2
Problem Formulation

As stated before, software product lines are usually more efficiently created when extracted from existing and similar software systems (Krueger 2001). This process involves the identification of features in existing source code (Kästner et al. 2008). Unfortunately, identifying features in existing source code of enterprise software systems might became a laborious and error-prone task (Cirilo et al. 2012), as we will present in this Chapter.

Indeed, enterprise software product lines are comprised of multiple different application domains and views (Recker et al. 2006). There are vertical domains such as accounting and inventory as well as horizontal domains such as user interfaces and messaging. Therefore, domain experts, interface designers, database experts and developers with different kinds of expertise all take part in the process of building such a software product line. Considering the important number of problem domains, there is a need for an equally important number of specialized languages. This diversity of knowledge and expertise is usually managed via a myriad of object-oriented frameworks, where each one offers domain-specific concepts and expert-specific implementation mechanism. Therefore, every participant of the development process has it own particular mean to solve problems specific to its expertise.

Two well-know and industrial-strength frameworks are Spring and Jadex. Spring delineates a service-oriented infrastructure based on the *Bean* concept. This term is used to denote the Spring-managed objects. The Spring container manages the object lifecycle by interpreting declarative data exposed in the form of XML documents, so called Spring Application Context. Such documents specify the classes and properties of the injectable objects, allowing developers to decouple the configuration and specification of dependencies from their actual program logic. Figure 2.1 illustrates a source code instantiating the Spring-provided domain concept (i.e., *Bean*) and how feature assignment occur in such context.

Jadex provides an agent-oriented development paradigm, whereby the *Agent* concept allows us to decompose software system into autonomous interacting entities with their own goals and rational manner. Jadex also

PUC-Rio - Certificação Digital Nº 0812626/CA

provides other concepts such as *Beliefs*, *Goals*, *Plans*, *Capabilities*, so on (see Figure 2.2). Programming Jadex *Agents* is done using its provided API and Agent Definition Files (ADF). The ADFs declare the initial *Beliefs*, *Goals* and *Plans* of an *Agent*. The Jadex reasoning engine reads this file to create and execute instances of *Agents*. It also observes the current values of *Goals* and *Beliefs* in order to execute *Plans* based on internal/external messages and events. *Plans* are Java classes that extends the Plan abstract class from the Jadex API. This abstract class provides methods for sending messages, dispatching goals or waiting for events. *Plans* are also able to read and alter the *Beliefs* of the *Agent* using the API of the belief base.

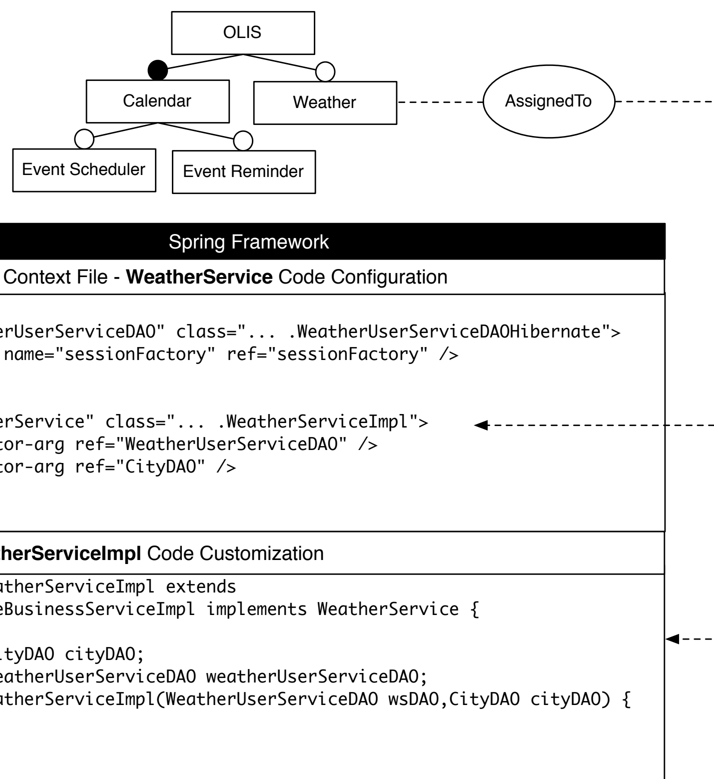

Figure 2.1: Source code instantiating Spring-provided concepts and their assignment to features.

In general, object-oriented frameworks provide convenient domain concepts and build-in generic functionalities, however they introduce a lot of complexity for identifying features in existing source code. These challenges stem from the fact that more diversity of implementation technology demands more coordination for configuration knowledge specification. We will refer to these challenges as the *heterogeneous configuration knowledge problem*. This problem has a conceptual and a technical aspect.

Conceptually, the main problem is to localize and comprehend concept instances implementing features. This happens because the instantia-

PUC-Rio - Certificação Digital Nº 0812626/CA

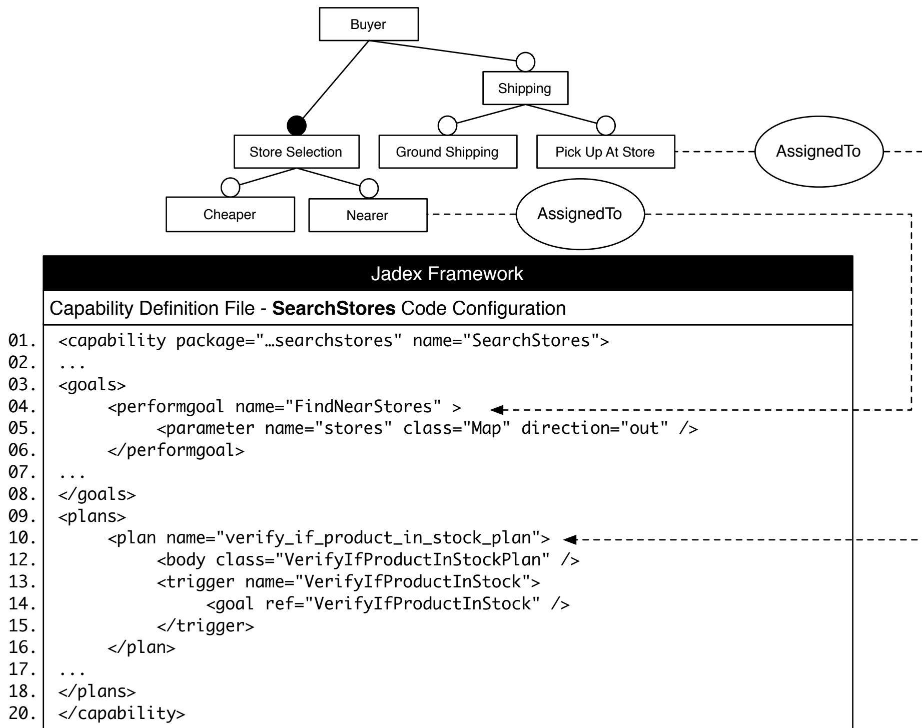

Figure 2.2: Source code instantiating Jadex-provided concepts and their assignment to features.

PUC-Rio - Certificação Digital Nº 0812626/CA

tion of a particular concept (e.g., Spring *Bean*) is often scattered across the source code, tangled with code instantiating other concepts, and hard to be modularized because their implementation in general involves code configuration and customization. One motivation for modularizing code implementing features is that developers can reason about it without being distracted with code of other features. Moreover, the high degree of scattering is considered as the main responsible for numerous problems (Garcia et al. 2005, Eaddy et al. 2008, Figueiredo et al. 2009). Consider the source code of Figure 2.1, which instantiates the *Bean* `WeatherService`. The code is scattered across the Spring application context and the `WeatherServiceImpl` Java class, in addition to be tangled with code declaring other Spring *Beans* (e.g., `WeatherUserServiceDAO`). To properly comprehend the behaviour of a feature, developers need to search the entire code based and observe more than one place, instead of just looking into a single location. For example, they need to reason about the interrelationship between XML and Java code to comprehend the code implementing the *Weather* feature.

A workaround solution to attenuate the diffusion of the configuration knowledge is the use of search (Janzen and De Volder 2003) and views (Kästner et al. 2008) mechanisms that answer the question about what code belongs to a feature. However, it still does not solve the fundamental problems

discussed above. The abstraction mismatch and diffusion remain, and require developers to spend additional time on performing non-trivial queries through the source code (Siau et al. 2004). This should not be an extra task assigned to developers who should resort only to domain-specific concepts that they are familiar with. A more straightforward solution should support high-level declarative expressions for representing the configuration knowledge, such as "The Spring *Bean* `WeatherService` implements the Feature *Weather*".

Technically, the main problem is to enforce consistency. Software product lines are inherently complex and prone to all kind of errors. Errors can be distinguished in three basic types, as discussed in (Kästner et al. 2009): syntactic errors, type errors and semantic errors. The first type occurs when a product does not respect the language's syntax in some sense. The second one refers to errors such as statements creating a class that does not exist in a product, that is, that product does not respect the language's type system. The last one occurs when a product behaves incorrectly according to some specification. When identifying features in existing source code, developers might also introduce errors regarding the framework's programming interface, as not assigning both code customization and configuration to the same feature, or not observing references between concept instances. Consider the source code in Figure 2.1. As a mandatory concern, developers must respect the reference between the *Beans* `WeatherService` and `WeatherUserService`, meaning that they must be related with coherent combination of features.

Note that in general syntax and type errors can be statically checked by language compilers or checkers. However, as discussed in (Kästner et al. 2009, Thaker et al. 2007), deriving and checking every product for consistency in separated is often infeasible because even from small product lines with a few numbers of features developers can derive thousands of products. Instead tools must be provided to detect errors early, and when it is possible, assist developers in resolving the problems. The idea is to ensure that no ill-product will be derived from a correct configuration knowledge. Even when it is feasible to derive and check every product, this problem can get worse because errors regarding the framework's programming interface frequently can be only detected at runtime. Since current techniques (Hessellund et al. 2007, Antkiewicz and Czarnecki 2006) for guaranteeing correctness of framework-based application is no general enough to express all circumstances in which feature assignment violates programming interface constraints, errors might remain undetected unless a product with a problematic feature combination is derived and executed, thus, for a long time.

In order to shed light on the frequency of the aforementioned challenges,

PUC-Rio - Certificação Digital Nº 0812626/CA

we computed some metrics over three framework-based software product lines (see Section 5): *distribution of feature over files* (DoFF), *distribution of concept instances over files* (DoCF), *number of concept instances implementing features* (NoCF), and *number of references between concept instances implementing features* (NoRF). The product lines were implemented using eight different frameworks. In some of them, such as Spring and Struts, type hierarchies are trivial. In Jadex, for example, the type hierarchies are more complex. All of them use XML documents as configuration files. Each product line realizes more than 30 features and their code size ranges from ∼4000 LOC to ∼14600 LOC.

Product Lines	OLIS		eShop		Buyer	
No. Features	7		8		7	
No. Files	270		93		30	
No. Concept Instances	1107		324		323	
No. References	242		72		47	
DoFF	135	50%	49	52%	16	53%
DoCF	167	15.08%	139	42.90%	6	1.85%
NoCF	104	9.39%	35	10.23%	20	6.19%
NoRF	44	18.18%	7	9.72%	2	4.24%

Table 2.1: Results of computed metrics.

Table 2.1 presents an overview of the results. Scattering of features over the source code does not depends much on the product line (see line DoFF) and the results reveal that a large percentage of scattering was found in all of them. For example, 50% of OLIS files contain at least a part of the 7 optional or alternative features. Therefore, $51.33\% \pm 1.52\%$ files implement part of at least one feature. Observe that we were not concerned on computing the files implementing more than one feature, that is, compute feature tangling.

The distribution of concept instance over files vary significantly across the product lines (see line DoFF). 42.90% of eShop concepts are implemented in two files. In Buyer, on the other hand, only 1.85% of concept instances are implemented in two files. According to the results, $19.94\% \pm 20.95\%$ concept instances are distributed over more than one file.

Regarding the number of concept instances implementing features, it also does not vary much from one product line to another (see line NoCF). The results reveal that $8.60\% \pm 2.13\%$ of concept instances implement features. In contrast to scattering, only a small percentage of concept instances implement features. For example, only 9.39% of the concept instances from OLIS implement features. Rather, taking only concept instances distributed over more than one file into account, the probability of they implement a feature in-

PUC-Rio - Certificação Digital Nº 0812626/CA

creases. 10.23% of eShop product line's concept instances implement features, but 65.72% of such concept instances are distributed over more than one file. The results show that $26.71\% \pm 34.53\%$ of concept instances distributed over more than one file implement features. Finally, Table 2.1 also indicates a small number of references between concept instances implementing features (see line NoRF). 18.18% of OLIS's references between concept instances implement any feature. In general, the results reveal that $10.71 \pm 1,52$ references between concept instances involve concept instances implementing a feature.

Although not representative, the metrics provide initial dimension of the problem, giving insight in how features are implemented in framework-based software product lines. We show next four scenarios that concretely illustrate the heterogeneous configuration knowledge problem. There are several occurrences of such scenarios in the framework-based product lines investigated by us. The scenarios also illustrate that the challenges aforementioned are real rather than hypothetical challenges.

2.1 Scenarios when Assigning Framework-based Software Product Lines Source-code to Features

The following scenarios reflect our experience in extracting features from four existing framework-based software product lines.

2.1.1 Configuration and Customization Code

The first scenario refers to the usual way of instantiating framework-provided concepts. The process of frameworks instantiation is in general characterized by two activities (Antkiewicz and Czarnecki 2006): concept configuration and customization. For example, creating an instance of a Spring Bean involves first, implementing its behaviour (customization) and second declaring its existence and dependencies in a Spring Application Context file (configuration) (see Figure 2.3). Therefore, to control the inclusion of the instantiated concepts in products, the developer must assign both, code configuration and customization, to the same feature.

Although the well-formedness rules that govern concept instantiation can be established by checking whether its configuration conforms to XML schemas and customization code meets the Java syntax and type system for example, those checkers do not provide any support for statically checking the integrity between them. Enforcing the existence of both code elements is a prerequisite for guaranteeing the correct behaviour of applications. For

PUC-Rio - Certificação Digital Nº 0812626/CA

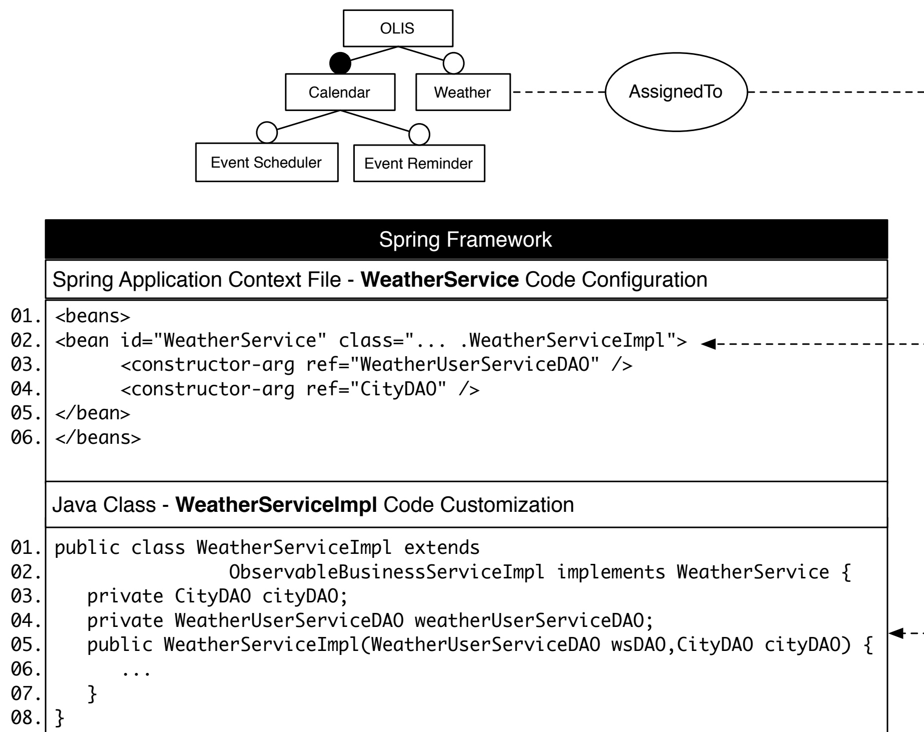

Figure 2.3: Configuration and customization Code.

example, the Spring container is responsible for instantiating, configuring, and assembling the *Beans*. The container obtains instructions about what classes to instantiate by reading the code configuration. The code configuration of the *Bean* `WeatherService` (see Figure 2.3), for example, explicitly indicates the existence of a class called `WeatherServiceImpl` that implements the code customization of the respective Spring *Bean*. However, observe that it is not uncommon for developers accidentally forget to assign the *Weather* feature to `WeatherService` code configuration. In this case, even products without *Weather* feature still containing the `WeatherService` configuration, but they are not containing the `WeatherServiceImpl` class. As the exemplified cross-reference is not evaluated until the `WeatherService` is selected to be instantiated at the execution time, the non-existence of the missing `WeatherServiceImpl` class will be only detect later, at running time.

Observe that typed references would offer only a partial solution to the problem since the knowledge about references can be hidden inside the framework logic. Spring provides two major variants of dependency injection, constructor-based and setter-based. Our focus here is the constructor-based variant. In this variant, dependency injection is resolved by the container invoking a constructor with a number of arguments, each one referring to a dependency. The order in which the *Construct-arg* concepts (see Figure 2.4) are defined in code configuration is in general the order in which those arguments

PUC-Rio - Certificação Digital Nº 0812626/CA

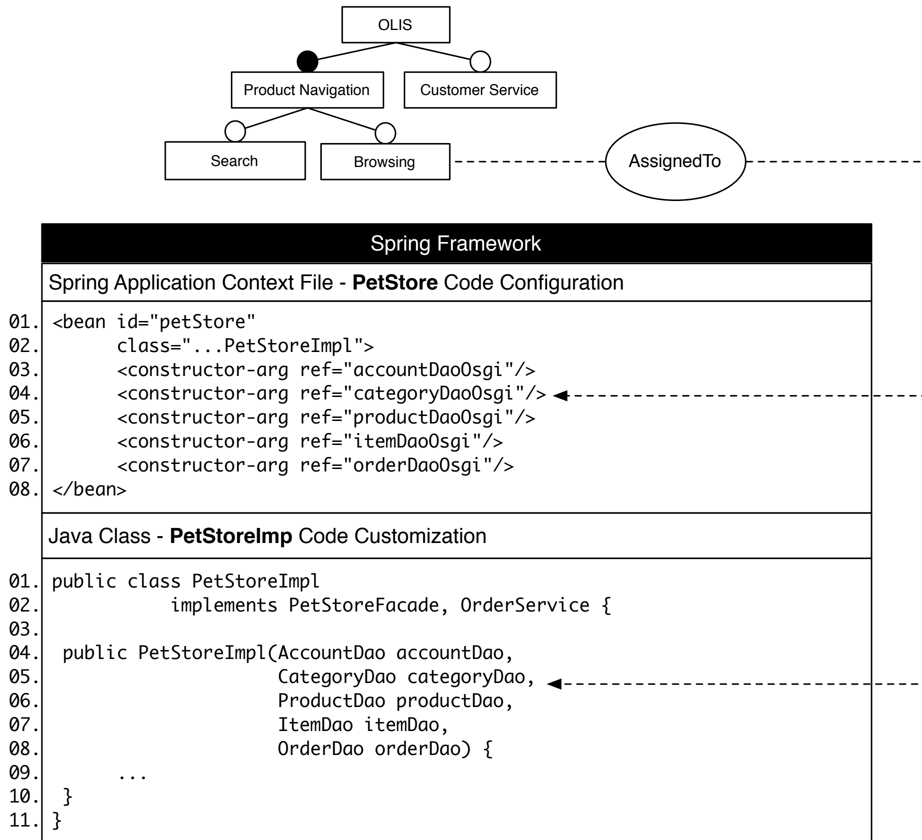

Figure 2.4: Spring constructor-based dependency injection.

are supplied to the construct when the *Bean* is being instantiated. In this case, every *Constructs-arg* and its respective constructor argument must be annotated with features in accordance. Figure 2.4 illustrates the usual way to define construct-based dependency injection.

Suppose that in the example shown in Figure 2.4 the developer did not map the constructor argument `CategoryDAO categoryDAO` to *Browsing* feature. Observe that in this case the code is well-formed for all products that contain the feature *Browsing*. However, in products without *Browsing* the order of parameter assignment changes, leading to a mismatching among the `constructor-arg` and their respective constructor parameters. Therefore, the product will fail at running time because the Spring probably will assign to constructor parameters values of incompatible type.

Due to the many lines of code and features in enterprise software product lines and the complexity associated with the framework's programming interface, the developers probably can get confused or even not properly analyse the source code and introduce such a category of error. To check the entire product line, we need to ensure that code configuration can reach a code customization in every product, for example. Moreover, in addition to have the

PUC-Rio - Certificação Digital Nº 0812626/CA

detection of error only at runtime we also have the problem of visualizing and comprehending feature implementation. The code of a particular concept is often scattered across the base source code and tangled with code configuration of other concepts.

2.1.2 Cross-references between Concept Instances

As mentioned, well-formedness of the concept instances in terms of syntax and type system can be established by checking whether its configuration and customization conforms to the XML schema and Java specification, respectively. However, the cross-references between concept instances inside code configuration might become another serious and frequent problem. For example, as mentioned, each Spring Bean needs to inform its dependencies. As illustrated in Figure 2.5, all such references across Spring Beans are name-based: the name of the attribute `ref` should match the `id` attribute of the corresponding Spring Bean in the case of constructor-based dependency injection.

Considering the code in Figure 2.5, suppose that the construction `<constructor-arg ref="categoryDaoOsgi"/>` is not assigned to any feature. Although this code will be well-formed for all products that actually select the feature *Browsing*, the reference is not resolved in product in which *Browsing* is not selected. In this case, the `construct-args` reference remains but the corresponding Spring Bean configuration is removed. Unfortunately, there is no mechanism in XML Schema to statically enforce this constraint, even when a

PUC-Rio - Certificação Digital Nº 0812626/CA

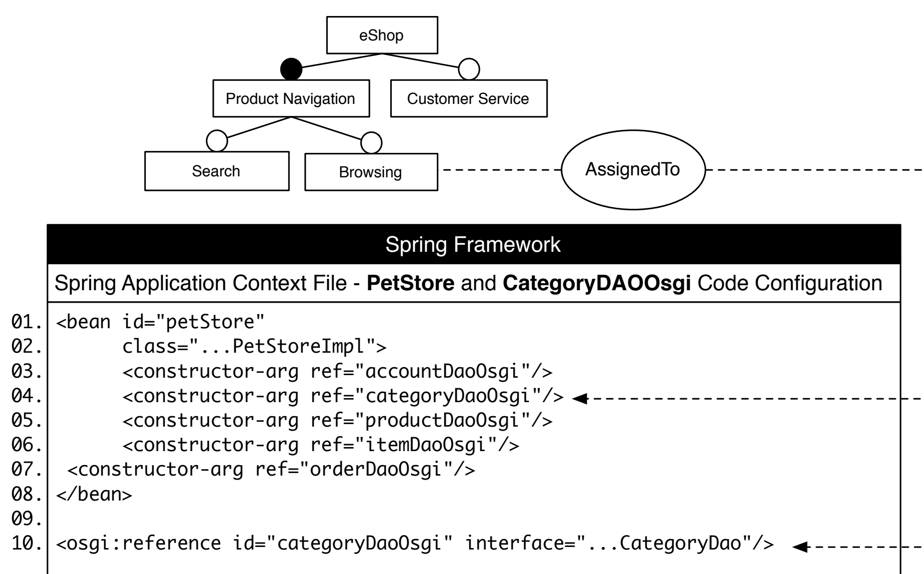

Figure 2.5: Cross-references between concept instances.

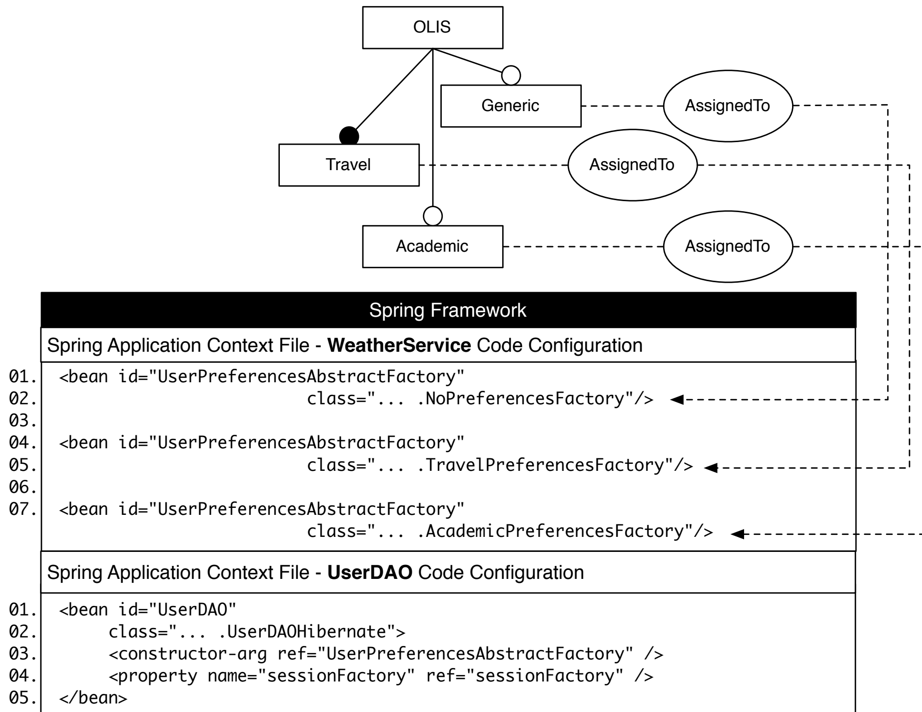

Figure 2.6: Source code of mutually exclusive concepts instances.

PUC-Rio - Certificação Digital Nº 0812626/CA

product without *Browsing* is requested. XML schemas are limited in the sense that element and attributes declarations are context insensitive. It means that XML Schemas do not express when the presence of an element or attribute depends on the presence of other elements. Therefore, the developers might be unaware of the errors unless a product with a problematic feature combination is derived and executed.

In this case, developers also face the difficult navigation problem. Searching for all uses of a certain concept instance without any guidance might increase developer effort. Depending on the number of concepts, they still analyzing them just to be sure that a feature assignment does not impact any other concept instance. The previous examples were relatively simple because they contained only assignment with single optional features. However, it is possible to have features that are mutually exclusive. Even worse, feature models might contain in practice complex constraints, such as "feature X excludes Y and requires W and Z".

Consider the source code in Figure 2.6. This is only well-formed if developers know (i) that *Travel*, *Academic*, and *Generic* are mutually exclusive - Spring might not work properly due to the existence of more than one *Beans* with the same `id`; and (ii) that *User Preferences* can only be selected if either *Travel*, *Academic* or *Generic* is selected. In the last case, an ill-formed product can be derived with a reference to `UserPreferencesAbstractFactory`. Figure

2.6 illustrates that techniques for guaranteeing correctness of framework-based application usually are no expressive enough to address all circumstances in which feature assignment violates programming interface constraints. In addition, it also shows that complex constraints among feature need to be considered, for ensuring well-formed framework-based software product lines. As this information might not be clear in source code, developers are susceptible to analyze unnecessary code or even forget to observe an important part of source code, increasing their development effort and the chances of introducing errors.

PUC-Rio - Certificação Digital Nº 0812626/CA

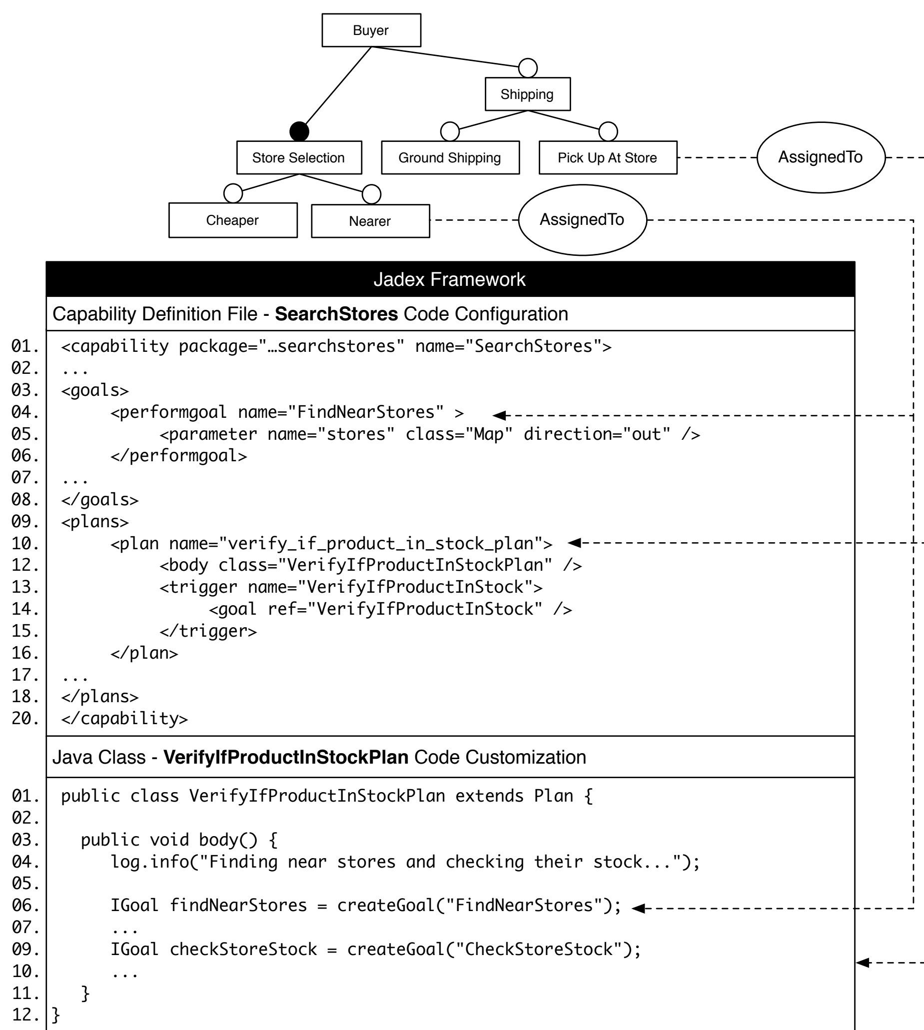

Figure 2.7: References to concept instances inside customization code.

2.1.3 References to Concept Instances inside Customization Code

The relation between code configuration and customization is that code configuration declares the existence of a concept instance and code cus-

tomization provides the implementation. For example, the definition file illustrated in Figure 2.7 declares the *Goal* `FindNearStores` and the *Plan* `VerifyIfProductInStockPlan`. The *Plan* is implemented in Java by the class `VerifyIfProductInStockPlan.java`. Actually, *Plans* define the concrete behaviours that an *Agent* might assume. There are three expected kinds of consistency constraints between such elements. First, as illustrated in Section 2.1.1, the Java implementation must exist. Second, the *Goal* `VerifyIfProductInStock` must exist (see Section 2.1.2). Finally, only declared concepts may be accessed inside the *Plan* implementation. For example, *Goals* can be also dispatched by *Plans* via the `createGoal` function, see line 06 in Figure 2.7. The code customization should only access keys in this function that correspond to declared concepts, such as *Goal* `FindNearStores`. Accordingly, code configuration and customization must agree on proper mapping with features, in addition to agree with use of common names and types.

Sadly, as mentioned neither XML Schemas nor Java specification state any consistency requirements on the relation between declared concept instances and their use. The use of attributes stored in maps is weakly typed. So, by using this means of accessing concept instances inside customization code we lose static guarantees. The code in Figure 2.7 is well-formed for all products that actually select the feature *Pick Up at Store*, which regards the Java syntax and type system. However the reference to *Goal* `FindNearStores` cannot be resolved in products in which *Nearer* is not selected and this problem will be detected only at runtime. Therefore, the expected constraints are implicit and its violation can lead to unpredictable behaviour or product malfunction. Again, here we also have the comprehension problem. To know about the correct means of using a certain concept instance, developers need to jump from the code customization to configuration. In addition, there is no easy way to know the impact of assigning a feature to a specific concept instance.

2.1.4
Context Sensitive Instantiation Constraints

The last scenario where developers must be careful when assigning source code elements to features refers to concept instantiation governed by context sensitive constraints. This scenario appears when developers are using concepts designed to be highly reusable. As they often are applicable in many different situations, they must satisfy particular requirements for each one.

For example the *Result* concept from the Struts framework implements a diversity of constrains. Depending on the value of the *Type* attribute a variety of *Param* concept instances might be required. For example, *Result* instances

PUC-Rio - Certificação Digital Nº 0812626/CA

Struts Framework

Struts Configuration File

```
01. <struts>
02. ...
03. <action name="Event_*" method="{1}" class="eventAction">
04.      <result name="input">/pages/Event.jsp</result>
05.      <result name="success" type="redirect-action">
06.           <param name="actionName">EventsBoard</param>
07.           <param name="namespace">/eventsannouncement</param>
08.           <param name="parse">true</param>
09.           <param name="month">${eventDataManager.month}</param>
10.           <param name="year">${eventDataManager.year}</param>
11.      </result>
12. </action>
13. ...
14. </struts>
```

Figure 2.8: Context sensitive instantiation constraints.

of the type `redirect-action` require instances of `actionName` and `namespace` Param (see Lines 6,7 and 8 in Figure 2.8). In this case, such concept instances are mandatory because the Struts framework relies on this information to properly redirect the control flow for *Actions*. For redirecting to *Pages* any *Param* instance is required. Therefore, depending on the value of one attribute in XML documents, different instantiations constraints might take place.

As XML Schema checkers are context insensitive there is no way to statically enforce when a feature assignment is violating context sensitive constraints. That is, it is hard to know when a *Param* instance is missing by accident or on purpose. Again, both problems appear: the detection of errors at runtime and difficulty in comprehending features in the source code.

PUC-Rio - Certificação Digital Nº 0812626/CA

2.2 Limitations of the Related Work

We give in this section an overview of different code-oriented techniques and discuss benefits and drawbacks that guided us in the search for better feature implementation mechanisms. We have categorized the related work into two groups: annotation-based techniques (Section 2.2.1); and model- based techniques (Section 2.2.2). This helps to abstract from concrete languages or tools and instead discuss more generally advantages and limitations of the common underlying mechanisms.

In annotation-based techniques, the variable code is annotated with features and conditionally removed during product derivation. The C preprocessor cpp and CIDE (Kästner et al. 2008) are typical examples. First of all, annotation-based techniques are simple, easy to use in existing projects and are common in practice. Nevertheless their expressiveness are limited to the

semantics of languages such as Java and XML. In Section 2.2.1, we discuss benefits and problems of current annotation-based techniques.

On the other hand, in model-based techniques, feature implementation are specified using general-proposal models and ad-hoc references to source code elements. The references are used to conditionally remove the variable code from the common one. They usually also provide support for annotating the source code with features. pure::variants (pure systems 2012) is an example of an tool that implements the model-based technique. As well as annotation-based techniques, they are also flexible and ready to use in existing projects. However they provide a more high-level and powerful mechanisms for constraint specification. Nevertheless, model-based techniques tend to force the use of general-purpose abstractions and constraint mechanisms. In Section 2.2.2 we will point out some limitations, especially regarding to readability and consistency check.

PUC-Rio - Certificação Digital Nº 0812626/CA

2.2.1 Annotation-based Techniques

Annotation-based techniques implement features by annotating source code elements in a common code. By removing the annotated code elements tailored products can be automatically derived. The most common form is `#ifdef` and `#endif` directives to conditionally remove code snippets before compilation. CIDE (Kästner et al. 2008) is an example of modern preprocessor tools that supports code annotation with feature. In contrast to traditional preprocessors as C++, CIDE does not use additional annotations. Instead, it uses colors to indicate the association among source code elements and features. In cases where a code is associated with more than one feature, representing composition of features, CIDE uses a mix of background colors. Even tough CIDE is based on preprocessor semantic, it is superior in the sense that it does not allow developers annotating arbitrary source code elements, therefore avoiding syntactic (Kästner et al. 2009) and type problems (Kästner and Apel 2008).

Annotation-based techniques are simple, easy to use in existing projects and are common in practice. However they present several problems when applied in the context of enterprise software product lines. We exemplify the limitations that we observed most frequently by means of the codes snippet in Figure 2.9 and Figure 2.10.

First, instead of providing a view that separates all source code elements that instantiate framework-provided concepts by features, the annotation-based techniques scatters feature codes across the entire code base, where

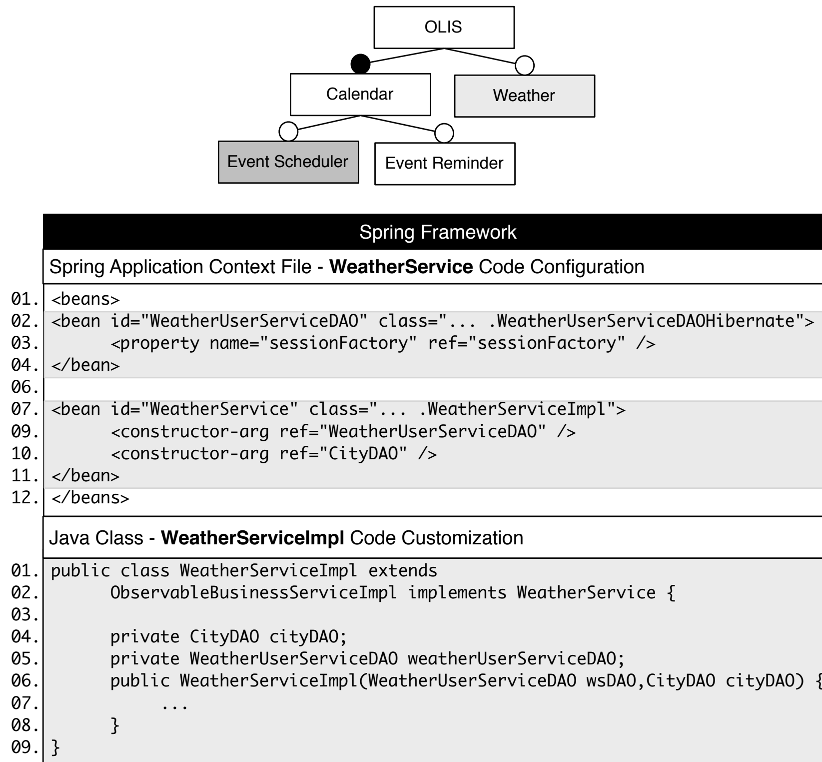

Figure 2.9: WeatherService instantiation code annotated with features.

they are also entangled with the code of other features and related concept instances. For example, in Figure 2.9, the code implementing the feature *Weather* is scattered throughout two files and tangled with code responsible for other feature, also implemented in the same Spring application context file. Even when the code implementing a concept instance is separated (see Figure 2.10), as the case of Capability `EventScheduler`, annotations can be misunderstood. As mentioned in Section 2.1, a high degree of scattering leads to the difficult navigation problem. Searching for all uses of a certain concept instance without any guidance might be cumbersome for developers, and in some case they might still be analyzing the source code just to be sure that a feature assignment associated to a concept instance does not impact any other concept instances (see Section 2.1.2). Therefore, such scattering and mismatching of configuration knowledge can make it hard for developers understanding when or why a certain source code element is included in a product, and even maintaining the source code becomes a tedious task. In general, some studies claim that annotation-based techniques work fine in small projects, however do not scale to large software product lines.

Also consistency checks among feature model, framework's programming interface and features used inside annotations are often missing. Using annotations to implement features can make it very difficult to detect the errors

PUC-Rio - Certificação Digital Nº 0812626/CA

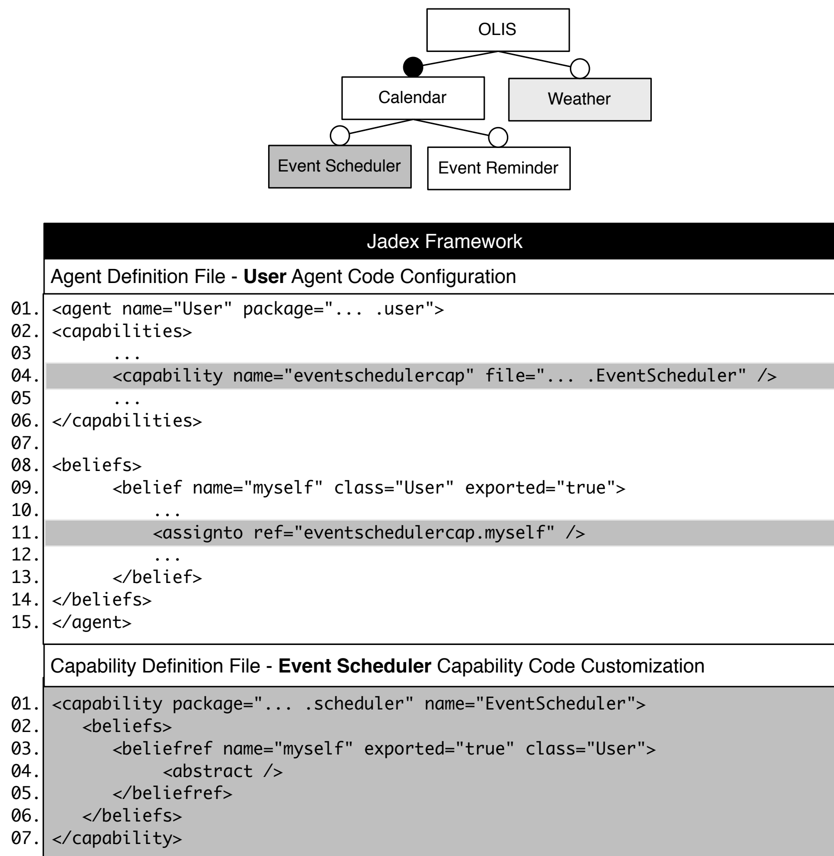

Figure 2.10: User agent and Event Scheduler capability instantiation codes annotated with features.

mentioned in Section 2.1. Even modern preprocessors, such as CIDE, make it difficult to detect errors introduced when annotations do not follow the framework's programming interface. They only operate at the level of programming languages, without observing the constraints that govern concept instantiation. Therefore, developers are prone to simple errors, like annotating the code customization but do not annotate the code configuration, as illustrated in Section 2.1.1. There is no way in CIDE to check this constraint. Even worse, as CIDE type check mechanism works only over the programming language type system, there is no easy and direct way to extend it to take into account the framework's programming interface.

2.2.2 Model-based Techniques

Model-based techniques (pure systems 2012, Czarnecki and Antkiewicz 2005) specify the configuration knowledge in one or more general-purpose models. pure::variants (pure systems 2012) is a commercial tool that implements this technique. In pure::variants the product line is defined in one or more

PUC-Rio - Certificação Digital Nº 0812626/CA

object-oriented- based models, called Family model. A Family model is organized into a logical hierarchy of Components, Parts and Source type. pure::variants provides general-purpose and predefined Part types: ps:class, ps:object, ps:package, ps:method, ps:flag, and so on. The physical representation of Components and Parts is defined by Sources. Source types can be: ps:file, ps:condfile, for example. In model-based techniques, model elements are only included in a product when: its parent is included; and any restrictions associated with it are fulfilled. Restrictions are used to express both features implementation and constraints among model elements.

Figure 2.11 illustrates a software product line defined in pure::variants. The Family model on the left side is the responsible for defining the architecture. For example, it contains numerous instances of ps:class type representing Spring *Beans* (e.g., `WeatherService`, `WeatherUserServiceDAO`, `CityDAO`).

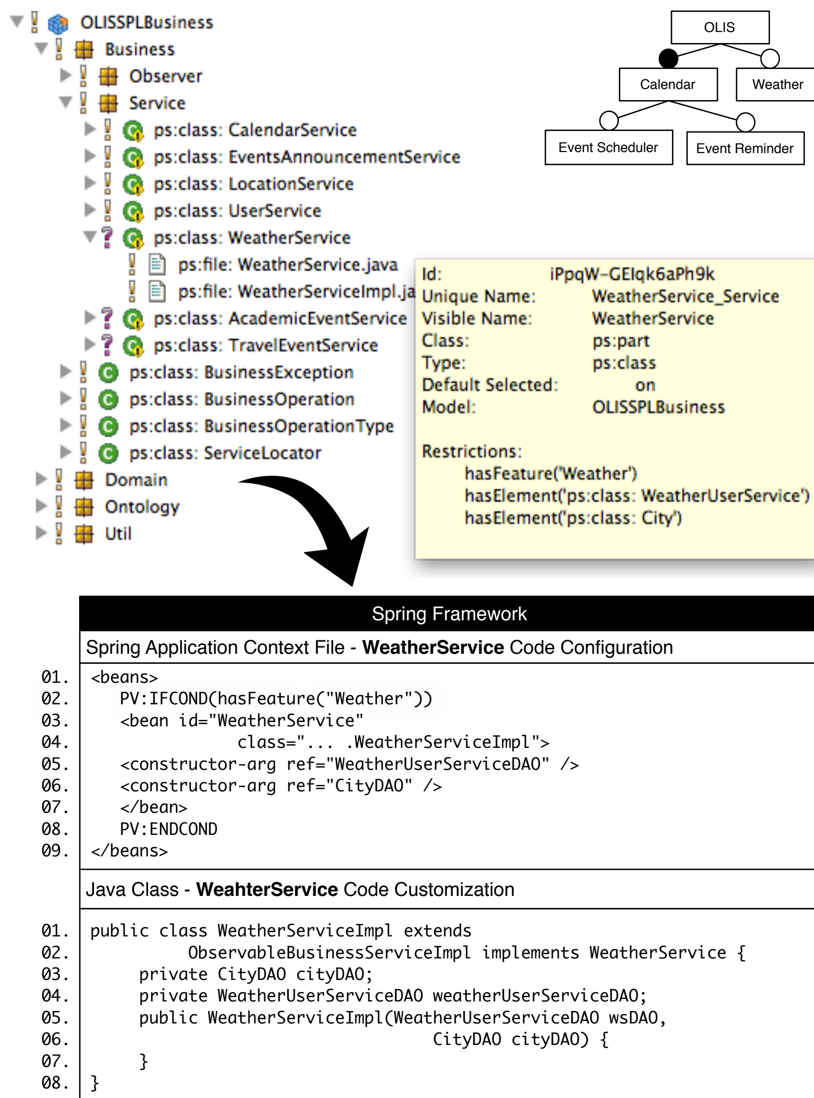

Figure 2.11: Software product line architecture defined in pure::variants.

PUC-Rio - Certificação Digital Nº 0812626/CA

There are also three restrictions attached to `WeatherService` element. The `hasFeature("Weather")` means that: the existence of the `WeatherService` element in the Family model depends on the selection of the `Weather` feature in any valid configuration of the feature model.

According to the product line growth, it might become increasingly difficult for developers to understand framework-based product lines defined in Family models. First, the use of general-purpose concepts does not support developers visualizing and reasoning about the configuration knowledge in terms of framework-provided concepts and programming interfaces they are already familiar with. Moreover, when creating the Family Model, the developers need to know what are the required and dependent source code elements. For example, they need to know that `WeatherService` is implemented by the `WeatherService.java` and `WeatherServiceImpl.java` files. Unfortunately, what defines the optional and required combination of source code is not specified explicitly in this technique. It can be only discovered by investigating the source code or the framework documentation which is often incomplete and does not cover the full range of concept variants.

In order to express fine-grained feature implementation in the source code model-based techniques often rely on templates engines. The code snippet in Figure 2.11 illustrates the Spring Application Context as a template file following the pure::variants notation. The template processing yields in a new file without codes related to expressions that evaluate to false. Note that such mechanism is similar to annotate the source code using features, as proposed in annotation-based techniques (see Section 2.11). Consequently, it suffers from the same fundamental problems discussed above: (i) tangled code (base code and template statements) distracts the programmer in the search for concept instances and uses; (ii) template statements are not part of the language but added on top by an external tool (e.g., pure::variants); (iii) there is no explicitly traceability from a feature to its implementation by means of framework-provided concepts.

An interesting aspect of model-based techniques is that they support the specification of requires and excludes constraints between model elements. Consider the Family model in Figure 2.11. The last two restrictions, `hasElement("WeatherUserService")` and `hasElement("City")`, express that `WathearService` element depends on two other elements (`WeatherUserService` and `City`) of the Family model. These restrictions express the dependencies defined in the code configuration of the Bean `WeatherService`, as we can observe in the right side of Figure 2.11 (see lines 05 and 06 from the Spring application context file).

PUC-Rio - Certificação Digital Nº 0812626/CA

Although restrictions provide an intuitive model for describing constraints in product line development, there is no mechanism in existing model-based techniques that ensures the correspondence between constraint applied to model elements and the framework's programming interface. As illustrated in Section 2.1, in real-world framework-based software product lines there are more complex constraints that a general-purpose technique dedicated to software product line implementation must take into account. For example, using pure::variants it is easy for developers forget to create a constraint that obligates the presence of *Bean* `WeatherUserSevice` when the *Weather* feature is selected. Therefore, without reading the framework's documentation, it is not possible to know whether a constraint is missing by accident or on purpose. Moreover, as mentioned, even when available, they are passive and partial, not covering the full range of concepts and their instantiation constraints.

In addition, since there is no connection between the restrictions in the modeling level (e.g., restrictions, requires and excludes constraints) and code level (e.g., template statements), determining which decisions lead to an exclusion of all source code elements implementing a given concept instance; and whether they are included in at least one product, is tedious and difficult to automated. For example, there is no way to check when the restriction related to `WeatherService` element is compatible with the `IF` statement (see line 02-08 in Figure 2.11) inside the Spring Application Context file. This misconnection leads to situations where the introduced errors are hard to discover and resolve.

2.3 Summary and Goals

Table 2.2 summarizes the discussed benefits and limitations of annotation and model-based techniques. It is apparent that both groups are, in some aspects, complementary, and that there is a lack of appropriated support for implementing features in the context of enterprise software product lines.

Our experience with four different enterprise product lines has shown that both groups are hard to use and some benefits like code navigability and consistency check are not appropriately achieved, mostly when applied in the context of framework-based software product lines. As discussed, some limitations (e.g., abstraction mismatch) are even conceptual and cannot be solved with just incrementing existing tools. Hence, instead of improving such existing solutions, our goal is to adjust the existing domain-specific modeling solutions, which are already broadly used on the implementation of framework-based software systems. We address the discussed scenarios, and propose a novel perspective to see features in domain-specific knowledge

PUC-Rio - Certificação Digital Nº 0812626/CA

models (DKMLs). Models builded from DKMLs are designed to represent the configuration knowledge of an implementation domain, defining a clearly mapping to code instantiating concepts. It is easy to build DKMLs for different domains of application. In addition, they encode the framework's programming interface, which allows us to automated some tasks, such as checking the consistency of the entire software product line.

Criteria	Annotation-based	General-purpose Model-based
Feature Assignment	Fine-grained and over the source code	Coarse-grained, via models. Fine grained, direct to source code. There is no formal connection between them
Error Detection	Some approaches can detect language syntax and type-system errors	General-purpose constraint checking and are not well connected to any implementation language
Language and Technology	In general are language independent	In general are language independent
Modularity	Scattered and tangled but can be virtual	Scattered and tangled
Uniformity	Simple and uniform	Mismatch between models and source code

Table 2.2: Benefits and limitations of annotation and model-based techniques.

PUC-Rio - Certificação Digital Nº 0812626/CA