AUTOMATIC DERIVATION OF SPRING-OSGI BASED WEB ENTERPRISE APPLICATIONS

[1]Elder Cirilo, [2]Uirá Kuleza and [1]Carlos Lucena
[1]*Computer Science Department, Pontifical Catholic University of Rio de Janeiro - PUC-Rio, Brazil*
[2]*Computer Science Department, Federal University of Rio Grande do Norte – UFRN, Brazil*

Keywords: Component-based Technologies, Software Product Lines, Product Derivation.

Abstract: Component-based technologies (CBTs) are nowadays widely adopted in the development of different kinds of applications. They provide functionalities to facilitate the management of the application components and their different configurations. Spring and OSGi are two relevant examples of CBTs in the mainstream scenario. In this paper, we explore the use of Spring/OSGi technologies in the context of automatic product derivation. We illustrated through a typical web-based enterprise application: (i) how different models of a feature-based product derivation tool can be automatically generated based on the configuration files of Spring and OSGi, and Java annotations; and (ii) how the different abstractions provided by these CBTs can be related to a feature model with the aim to automatically derive an Spring/OSGi based application or product line.

1 INTRODUCTION

Component-based technologies (CBTs) are nowadays widely adopted in the development of different kinds of enterprise applications. They enable the management – assembling, adapting and connecting – of the application components and their different configurations.

Over the last years, two important CBTs have been adopted by the Java development community: Spring frameworks (http://www.springframework.org) and OSGi (http://www.osgi.org). These CBTs provide specific mechanisms and abstractions to specify and manage the system components. Spring framework (http://www.springframework.org) allows defining an application as a set of components implemented as simple classes. It provides ways: (i) to link the application components by means of the dependency injection mechanism (Johnson, 2002); and (ii) to extend their base functionality using aspect-oriented techniques. On the other hand, the OSGi specifies the application components as a set of Bundles. Each Bundle aggregates a set of classes, interfaces, aspects and files that implement the services provided by that component. One of the main benefits of OSGi is to provide an environment to dynamically install, deploy and remove Bundles. Recently, the Spring Dynamics Modules (SDM) (http://www.springframework.org/osgi) has been proposed with the aim to integrate both Spring and OSGI technologies. Spring DM enables the development of Spring applications that can be deployed in an OSGi execution environment,

Despite the benefits that CBTs can bring to the development and management of components in enterprise applications, it also contributes to improve their complexity due to: (i) the big amount of new code assets (configuration files, annotations) that applications need to deal with: and (ii) the need to specify and configure each of the application components and their respective relationships. Additionally, the increasing demand for producing flexible and customizable enterprise applications or software product lines (Clements and Northrop, 2001) in order to address different market segments also contributes significantly to bring difficulties to this scenario.

Modern software engineering approaches motivate the use of model-driven techniques (Stahl and Voelter, 2006) to help both the processes of development and automatic instantiation of customizable application and software product lines. Domain-specific languages (Czarnecki et al., 2000) and code generators (Czarnecki et al., 2000) are the main technologies that contribute to address these aims. While these approaches can clearly bring benefits to the development of enterprise application and software product lines, there is no systematic method or technique that show how mainstream CBTs can take advantage of them.

In this context, this paper presents an approach for automatic derivation of enterprise applications or software product lines implemented using the Spring and OSGi technologies. We extend the GenArch, a model-based product derivation tool previously proposed (Cirilo et al., 2007), to incorporate functionalities that allow automatically instantiate Spring/OSGi enterprise based applications developed over the SDM platform.

The remainder of this paper is organized as follows. Section 2 presents the new mainstream CBSE technologies Spring and OGSi, and how this two technologies work together in the SDM. Section 3 describes an overview of GenArch, a model-based product derivation tool developed by our research group. Section 4 describes the extensions proposed to the GenArch tool in order to address Spring-OSGi model-based applications. And, finally, on the section 5 we present our conclusions.

2 NEW MAINSTREAM CBSE TECHNOLOGIES

In this section we summarize Spring and OSGi component technologies, showing, their differences and how they are integrated in the context of Spring DM. We also point out the advantages and disadvantages that the union of these component technologies can bring for feature modularization and variability management.

2.1 Spring Framework

Spring (http://www.springframework.org) is an open-source framework created to address the complexity of Java enterprise application development. Spring enables the development through use of components based on POJOs (*Plain Old Java Objects*), where each POJO contains only *business logic*. The Spring framework is responsible for addressing the additional features (transaction, security, logging, etc), thus incrementing the base functionality provided by POJOs and needed to build enterprise applications.

Spring makes it possible to use a simple component model to achieve things that were previously only possible with complex component models like Enterprise Java Beans (Burke and Monson, 2006). In this way, server-side application, like web information systems, can benefit from Spring in terms of simplicity, testability, and loose coupling. These benefits is reached by the inversion of control principle (Johnson, 2002) (IoC) and aspect-oriented container provided by the Spring framework.

Spring framework uses a XML configuration file to specify the application components metadata and the dependency between then. This file is called application context in the Spring terminology, and is the primary unit of modularity of one Spring application. It contains one or more Bean definitions which typically specify the class that implements the Bean, the Bean properties and the respective Bean dependencies to be injected. Additionally, this configuration file also defines which aspects will be applied to each Bean of the application.

2.2 OSGi

The OSGi specification (http://www.osgi.org) defines a framework that facilitates modularizing Java applications into smaller and more manageable pieces (Bundles). It defines a standardized module packaging, life-cycle management and service registration. A Bundle is deployed as a “plain” JAR file that contains, besides other resources (e.g., classes, aspects, pictures), a MANIFEST file which have some specific metadata. This information must includes a symbolic name for the Bundle and, optionally, can have the location of a activator class which is called when the Bundle is started or stopped, exported package, package and Bundle dependencies, and general information about the Bundle. Each Bundle represents an application module that also can exports one or more services to the end-user or other Bundles. The exported services are registered in a specific OSGi service registry, that expose this services for other Bundles to discover and to use. A newly formed OSGi Enterprise Expert Group is a initiative to introduce OSGi on the server side. This group is looking for extend the OSGi specification to support the needs of Enterprise Java vendors and developers, such as: distributed and extended service model, enterprise life-cycle and configuration management.

2.3 Integrating Spring and OSGi for SPL Architecture Implementation

In the context of SPL, the use of Spring and OSGi implementation model can bring several advances for enterprise application and SPL’s architecture development process. In one hand, The Spring dependence injection and AOP container provides a flexible development approach in which optional and alternatives features can be easily addressed by components and aspect-oriented composition. On the other hand, the OSGi module system can be used to separate the application features on a single and consistency Bundle, to promote a better separation of concerns and to make easy the evolution and maintenance of the SPL's architecture.

The Spring DM is an initiative in the direction of combine Spring and OSGi component model. It makes easy to write Spring applications that can be deployed in an OSGi execution environment, in such way that applications can take advantage of the services offered by Spring and the dynamic environment provide by OSGi platform.

By the Spring DM implementation model, a Bundle may contain a Spring application context, in charge to describe and configure its Beans. Some of these Beans may optionally be made public and exported as OSGi services and thus made available to end-user or other Bundles. Beans within the Bundle may also be transparently injected with references to OSGi services.

We have implemented some features of a Web Shop application as Spring Beans and modularized them in four OSGi Bundles: (i) `eshop` – core implementation of the Web Shop application; (ii) `eshop.payment` – modularizes a set of payment methods and resources related to the payment process; (iii) `eshop.customer` – modularizes the customer service; (iv) `eshop.shipment` – encompasses services to calculate taxes and resolve the shipment process. Some Beans that realize different application features (payment services, shipment services and customer services) were made public in order to be used by the eshop Bundle, whose implements the application core features. Spring DM was used to structure our application in order to enable us to combine OSGi Bundles to form different versions (customization). From the viewpoint of SPL the Web Shop application satisfies a large amount of variabilites (18 optional and alternative features) and complex dependencies between them.

The SPL derivation process demands the resolving of all feature dependencies in both problem space and solution space in order to guarantee the reliability of the derived product. Looking for the derivation of the Web Shop case study, if we desire a product with registered checkout feature, it also must have the register customers feature. Looking for this constraint on solution space, the components that implement the registered checkout feature depend, indirectly, on the components that implement the customer services. To derive a product from the Web Shop application the application engineer may configure, by hand, four Spring application context and MANIFEST files, one of these for each Bundle, resolving the dependencies and configuration. Moreover, he needs to know how to separate the SPL's implementations elements in their respective Bundles.

Without the aid of an automated tool, is evident that the number of variabilites, dependencies between features, and the amount of configuration files and code assets become the manually process of derive a Spring/OSGi product time consume and non trivial. Thus, it is clear that we need appropriated mechanism for enable automatic product derivation of Spring/OSGi applications and product lines.

In this context, section 4 details an extension of the GenArch tool specific for automate product derivation and variability management of Spring/OSGi based applications and product lines. Firstly, in next section, we give an overview of our model-based product derivation approach.

3 GENARCH – A MODEL-BASED PRODUCT DERIVATION TOOL

GenArch (Cirilo et al., 2007) is a model-based tool that enables the mainstream software developer community to use the concepts and foundations of the SPL approach in the product derivation process (Czarnecki and Eisenecker, 2000). The main idea is to provide conveniences for developers create a set of simple derivation models from existing implementation assets made available during the SPL development. In this section, we give an overview of the derivation approach implemented by GenArch tool.

3.1 Approach Overview

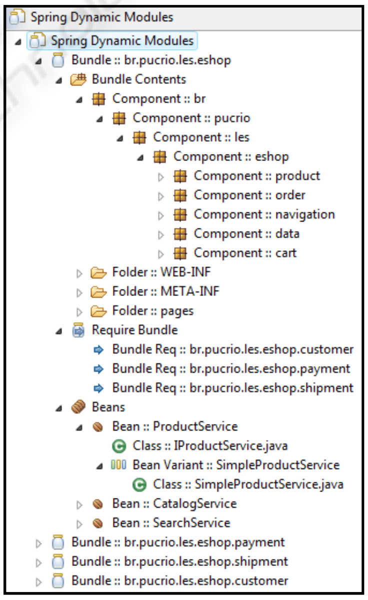
GenArch implements a model-based software product line derivation approach founded on generative programming (Czarnecki and Eisenecker, 2000). Based on three main steps: (i) automatic models construction; (ii) artifacts synchronizations; and (iii) product derivation; our approach offers a code-oriented variability management which supports automatic product derivation. The variability management in GenArch is accomplished by three models: (i) implementation model (solution space); (ii) feature model (problem space); and (iii) configuration model (configuration knowledge).

The implementation model defines a visual representation of the SPL implementation elements (classes, aspects, templates, configuration and extra files) in order to relate them to feature models. Feature models (Czarnecki and Eisenecker, 2000) are used in our approach to represent the variabilities that exist in SPL architectures. The configuration model is responsible to define the mapping between features and implementation elements. It represents the configuration knowledge from the generative programming (Czarnecki and Eisenecker, 2000), being fundamental to link the problem space (features) to the solution space (implementation elements).

The approach steps were implemented in GenArch through three modules: (i) importing module; (ii) synchronizer module; and (ii) derivation module. The importing module enables the creation of initial versions of the derivation models by parsing the code assets that implement the SPL architecture. Initially, in the importing process, the domain engineers are responsible to annotate the existing code (classes, interfaces and aspects) of SPL architectures using GenArch annotations. Two kinds of annotations are supported by our tool `@Feature` and `@Variability`.

During the importing process, the GenArch tool parses all the implementation elements from specific directories, including all the annotations created inside the implementation elements, and it generates initial versions of the derivation models. In this parsing step, each `@Feature` annotation encountered demands the creation of: (i) a new feature in the feature model; and (ii) a mapping relationship between the feature created and the respective implementation element annotated, in the configuration model. The GenArch tool also generates code templates based on the `@Variability` annotation. Code templates are used to implement variabilities which will be customized based on information collected by models instances during application engineering. Every code template created by our tool is included in the product line implementation model and a dependency relationship with a feature is created in the configuration model. The feature related with the code template is specified directly in the `@Variability` annotation. After the generation of the initial versions of GenArch derivation models, the domain engineer can refine them by including, modifying or removing any feature, implementation element or mapping relationship. The GenArch tool defines a specific module, called synchronizer, to keep the consistency between its derivation models and the code assets that implements the SPL architecture. The synchronizer is responsible to observe changes in the code assets and automatically reflects these changes to the models.

The derivation module is responsible to produce an instance of the derivation models and customize the requested product. In the first step of the derivation process the application engineer must choose and configure features by means of the specification of a feature model configuration. The customization and compositions of the SPL architecture are driven by this feature model configuration, in a next step. GenArch perform this step by deciding which implementation elements must to be instantiated to constitute the product and by customizing classes, aspects or configuration files. Each element that must be customized is represented by a template. Each template uses information collected by the instance of the derivation models to customize its respective variable parts of code. The derivation process in GenArch is concluded with code generation based on the processing of templates. Both selected and generated code assets are loaded in a specific source folder of a specified Eclipse Java project. The complete algorithm used by GenArch tool can be found in (Cirilo et al., 2007) .



```
Spring Dynamic Modules
Spring Dynamic Modules
  Bundle :: br.pucrio.les.eshop
    Bundle Contents
      Component :: br
        Component :: pucrio
          Component :: les
            Component :: eshop
              Component :: product
              Component :: order
              Component :: navigation
              Component :: data
              Component :: cart
      Folder :: WEB-INF
      Folder :: META-INF
      Folder :: pages
    Require Bundle
      Bundle Req :: br.pucrio.les.eshop.customer
      Bundle Req :: br.pucrio.les.eshop.payment
      Bundle Req :: br.pucrio.les.eshop.shipment
    Beans
      Bean :: ProductService
        Class :: IProductService.java
        Bean Variant :: SimpleProductService
          Class :: SimpleProductService.java
      Bean :: CatalogService
      Bean :: SearchService
  Bundle :: br.pucrio.les.eshop.payment
  Bundle :: br.pucrio.les.eshop.shipment
  Bundle :: br.pucrio.les.eshop.customer
```

Figure 1: EShop Spring-OSGi architecture model.

4 INTEGRATING SPRING/OSGI IN THE GENARCH TOOL

Section 3 illustrated how SPL architectures based on the adoption of typical code assets (classes, interfaces, aspects and extra files) can be addressed by GenArch tool. In this section, we describe how our tool was extended to address the derivation of Spring/OSGi based enterprise applications.

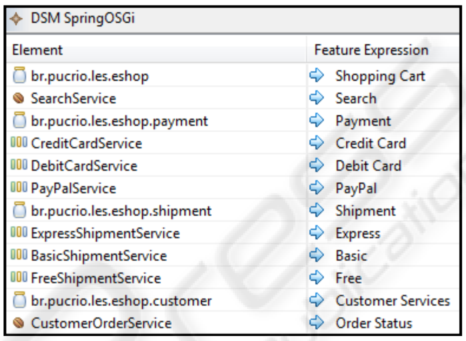
4.1 Extending GenArch Tool to Support Spring/OSGi

In order to enable automatic product derivation of Spring/OSGi applications, we extended the GenArch tool to incorporate a Spring-OSGi-specific architecture model (Figure 1). This model provides: (i) a vision of the SPL architecture implementation in the level of Spring/OSGi components; and (ii) a concise way to document and trace Spring/OSGi architecture level variability. Tracing features in SPL development at different levels can help engineers to better analyze the features covering and the impact of changing requirements. The use of Spring-OSGi architecture models also allows GenArch working not only with Java and AspectJ elements, as described in Section 3, but also with Beans, Bundles and the respective Spring and OSGi configuration files in the derivation process.

The Spring-OSGi architecture model (Figure 2) expresses OSGi Bundles by means of their name, contents and dependency Bundles. The Bundle's content encompasses components, classes, folders, files and configuration files. If the Bundle depends on other existing ones, the list of required Bundles can also be specified. This last property is essential for GenArch figures out which Bundles need to be part of the application under customization. Spring Beans are expressed by their variants, where each variant is identified by a name and can be related with other dependent Beans. GenArch uses the concept of Bean Variant to distinguish between different implementations of an interface.

Figure 1 shows the Spring-OSGi architecture model instantiated for the EShop application. It encompasses the four Bundles that modularize this application. For the `eshop` Bundle, for example, the model describes: (i) the packages (components) and folder that implement it; (ii) the required Bundles; and (iii) the Spring Beans that are modularized by the Bundle.

The Spring/OSGi extension also allows the domain engineer to define different levels of configuration. Fine-grained configurations can be created by the default mapping relationships of specific implementation elements (classes, aspects, GIF files) to any product line feature (Cirilo et al., 2007). Cross-grained mapping relationships of both Spring Beans and OSGi Bundles to product line features can be defined by domain engineers in a new view into the configuration model (Figure 2). Figure 2 shows, for example, that the `eshop.payment` Bundle depends on the Payment optional feature and that the `CreditCardService`, `DebitCardService`, and `PayPalService` payment services Bundles depend on, respectively, the Credit Card, Debit Card and PayPal alternative features.



Element	Feature Expression
br.pucrio.les.eshop	Shopping Cart
SearchService	Search
br.pucrio.les.eshop.payment	Payment
CreditCardService	Credit Card
DebitCardService	Debit Card
PayPalService	PayPal
br.pucrio.les.eshop.shipment	Shipment
ExpressShipmentService	Express
BasicShipmentService	Basic
FreeShipmentService	Free
br.pucrio.les.eshop.customer	Customer Services
CustomerOrderService	Order Status

Figure 2: Spring-OSGi configuration knowledge.

This knowledge is used during the derivation process and enables GenArch to customize applications at different levels, such as: (i) the definition of resources (classes, files, etc) that compose the Bundles, specified on the traditional view; and (ii) the definition of Bundles and Beans that will be part of the final application. In section 4.3, we describe how GenArch use them to automatically derivate customized products.

4.2 Automatic Parsing of Spring/OSGi Applications

Once the main target of GenArch approach is to provide functionalities for parsing code assets metadata in order to automatic generate an initial version of the derivation models. We have extended this mechanism to also accomplish parsing of both Spring/OSGi specific code assets (Spring application context and OSGi Manifest configuration files) and Spring-specific Java annotations.

We created a new Java annotation, called `@SpringBean`. It is used to specify the Bean name, a version name, and the Bundle name. Thus, based on this metadata, Spring Bean abstractions can be created in the Spring-OSGi architecture model by parsing the Java classes marked with this annotation. It follows the same approach described in Sction 3. The use of this annotation also informs for GenArch that the annotated class is the realization of a specific Bean. So, based on this information, GenArch can also infer mapping between Beans and class elements.

Each MANIFEST file in the project demands the creation of a Bundle element in the Spring-OSGi architecture model. Each subelement of the Bundle abstraction is created by parsing the `Bundle-Name`, `Require-Bundle`, and `Export-Package` properties. The content of the `Export-Package` is used by GenArch to infer the relationship between Bundles and implementation elements (package, in this case). The `Require-Bundle` provides the list of dependency Bundles. And, the `Bundle-name` is used for both identify the Bundle element in the Spring-OSGi architecture model and to name the respective jar file during the derivation process.

After the creation of the initial version of the derivation models, the domain engineers must verify the adhesion of the generated models with the SPL specification, and if necessary they can perform, by hand, some activities such as: inclusion, moving or exclusion of any model's element and mapping relationship between them.

4.3 Automatic Derivation of Spring/OSGi Applications

Besides the tasks that are realized during the common derivation process described in Section 3.1, we incorporate new ones that are specific to customize Spring/OSGi artifacts: (i) spring configuration files; and (ii) OSGi MANIFEST files.

Each Spring configuration file (application context) must be defined as a template in our approach. During the product derivation process, this template is processed to customize its respective variabilities – XML tags that describe the Beans of the SPL architecture. Spring applications context are processed in two main steps: (i) decision of Beans that will compose the final application; and (ii) customization of the respective Bean's tags – it means that the code of a Bean tag, whose Bean class depends on specific features, is only included in the Spring configuration file of a product, if the corresponding feature is selected.

During the derivation process, the GenArch tool also decides based on the feature selection, which Bundles will compose the final product. For each Bundle to be included in the product being generated, the GenArch tool proceeds in the following way: (i) it creates an Eclipse plug-in project; (ii) it loads the selected implementation elements and template generated elements in this Eclipse project; and finally, (iii) it customizes the `Require-Bundle` and `Export-Package` fields in the OSGi manifest file. Different from the previous versions of GenArch, which demands the derivation of only one Eclipse Java project, the OSGi extension generates one Eclipse project per Bundle.

5 CONCLUDING REMARKS AND FUTURE WORKS

In this paper, we presented an extension to GenArch, a model-based tool, that addresses the product derivation and automatic instantiation of product lines or customizable applications implemented using the mechanisms available in Spring and OSGi. Automatic mechanisms are used to generate partial version of these models based on the specific artifacts (configuration and manifest files) and Java annotations.

As future work, we are interested: (i) in studying the incorporation in our approach of domain-specific languages that models behavioral semantics - currently, our approach is only involved with the definition of languages that capture static semantics of the requirements and features of the SPL architecture; (ii) enable specific architecture models composition. Finally, we intend to apply the tool in more complex SPL case studies.

REFERENCES

Cirilo, Elder, Kulesza, Uirá and Lucena, Carlos. A Product Derivation Tool Based on Model-Driven Techniques and Annotations. *Journal of Universal Computer Science.* 2007, Vol. 14, 8, pp. 1344-1367.

Czarnecki, Krzysztof and Eisenecker, Ulrich. *Generative Programming: Methods, Tools, and Applications.* s.l. : Adisson-Wesley, 2000. 0201309777.

Burke, Bill and Monson-Haefel, Richard. *Enterprise JavaBeans 3.0.* s.l. : O'Reilly Media, Inc., 2006. 059600978X.

OSGi. [Online] http://www.osgi.org.

Spring Framework. [Online] http://www.spring framework.org.

Spring Dynamic Modules. [Online] http://www.spring framework.org/osgi.

Stahl, Thomas and Voelter, Markus. *Model-Driven Software Development: Technology, Engineering, Management* . s.l. : Wiley, 2006. 0470025700.

Johnson, Rod. *Expert One-on-One J2EE Design and Development (Programmer to Programmer).* s.l. : Wrox, 2002. 0764543857.

Clements, Paul and Northrop, Linda. *Software Product Lines: Practices and Patterns.* s.l. : Addison-Wesley Professional, 2001. 0201703327.