PUC-Rio - Certificação Digital Nº 0812626/CA

4 GenArch$^+$: Building Software Product Lines with Domain Knowledge Modeling Languages

4.1 Tool Architecture Overview

GenArch$^+$ is a model-based tool that supports the engineering of framework-based software product lines (Cirilo et al. 2009, Cirilo et al. 2011a). This tool implements the ideas discussed in previous Chapter. Figure 4.1 presents an architectural overview of the tool.

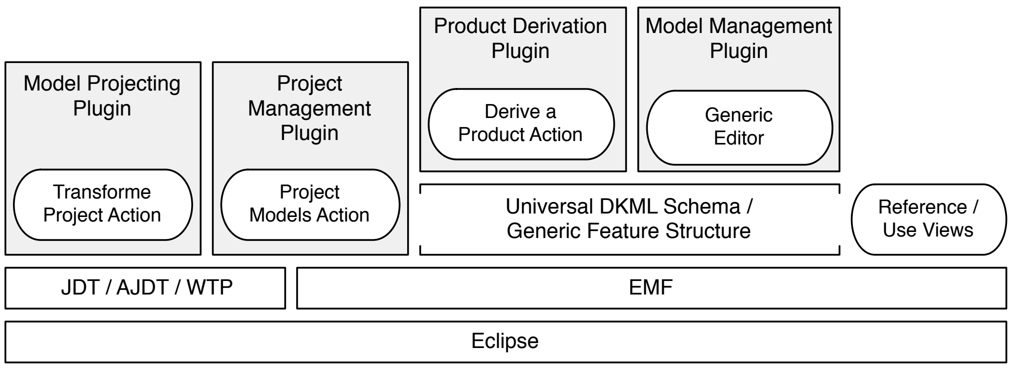


Figure 4.1: GenArch$^+$ architectural overview.

GenArch$^+$ models build on Eclipse Modeling Framework (EMF) (Budinsky et al. 2003). EMF is a eclipse plugin for defining model-based tools using structured data models. EMF provides some general functionalities, such as typed references, many-to-many relationships, and standard cross-model reference mechanism that facilitates our implementation. It also provides some functionalities that automatically generates a set of Java classes to generically manipulate and persist models.

The feature model is implemented by the Feature Modeling Plugin (FMP) (Antkiewicz and Czarnecki 2004). This plug-in allows modeling the feature model proposed by Czarnecki et al. (Czarnecki and Eisenecker 2000), which supports mandatory, optional, and alternative features, and their respective cardinality. However, GenArch$^+$ is agnostic of one specific feature modeling technology. To date GenArch$^+$ implements internally its own feature model structure, which supports to plug any modeling technology that can be mapped to the provided representation.

GenArch$^+$ contributes two Eclipse views: Reference View and Usage View. The Reference View is an automated support for a deep analysis of concept instances association. The Usage view shows which source-code elements are using a given concept instance. GenArch$^+$ also provides a generic Editor of DKMLs. Basically the editor supports uniform editing and analysis of DKMs. These two views are activated via the generic Editor of DMKLs. Therefore, as we will present next, in Section 4.2, GenArch$^+$ is also a generic DKML infrastructure, that is, the tool is also a framework for implementing DKMLs.

The tool also contributes with three actions: Transform a Existing Java Project to GenArch$^+$ Project; Project Models from Source-code; and Derive a Product. They are structured over three plugins: (i) project management; (ii) models projection; (iii) product derivation, respectively. The project management is a plugin that supports flexible management of different types of Eclipse projects. To date, GenArch$^+$ provides support for handling Java projects. The project derivation plugin is in charge of deriving a product form the product line project in a new eclipse project. The derivation is performed by removing from the elements in the new project the elements to which assigned feature expression is evaluated to false. As mentioned, the knowledge needed to perform this task is specified in the models. Finally, the model projection plugin is in charge of parsing meta-data defined in DKML's abstract syntax and projecting the elements defined in the source-code which matches to those meta-data patterns by creating domain knowledge model elements. This module is implementation language agnostic, that is, it supports pluggable mapping parsers that implement interpreters for a given asset type. Each mapping interpreter is implemented in a separated Eclipse plug-in.

The derivation model and projection model also use different Eclipse APIs. The mapping interpreter uses Eclipse Java Development Tools (JDT) API and it relies on the parser, abstract syntax trees API (Shavor et al. 2003). The XML mapping interpreter uses Eclipse Web Tools Plataform (WTP) API form handling XML files. The remaining modules handles both plain Java as well as AspectJ projects.

4.1.1 Domain Knowledge Schema

The basic requirements for the successful use of our proposed technique are that DKMLs must be easy to be built and composed with each other

PUC-Rio - Certificação Digital Nº 0812626/CA

PUC-Rio - Certificação Digital Nº 0812626/CA

without being detrimental to their domain-specific capabilities. Our solution to achieve these requirements is the conception of a product line implementation infrastructure based on a domain-independent schema and parametric polymorphism. Following we present the key characteristics of our solution.

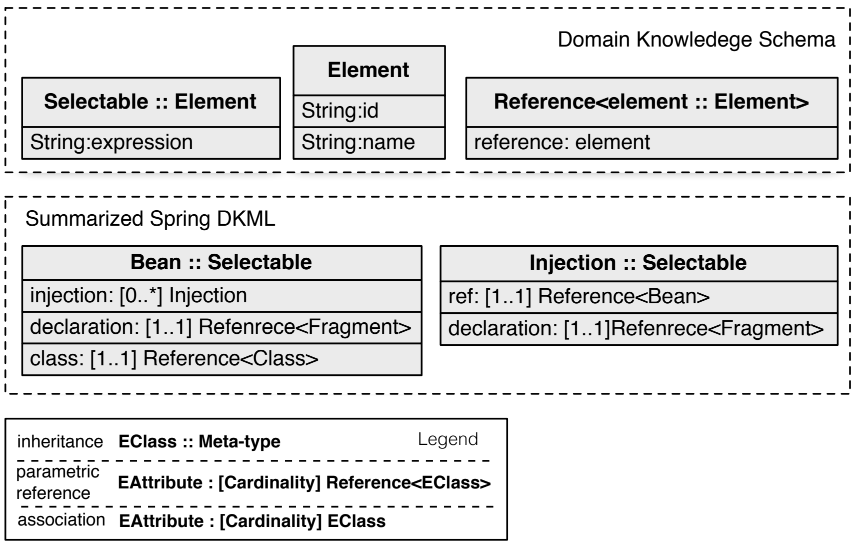

Domain Knowledege Schema
Element
String:id
String:name
Selectable :: Element
String:expression
Reference<element :: Element>
reference: element
Summarized Spring DKML
Bean :: Selectable
injection: [0..*] Injection
declaration: [1..1] Refenrece<Fragment>
class: [1..1] Reference<Class>
Injection :: Selectable
ref: [1..1] Reference<Bean>
declaration: [1..1]Refenrece<Fragment>
Legend
inheritance EClass :: Meta-type
parametric reference EAttribute : [Cardinality] Reference<EClass>
association EAttribute : [Cardinality] EClass

Figure 4.2: Domain Knowledge Schema

GenArch$^+$ DKML construction method consists of two distinct layers of semantics definition (see Figure 4.2). The first layer, domain knowledge schema, is a defined minimal set of concepts that serves to give the second layer, DKMLs, a configuration foundation. This way, DKMLs are given unambiguous definitions, and can be interpreted by the tool. DKML is the only language to be manipulated directly by the developer. The domain knowledge schema is used by the meta-modelers. They can determine the universe of DKMLs as composed of domain concepts (*Element*), where some ones are accountable to have those instances present in every DKM, others can have those instances optionally excluded (*Selectable*). Finally, model elements are also likely to be related to other elements (*Reference*).

DKMLs are constructed via inheritance (see Figure 4.2). Every DKML concept representing a framework-provided concept must be a sub-type of *Element*. When a concept represents a valid optional concept it needs to be a sub-type of *Selectable*. In this case, the tool enables the mapping between concept instances and feature expressions through `expression` property. The *Reference* element is used to denote all types of relations between framework concepts and their mapping to source-code elements. Typed references are captured via parametric polymorphism. For example, the `ref` property of the Injection concept only assumes a reference to model elements that are instances of the Bean concept. It allows the infrastructure to interpret the

DKML abstract syntax in domain-specific ways and guide the developer when creating the configuration knowledge (see Section 3.4). Therefore, in spite of its universality, the proposed schema is powerful enough to express the domain-specific knowledge associated with heterogenous application frameworks.

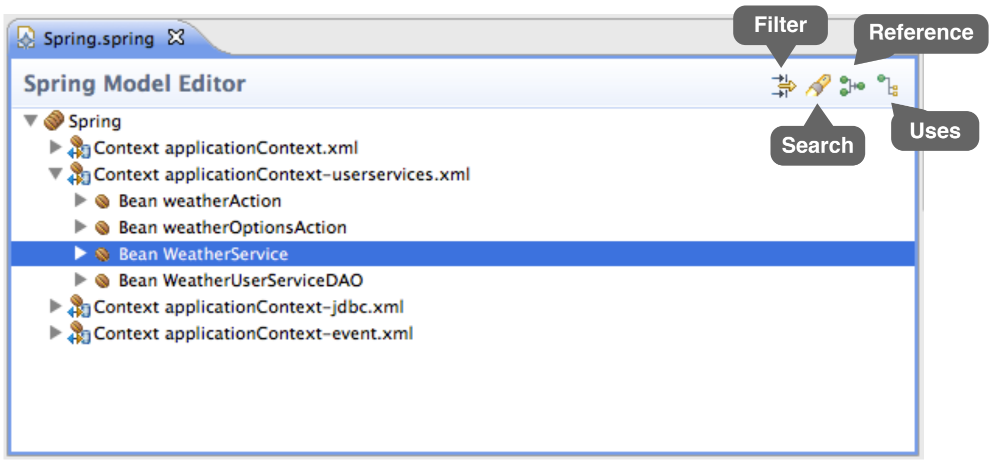
4.2 DKML Editor and Views

GenArch$^+$ provides a reflective DKML editor that exploits the Domain Knowledge Schema and DKMLs abstract syntax. It is a tree-based editor implemented from Eclipse EMF editors (see Figure 4.3). It enables developers to access and modify domain knowledge models of different DKMLs in a uniform way. The editor uses reflective capabilities of the EMF object model in order to structure the user interface and properties view. When the developer selects an element, three different piece of information about it are exhibited in the properties view:

– **Basic** - Only basic properties information are exhibited in this tab, such as the element's name.

– **Feature Mapping** - In this tab is exhibit to which features the element is assigned to. This can be a boolean feature expression.

– **Code Mapping** - Here is exhibit to which source code elements the element is mapped to. Every mapping refers to one element from the implementation model, as mentioned in Section 3.2.

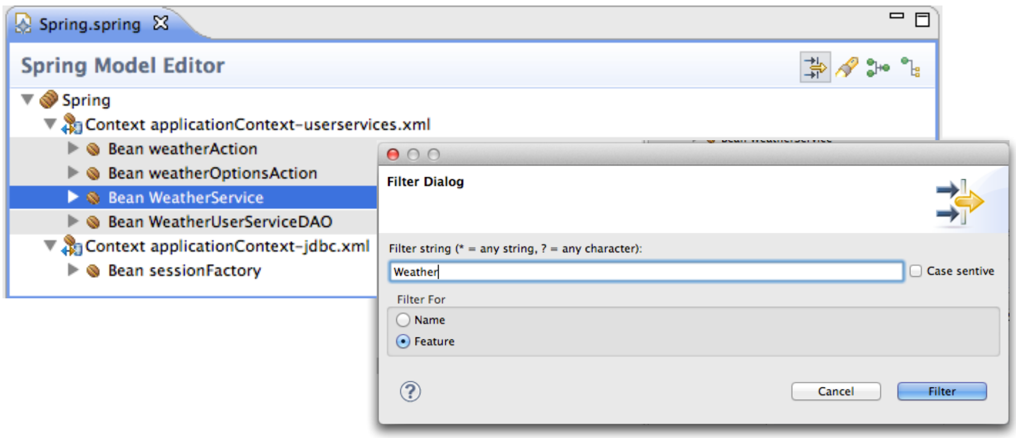


Spring
Context applicationContext.xml
Context applicationContext-userservices.xml
Bean weatherAction
Bean weatherOptionsAction
Bean WeatherService
Bean WeatherUserServiceDAO
Context applicationContext-jdbc.xml
Context applicationContext-event.xml

Figure 4.3: GenArch$^+$ reflective DKML editor.

The editor also provides some advanced generic capabilities:

PUC-Rio - Certificação Digital Nº 0812626/CA

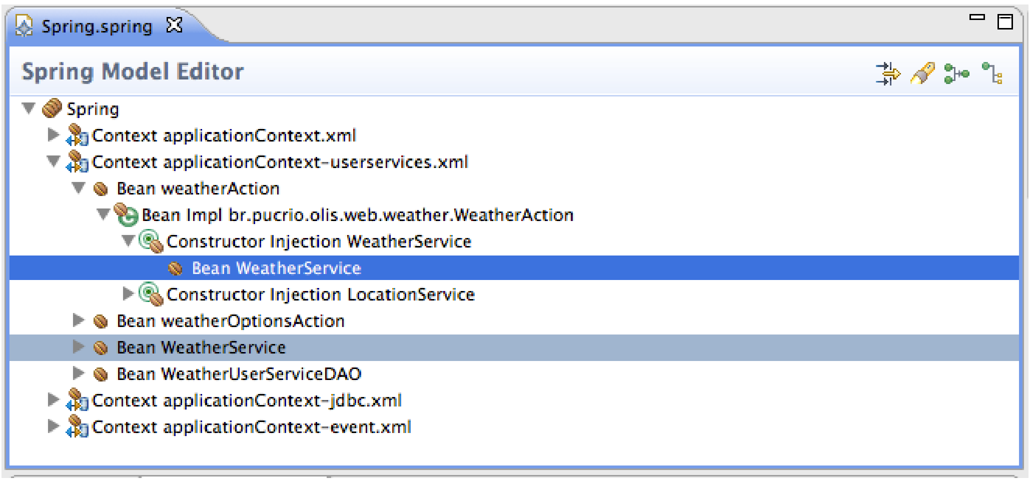


```
▼ Spring
  ▼ Context applicationContext-userservices.xml
    ▶ Bean weatherAction
    ▶ Bean weatherOptionsAction
    ▶ Bean WeatherService
    ▶ Bean WeatherUserServiceDAO
  ▼ Context applicationContext-jdbc.xml
    ▶ Bean sessionFactory
```

Filter string (* = any string, ? = any character): Weather
☐ Case sentive
Filter For
○ Name
◉ Feature

Figure 4.4: Filtered exemplar Spring-DKML

Visualizing on Feature. Visualizing on feature shows the elements in domain knowledge models or implementation model of one or more features. In this resumed visualization developer can quickly identify features assignment. There are basically two kinds of visualizing on feature implemented in the editor. First, the editor can only show model elements assigned to one or more feature (and some necessary context) and hide everything else. This corresponds to the Filter functionality (see Figure 4.4). Second, the editor can still show the entire tree, but it highlights the model elements assigned to the desired feature. This operation corresponds to the Search functionality. Observe that it provides some form of modularity.



```
▼ Spring
  ▶ Context applicationContext.xml
  ▼ Context applicationContext-userservices.xml
    ▼ Bean weatherAction
      ▼ Bean Impl br.pucrio.olis.web.weather.WeatherAction
        ▼ Constructor Injection WeatherService
            Bean WeatherService
        ▶ Constructor Injection LocationService
    ▶ Bean weatherOptionsAction
    ▶ Bean WeatherService
    ▶ Bean WeatherUserServiceDAO
  ▶ Context applicationContext-jdbc.xml
  ▶ Context applicationContext-event.xml
```

Figure 4.5: `WeatherService` occurrence marked.

PUC-Rio - Certificação Digital Nº 0812626/CA

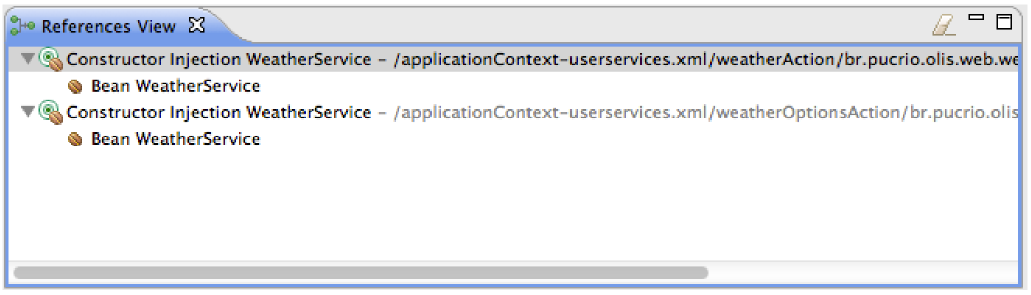

References View
Constructor Injection WeatherService – /applicationContext-userservices.xml/weatherAction/br.pucrio.olis.web.we
Bean WeatherService
Constructor Injection WeatherService – /applicationContext-userservices.xml/weatherOptionsAction/br.pucrio.olis
Bean WeatherService

Figure 4.6: Reference View: references to the *Bean* `WeatherService`

Marking occurrences and Searching for Elements. As elements with the same name are common to appear in distinguished paths in the tree structure, developers can get lost when they are analyzing `references`. To overcome this, the generic editor provides a functionality that helps developers to locate model elements from selected `references` (see Figure 4.5). This marks the concrete referred element in the same domain knowledge model. Alternatively, to the mark occurrences, developers can also search over the domain-knowledge models by using the general Search functionality.

Reference View. The reference view is an automated support for developers analyzing the impact of feature assignment. When the developer selects an element and clicks on the reference button, the editor queries the underlying

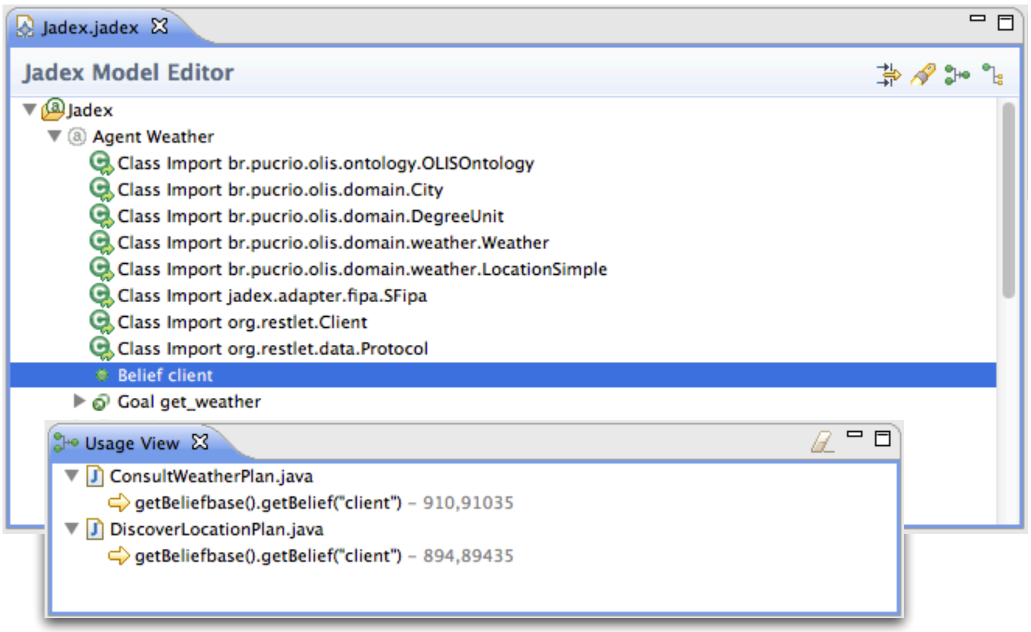

Jadex.jadex
Jadex Model Editor
Jadex
Agent Weather
Class Import br.pucrio.olis.ontology.OLISOntology
Class Import br.pucrio.olis.domain.City
Class Import br.pucrio.olis.domain.DegreeUnit
Class Import br.pucrio.olis.domain.weather.Weather
Class Import br.pucrio.olis.domain.weather.LocationSimple
Class Import jadex.adapter.fipa.SFipa
Class Import org.restlet.Client
Class Import org.restlet.data.Protocol
Belief client
Goal get_weather
Usage View
ConsultWeatherPlan.java
getBeliefbase().getBelief("client") – 910,91035
DiscoverLocationPlan.java
getBeliefbase().getBelief("client") – 894,89435

Figure 4.7: Usage View: usage of the *Belief* `client`

PUC-Rio - Certificação Digital Nº 0812626/CA

representation for all occurrence of the selected element in a `reference`. The resulting references are present in the reference view in the form of a list (see Figure 4.6). For each element of the list, is exhibit the reference's parent and its complete path in the model. A double click on elements from the list in the reference view can be used to directly navigate to concrete element in the domain knowledge model.

Usage View. The usage view is an automated support for developers navigate through the uses of a certain concept instance. When the developer selects an element and click on the uses button, the editor queries the entire source code for a pattern that identifies the use of the selected element. The pattern is generic specified in the DKML abstract syntax, as it will be described next (Section 4.3). The resulting references are present in the usage view in the form of a list (see Figure 4.7). For each element in the list, it is exhibited the source code element that encompasses the code using the concept instance and attached to this element a reference to the actual source code using the concept instance. Also the complete path of the source code element in the project is also exhibit. A double click on elements from the list can be used to directly navigate to the actual source code.

PUC-Rio - Certificação Digital Nº 0812626/CA

4.3 A Language For Reverse Engineering Domain Knowledge Models

We have defined a language that is based on the idea presented in (Antkiewicz and Czarnecki 2006) for reversing engineering domain knowledge models from existing source code. As DKMLs do not specify attributes and contain explicitly references, the strategies presented in (Antkiewicz and Czarnecki 2006) do not generally apply for DKMLs. Therefore, we have implemented a summarized version of the reverse engineering strategy proposed in (Antkiewicz and Czarnecki 2006), that therefore fits to our case. Basically, the idea is to use source code mapping patterns in order to specify the semantics of DKMLs elements.

Mapping patterns are defined for a certain asset type, such as Java or XML. That way, the DKMLs meta-data is not tied to a particular implementation language, therefore it provides support for handling new implementation languages. Mapping patterns in must of case correspond to attributes of structural elements of the source-code, but they also might correspond to behavioural code pattern in the case of code customization. Figure 4.8 presents mapping patterns for the DKML from Figure 3.2.

Mappings are associated to `classes` and `references`. One `reference`

PUC-Rio - Certificação Digital Nº 0812626/CA

Mappings Elements	Properties
Reference	
Asset	Extension, Dialect, Pattern
Member	Pattern, <Singleton>
AssetMember	Extension, Dialect, Pattern, MemberPattern
Reference	From, <Scope>
Class	
SimpleName	Pattern, <Matching-To>
QualifiedName	Pattern, <Token>, <Matching-To>

Table 4.1: Mapping Types

might correspond to an Asset, Member, AssetMember, or a Reference. Table 4.1 presents the mappings types and their respective properties. The Asset specifies that the concept instance is implemented by a source code file, for example, a Java class, a XML document, a Property File. The Member specifies that the concept instance source code is a member of Asset of its parent. For example, a method of a Java class, or tags in XML documents. Observe that in the case of code customization, the member also can be a behavioural pattern, for example a method calling. The AssetMember specifies that the concept instance is implemented by a member of a Asset different from its parent Asset. Finally, the Reference specifies a reference to a existing element in the model.

Each type is associated to a Pattern, which is concretely defines as a code query. For example, the *bean* `reference` attached to *Context* `class` is associated to a Member mapping whose Pattern is the code query *$pattern/bean*. The *$pattern* is replaced by the Pattern of the parent in the abstract syntax hierarchic. In this case, it will be replaced by */beans* value, defined in the Asset mapping attached to the *Context* `class`. The the `classes`, which type the references, must define to which attribute of the structural element that matches to the specified code query its name came from. For example, the *Bean* `class`, which types the *bean* `reference` of the previous example, defines that its name must be set with the values of the attribute `id` of the tag that matches to the Pattern *$pattern/bean*. It can refer to a simple name or even a qualified name. In case of qualified names, it must be defined a Token that enable the infrastructure stripe out the qualified name. A more specific filter can also be applied over the elements that results from the code query, which are specified by the Matching-To attribute and evaluated over a specified structural attribute value. For example, the *ConstructorParameter* `class` determines that only elements that also equally matches to its parent name are candidate to be instances. To date it is defined as a equals to a given predefined value or respecting a specific regular expression.

Observe that in some cases, in addition to specifying parameters explicitly, parameter values can also be defined implicitly using context values. The context retrieves the value for a parameter from the instance of: (i) the closes parent feature with the defined value; (ii) or from an explicitly defined reference.

4.3.1 Reverse Engineering Domain Knowledge Models

Reverse engineering creates a domain knowledge model from a given source code. Reverse engineering is driven by the annotation aggregated to the abstract syntax of a DKML. The algorithm traverses the metamodel, executes code queries for patterns in the abstract syntax, and creates instances of these `classes` for the patterns matched by the queries. After creating an instance of a `class` or `reference` in the model, mappings are established between the model and the corresponding source code element. These mapping enable navigation from the model to code. In general, there are 3 ways in which an model element is created:

1. if a `reference` does not have any mapping definitions attached, an instance of the `class` is created and its `references` are processed next.

2. if a `reference` has a mapping definition attached and the Name annotation attached to the `class` does not expresses a MatchTo, an instance of the `class` is created for each source code element that matches with the pattern.

3. if a `reference` has a mapping definition attached and the Name annotation attached to the `class` expresses an MatchTo, an instance of the `class` is created for each source code element that matches with the pattern and the Matching-To rule.

After creating model elements in one of the ways mentioned above, the algorithm evaluates mappings of the type Reference. For each `reference` associated to a Reference the algorithm queries the model to find a element which matches to the value defined in the From property. In the simplest case, the algorithm search in the entire model associate the first occurrence, however the Reference also supporting restrict the context of search. In this case, the property Context defines from which places the algorithm must search for elements containing the value specified in the From property.

PUC-Rio - Certificação Digital Nº 0812626/CA

Spring Domain Knowledge Modeling Language

```
[DKML] Spring
   <1-*> [reference] context : Context
      [mapping] Asset
         [property] Extension -> xml
         [property] Dialect -> xml
         [property] Pattern -> /beans
[class] Context : Selectable
   <1-*> [reference] bean : Bean
      [mapping] Member
         [property] Pattern -> $pattern/bean
[class] Bean : Selectable
   [mapping] SimpleName
      [property] Pattern = @id
   <1-1> [reference] interface : BeanInterface
   <1-*> [reference] implementation: BeanImplementation
[class] BeanInterface
[class] BeanImplementation : Selectable
[class] ConstructorInjection : Selectable
   <1-1> [reference] constructorParameter : ConstructorParameter
      [mapping] AssetMember
         [property] Extension -> java
         [property] Dialect -> java
         [property] Pattern -> &name.lastSegment
         [property] MemberPattern -> constructorArg
   <1-1> [reference<Bean>] bean : Bean
      [mapping] Reference
         [property] From -> $name
[class] ConstructorParameter
   [mapping] Name
      [property] Pattern -> @name
      [property] Matching-To -> ==[$name]
[class] PropertyInjection : Selectable
[class] PropertyMethod
```

Figure 4.8: Spring-DKML mappings patterns.

PUC-Rio - Certificação Digital Nº 0812626/CA

4.4 Summary

In this Chapter we completed our efforts to improve the engineering of framework-based software product lines. We presented our code-oriented technique based on DKMLs with tool support, which addresses almost all problems of traditional code-oriented techniques (see Chapter 2). The tool is built on universal domain knowledge schema that supports a generic editor of DKMLs enriched with many advanced functionalities: visualization of features, references and uses inspection, so on. Finally, we described that DKMLs elements represent code patterns in the source code and that domain knowledge model can be automatically projected by attaching such patterns definition to `classes` and `references` elements of the abstract syntax. In next Chapter we evaluate in what extends the benefits of our proposed improved code-oriented product line implementation technique.

PUC-Rio - Certificação Digital Nº 0812626/CA