquivos` e colocando esta funcionalidade na classe `PublicadorNoticias`.

2 – Descubra quanto tempo os métodos demoram para executar utilizando advices. Primeiro tente com Before e After, depois com Around.

6. Aplicações Baseadas em POJOs

Como vimos, o uso do Spring Framework tem como grande vantagem o fato de não interferir diretamente na modelagem da sua aplicação. O uso de um container de IoC como o Spring afeta a arquitetura do sistema (como quando se escolhe IoC ao invés do uso de Registry, ou *POJOs* ao invés de EJBs) mas os objetos de negócio não sofrem muitos impactos.

Como ***Arquitetura*** *entendemos o conjunto de decisões sobre a divisão de um sistema em módulos e a forma como estes módulos interagem.*

Mais que isso, o Spring induz ao uso de boas práticas de programação ao se basear ostensivamente em interfaces ao invés de classes. Esta característica, além de permitir extensão e adaptação fácil do framework e ser um conceito da Orientação a Objetos, possibilita o uso de testes unitários com muita facilidade.

Projetando Aplicações J2EE

A cultura de desenvolvimento J2EE difundida por livros e fornecedores até pouco tempo (e que continua enraizada na maioria dos ambientes de desenvolvimento) é de que se deve ter dois tipos de objetos num sistema: objetos de dados e objetos de lógica.

A prática de separar objetos nestes dois seguimentos é completamente avessa aos objetivos originais da Orientação a Objetos, que é de ter estado (dados) e comportamento (lógica) no mesmo componente (veja *[Fantoches]*).

O framework EJB divide seus componentes em três tipos básicos:

- **Message-Driven Beans:** Classes que consomem mensagens assíncronas via JMS
- **Session Beans:** Objetos que implementam a lógica de negócios a ser executada
- **Entity Beans:** Objetos que contêm os dados sendo manipulados, com mapeamento direto do banco de dados relacional

Para que sua aplicação disponha de recursos sofisticados de J2EE padrão (sem uso de ferramentas de terceiros) como controle de transações é indicado que sua modelagem siga esta divisão. Desta forma, se descobrimos durante a análise do sistema que devemos ter um objeto como o abaixo:

Figura 6.1– Objeto Encontrado na Análise

Este objeto reúne estado e comportamento relativos ao conceito que queremos modelar em software, no caso uma música. Ele mapeia da melhor maneira possível o conceito que entendemos como “música” para objetos.

Ao utilizar EJBs, entretanto, teríamos que mapear este objeto de negócio em pelo menos dois objetos:

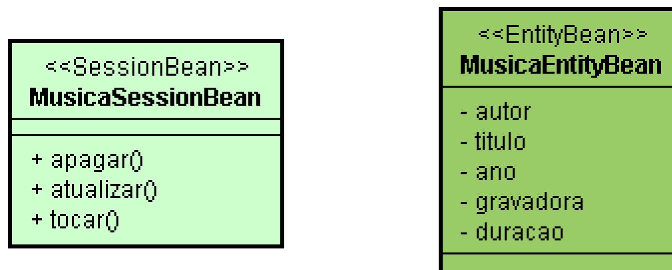


Figura 6.2– Objeto Separado em EJBs

Note que estes objetos não fazem parte da análise nem fazem sentido ao domínio. Para o usuário uma música é algo com título, autor, gravadora, duração e ano que ele toca no rádio, explicar que este conceito fica dividido em duas metades e que o ato de tocar uma música não é gerenciado pelo mesmo componente que gerencia a própria música é confuso. Como se não bastassem estes problemas, para cada EJB nós devemos criar uma série de arquivos de configuração e interfaces que não servem para nada mais que satisfazer o framework.

Nada impede que EJBs sejam utilizados sem esta estrutura. Podemos ter SessionBeans como *Services* que manipulam *POJO*s, e DAOs JDBC ou um framework como Hibernate substituindo Entity Beans. Ainda assim existe a necessidade de adaptar a modelagem dos *Services* ao framework EJB, criar vários arquivos de configuração, interfaces remotas e etc.

Utilizando POJOs

A *Figura 6.1* traz um exemplo claro de *POJO*. É uma classe simples que representa um conceito de negócio e não está presa à nenhum framework ou biblioteca.

POJO *(Plain Old Java Object) é um objeto Java normal, que não implementa nem estende nenhuma classe de infra-estrutura (de um framework, por exemplo). Um Servlet não é um POJO porque precisa estender* `javax.servlet.http.HttpServlet` *para funcionar.*

Idealmente o desenvolvimento OO em Java deve ser feito utilizando POJOs. Os conceitos da aplicação devem ser implementados de forma extremamente simples, sem influência da arquitetura escolhida.

Como mencionado, esta abordagem possui problemas quando estamos utilizando J2EE. O maior deles é que a infra-estrutura provida por este framework não foi pensada para o uso de objetos simples e sim de objetos integrados ao framework como EJBs e Servlets.

Para não deixar que a infra-estrutura J2EE influencie diretamente sua aplicação e ainda assim obter as funcionalidades que J2EE traz para os sistemas é preciso utilizar um chamado *Container Leve.*

Container Leve *(Lightweight Container) é um container que dá a objetos Java normais (POJOs) características que normalmente não estariam disponíveis para estes, como mecanismos de segurança, transações e IoC.*

Ao utilizar um container leve como o Spring é possível utilizarmos objetos simples como os da figura *Figura 6.1* ao invés da estrutura "artificial" mostrada na *Figura 6.2*, exigida pelos EJBs.

Esta é a característica que distingue aplicações baseadas em containeres leves (como Spring)de outras baseadas em herança e frameworks intrusivos (como EJB).

7. Projeto de Aplicações com Spring

Além de possibilitar o desenvolvimento baseado em POJOs, o Spring induz o uso de boas práticas que fazem esta abordagem realmente interessante. Duas destas práticas são: foco em interfaces e o uso de *Domain Models*.

Programando para Interfaces

Nos exemplos executados até agora tivemos sempre nossos beans implementados diretamente por classes. O bom uso de Orientação a Objetos, entretanto, pede que os sistemas foquem em contratos, não em implementação.

Uma interface é um contrato a ser obedecido por uma classe. Ao implementar uma interface uma classe se compromete a implementar o comportamento descrito nesta interface. Ao vincular sua classe apenas ao contrato ao temos flexibilidade de mudar a implementação sem alterar as classes que dependem desta. Como exemplo, verifique o domínio abaixo.

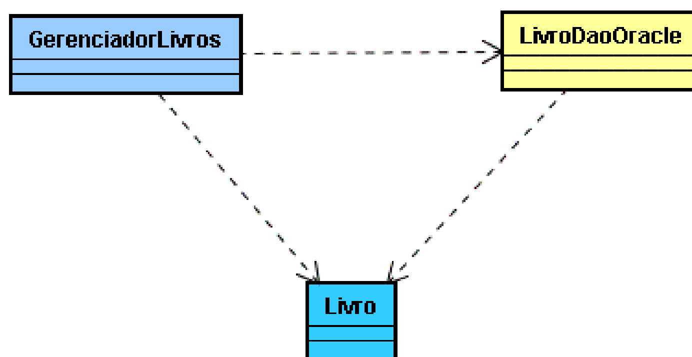


Figura 7.1- Domínio com DAO

Neste exemplo temos um `GerenciadorLivros` que utiliza um DAO para persistir os `Livro`s. O DAO em questão é uma implementação exclusiva para lidar com banco de dados Oracle. O fonte do `GerenciadorLivros` fica parecido com o código abaixo.

```
public class GerenciadorLivros {
	private LivroDaoOracle livroDao = null;
	public void setLivroDao(LivroDaoOracle livroDao) {
		this.livroDao = livroDao;
	}

	//métodos de negócio
}
```

E este seria o `applicationConfig.xml`:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
        "http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
	<bean id="livroDaoOracle"
	class="br.com.fragmental.cursos.spring.apostila.capitulo6.LivroDaoOracle"
```

```
/>

	<bean id="gerenciadorLivros"
	class="br.com.fragmental.cursos.spring.apostila.capitulo6.GerenciadorLivro
s">
		<property name="livroDao" ref="livroDaoOracle" />
	</bean>
</beans>
```

Se quisermos que nosso sistema também possa utilizar MySQL, precisamos alterar o `GerenciadorLivros` como mostrado no diagrama abaixo.

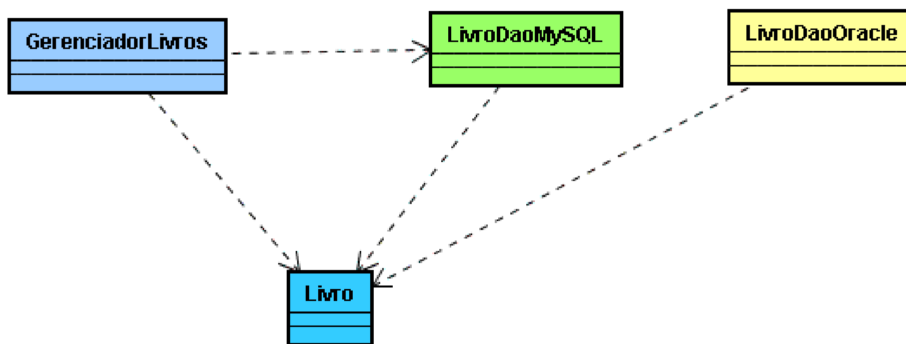


Figura 7.2- Alterando para MySQL

Efetuando as mudanças destacadas no código:

```
public class GerenciadorLivros {

	private LivroDaoMySQL livroDao = null;

	public void setLivroDao(LivroDaoMySQL livroDao) {
		this.livroDao = livroDao;
	}

	//métodos de negócio
}
```

Além da mudança no `applicationContext.xml`:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
        "http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
	<bean id="livroDaoOracle"
	class="br.com.fragmental.cursos.spring.apostila.capitulo6.LivroDaoOracle"/
>
```

```
	<bean id="livroDaoMySQL"
	class="br.com.fragmental.cursos.spring.apostila.capitulo6.LivroDaoMySQL"/>

	<bean id="gerenciadorLivros"
	class="br.com.fragmental.cursos.spring.apostila.capitulo6.GerenciadorLivro
s">
		<property name="livroDao" ref="livroDaoMySQL" />
	</bean>
</beans>
```

O simples fato de isolar a lógica de acesso a dados no DAO permite que o `GerenciadorLivros` possa ser alterado sem muita alteração de código, porém **existe** alteração no `GerenciadorLivros` e isso não é necessário nem desejável.

Num sistema de verdade estas mudanças simples deveriam ser feitas em diversos lugares para cada mudança deste tipo, e este não é o tipo de problema que ocorre apenas com mudanças drásticas como a de o SGBD utilizado, ele se manifesta toda vez que alguma coisa na implementação mudar.

Pensando no problema, podemos abstrair o modelo chegando a conclusão que *o* `GerenciadorLivros` *precisa de um DAO para persistir os objetos.* O problema que motiva a mudança de código no `GerenciadorLivros` para mudarmos de banco de dados é que da forma como está implementado hoje este não depende ***de um DAO***, ele depende ***do LivroDaoOracle*** especificamente (a nossa mudança apenas o fez depender de outro DAO). Esta é a dependência a ser eliminada.

Para eliminarmos a dependência direta, vamos fazer com que a classe `GerenciadorLivros` dependa de fato de um DAO qualquer criando uma interface genérica para DAOs.

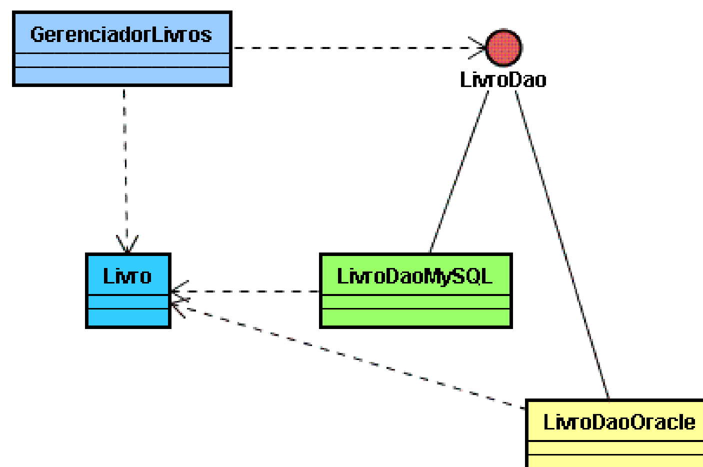


Figura 7.2- Alterando para MySQL

Temos, então, este código no `GerenciadorLivros`:

```
public class GerenciadorLivros {

	private LivroDao livroDao = null;
```

```
	public void setLivroDao(LivroDao livroDao) {
		this.livroDao = livroDao;
	}

	//métodos de negócio
}
```

E agora para alterar o DAO utilizado basta configurar o `applicationContext.xml` de conforme os exemplos anteriores.

Este mesmo princípio deve ser utilizado ao relacionar os beans de negócio, não apenas de infra-estrutura. Um bean deve depender do *conceito* representado por outro bean, não pela forma como ele está implementado. Por exemplo, considere outro modelo de objetos.

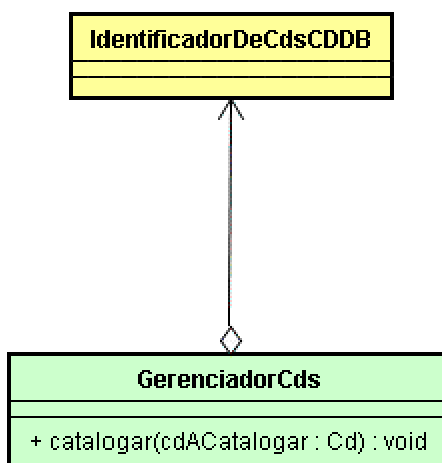


Figura 7.3 – Catalogando CDs

Neste exemplo temos o bean `GerenciadorCds` que depende de um serviço de identificação de CDs. Atualmente esta identificação é feita através de uma consulta ao CDDB (http://en.wikipedia.org/wiki/CDDB), um banco de dados disponível na Internet com informações sobre os CDs catalogados (o mesmo utilizado, por exemplo, pelo iTunes).

Durante o desenvolvimento da aplicação, entretanto, não é desejável conectar a esta base de dados o tempo todo. O desenvolvedor pode ter que parar de trabalhar porque a conexão com a Internet está com problemas ou pode precisar ter controle sobre a resposta dada pelo CDDB para simular comportamentos possíveis (o que acontece se não houver conexão? E se o CDDB retornar uma resposta formatada de maneira errada?).

Mesmo se a aplicação for desenvolvida e estiver em produção existe a possibilidade de se alterar a forma como os dados do CD são acessados. Imagine que passamos a catalogar CDs de bandas alternativas que ainda não estão no CDDB, podemos precisar de um objeto que tente encontrar o CD no CDDB e caso não ache procure em outro lugar. Ou podemos precisar manter informações em cachê local para minimizar as conexões via internet... todas estas alterações vão acabar se refletindo em mudanças no código do `GerenciadorCds`, o que não é nem um pouco desejável.

Este é o mesmo problema que tivemos com o DAO anteriormente e podemos resolve-lo da mesma forma. O `GerenciadorCds` não precisa especificamente do `IdentificadorDeCdsCDDB`, ele precisa de um `IndentificadorDeCDs` qualquer. Isso nós podemos modelar com uma interface.

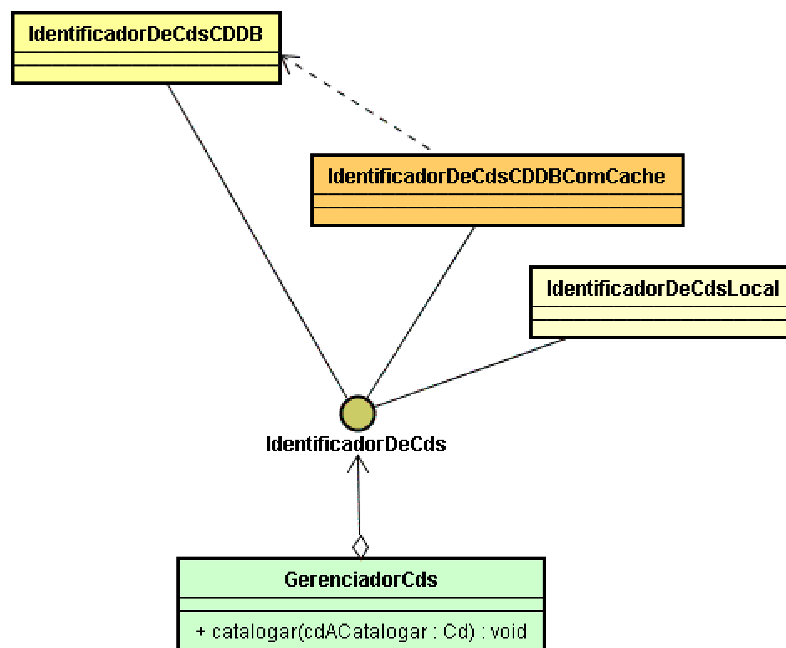


Figura 7.4 – O identificador de CDs genérico

E assim como no exemplo do DAO para escolher qual identificador utilizar basta alterar o `applicationContext.xml`.

O padrão do Spring, **que será utilizado deste ponto em diante neste texto**, é utilizar o sufixo `Impl` para as implementações quando o nome da interface coincidir com o da implementação, como no exemplo abaixo:

```
public interface GerenciadorAutores {
	//metodos de negocio
}
```

```
public class GerenciadorAutoresImpl implements GerenciadorAutores{
	//metodos de negocio
}
```

```
public interface GerenciadorEditoras {
	//metodos de negocio
}
```

```
public class GerenciadorEditorasImpl implements GerenciadorEditoras{

	private GerenciadorAutores gerenciadorAutores = null;

	public void setGerenciadorAutores(GerenciadorAutores gerenciadorAutores) {
```

```
            this.gerenciadorAutores = gerenciadorAutores;
      }
      // metodos de negocio
}
```

Configurados no `applicationContext.xml` desta forma:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
        "http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
      <bean id="gerenciadorAutores"
      class="br.com.fragmental.cursos.spring.apostila.capitulo6.GerenciadorAutoresImpl"/>

      <bean id="gerenciadorEditoras"
      class="br.com.fragmental.cursos.spring.apostila.capitulo6.GerenciadorEditorasImpl">
            <property name="gerenciadorAutores" ref="gerenciadorAutores" />
      </bean>
</beans>
```

Padrões de Modelagem de Domínio

Ao modelar o Domain Model (veja *[PadroesDeNegocio]*) da sua aplicação existem algum Padrões que podem ser observados. Eles definem alguns dos componentes básicos de um Domain Model, as pedras fundamentais para a modelagem de Regras de Negócio de maneira eficiente.

O Spring Framework e outros containeres leves são ideais para o uso destes padrões em Java. Segue um resumo destes padrões, mais detalhes na bibliografia.

Entity

Entities (ou Entidades, nenhuma relação com Entity Beans) são objetos que possuem uma identidade única. Um carrinho de compras numa loja virtual web não é igual a outro. Não importa que possuam os mesmos produtos, o carrinho A é o carrinho do usuário A, o carrinho B é do usuário B. Mesmo que contenham os mesmos produtos você não pode exibir o carrinho B ao usuárioA, eles são diferentes! O carrinho neste exemplo segue o Padrão **Entity**., ele é uma entidade de negócios única.

Value Object

Nem todos os objetos precisam de identidade. Muitos objetos “menores” presentes em nosso modelo de negócios têm significado apenas pelo valor que possuem. Imagine um objeto que representa uma data, digamos “01/06/1979”. Apesar de ser diferente de um objeto que representa a data “21/02/1983” é igual a (e pode ser utilizado no lugar de) qualquer outro que represente a mesma data que ele. Datas são geralmente **Value Objects**, o que importa é seu valor.

Service

Services são classes que não implementam diretamente as regras de negócio da aplicação, apenas coordenam a interação entre os componentes. Elas são quase sempre beans gerenciados pelo Spring todas as nossas classes `GerenciadorXyz` deste texto são Services. É muito importante que Services não implementem as regras de negócio, apenas atuem como Façades coordenando as interações.

Repository

Um Repository é onde os objetos ficam armazenados. Para as classes de negócio não importa se os objetos são armazenados em arquivos XML, num banco de dados ou qualquer outro lugar, eles apenas precisam que ao consultar o Repository possam obter estas classes. O Padrão Data Access Object – *DAO* geralmente é utilizado para implementar o **Repository**. O Repository quase sempre é definido como uma interface implementada pelo DAO.

Nota: *A Sun publicou em seu Core J2EE Patterns um Padrão chamado* Value Object. *Este Padrão é uma especialização do Padrão* Data Transfer Object *posteriormente catalogado por Martin Fowler. Na segunda edição do livro a Sun alterou o nome deste para* Transfer Object*, porque Martin Fowler e Eric Evans já haviam incluído um Padrão chamado* ***Value Object*** *(que é o descrito acima) em suas obras.*

Geralmente programadores Java se referem ao Transfer Object *como* Value Object*. Além do fato desta nomenclatura estar desatualizada segundo o próprio catálogo de Padrões oficiais de Java EE, Transfer Objects não devem ser utilizados em sistemas que utilizam apenas uma JVM. Um TO utilizado desta maneira acaba com a Orientação a Objetos criando um design altamente Procedural. Este Padrão foi publicado visando uso em sistemas onde Entity Beans trocam informações numa rede.*

O diagrama abaixo mostra uma modelagem feita com estes padrões.

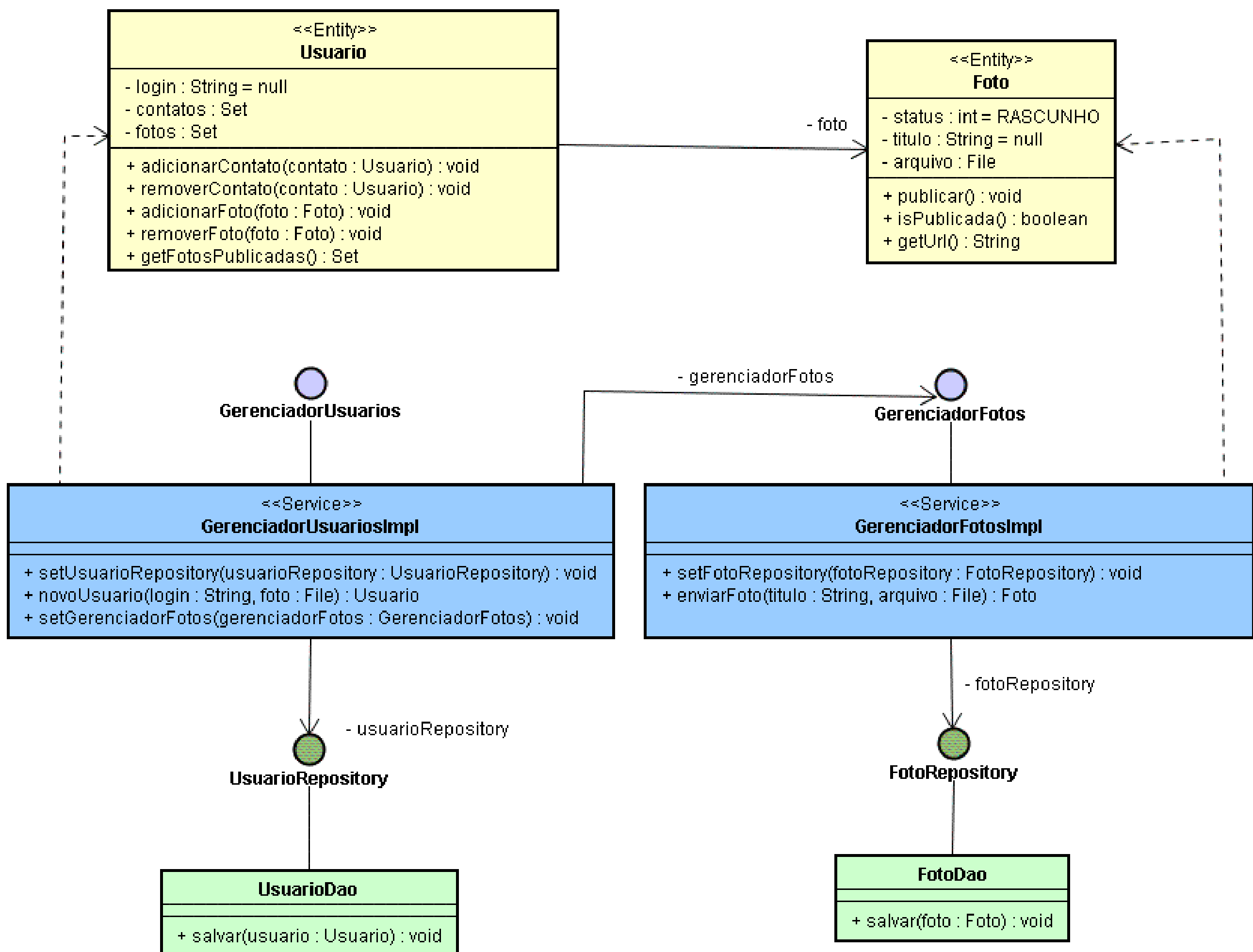


Figura 7.5 – Modelagem utilizando Padrões de Negócio

Neste modelo temos o `Usuario` que possui `Foto`s como Entities. As regras de negócio estão implementadas dentro destas Entities, como vemos abaixo:

```
public class Usuario {
	private Set contatos = new HashSet();
	private Set fotos = new HashSet();
	private Foto foto = null;
	private String login = null;

	public Usuario(String login) {
		super();
		this.login = login;
	}
	public void setFoto(Foto foto) {
		this.foto = foto;
	}
	public void adicionarContato(Usuario contato) {
		contatos.add(contato);
	}
	public void removerContato(Usuario contato) {
		contatos.remove(contato);
	}
	public void adicionarFoto(Foto foto) {
		fotos.add(foto);
	}
	public void removerFoto(Foto foto) {
		fotos.remove(foto);
	}
	/**Retorna todas as fotos deste usuário que estão publicadas atualmente.*/
	public Set getFotosPublicadas() {
		Set fotosPublicadas = new HashSet();
		for (Iterator it = fotos.iterator(); it.hasNext();) {
			Foto foto = (Foto) it.next();
			if(foto.isPublicada()) fotosPublicadas.add(foto);
		}
		return fotosPublicadas;
	}
	public String getLogin() {
		return login;
	}
}
```

```
public class Foto {
	public static final int RASCUNHO = -1;
	public static final int PUBLICADA = 10;
	public static final int DESPUBLICADA = 100;
	private int status = RASCUNHO;
	private File arquivo = null;
	private String titulo = null;
	public void setTitulo(String titulo) {
		this.titulo = titulo;
	}
	public String getTitulo() {
		return titulo;
	}
	public Foto(File arquivo) {
		super();
		if (arquivo == null)
			throw new IllegalArgumentException("Deve ser definido um 
arquivo!");
		this.arquivo = arquivo;
	}
	public void publicar() {
		this.status = PUBLICADA;
	}
	public boolean isPublicada() {
		return this.status == PUBLICADA;
	}
	public String getUrl() {
		String url = null;
		try {
			url = arquivo.toURL().toString();

		} catch (MalformedURLException e) {
			throw new IllegalStateException("Problemas ao gerar URL da 
foto", e);
		}
		return url;
	}
}
```

O que os Services fazem é apenas coordenar o fluxo de interações. Vamos ver a implementação da criação

de um novo usuário:

```
public class GerenciadorUsuariosImpl implements GerenciadorUsuarios {
	private GerenciadorFotos gerenciadorFotos = null;
	private UsuarioRepository usuarioRepository=null;
	public void setUsuarioRepository(UsuarioRepository usuarioRepository) {
		this.usuarioRepository = usuarioRepository;
	}
	public Usuario novoUsuario(String login, File foto){
		Usuario novoUsuario = new Usuario(login);
		Foto fotoUsuario = gerenciadorFotos.enviarFoto(login, foto);
		novoUsuario.setFoto(fotoUsuario);
		usuarioRepository.salvar(novoUsuario);
		return novoUsuario;
	}
	public void setGerenciadorFotos(GerenciadorFotos gerenciadorFotos) {
		this.gerenciadorFotos = gerenciadorFotos;
	}
}
```

```
public class GerenciadorFotosImpl implements GerenciadorFotos {
	private FotoRepository fotoRepository=null;
	public void setFotoRepository(FotoRepository fotoRepository) {
		this.fotoRepository = fotoRepository;
	}
	public Foto enviarFoto(String titulo, File arquivo){
		Foto foto = new Foto(arquivo);
		foto.setTitulo(titulo);

		//processa o arquivo da foto
	    arquivo.renameTo(new File(DIRETORIO_FOTOS, arquivo.getName()));
		arquivo.setReadOnly();

		fotoRepository.salvar(foto);

		return foto;
	}


}
```

Por último, os Repositories são implementados por DAOs que, neste exemplo fictício, apenas escrevem na tela:

```
public class UsuarioDao implements UsuarioRepository{
	public void salvar(Usuario usuario) {
		System.out.println("SALVANDO USUARIO ["+usuario.getLogin()+"]");
	}
}
```

```
public class FotoDao implements FotoRepository {
	public void salvar(Foto foto) {
		System.out.println("SALVANDO FOTO [" + foto.getTitulo() + "]");
	}
}
```

Para unirmos todos estes objetos, contamos com o seguinte `applicationContext.xml`:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
        "http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
	<bean id="usuarioRepository"
class="br.com.fragmental.cursos.spring.apostila.capitulo6.exemplo.UsuarioDao" />
	<bean id="fotoRepository"
class="br.com.fragmental.cursos.spring.apostila.capitulo6.exemplo.FotoDao" />

	<bean id="gerenciadorFotos"
class="br.com.fragmental.cursos.spring.apostila.capitulo6.exemplo.GerenciadorFot
osImpl">
		<property name="fotoRepository" ref="fotoRepository" />
	</bean>

	<bean id="gerenciadorUsuarios"
class="br.com.fragmental.cursos.spring.apostila.capitulo6.exemplo.GerenciadorUsu
ariosImpl">
		<property name="gerenciadorFotos" ref="gerenciadorFotos" />
		<property name="usuarioRepository" ref="usuarioRepository" />
	</bean>
</beans>
```

Exercícios

1 – No modelo apresentado na seção sobre Padrões de Modelagem de Domínio, implemente a funcionalidade que permite aos usuários:

a) Indicarem e removerem seus contatos

b) Enviarem e removerem fotos

c) Alterar sua foto de usuário

Bibliografia

[ContratosNulos]

Calçado, Phillip - Contratos Nulos - Fragmental TI LTDA, 2006

http://fragmental.com.br/wiki/index.php?title=Contratos_Nulos

[DomainDrivenDesign]

Evans, Eric – Domain Driven Design – Addison-Wesley, 2004- ISBN 0321125215

[Fantoches]

Calçado, Phillip - Fantoches - Fragmental TI LTDA, 2006

http://fragmental.com.br/wiki/index.php?title=Fantoches

[PadroesDeNegocio]

Calçado, Phillip - Desenvolvendo Sistemas OO com Padrões de Negócio – MundoJava #17 – Editora Mundo

[ProSpring]

Harrop, Rob; Machacek, Jan – Pro Spring – Apress, 2005 – ISBN 1590594614

[SpringInAction]

Walls, Craig; Breidenbach, Ryan – Spring in Action - Manning, 2005 – ISBN 1932394354

[SpringReference]

Johson, Rod et Al – Spring Java/J2EE Application Framework

http://static.springframework.org/spring/docs/1.2.x/reference/index.html