5
Derivando Aplicações Baseadas em Spring e OSGi

O capítulo 3 apresentou como um conjunto de artefatos (classes, interfaces, aspectos e arquivos extras) de uma arquitetura de LPS pode ser modelado na ferramenta e como essa automaticamente deriva produtos com diferentes customizações. A simples utilização das abstrações referentes aos artefatos citados não fornece informação suficiente para endereçar a customização de arquiteturas implementadas com abstrações de mais alto nível, como, por exemplo: componentes, agentes, serviços. Dessa forma, para que seja viável a customização de arquiteturas mais complexas, é necessário o acréscimo de novas abstrações, relacionadas às tecnologias utilizadas, nos modelos já existentes (implementação e configuração) ou até mesmo a criação de novos modelos. Este capítulo descreve como a ferramenta GenArch foi estendida para endereçar a derivação de produtos baseados nas tecnologias de componentes Spring e OSGi. O objetivo dessa extensão é permitir a customização de: (i) propriedades nos descritores dos componentes (*beans* e *bundles*); e (ii) seleção de quais componentes constituirão o produto final gerado.

5.1.
Spring e OSGi – Tecnologias de Componentes

Spring é um framework open-source criado para endereçar a complexidade do desenvolvimento de aplicações corporativas implementadas na linguagem Java. Ele permite o desenvolvimento de aplicações, através do uso de um modelo de componentes baseado em Java Beans (Johnson et al. 2005). Um componente é denominado *Bean* e implementa a lógica de negócio da aplicação. Nesse caso, o desenvolvedor fica responsável somente pela implementação da lógica do negócio (*Beans*), enquanto o framework Spring fica responsável por atender características adicionais como: transação, segurança, logging, etc. Permitindo dessa forma, ampliar as funcionalidades básicas provida pelo *Bean* com requisitos indispensáveis no desenvolvimento de aplicações corporativas. Spring torna possível o uso de um modelo de componentes simples, testável e de baixo acoplamento para endereçar características que

previamente somente eram contempladas por modelos de componentes complexos, como Enterprise Java Beans (Roman et al. 2004). A simplicidade e baixo acoplamento do modelo de componentes Spring é realizado pelo princípio de inversão de controle (IoC – *Inversion of Control*) e pelo container orientado a aspectos. Na técnica de IoC, também chamada de injeção de dependência (Johnson 2003), objetos são passivamente dotados de suas dependências ao invés de criarem ou procurarem por elas. No framework Spring, um bean expressa suas dependências através da exposição de métodos de configuração de atributos (*setters*) ou através de argumentos nos métodos construtores. O container orientado a aspectos provê uma solução flexível para o desenvolvimento de interesses transversais corporativos, tais como, gerenciamento de transação, *logging* e segurança.

A unidade de instalação primária do Spring é uma aplicação composta por *beans*. Tal aplicação é normalmente descrita em um arquivo de configuração XML, denominado *application context*. Um *application context* especifica os componentes (*beans*) que compõem o contexto de execução da aplicação Spring. Geralmente esse arquivo contém a definição de um ou mais beans, que tipicamente especifica a classe Java que o implementa, valores iniciais para algumas propriedades e a definição dos respectivos beans que devem ser injetados. Adicionalmente, este arquivo também define quais aspectos serão aplicados a cada bean da aplicação.

Open Services Gateway Initiative (OSGi) (OSGi 2003) é um consórcio de aproximadamente oito companhias de diversos lugares do mundo que define uma plataforma e uma infra-estrutura que permite a instalação de serviços em redes locais e dispositivos. A especificação OSGi define uma arquitetura comum e aberta para provedores de serviços, desenvolvedores, vendedores de software, operadores de gateway, instalarem e gerenciarem serviços de forma coordenada.

De acordo com a especificação OSGi, aplicações Java são estruturadas em um conjunto de *bundles*, onde cada *bundle* representa um componente da aplicação que provê serviços para um usuário final ou outros componentes. Um *bundle* é empacotado como um arquivo .jar da linguagem Java que contém, além de outros recursos (classes, aspectos, figuras), um arquivo de manifesto que mantém informações sobre o *bundle* que precisam ser fornecidas para o container OSGi no momento da instalação. Incluem nas informações: (i) a localização de uma classe ativadora, que é ativada quando o *bundle* instalado é iniciado ou finalizado; (ii) dependências; (iii) serviços usados; (iv) serviços

providos; e (v) informações em geral sobre o *bundle*. *Bundles* são instalados e executados em um container que implementa a especificação OSGi. Um container OSGi é responsável pelo gerenciamento do ciclo de vida dos componentes, permitindo a instalação e execução desses de forma dinâmica, e a coexistência de diferentes versões de um mesmo *Bundle*.

5.2. Integração das Tecnologias Spring e OSGi

A iniciativa *Spring Dynamic Modules* (SDM) (SpringDynamicModules 2008) tem por objetivo integrar o modelo de componente Spring com o modelo de componentes dinâmico fornecido pela plataforma OSGi. SDM permite a exportação e importação transparente, gerenciamento e controle do ciclo de vida de serviços OSGi dentro de um Spring *application context*. Esse modelo de desenvolvimento torna fácil a escrita de aplicativos Spring que podem ser instalados em um ambiente de execução dinâmico e faz o desenvolvimento de aplicativos OSGi mais simples e produtivo através do uso do modelo de programação Spring. Usando SDM, um *bundle* ativo pode conter um Spring application context, responsável pela instanciação, configuração, montagem e decoração de objetos (*beans*) dentro de um *bundle*. Alguns desses beans podem ser exportados como serviços OSGi e para ser disponibilizados para outros *bundles*. Beans também podem ser transparentemente injetados como referência em serviços OSGi.

A combinação entre Spring e OSGi provê (SpringDynamicModules 2008): (i) melhor separação da lógica da aplicação em módulos; (ii) habilidade de instalação de múltiplas versões de um mesmo módulo (ou biblioteca) concorrentemente; (iii) habilidade de dinamicamente descobrir e usar serviços providos por outros módulos no sistema; (iv) habilidade de dinamicamente instalar, atualizar e remover módulos em tempo de execução; (v) uso do Spring para instanciar, configurar, montar e decorar componentes dentro e entre módulos; e (vi) um modelo de programação simples e familiar para os desenvolvedores explorarem as características da plataforma OSGi.

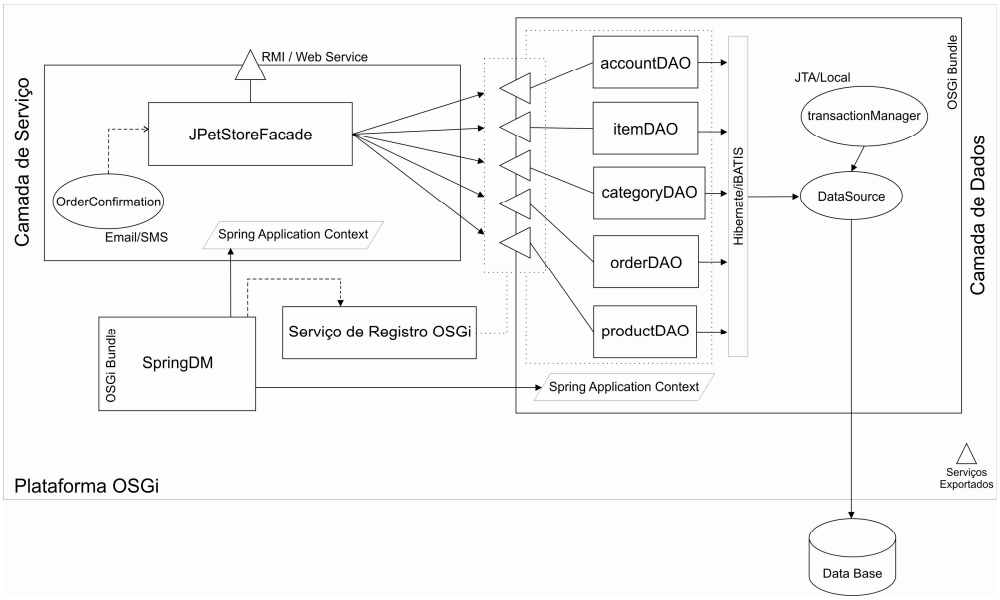


Figura 26. Arquitetura Spring/OSGi da aplicação JPetStore

A Figura 26 apresenta a arquitetura de uma aplicação Spring/OSGi. Tal aplicação é uma adaptação da aplicação JPetStore distribuída em conjunto com o Spring para um ambiente Spring/OSGi. JPetStore é uma aplicação de e-commerce que basicamente permite a consumidores à busca e compra de animais de estimação. A arquitetura desta versão esta logicamente particionada em duas camadas: (i) serviços – oferece uma fachada (RMI ou Web Service), com diversas funcionalidades, que pode ser acessada por uma camada superior; e (ii) dados – provê diversos DAOs (Alur et al. 2003), exportados como serviços OSGi, e diferentes tipos de gerenciadores objeto-relacional (Hibernate ou iBATIS) e transacionais (Local, JTA).

Cada camada de tal aplicação é implementada como um *bundle* OSGi. O *bundle* que implementa a camada de dados apresenta um Spring *application context* que contém diversos *beans* que fornecem um modelo e funcionalidades para acesso e armazenamento de informações em banco de dados. Cinco desses *beans* foram disponibilizados como serviços OSGi (accountDAO, itemDAO, categoryDAO, orderDAO e productDAO). Além disso, tal *application context* oferece dois outros tipos de *beans*, com diferentes implementações alternativas: (i) um tipo responsável por definir o mapeamento entre DAOs e o banco de dados, sendo uma versão implementada com Hibernate e a outra implementada com iBATIS; e (ii) outro tipo que habilita a conexão de DAOs com o banco de dados via *datasource*, sendo uma versão preparada para trabalhar

com transações locais e outra habilitada para trabalhar com transações distribuídas via Java Transaction API JTA (JAT 2008).

O *bundle* que implementa a camada de serviços provê um Spring *application context* que contém somente dois beans internos: (i) um *bean* que implementa a fachada e a expõe como um serviço OSGI com duas opções de acesso remoto (RMI ou Web Service), e (ii) um *bean* opcional que realiza a confirmação de compras, através de dois serviços de comunicação alternativos (Email ou SMS).

Por essa descrição, pode-se ver, que a aplicação JPetStore satisfaz a uma enorme variedade de requisitos e possui um grande número de variabilidades e dependências complexas que precisam ser customizadas durante a derivação de uma versão específica. Essa característica torna o processo de construção de um produto, não trivial e sujeito a erros. Considere, por exemplo, uma versão onde a aplicação JPetStore usa: (i) a tecnologia de banco de dados MySql; (ii) o framework de mapeamento objeto-relacional iBATIS; (iii) o gerenciamento de transação local; (iv) com seus serviços exportados usando uma interface RMI e (v) notificação das compras efetuadas através do envio de emails. Sem a ajuda de uma ferramenta de derivação automatizada, o engenheiro de aplicação precisaria configurar, manualmente, 13 linhas no arquivo *aplication context* da camada de serviços e 33 linhas no arquivo da camada de dados, e empacotar 19 elementos de implementação no arquivo .jar de instalação do *bundle* que implementa a camada de serviços e 22 elementos de implementação no arquivo .jar de instalação do *bundle* que implementa a camada de dados.

Este cenário mostra que é evidente a necessidade de uma ferramenta que automatize o processo de derivação de aplicações com características similares a apresentada. A próxima seção mostra como a ferramenta GenArch foi estendida para endereçar a derivação automática de aplicações corporativas implementadas com a combinação das tecnologias Spring e OSGi.

5.3. Arquitetura da Extensão

A abordagem definida pela ferramenta GenArch tem como base três passos principais: (i) geração parcial e automática dos modelos; (ii) sincronização entre artefatos e modelos; e (iii) derivação de produtos. As próximas subseções apresentam como se deu a extensão de cada um desses passos para que a ferramenta enderece a modelagem e derivação de produtos

Spring/OSGi. Primeiramente, a seção 5.3.1 apresenta como os modelos existentes foram estendidos para suportar as abstrações definidas pelas tecnologias Spring e OSGi. Posteriormente, a seção 5.3.2 detalha como artefatos do Spring e do OSGi podem ser automaticamente processados pala ferramenta para endereçar a criação e sincronização automática do modelos. Finalmente, a seção 5.3.3 ilustra como produtos baseados na tecnologia Spring e OSGi podem ser derivados.

5.3.1. Estendendo os modelos GenArch para suportar Spring e OSGi

As tecnologias Spring e OSGi foram integradas na ferramenta, através de duas extensões principais: (i) incorporação da abstração Spring *Bean* no modelo de implementação; e (ii) criação de uma visão de *deployment* que habilita a especificação da estrutura de cada um dos *bundles* que compõe o núcleo de artefatos. É importante frisar que só foram incorporadas duas novas abstrações: Spring *Bean* e *Bundle* OSGi. A base de implementação do núcleo de artefatos continua sendo formada por classes, interfaces, aspectos e arquivos extras.

O framework Spring demanda, para cada aplicação, a especificação de um arquivo XML que descreve todos os *beans* que a compõe. Já a plataforma OSGi demanda, para cada *bundle*, a especificação de um arquivo que descreve as propriedades de tal *bundle*. Dessa forma, durante o processo de derivação, se torna necessário a customização de tais arquivos, de acordo com a configuração dos elementos de implementação que farão parte do produto final.

Como foi dito, o modelo de implementação da ferramenta GenArch foi estendido para incorporar informações referentes aos *beans* do Spring implementados. Nessa nova versão do modelo, cada classe Java pode ser acrescida com um elemento *Bean*. Esse novo elemento contém informações relativas ao *bean* Spring que tal classe implementa. Dentre o conjunto de informações necessárias para descrever um *bean* Spring, o elemento *Bean* somente suporta a descrição: (i) do nome do *bean*; (ii) dos *beans* que são requeridos/usados pelo bean sendo especificado e que fazem parte do mesmo *application context;* e (iii) dos *beans* exportados como serviços OSGi requeridos. A criação de um elemento *Bean* no modelo de implementação permite, indiretamente, através das classes associadas à *beans*, definir o mapeamento entre *beans* e características no modelo de configuração. Baseada nos mapeamentos e nas informações descritas no modelo de implementação, a

ferramenta consegue escolher quais *beans* irão compor um produto, e, automaticamente, customizar um arquivo de configuração específico para tal produto.

Para permitir a customização de *bundles*, uma nova visão foi adicionada a ferramenta. Nesta nova visão, o engenheiro de domínio consegue especificar a estrutura de implementação dos *bundles* que compõem a aplicação. Para cada *bundle* deve se especificar: (i) o conteúdo de implementação do *bundle* – componentes com respectivos elementos e outros recursos (pastas, arquivos, figuras); (ii) o arquivo MANIFEST.MF que contém as informações de *deployment*; e (iii) se o *bundle* depende de outros *bundles* existentes, uma lista de *bundles* requeridos também pode ser especificada. Esta visão permite a ferramenta executar a customização em dois níveis, decidindo: (i) quais recursos (classes, arquivos, etc) formam o *bundle*; e (ii) quais bundles irão compor o produto derivado. O modelo de configuração foi acrescido com a nova visão de *deployment*, que permite mapear *bundles* para características. Esse mapeamento permite a ferramenta decidir quais *bundles* serão derivados. Os elementos que compõem um *bundle* são escolhidos, indiretamente, pelos mapeamentos entre elementos de implementação e características, definidos na visão antiga do modelo de configuração.

5.3.2. Construção automática dos modelos a partir de artefatos Spring e OSGi

A criação automática das abstrações *Beans* e *Bundles* nos modelos de implementação e *deployment*, respectivamente, também é realizada a partir dos artefatos de implementação presentes no núcleo. A criação de *Beans* no modelo de implementação é baseada nas anotações `@SpringBean`, `@LocalRef` e `@OSGiRef`. A Tabela **4** mostra os três novos tipos de anotações oferecidas nesta extensão: (i) `@SpringBean` – essa anotação é utilizada para indicar que uma classe implementa um *Bean* em particular, cujo nome deve ser especificado na propriedade `name`; (ii) `@LocalRef` – esta anotação é utilizada para indicar que o *Bean* implementado pela classe anotada, possui uma referência local (mesmo *application context*) para o *Bean* indicado na propriedade `name`; e `@OSGiRef` - esta anotação é utilizada para indicar que o *Bean* implementado pela classe anotada, possui uma referência para um *Bean*, cujo nome deve ser especificado

na propriedade `name`, exportado como um serviço OSGi pelo *Bundle* indicado na propriedade `bundleName`.

Tabela 4. Anotações GenArch para Spring/OSGi e seus atributos

@SpringBean	
Atributos	
Name	Nome do Bean Spring implementado
@LocalRef	
Atributos	
Name	Nome do Bean referenciado
@OSGiRef	
Atributos	
Name	Nome do Bean referenciado
BundleName	Nome do Bundle que exporta o Bean referenciado

A Figura 27 apresenta uma classe Java marcada com as anotações `@SpringBean`, `@LocalRef` e `@OSGiRef`. Cada anotação `@SpringBean` demanda a criação de um elemento *Bean* no modelo de implementação. Tal elemento é criado como um filho do elemento *Class* que representa a classe anotada. Cada anotação `@LocalRef` demanda a criação de um sub-elemento *Local Reference* que indica o uso de um *Bean* presente no mesmo contexto de aplicação do elemento *Bean* pai.

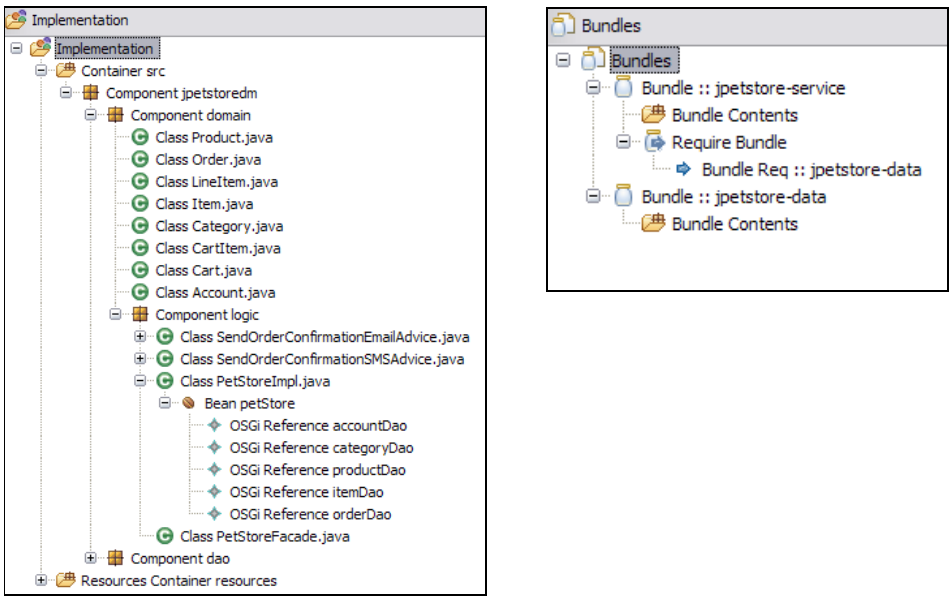
```
@SpringBean(name="petStore")
@OSGiRefs({@OSGiRef(name="accountDao",bundleName="jpetstore-data"),
                @OSGiRef(name="categoryDao",bundleName="jpetstore-data"),
                @OSGiRef(name="productDao",bundleName="jpetstore-data"),
                @OSGiRef(name="itemDao",bundleName="jpetstore-data"),
                @OSGiRef(name="orderDao",bundleName="jpetstore-data")})
public class PetStoreImpl implements  PetStoreFacade {
        private AccountDao accountDao;
        private CategoryDao categoryDao;
        private ProductDao productDao;
        private ItemDao itemDao;
        private OrderDao orderDao;

        ...
}
```

Figura 27. Classe Java anotada

As anotações do tipo `@OSGiRef` demandam a criação de um sub-elemento OSGi Reference que é uma referência para um *Bean* exportado como serviço OSGi por um determinado *Bundle*. Toda essa descrição é utilizada pela ferramenta GenArch para decidir quais *Beans* precisam ser derivados e também, indiretamente, quais *Bundles* também precisam ser. Nesse último caso, se um *Bean* depende de um serviço exportado é necessário que o *Bundle* que o implementa também seja derivado.

A criação de um elemento *Bundle* na visão de *deployment* é feita a partir de arquivos *MANIFEST*. Durante a criação da estrutura de descrição dos Bundles na visão de *deployment*, a ferramenta lê alguns campos do arquivo de propriedades *MANIFEST* e extrai informações, tais como: nome do *bundle*, *bundles* requeridos e pacotes exportados. Do campo `Bundle-Name`, a ferramenta retira o nome do *bundle*. O campo `Required-Bundle` é usado para criar a lista dos *bundles* requeridos. E, finalmente, o campo `Export-Package` permite a criação inicial do conteúdo de implementação. A criação automática do conteúdo de implementação não é completamente realizada, principalmente, porque não existe ainda, uma maneira fácil de mapear elementos de implementação para *Bundles*. Por isso, após a criação dessa visão, o engenheiro de domínio deve refiná-la, incluindo novos elementos ao conteúdo de implementação.

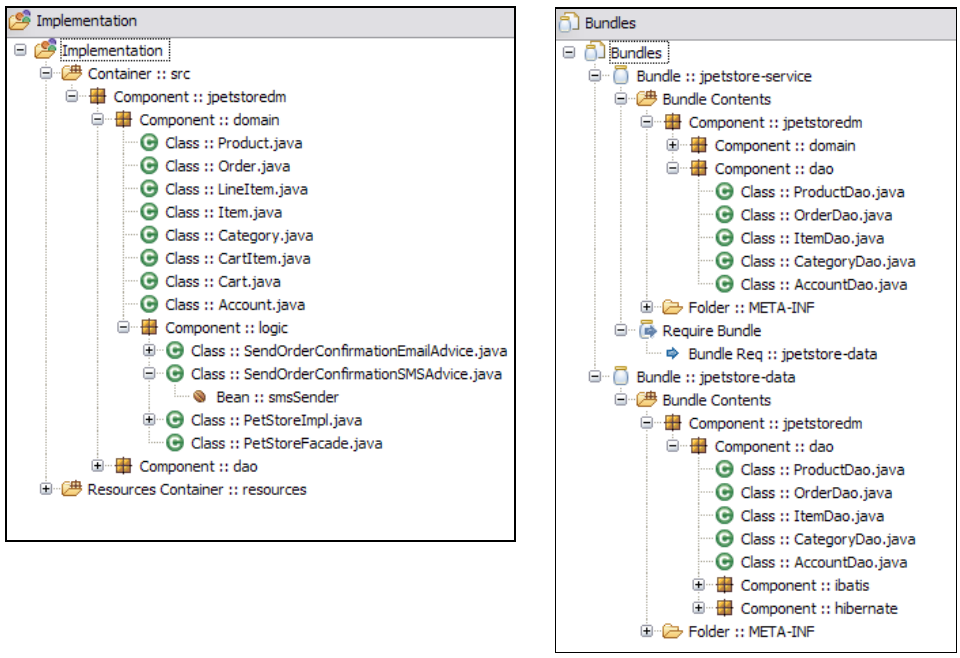
A Figura 28 apresenta a versão inicial dos modelos de implementação e *deployment* da aplicação JPetStore, criada automaticamente a partir dos elementos de implementação e anotações. Cada elemento de implementação da aplicação foi convertido para a sua respectiva abstração no modelo de implementação (Figura 28(a)). O elemento *Bean* `petStore`, relacionado ao elemento Class `PetStoreImpl`, mostrada na Figura 28(a), foi criado a partir da anotação `@SpringBean` marcada na classe `PetStoreImpl` (Figura 27). A lista de *Beans* referenciados pelo Bean `petStore` foi criada a partir das anotações `@OSGiRef` também marcadas na classe `PetStoreImpl` (Figura 27). A versão inicial do modelo de *deployment*, apresentada na Figura 28 (b), foi criada a partir de uma versão incompleta do arquivo MANIFEST de cada um dos Bundles.



(a) Modelo de Implementação (b) Modelo de *Deployment*

Figura 28. Versão inicial dos modelos de implementação e *deployment* da aplicação JPetStore

A Figura 29 apresenta a versão final dos modelos de implementação e *deployment* da aplicação JPetStore. O modelo de implementação, detalhado na Figura 29(a), não sofreu nenhuma modificação. No modelo de *deployment,* a grande modificação ficou no conteúdo dos *Bundles* (`jpetstore-service` e `jpetstore-data`). Para cada um dos *Bundles*, diversos elementos de implementação foram adicionados à propriedade `Bundle Contents`. No Bundle `jpetstore-service` foram adicionados os elementos que implementam a fachada, o serviço de notificação de compra (Email e SMS) e as interfaces dos DAOs (Alur et al. 2003) referenciados. No *Bundle* `jpetstore-data` foram adicionados as interfaces dos DAOs e as respectivas implementação usando iBatis e Hibernate.



(a) Modelo de Implementação (b) Modelo de *Deployment*

Figura 29. Versão final dos modelos de derivação da aplicação JPetStore

O modelo de *deployment* permite mapear *Bundles* para características. A versão atual do JPetStore possui apenas dois *Bundles* e ambos são obrigatórios. Dessa forma, não foram definidos mapeamentos explícitos entre características e elementos do modelo de *deployment*, por ser desnecesssário. A

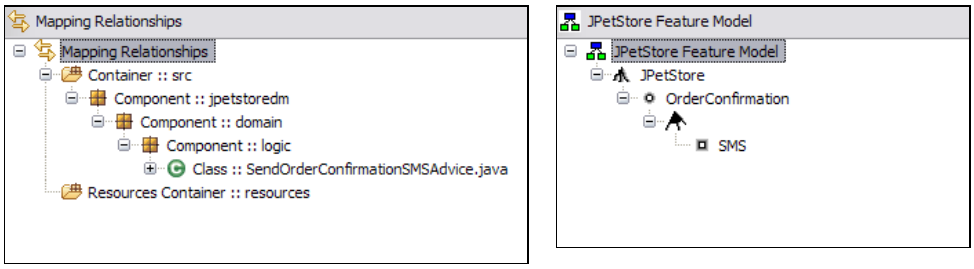
Figura 30(a) mostra o modelo de configuração gerado a partir de anotações `@Feature` presentes nos elementos de implementação da aplicação JPetStore. Somente os elementos que implementam as características **SMS** e **Email** foram mapeados nessa versão inicial. A

Figura 30(b) mostra o conteúdo do modelo de características que também foi criado a partir de anotações `@Feature`. As demais características, apresentada na versão final do modelo de configuração,

Figura 31(b), são implementadas por um conjunto de classes, presentes em um único pacote (**Hibernate** e **iBatis**), ou são implementadas por Beans Spring (**RMI**, **WebService**, **Email**, **JTA**, **Local**), por isso, não puderam ser criadas automaticamente. O modelo de configuração,

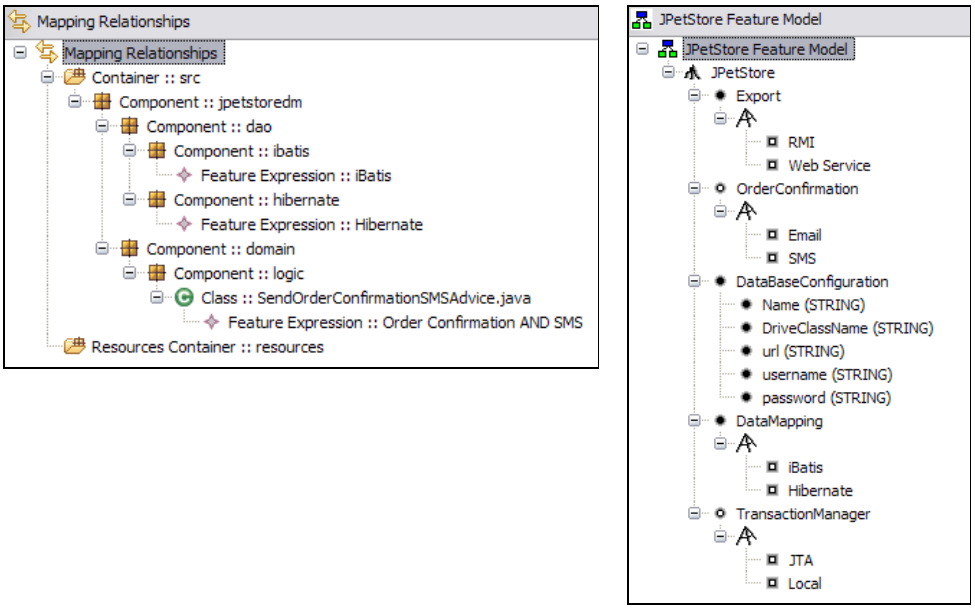
Figura 31(a), foi acrescido com dois novos mapeamentos, referentes aos pacotes que implementam as estratégias de mapeamento Objeto-Relacional (`hibernate` e `ibatis`). Como não é possível anotar pacotes, esses

mapeamentos não foram resolvidos automaticamente durante a criação da versão inicial do modelo de configuração.



(a) Modelo de Configuração (b) Modelo de Características

Figura 30. Versão inicial dos modelos de configuração e características



(a) Modelo de Configuração (b) Modelo de Características

Figura 31. Versão final dos modelos de configuração e características

5.3.3. Derivando produtos Spring/OSGi

O primeiro passo do processo de derivação especificado pela ferramenta GenArch é composto pela seleção e configuração de características em uma instância do modelo de característica. A partir da instância do modelo de

características criada, a ferramenta é capaz de customizar um produto selecionando quais elementos de implementação derivar e da geração de código através de templates. A derivação de um produto Spring/OSGi, em particular, se dá através da seleção de quais elementos de implementação devem ser instanciados e da customização de arquivos descritores (Spring *application context* e OSGi *MANIFEST*).

Em um arquivo de *application context* do Spring, pode-se customizar as propriedades que descrevem as dependências de um *Bean* e os *Beans* que formarão a aplicação. Cada arquivo de configuração deve ser implementado como um template XPand para que seja possível, através das informações presentes no modelo de características, customizar partes mais específicas, como a descrição de propriedades de um Bean. A seleção de quais Beans serão descritos no arquivo de configuração é feita, após a geração do template. A ferramenta GenArch, primeiramente, faz um *parse* do arquivo XML gerado para uma estrutura DOM (Holzner 2000). A partir da estrutura DOM, a ferramenta retira as tags relativas aos Beans que não serão derivadas e então, a estrutura é novamente gravada no arquivo XML. Essa estratégia evita a necessidade de se definir uma seleção para cada *Bean* que implementa uma variabilidade. Isso permite a construção de templates mais simples, com menos código, de fácil manutenção e com crescimento ordenado.

A Figura 32 contém o código-fonte parcial do template do arquivo de configuração Spring relativo ao Bundle da camada de serviços. O Bean `mailSender` (linhas 7-9) implementa a característica de notificação por email das compras do usuário. Tal código só será derivado (fará parte do arquivo final gerado) se a característica **Email** estiver selecionada na instância do modelo de características em uso. Essa seleção é feita após a geração do template e toma como base o mapeamento entre a classe `SendOrderConfirmationEmailAdvice` e a característica **Email**. O `order-rmi` (linhas 26-33) implementa a característica de exportação da fachada como uma interface RMI. A seleção desse *Bean* se dá através dos mecanismos oferecidos pela linguagem de template XPand. Primeiramente uma instância da característica **RMI** é obtida da instância do modelo de características (linha 24). O status da seleção de tal característica é verificado (linha 25). Se a característica foi selecionada, o código da linha 26-33 é adicionado ao arquivo que será gerado, se não, o código não é adicionado.

```
01. «IMPORT br::pucrio::inf::les::genarch::models::models»
02. «EXTENSION br::pucrio::inf::les::genarch::models::extension»
03. «DEFINE Main FOR Instance»
04. «FILE "jpetstore-service-context.xml"»
05. <beans xmlns="http://www.springframework.org/schema/beans"
06.     ...
07.     <bean id="mailSender"
08.     class=" jpetstoredm.logic.SendOrderConfirmationEmailAdvice">
09.     </bean>
10.
11.     <bean id="petStore"
12.     class=" jpetstoredm.domain.logic.PetStoreImpl">
13.     <property name="accountDao">
14.             <osgi:reference interface="jpetstoredm.dao.accountDao"
15.             timeout="5000"/>
16.     </property>
17.     <property name="categoryDao">
18.             <osgi:reference interface="jpetstoredm.dao.categoryDao"
19.             timeout="5000"/>
22.     </property>
21.     ...
22.     </bean>
23.     ...
24.     «LET feature("RMI",featureModelInstance) AS rmiFeature»
25.     «IF rmiFeature.isSelected »
26.     <bean id="order-rmi"
27.             class="org.springframework.remoting.rmi.RmiServiceExporter">
28.     <property name="service" ref="petStore"/>
29.     <property name="serviceInterface"
30.                     value="jpetstoredm.domain.logic.PetStoreImpl"/>
31.     <property name="serviceName" value="order"/>
32.     <property name="registryPort" value="1099"/>
33.     </bean>
34.     «ENDIF»
35.     «ENDLET»
36. </beans>
37. «ENDFILE»
38. «ENDFOREACH»
39. «ENDDEFINE»
```

Figura 32. Template de um arquivo de configuração Spring

A seleção e customização de *Bundles* também são feitas a partir de uma instância do modelo de características. Para cada *bundle* derivado, a ferramenta executa as seguintes atividades: (i) cria um projeto Eclipse do tipo plug-in; (ii) copia o conteúdo do *bundle*, especificado na visão de *deployment*, para o projeto criado; e (iii) finalmente, customiza os campos `Require-Bundle` e `Export-Package` do arquivo MANIFEST. A customização do arquivo MANIFEST é realizada, através da API *Properties*, oferecida pelo Java e que possui diversas funcionalidades que facilitam a leitura e escrita de arquivos de propriedades. A ferramenta GenArch faz uso dessa API para customizar os campos `Require-Bundle` e `Export-Package`. Essa customização se dá através da informação presente na descrição da estrutura do Bundle no modelo de *deployment*. O campo `Required-Bundle` é customizado a partir da lista de *Bundles* requeridos e o campo `Export-Package` é customizado a partir dos pacotes descritos no conteúdo do *Bundle*. O campo *Required-Bundle* é preenchido com os *Bundles* requeridos derivados. O campo *Export-Package* é preenchido com

todos os pacotes que foram derivados. É responsabilidade do engenheiro de aplicação remover os pacotes que não devem ser exportados.

Este trabalho utiliza a ferramenta ANT (Hatcher 2002) com o objetivo de automatizar e procurar eliminar erros durante o processo de *deployment* dos componentes derivados. O uso do ANT é uma funcionalidade fora do escopo da ferramenta GenArch. Então, para cada produto derivado, a ferramenta gera um script ANT responsável por fazer o *deployment* dos bundles que implementa tal produto. Cada script é gerado a partir de um único template XPand, que utiliza as informações presentes no modelo de implementação e na instância do modelo de características para customizar seu conteúdo. A ferramenta percorre os elementos da instância do modelo de características e cria, baseado na seleção das características e nas relações de mapeamento, um subconjunto do modelo de implementação. Esse subconjunto contém somente os elementos que serão derivados. Nesse caso, tanto o modelo de implementação quanto o modelo de característica estão sendo utilizados como DSLs de configuração

O arquivo gerado será responsável por criar arquivos jars que empacotam cada um dos *bundles* e fazer uma cópia destes no diretório de instalação do container OSGi. O conteúdo de cada arquivo jar é extraído do projeto Eclipse que contém os elementos de implementação do respectivo *Bundle*.

5.4. Conclusões e Resultados

Este capítulo apresentou uma extensão da ferramenta GenArch que endereça a derivação de produtos implementados com as tecnologias Spring e OSGi. Essa extensão promoveu o acréscimo de novas abstrações no modelo de implementação que permitiram um maior poder de customização a ferramenta. Com a configuração feita através da ferramenta pode-se ter todas as partes comuns e variáveis de arquivos descritores codificadas em somente um único arquivo. Utilizando uma implementação específica, o número de linhas criadas para permitir a customização dos arquivos de configuração Spring foram 16. Após a derivação não foi necessária nenhuma alteração ou acréscimo nos arquivos de configuração Spring gerados. A versão do sistema JPetStore, implementada pelo Spring, contém 3 arquivos de configuração, sendo que esses arquivos representam somente algumas das diversas configurações possíveis da aplicação. Nessa versão, onde somente algumas funcionalidades variam, temos 18 linhas de código replicadas e as 38 linhas que precisam ser configuradas

manualmente. Essa replicação de código dificulta a manutenção e o gerenciamento do sistema. A configuração manual é sujeita a erros e pode provocar diversos problemas durante o processo de *deployment*.

Em relação à *Bundles*, a partir do estudo de caso executado, dá para se observar que o número de linhas que precisam ser configuradas nos arquivos *MANIFEST*, após a derivação, tende a ser inferior ao número que seria necessário sem o uso da ferramenta. Em relação à seleção de quais elementos de implementação devem ser derivados, é evidente, pelos estudos de casos apresentados (Capítulo 4 e JPetStore), que o uso da ferramenta é interessante também nesse cenário. O uso da ferramenta ANT para o *deployment* dos *Bundles* derivados também se mostrou um recurso interessante, principalmente, porque essa ferramenta utiliza arquivos de script que podem ser facilmente customizados pela ferramenta, como mostrado na Seção 5.3.3.

Da forma como a customização de arquivos descritores foi implementada, os mapeamentos entre determinados *Beans* e características não são representados no modelo de configuração. Tais mapeamentos são feitos diretamente nos arquivos descritores. Isso provocou uma perda de visão do conhecimento de configuração, ou seja, está mais complicado descobrir onde uma determinada característica afeta a arquitetura de implementação da LPS. Uma solução seria a incorporação de tais fragmentos de código no modelo de implementação, de forma que esses sejam diretamente mapeados para características no modelo de configuração. Durante o processo de derivação, tais fragmentos somente seriam incorporados ao arquivo de configuração se as seleções feitas na instância do modelo de características validassem os respectivos mapeamentos no modelo de configuração.

O trabalho apresentado nesse capítulo também mostrou ser interessante a flexibilização do núcleo da ferramenta de forma que este aceite facilmente o acréscimo de novos modelos. Esse novo recurso possibilitaria a derivação de arquiteturas de LPS mais complexas e implementadas por diferentes tecnologias e abstrações, como apresentado nesse capítulo. Essa flexibilização se daria a partir da criação de pontos de extensão que aceitariam o registro de meta-modelos EMF e de processadores, que podem ser acoplados antes e após o processo de derivação.