A Flexible Model for Crosscutting Metadata-based Frameworks

Eduardo Guerra[1], Eduardo Buarque[1], Clovis Fernandes[1], Fábio Silveira[2]

[1] Aeronautical Institute of Technology (ITA) - Praça Marechal Eduardo Gomes, 50
CEP 12.228-900 - São José dos Campos – SP, Brazil
guerraem@gmail.com, skyedu_b@yahoo.com, clovistf@uol.com.br

[2] Federal University of São Paulo (UNIFESP)
Rua Talim, 330 - CEP 12231-280 - São José dos Campos – SP, Brazil
fsilveira@unifesp.br

Abstract. Frameworks aims to provide a reusable functionality and structure to be used in distinct applications. Aspect-oriented frameworks address crosscutting concerns and provide ways to attach itself in the application in a transparent way. However, using aspects the variations in the behavior can only be customized by aspect inheritance, which can increase exponentially the number of aspects and difficult the pointcut management. This paper proposes a flexible model which combines techniques for the insertion of crosscutting functionality with the structure of a metadata-based framework. This model allows (a) the maintenance of the class obliviousness, (b) the independence of the crosscutting technology and (c) the framework customization by composition. Additionally, the paper presents SystemGlue, which is a crosscutting framework that implements the proposed concepts. A modularity analysis was performed in an application that uses this framework to evaluate if the objectives were achieved.

Keywords: framework, aspect orientation, metadata, software design, software architecture.

1 Introduction

A framework is a set of classes that supports reuses at larger granularity. It defines an object-oriented abstract design for a particular kind of application which does not enable only source code reuse, but also design reuse [1]. The framework's abstract structure can be filled with its own classes or application-specific ones, providing flexibility for the developer to adapt its behavior to each application. Besides flexibility, a good framework also increases the team productivity and makes application maintenance easier [2] [3].

A framework can contain points, called hot spots, where applications can customize their behavior [4]. They represent domain pieces that can change among applications. Points that cannot be changed are called frozen spots, which usually

define the framework's general architecture, which consists in its basic components and the relationships between them. There are basically two different types of hot spots, that respectively uses inheritance and composition to enable the application to add behavior [5]. The use of composition allows the creation of black box frameworks [6], which scales better and provides a more flexible structure than the ones that use inheritance.

An aspect-oriented framework [7], like object-oriented frameworks, can be considered a incomplete reusable application that must be instantiated to create a concrete software. They can be classified as cross-cutting framework, which implement non-functional requirements, and application frameworks, that implement business rules. To specialize the framework behavior to a target application, an abstract aspect should be specialized, implementing the abstract methods and configuring the desired pointcuts. This structure based on inheritance is not suitable for frameworks with a large number of possible behavior variations [8]. For instance, the number of necessary aspects can grow exponentially based on the number of possible variabilities.

This paper introduces a flexible model that can be used to create aspect-oriented frameworks which uses a metadata-based processing to eliminate the drawbacks of the approach based on aspect inheritance. The present work proposes the usage of metadata to configure framework variabilities, an internal structure to enable composition and a metadata definition technique to maintain obliviousness. To evaluate this properties a complex framework for system integration, ready to be used in production environments, was implemented and used in a case study. Based on that, a modularity analysis was performed to verify the proposed model properties.

2 Frameworks

This section aims to describe different kinds of frameworks highlighting their main characteristics and the way that they provide behavior adaptation. In subsection 1.2, the mechanisms based on inheritance and composition in object-oriented framework are described. Next, subsection 2.2 presents the aspect-oriented frameworks and the drawbacks of using only inheritance to implement the behavior variabilities. Further, subsection 2.3 introduces the metadata-based frameworks, how they work and how they are internally structured.

2.1 Object-oriented Frameworks

A framework can be considered an incomplete software with some points that can be specialized to add application-specific behavior, consisting in a set of classes that represents an abstract design for a family of related problems. It provides a set of abstract classes that must be extended and composed with others to create a concrete and executable application. The specialized classes can be application-specific or taken from a class library, usually provided along with the framework [1].

Another important characteristic of a framework is the inversion of control [3, 9]. A framework's runtime architecture enables the definition of processing steps that can call applications handlers. This allows the framework to determine which set of application methods should be called in response to an external event.

An abstract class can define abstract methods that are invoked from a more general method in the same class. Those general methods are called template methods [10] and they define the skeleton of an algorithm in an operation, deferring to subclasses the redefinition of certain steps without changing the algorithm's structure. Those abstract methods are called hook methods [5] and must be implemented in the subclasses for framework adaptation [3].

The main framework class can also have some instance variables and delegate part of the execution to them. Those instances must obey a known protocol, extending an abstract class or implementing an interface, for the framework to be able to invoke methods on them. In this case, the hook methods invoked by the template methods are located in other classes, which are called hook classes. Thus, for framework adaptation it is not necessary to extend the framework main class and the developer must only change the instance that composes it. That instance can be taken from the framework's own class library or can be application-specific.

A framework is neither pure blackbox nor pure whitebox. The whitebox strategy is more difficult to use, because the developer must know details about the framework's internal structure. It is also more flexible, because it gives more freedom for choosing what should be overridden by the subclass. The blackbox strategy hides the implementation details and composes the application functionality with hook classes. It is also less flexible, since the application can interfere only in certain points. In whitebox, the implementation must be chosen when the class is instantiated, and in blackbox it can be changed later. A pattern language for framework evolution [6] suggests that a framework should start being whitebox, which is more flexible, and when the extension points became more clear, it should evolve to a blackbox strategy.

2.2 Aspect-oriented Frameworks

Aspect-oriented programming [12] is a programing paradigm, whose main goal is to modularize cross-cutting concerns. The adoption of this paradigm by the software development community is still happening and it is usually used encapsulated inside tools and frameworks, such as JBoss Application Server [13, 14] and Spring [15,16].

The modularization capabilities of aspect-oriented programing can be used to improve object-oriented frameworks. Using aspects, it is possible to add features in an existent object-oriented framework without the modification of the original source-code [17]. This modularization of framework's features brings other benefits such as functionalities that can be easily disabled and potentially used in other contexts.

Other possibility is the creation of an aspect-oriented framework [7] that can be classified as cross-cutting framework, which implement non-functional requirements, and application frameworks, that implement business rules. Like object-oriented frameworks, those can be considered a incomplete reusable application that must be instantiated to create a concrete software. A framework's abstract aspect must be

specialized to be weaved in the desired pointcuts and to add implementation in the hook methods, like represented in Fig. 1.

An abstract aspect cannot use composition in extension points, because its invocation is transparent for the application, which do not have direct access to the aspect to set the hook classes. The composition can be used in this context only if the hook classes are instantiated using a Factory Method [10], which is a type of hook method.

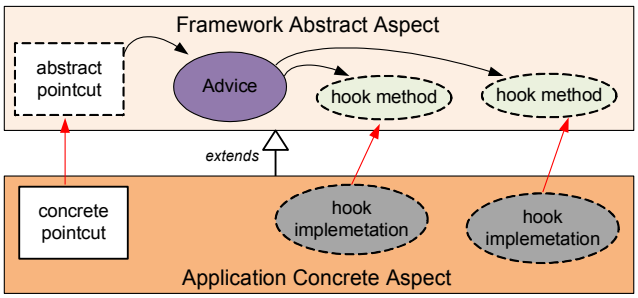

Fig. 1. The structure of an aspect-oriented framework.

A study about existent aspect-oriented frameworks [7] analyzed 13 frameworks and all of them contains a small number of functional variabilities. That can indicate that the existent model does not scale for a large number of possible behavior variations.

Indeed, based on this structure, every variability in those frameworks should be modeled as hook methods in the main abstract aspect. For variabilities whose behaviors can be combined the number of possible advices grows exponentially with the number of variabilities [8]. The concrete pointcuts also became granular and hard to manage.

2.3 Metadata-based Frameworks

The framework structures has evolved and recent ones make use of introspection [18] [19] to access at runtime the application classes metadata, like their superclasses, methods and attributes. As a result, it eliminates the need for the application classes to be coupled with the framework abstract classes and interfaces. The framework can, for instance, search in the class structure for the right method to invoke. The use of this technique provides more flexibility to the application, since the framework reads dynamically the classes structure allowing them to evolve more easily [20].

When a framework uses reflection [20][21] to access the class elements and execute its responsibilities, sometimes the class intrinsic information is not enough. If framework behavior should differ for different classes, methods or attributes, it is necessary to add a more specific meta-information to enable differentiation. For some domains, it is possible to use marking interfaces, like Serializable in Java Platform, or naming conventions [22], like in Ruby on Rails [23]. But those strategies can be used only for a limited information amount and are not suitable for situations that need more data.

Metadata-based frameworks can be defined as frameworks that process their logic based on the metadata of the classes whose instances they are working with [24]. In those, the developer must define into application classes additional domain-specific or application-specific metadata to be consumed and processed by the framework. The use of metadata changes the way frameworks are build and how they are used by software developers [25].

The developer's perspective in the use of those frameworks has a stronger interaction with metadata configuration than in method invocation or class specialization. In traditional frameworks, the developer must extend its classes, implement its interfaces and create hook classes for the behavior adaptation. He also have to create instances of those classes, setting information and hook class instances. Using metadata-based frameworks, programming focus is on declarative metadata configuration and the method invocation in framework classes is smaller and localized.

The basic processing in a metadata-based framework consists in the metadata reading from the target object, followed by its processing. In this process, the metadata read is used to adapt framework behavior and to apply introspection to access and modify the application object.

In [24], a pattern language for metadata-based frameworks was described, addressing the main issues about how to structure internally metadata-based frameworks. The patterns Delegate Metadata Reader and Metadata Processor combined enable the extension of the metadata schema, providing a solution that allow the insertion of new application-specific hook classes in the framework execution. This solution is used in APIs like Bean Validation [26] and frameworks like JColtrane [27].

The metadata consumed by the framework can be defined in different ways. Naming conventions [22] uses patterns in the name of classes and methods that has a special meaning for the framework. To exemplify this there are the Java Beans specification [28], which use method names beginning with 'get' and 'set', and the JUnit 3 [29], which interprets methods beginning with 'test' as test cases implementation. Ruby on Rails [23] is an example of a framework known by the naming conventions usage.

Conventions usage can save a lot of configurations but it has a limited expressiveness. For some scenarios the metadata needed are more complex and naming conventions are not enough. An alternative can be setting the information programmatically in the framework, but it is not used in practice in the majority of the frameworks. Another option is metadata definition in external sources, like XML files and databases. The possibility to modify the metadata at deploy-time or even at runtime without recompile the code is an advantage of this type of definition. However, the definition is more verbose because it has to reference and identify program elements. Furthermore, the distance that configuration keeps from the source code is not intuitive for some developers.

Another alternative that is becoming popular in the software community is the use of code annotations, that is supported by some programming languages like Java [30] and C# [31]. Using this technique the developer can add custom metadata elements directly into the class source code, keeping this definition less verbose and closer to

the source code. The use of code annotations is called attribute-oriented programing [32].

Prior studies report a successful use of attribute-oriented programming in different contexts [33], like serialization, web service endpoints and interface to databases. It is also used in a fractal component model implementation [34] and in conjunction with Model-driven Development [35]. A recent experiment about the usage of metadata revels that the use of these frameworks reduces the application coupling and can increase the team productivity [36].

3 Proposed Model

This section presents the proposed model for metadata-based crosscutting frameworks. The word “crosscutting” was used instead of “aspect-oriented” since the model can also be applied to other implementation strategies like the use of dynamic proxies [19] and composition filters [37]. For simplification, in the model description the strategies are referenced as aspects, unless the differentiation is relevant in the context.

This model’s goal is to provide a flexible structure for a crosscutting framework to be able to deal with a large number of variabilities. Other characteristics considered were the preservation of the class obliviousness and an easy framework adaptation for distinct architectures. The following subsections present the proposed practices to achieve these goals.

3.1 Metadata for Behavior Adaptation

Since an aspect can intercept the execution of different classes without their knowledge, it is hard to differentiate the execution for each one. The main strategy of the existent aspect-oriented frameworks for behavior differentiation is to provide different aspects for each possibility [7]. These aspects inherit from a framework abstract aspect specializing its behavior. As presented in the previous section, this model has serious drawbacks for a large number of variabilities, specially when they differ in a granular way among the classes and methods.

The foundation of the proposed model is to use class metadata to differentiate framework behavior. In aspects, the pointcuts are already defined based on class metadata, like class package, class name, method name, method return, parameter types and others. It can even use domain-specific or application-specific metadata defined in code annotations. Despite metadata defined can also be used for pointcut definition, this model proposes that this metadata should be consumed by the framework to enable differentiation of the execution logic among the classes.

It should define which variations are possible in the framework execution and provide a metadata schema to enable this differentiation. The metadata can be defined using code annotations, XML files, databases, code conventions or using a combination of this strategies. When a method execution is intercepted, the

framework should read its intrinsic and domain-specific metadata and use it to parameterize its execution.

Among the benefits of this approach, it is possible to highlight that the use of metadata enable the existence of a single framework aspect. That aspect should be specialized only to define a more specific pointcut where it should be applied in the target application.

3.2 Intercepting Technology Independence

One of the requirements that should considered in the construction of a framework is the adaptability for different architectures and applications. The actual aspect implementations in Java language are not a standard adopted by all applications. Examples of aspect implementations in Java are AspectJ [15], Spring AOP [16] and JBoss AOP [38]. Additionally, other solutions provide functionality that allow the insertion of components that can intercept the execution of a component method, such as dynamic proxies [19], EJB 3 interceptors [39] and CDI interceptors [40].

To enable framework independence about how execution should be intercepted in the architecture, this model proposes the encapsulation of the framework main functionality in a component, like illustrated in Fig. 2. This component can be invoked by different kinds of software components which can intercept the application execution, such as aspects, filters and proxies.

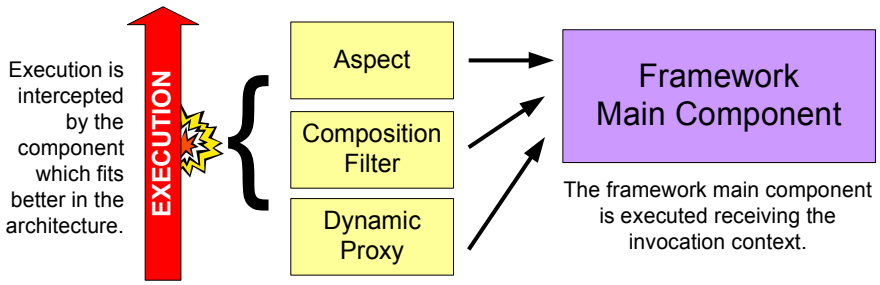

Fig. 2. Independence of the framework and intercepting component.

This practice allow the application to choose how the crosscutting framework should be attached to it. It makes the framework invocations more flexible and enable it to adapt easily to distinct environments. The authors consider this practice is advisable, not only to frameworks based on metadata, but for every crosscutting framework.

3.3 Metadata Extension

As presented in the previous sections, one of the weaknesses of the current model adopted for aspect-oriented frameworks relies in the usage of aspect inheritance for behavior specialization. By using metadata for framework adaptation (subsection 3.1)

and decoupling the main component from the execution interception (subsection 3.2), it is possible to use a model based on composition.

Fig. 3 illustrates the process proposed in this model. When the framework main component receives an invocation by one of the intercepting components, it invokes a class responsible for metadata reading that returns a representation of that information. This representation, called Metadata Container [24], can contain only the metadata retrieved, or moreover classes for which part of the execution can be delegated. These classes, created based on the class metadata, are called Metadata Processors [24].

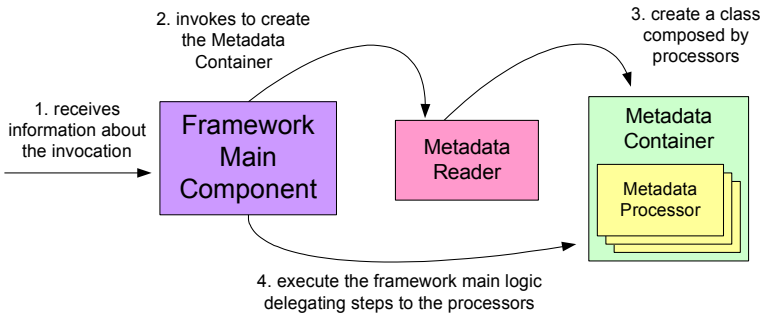

Fig. 3. Creation and execution of metadata processors.

Using this structure, it is possible to create application-specific metadata processors, enabling the extension of the framework functionality. To make it possible, a mapping that links each metadata type to a class that reads it should be created. Based on that mapping, the class responsible for reading metadata delegate the reading of these types to the Metadata Reader Delegate classes [24]. These classes are responsible for the creation of the Metadata Processors, like presented on Fig. 4.

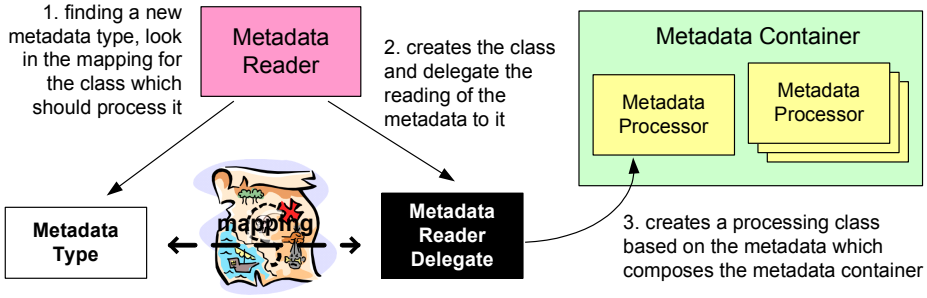

Fig. 4. Delegating the metadata reading.

If an application needs to extend the framework functionality, the first step is to create a new metadata type, which can be for instance an annotation or an XML element. The next step is to create the reader delegate class and map it to the created metadata type. Further, the metadata processor with the desired behavior should also be implemented and returned as the result of the reader delegate execution. Since the

processor would compose the metadata container, the execution of framework should be delegated to it.

This approach provides a solution that enables the extension of the framework behavior using composition. It allows processors to be combined in a more natural way without an explosion on the number of classes to support the combination of variabilities.

3.4 Domain Annotations

Especially when the metadata is defined using code annotations, the application class receives directly information about the framework concern. This creates a semantic coupling between the class and the framework, which reduces the application modularity.

To enable the usage of attribute-oriented programming without compromising the obliviousness, the present model proposes the use of domain annotations. The domain annotation concept was introduced by [41] in an attempt to introduce annotations in the context of Domain-driven Design [42]. The main idea is to represent domain concepts using annotations and not others related to non-functional and crosscutting concerns.

This model proposes the mapping of domain annotations to framework annotations, providing a decoupling of the application classes with the framework metadata. This mapping represents a translation of how the framework should deal with a class or a method which represents a given domain concept. This mapping also brings other benefits like a better modularization [43] and a reduction in the duplication of configurations [44].

Fig. 5 illustrates this mapping. The framework annotation should annotate the domain annotation instead of the class directly. The mapping can be called dynamic when the framework is prepared to search at runtime for its annotations inside other annotations. The mapping is static if a tool change the domain annotations to the framework annotations at compile time. For instance, Daileon is a tool which provides a function library that facilitate the implementation of a dynamic mapping and a tool for the static mapping [45].

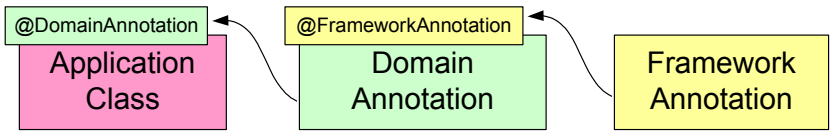

Fig. 5. Domain annotations mapping.

4 Implemented Framework – Esfinge SystemGlue

The software developed to demonstrate the proposed model in the present work is Esfinge SystemGlue [46], which is an open-source framework which aims to enable

the creation of distinct integration profiles for a given application. It was developed to solve a problem in a real application in which different clients needs to invoke distinct methods to integrate with their systems. The framework had already been functionally tested using ClassMock [47] and can be considered ready to be used in a production environment.

The next subsections describes the framework functionalities, its strategy for metadata definition and its internal structure.

4.1 General View

SystemGlue aims to provide an structure that allows the application to configure distinct integration profiles, enabling the invocation of different functionality according to the context. It uses metadata to define what should be executed after or before an application method execution. It supports method invocation, scheduling and message sending that can be executed based on conditions and asynchronously. The framework also uses metadata and code conventions to map the parameters and the returns among the invocations. The metadata definition can be defined in a flexible way using a combination of annotations and XML documents.

The following example exemplify the usage of annotations to configure the execution of functionality before and after a method execution. While the methods are invoked, their parameters and returns can be mapped and used among subsequent executions based on their names, which can be defined respectively by the parameter annotation *@Param* and the method annotation *@ReturnName*.

SystemGlue metadata configuration with annotations.

```
    @Executions ({
       @Execute(clazz=InteligenceIntegration.class,
             method="getTargetInfo",
             when=ExecutionMoment.BEFORE,
             rule="order.targets.size==0") ,
     @Execute(clazz= UnitsIntegration.class,
             method="sendOrder",
             when = ExecutionMoment.AFTER,
             async = true)
})
public void saveOrder(@Param("order") Order order){
  //core functionality implementation
 }
```

4.2 Flexible Metadata Definition

The use of framework annotations direct in the application methods can be useful for executing functionality which should always be invoked. Since to change the code annotations the code should be re-compiled, it is not a good solution to allow the configurations to be changed for distinct integration profiles.

SystemGlue also supports the metadata definition using XML files. This approach allow a more decoupled definition, which is more suitable for define metadata in situations where more than one metadata set is necessary for one class [48], which is the case for integration profiles. The next code presents an example of the same metadata defined in the previus example represented in an XML file.

SystemGlue metadata configuration using XML.

```
<systemglue>
 <class name="expl.OrderService">
  <method name="saveOrder" params="expl.Order">
   <execute class="expl.InteligenceIntegration"
       method="getTargetInfo" when="BEFORE"
       rule="order.targets.size == 0"/>
   <execute class="expl.UnitsIntegration"
     when="AFTER" method="sendOrder" async="true"/>
   <execute/>
  </method>
 </class>
</systemglue>
```

For the framework to load an XML file it is necessary to invoke the method loadXMLFile() in the class MetadataRepository. This file can define metadata for more than one class and a class can have metadata defined in more than one file.

A drawback of the presented approaches is that if different methods needs the same metadata configuration, the code to define it should be duplicated. It reduces maintainability making difficult general modifications in the metadata definition.

To avoid this problem, an alternative for metadata definition is the usage of domain annotations [41], which represents concepts related to the application domain and are defined by the application. These annotations can be mapped to the SystemGlue metadata using annotations or in the XML file, providing an indirect configuration. Considering that the framework functionality is crosscutting, the domain annotations preserve the classes obliviousness [43], since they would not contain information about a crosscutting concern.

Next code listing presents an example of the domain annotation mapping using annotations. The SystemGlue annotations are used in the domain annotation @OrderModification instead of directly on the class method. The framework recognize this indirect configuration and add this metadata to all methods configured with it. This practice facilitate changes, since the modification of the framework annotations would affect all methods annotated with the domain annotation. A domain annotation can annotate other domain annotation providing an specialization mechanism.

SystemGlue configuration of domain annotations.

```
//annotation definition
@Executions({
  @Execute(clazz = InteligenceIntegration.class,
    when=ExecutionMoment.AFTER, method="getTargetInfo",
    rule="order.targets.size==0"),
```

```
  @Execute (clazz= UnitsIntegration.class,
    when = ExecutionMoment.AFTER, method="sendOrder",
    async = true)
})
public @interface OrderModification{}

//method definition
@OrderModification
public void saveOrder(@Param("order") Order o){}
```

The use of domain annotations can also be combined with XML definition. The metadata configuration can refer to an annotation instead of the method directly. Using this approach, the annotation can be simply defined without framework annotations. Next code listing presents an instance of the domain annotation metadata definition in the XML.

Referencing the domain annotation in the XML file.

```
<systemglue>
  <annotation name="expl.OrderModification">
    <execute class="expl.InteligenceIntegration"
         method="getTargetInfo" when="BEFORE"
         rule="order.targets.size == 0"/>
    <execute class="expl.UnitsIntegration"
         when="AFTER" method="sendOrder" async="true"/>
  </annotation>
<systemglue>
```

It is important to highlight that any combination of these techniques can be used together in the same method to define the invocation of distinct functionality. Despite the advantages and drawbacks, each one is more applicable to a different scenario.

4.3 SystemGlue Internal Structure

One of the requirements considered in the construction of SystemGlue is that it should to be adaptable for different architectures. To enable SystemGlue functionality to be inserted in the most natural way to the application architecture, the main functionality is encapsulated in a component, named SystemGlueExecutor, which does not crosscut the application functionality.

Other components, such as dynamic proxies or aspects, are responsible to intercept the application methods invocation and delegate the execution to the main component. This flexibility is important to allow the execution to be inserted in a way which fits better in the application architecture. SystemGlue provides implementation of reflection dynamic proxies [19], which creates proxies based only on interfaces, and CGLib proxies [49], which supports proxies based on classes. The framework was also tested using Spring AOP [15] and EJB3 Interceptors [39], however these implementations are not provided with the framework to avoid more dependencies.

The framework follows the basic structure proposed in the section 3, as presented in Fig. 6. The SystemGlue main component retrieves the metadata container from a

metadata repository when it receives an invocation. The repository is populated with information retrieved from XML files and from the class annotations. The metadata container is composed by instances of the type MethodExecutor, which represents the executions that should be made after and before the application method.

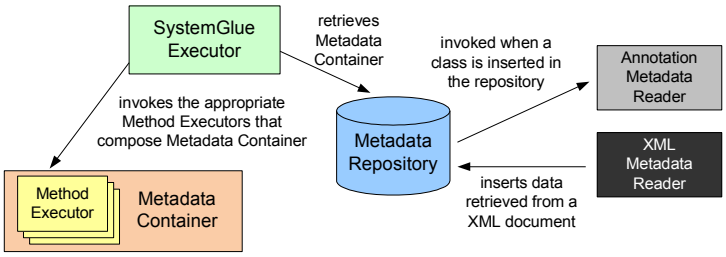

Fig. 6. SystemGlue internal organization.

5 Modularity Analysis

This section presents an evaluation of the model modularity, by analyzing a case study that used Esfinge SystemGlue framework and verifying if it was able to achieve the proposed characteristics. As a tool to this analysis, this work used a Dependency Structure Matrix [50], which is a matrix that basically shows the dependence between all the elements in a given software.

The interpretation of a DSM is made by noticing that both rows and columns have the same information: they represent a complete list of system entities whose dependence should be mapped. Each cell of the matrix represent the number of dependences between the entity represented by the line to the entity represented in the column.

To evaluate if the model allows the fulfillment of the modularity requirements, a fictitious case study was prepared with an application that plays the role of a Hospital ERP and three other applications representing softwares that integrate with it. It uses Esfinge SystemGlue to integrate the applications by using the domain annotations functionality. Figure 7 shows the DSM created based on the developed software.

The domain annotations are in the package *br.com.lab.integration* (C, D, E, G, H, I and J), classes responsible to activate the main features of the application are in the package *br.com.lab.controller* (B and F), SystemGlue's annotations are in the package *net.sf.systemglue.annotations (*K, L, M, N, O and P), and the remaining packages represent the classes responsible for the integration functionality (Q, R and S).

Based on the DSM extracted from the case study, it is possible to draw some conclusions about the system modularity. The main application classes only depends on the domain annotations. This dependence is highlighted by the yellow rectangles. Since the domain annotations express domain information, the application classes does not contain even a semantic dependence with integration concerns.

The domain annotations depend on the SystemGlue annotations to define each one's configuration. The SystemGlue annotations are highlighted by the green rectangle, while the dependences are highlighted by the orange rectangles. The classes responsible for the integration concern, highlighted in the blue rectangle, are completely decoupled of the rest of the system.

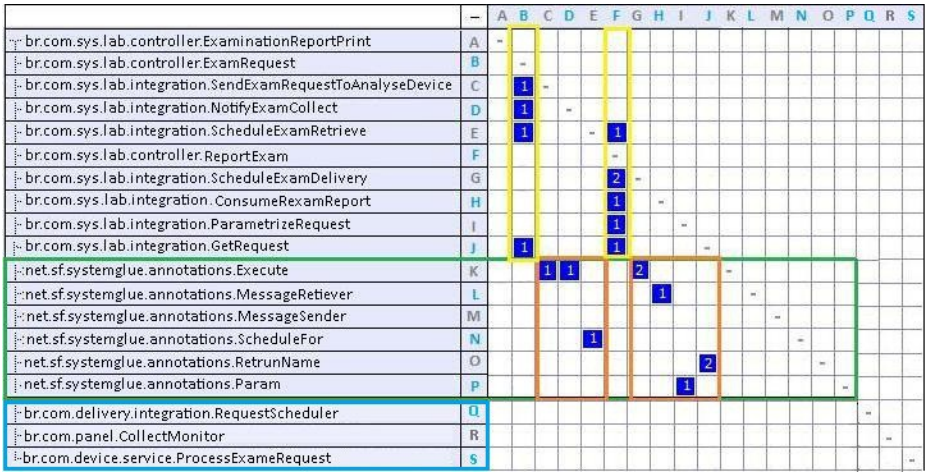

Fig. 7. Modularity analysis using a DSM.

Hence, the framework enables configuration profiles on metadata integrations with domain annotations, since the application classes deal with the main features and have no syntactic or semantic dependencies of classes that perform the integrations. Then SystemGlue has the responsibility to activate the points of integration. Based on that it is possible to conclude that the proposed model allows the fulfillment of this modularity requirements.

6 Conclusions

This work proposes a new model for crosscutting frameworks which enables it to deal with a high number of behavior variations. It is probably not suitable for domains with a small number of behavior variations. It proposes the use of metadata to enable the framework to use composition as the strategy for behavior extension. The model also proposes techniques for decoupling the component responsible for the method interception and the use of domain annotations to enable the usage of attribute-oriented programming without compromising the obliviousness.

This model was used to build a framework named SystemGlue which aims to provide a flexible structure to enable the creation of distinct integration profiles for one application. It naturally deals with a high number of variations, including the possibilities of parameters and return mapping and the combinations of functions to be invoked before and after the application method execution. The integration functions can also be invoked conditionally and be executed asynchronously

according to the configurations. The framework also provide flexible alternatives for metadata configuration and for attaching it in an architecture. A modularity analysis was performed in a case study that instantiated SystemGlue and the decoupling model requirements were evaluated.

References

1. R. Johnson; B. Foote. “Designing reusable classes”, In Journal Of Object-Oriented Programming, v.1, n. 2, p. 22-35, Jun./Jul. 1988.
2. Wirfs-Brock, R.; Johnson, R.. “Surveying current research in object-oriented design”. In: Communications of the ACM, September 1990, Volume 33 , Issue 9, p. 104-124.
3. M. Fayad, D. Schmidt, R. Johnson, “Application frameworks”, in Building Application Frameworks: Object-oriented Foundations of Frameworks Design, New York: Wiley, 1999. Chap. 1, p. 3-27.
4. Pree, W. Design Patterns for Object-Oriented Software Development. Addison Wesley / ACM Press, 1995.
5. Pree, W. “Hot-spot-driven development”, In “Building application frameworks: object-oriented foundations of frameworks design”, New York: Wiley, c1999, Chap. 16, p. 379-393.
6. Don, R.; Ralph, J. “Evolving Frameworks: A Pattern Language for Developing Object-Oriented Frameworks”. In proceedings of the Third Conference on Pattern Languages of Programming, 1996.
7. Camargo, V.; Masiero, P. “Frameworks Orientados a Aspectos”. In: XIX Simpósio Brasileiro de Engenharia de Software – SBES 2005. Uberlândia : 2005. Proceedings... p. 200-216.
8. Guerra, E.; Silva, J.; Silveira, F.; Fernandes, C. “Using Metadata in Aspect-Oriented Frameworks”. In: 2nd Workshop on Assessment of Contemporary Modularization Techniques (ACoM.08) at OOPSLA, 2008, Nashville - EUA.
9. Bosch, J.; Molin, P.; Mattsson, M.; Bengtsson, P.; Fayad, M. “Framework Problems and Experiences”. In: Building Application Frameworks – Object-oriented Foundations of Frameworks Design. Wiley, c1999, Chap. 3, p. 55-83.
10. E. Gamma, R. Helm, R. Johnson, J. Vlissides, “Design Patterns: Elements of Reusable Object-Oriented Software”, Addison-Wesley, 1994.
11. Jacobsen, E.; Nowack, P. “Frameworks and Patterns: Architectural Abstractions”. In: Building Application Frameworks – Object-oriented Foundations of Frameworks Design. Wiley, c1999, Chap. 2, p. 29-54.
12. Kiczales, G.; Lamping, J.; Menhdhekar, A.; Maeda, C.; Lopes, C.; Loingtier, J.; Irwin, J. “Aspect-oriented programming”. In European Conference on Object-oriented Programming, 1997. Proceedings... p. 220–242.
13. Fleury, M.; Reverbel, F. “The JBoss Extensible Server”. In ACM/IFIP/USENIX 2003 International Conference on Middleware. Rio de Janeiro: 2003. Proceedings... p. 344-373.
14. Jamae, J.; Johnson, P. “JBoss in Action: Configuring the JBoss Application Server”. Manning Publications, 2009.
15. Laddad, R. “AspectJ in Action: Enterprise AOP with Spring Applications “. Manning Publications, 2nd edition, 2009.
16. Walls, C.; Breidenbach, R. “Spring in Action”. Manning Publications; 2nd edition, 2007.
17. Silva, Maria Tania; Braga, Rosana; Masiero, Paulo Cesar. Evolução Orientada a Aspectos de um Framework OO. In: 1º Workshop de Manutenção de Software Moderna, 2004, Brasilia - DF.
18. F. Doucet, S. Shukla, R. Gupta, “Introspection in system-level language frameworks: meta-level vs. Integrated.” In Source Design, Automation, and Test in Europe, 2003. Proceedings... [S.l.: s.n], 2003. p. 382-387.
19. I. Forman, N. Forman, “Java reflection in action”. Greenwich: Manning Publ., 2005.
20. B. Foote, J. Yoder, “Evolution, architecture, and metamorphosis”, In Pattern Languages of Program Design 2. Boston: Addison-Wesley Longman, 1996. Chap. 13, p. 295-314.
21. Maes, P. “Concepts and Experiments in Computational Reflection”. In The International Conference on Object-oriented Programming, Systems, Languages and Applications – OOPSLA 1987. Proceedings... p. 147-169.
22. N. Chen, “Convention over configuration”, 2006. Available at <http://softwareengineering.vazexqi.com/files/pattern.html>, accessed on 17 dez. 2009.
23. Ruby, S.; Thomas, D.; Hansson, D. “Agile Web Development with Rails”. Pragmatic Bookshelf, Third Edition, 2009.
24. E. Guerra, J. Souza, C. Fernandes, “A pattern language for metadata-based frameworks”, In Conference on Pattern Languages of Programs, 16., 2009, Chicago. Proceedings… , 2009.

25. L. O'Brien, “Design patterns 15 years later: an interview with Erich Gamma, Richard Helm and Ralph Johnson”. InformIT, Oct. 22, 2009. Available at <http://www.informit.com/articles/article.aspx?p=1404056>, accessed on 26 dez.2009.
26. JSR 303: Bean Validation. 2009. Available at <http://www.jcp.org/en/jsr/detail?id=303> acessed in 17 dez 2009.
27. R. Nucitelli, E. Guerra, C. Fernandes, “Parsing XML Documents in Java Using Annotations”, In XML: Aplicações e Tecnologias Associadas, 2010, Vila do Conde, Portugal.
28. JavaBeans(TM) Specification 1.01 Final Release. 1997. Available at <http://java.sun.com/javase/technologies/desktop/javabeans/docs/spec.html> acessed in 27 dez 2009.
29. Massol, V.; Husted, T. “JUnit in Action”. Manning Publications, 2003.
30. JSR 175: a metadata facility for the java programming language. 2003. Available at <http://www.jcp.org/en/jsr/detail?id=175>, accessed on 17 dez. 2009.
31. Miller, J.; Ragsdale, S. “Common language infrastructure annotated standard”. Boston: Addison-Wesley, 2003.
32. Schwarz, D. “Peeking inside the box: attribute-oriented programming with Java 1.5.” [S.n.t.], 2004. Available at <http://missingmanuals.com/pub/a/onjava/2004/06/30/insidebox1.html>, accessed on 17. dez. 2009.
33. Cisternino, A.; Cazzola, W.; Colombo, D. “Metadata-driven library design”. In Library-centric Software Design Workshop. Proceedings... 2005.
34. Rouvoy, R.; Pessemier, N.; Pawlak, R.; Merle, P. “Using attribute-oriented programming to leverage fractal-based developments”, In International ECOOP Workshop on Fractal Component Model, 5., Nantes, 2006. Proceedings… , 2006.
35. Wada, H.; Suzuki, J. “Modeling Turnpike Frontend System: a Model-Driven Development Framework Leveraging UML Metamodeling and Attribute-Oriented Programming”. In Proc. of the 8th ACM/IEEE International Conference on Model Driven Engineering Languages and Sytems (MoDELS/UML 2005), 2005.
36. Guerra, E.; Fernandes, C. “An Experimental Evaluation on Metadata-based Frameworks Usage”, unpublished.
37. Bergmans, L.; Aksit, M. “Composing crosscutting concerns using composition filters” Commun. ACM, vol. 44, no. 10, pp. 51–57, 2001.
38. JBoss AOP: Framework for Organizing Cross Cutting Concerns.. Available at <http://www.jboss.org/jbossaop> acessed in 01 jun 2010.
39. JSR 220: Enterprise JavaBeans 3.0. 2006. Available at <http://www.jcp.org/en/jsr/detail?id=220>, accessed on 17 dez. 2009.
40. JSR 299: Contexts and Dependency Injection for the JavaTM EE platform. 2009. Available at <http://www.jcp.org/en/jsr/detail?id=299>, accessed on 17 dez. 2009.
41. Doernenburg, E. “Domain Annotations”. In The Thoughtworks Anthology: Essays on Software Technology and Innovation, Chapter 10, p. 121-141. Pragmatic Bookshelf, Raleigh, NC, USA, March 2008.
42. Evans, E. “Domain-Driven Design: Tackling Complexity in the Heart of Software”. Addison-Wesley Professional, 2003.
43. J. Perillo, E. Guerra, J. Silva, F. Silveira, C. Fernandes, “Metadata Modularization Using Domain Annotations”, In Workshop on Assessment of Contemporary Modularization Techniques 3.,2009, Orlando. Proceedings…[S.l.: s.n], 2009.
44. J. Perillo “Daileon: Uma Ferramenta Para Habilitar o Uso de Anotações de Domínio”, Trabalho de Curso (Engenharia de Software) - Curso de Especialização em Tecnologia da Informação, São José dos Campos: Instituto Tecnológico de Aeronáutica, 2010.
45. Perillo, R. ; Guerra, E.; Fernandes, C. “Daileon: A Tool for Enabling Domain Annotations”. In: 6th ECOOP'2009 Workshop on Reflection, AOP and Meta-Data for Software Evolution, 2009, Genova.
46. SystemGlue. Available at <http://systemglue.sf.net/> accessed in 14 mai 2010.
47. Guerra, E.; Silveira, F. ; Fernandes, C. “ClassMock: A Testing Tool for Reflective Classes Which Consume Code Annotations”. In: Workshop Brasileiro de Métodos Ágeis (WBMA 2010), 2010, Porto Alegre.
48. Fernandes, C.; Ribeiro, D.; Guerra, E.; Nakao, E. “XML, Annotations and Database: a Comparative Study of Metadata Definition Strategies for Frameworks”. In: XML: Aplicações e Tecnologias Associadas, 2010, Vila do Conde, Portugal.
49. Code Generation Library - CGLIB. Available at <http://cglib.sourceforge.net/> accessed in 31 jan 2010.
50. Yassine, A. “An Introduction to Modeling and Analyzing Complex Product Development Processes Using the Design Structure Matrix (DSM) Method”. Quaderni di Management (Italian Management Review), No.9, 2004.