A Design Strategy to Facilitate the Instantiation Process of White-Box Frameworks

André L. de Oliveira[1], Fabiano C. Ferrari[2], Paulo C. Masiero[1], Rosângela A. D. Penteado[2], Valter V. de Camargo[2]
[1]Mathematics and Computer Science Institute, University of São Paulo, São Carlos-SP, Brazil
[2]Computing Department, Federal University of São Carlos (UFSCar), São Carlos-SP, Brazil
{andre_luiz, masiero}@icmc.usp.br, {fabiano, rosangela, valter}@dc.ufscar.br

Abstract — **Frameworks aim at supporting the reuse of design and implementation in specific domains. A well-known type of framework is the white-box, whose instantiation takes place through inheritance. The instantiation process of this type of framework is heavily based on the implementation of hook methods, *i.e.*, it is necessary to provide concrete implementation for abstract methods. In general, making these methods concrete demands knowledge of the internal structure of the framework, which hardens the instantiation process. Moreover, the greater number of hook methods the more difficult is to reuse this kind of framework. In order to solve these problems, we propose a design strategy which combines Aspect-Oriented Programming, interfaces and annotations to ease the instantiation process of white-box frameworks. A framework called GRENJ was used as an example to show the applicability of our strategy. As a result, we observed a reduction in complexity of the instantiation process and in the amount of hook methods to be overridden.**

Keywords-Annotation; aspect-orientation; interfaces; instantiation.

I. Introduction

Frameworks promote the reuse of design and implementation in specific domains by means of encapsulating generic behaviors into abstract programming units [9]. They offer a set of variabilities, which must be selected to instantiate specific applications. Frameworks can be classified as white-box, black-box or gray-box. White-box are frameworks whose reuse process is heavily based on inheritance, *i. e.*, one must create concrete units which specialize the abstract ones provided by the framework [18].

In another research field, Aspect-Oriented Programming (AOP) contributes to a proper modularization of "crosscutting concerns" by providing abstractions specifically to address this kind of concern. The motivation behind AOP is that object-oriented programming is not able to properly modularize crosscutting concerns, leading to the known problems of code spreading and tangling [10]. As AOP has become an important research topic in the late 90's, several researchers started to investigate how its concepts impact framework development [3][13].

In general, the instantiation process of either, object-oriented (OO) or aspect-oriented (AO) white-box frameworks consists in providing concrete implementation for a great number of hook methods [18]. Usually, some of these methods must be overridden just to conform to internal framework policies or even to choose some variabilities. Besides, in some situations, the application engineer must be aware of architectural details of the framework to override some of its abstract methods. Yet another factor that harms the framework reuse is the amount of hook methods to be implemented. So, due to the high complexity of these methods the framework reuse process is more time-consuming and error-prone because the more hook methods to be implemented greater is the time and the probability of making mistakes.

Aiming at providing a process that cope with these problems, we propose a strategy that makes the instantiation of white-box frameworks a less time-consuming and error-prone task. Our strategy allows the application engineer to concentrate on what is really important, leaving out architectural details. This strategy demands less knowledge of the internal structure of the framework and reduces the amount of hook methods to be implemented. This strategy is a combination of AO concepts, interfaces and annotations [12].

Using the proposed strategy, the instantiation process involves labeling application classes with annotations, rather than only supplying concrete implementation for hook methods. So, there is a reduction in the amount of hook methods of the framework that needs to be implemented. This can be justified by the fact we can encapsulate the returning types of a group of inter-related hook methods in one attribute of one annotation. Note that, in our instantiation strategy, based on information provided by annotations labeled in application classes it is possible to provide concrete implementation for constructors, database mapping hook methods and other framework hook methods automatically, therefore simplifying the instantiation process. The contributions of this paper are twofold: i) the elicitation of instantiation problems which are inherent to white-box frameworks and ii) the proposal of a strategy for supporting the instantiation of white-box frameworks.

In Section II, the concepts of Annotations are presented. In Section III, the instantiation problems of white-box frameworks are presented. In Section IV, there is a description of how AOP, interfaces and annotations may be used to support the process of framework instantiation. In Section V, we present a preliminary assessment of the use of our strategy. In Section VI, the related works are discussed and in Section VII, the conclusion and proposal of future research are presented.

II. Annotations

Annotations are structures to mark parts of a program with data about the program itself – a kind of metadata. Annotations may be used to: supply information to the compiler in order to detect errors or emit warnings; performing processing in compile and deployment time and perform processing at runtime [12]. Annotations may be applied in declarations of classes, attributes, methods and other program elements. They may appear before the declaration of classes and may include elements with named or unnamed values.

Usually, an annotation is created for encapsulating a set of metadata about the program structure [12]. For example, suppose that a team needs to insert several comments in a class in order to register the author, implementation date, last modification date, current version, author of the modification and reviewers. Using annotations, it is possible to use these metadata by the definition of an *annotation type*, as shown in Figure 1. The definition of an annotation is similar to the definition of an interface, in which the reserved word `interface` is preceded by the character '`@`' [12]. As shown in Figure 1, an annotation includes the declaration of *annotation type* elements, which is similar to the definition of methods and that may define optional pattern values. To make the annotation available for processing at execution time, the annotation itself must include the declaration `@Retention (RetentionPolicy.RUNTIME)` (line 03 of Figure 1).

```
01 import java.lang.annotation.*;
02
03 @Retention(RetentionPolicy.RUNTIME)
04 @interface ClassPreamble {
05   // annotation types
06   String author();
07   String date();
08   int currentRevision() default 1;
09   String lastModified() default "N/A";
10   String lastModifiedBy() default "N/A";
11   String[] reviewers(); //array
12 }
```

Figure 1. Declaration of an annotation (Adapted from Oracle [12]).

III. White-Box Framework Instantiation

White-box frameworks as Spring [17] and Hibernate [7] use annotations as their main technique for the reuse process. Hibernate uses annotations for mapping classes and it fields to database tables. Spring uses annotations for mapping classes and their attributes to web user interface and their fields. Our annotation-based design strategy, presented in Section IV, has a different proposal; we use annotations to map the retuning types of white-box framework hook methods to concrete classes that should implement them. Note that such mapping is not possible using the annotation mechanisms provided by the Hibernate and Spring frameworks. However, white-box frameworks are still instantiated based only on overridden hook methods [3][4][13]. Therefore, when the amount of hook methods is large or when the implementation involves many details related to the internal structure of the framework, this task is harmed, which makes it expensive and error-prone.

The worst problem happens when the implementation of hook methods requires knowledge of details of the internal structure of the framework which are not directly related to the requirements of the application to be instantiated. This fact makes the process of instantiation a complex task that demands a deep knowledge on the internal structure of the framework. For example consider the piece of code shown in Figure 2. This is a concrete implementation for a hook method called `insertionFieldClause()` of a framework called GRENJ [4]. The main reason for the implementation of this method is to tell to the framework the attributes that have corresponding columns in the database table. However, to supply this simple information one needs to know the framework internal architectural characteristics, such as:

i) Creating a `String` object;

ii) Calling the `insertionFieldClause()` method of the base-class, which is also another requirement of the instantiation process. This is an internal architectural characteristic of the framework;

iii) Concatenating the attribute names with the `String` value, and

iv) Returning the `String` object.

So, although this implementation encompasses architectural details, some lines of code have no relation with the initial purpose of this method [4][8][15].

```
package grenj.model.instantiation;

public class Film extends Resource {
  ...
  public String insertionFieldClause() {
     String insertionClause = super.insertionFieldClause();
     return insertionClause + ", year";
  }
}
```

Figure 2. Example of instantiation in GRENJ framework.

IV. A Strategy To Instantiate White-Box Frameworks

Figure 3 shows a UML (Unified Modeling Language) class diagram detailing the strategy. The upper part of the figure represents framework units (classes/interfaces/aspects) and the lower part application units, that is, those that must be created by an application engineer to instantiate the framework. To represent aspects, we are using a UML profile for AspectJ defined by Everman et al. [5]. Annotations are represented by classes stereotyped with the `<<annotation>>` and the compartment of methods of these classes corresponds to attributes of the annotation.

The diagram of Figure 3 represents an instantiation scenario in which the application engineer needs to create two classes, `AppClass1` and `AppClass2`, to extend `FrameworkClass1` and `FrameworkClass2`. The engineer also needs to return the name of the second application class created (*i. e.*, `AppClass2`) to give a concrete implementation for `getFrameworkClass2()` method from `Framework Class1`. The framework needs to know the name of the second class to be able to instantiate objects from it internally. In conventional instantiation processes, the way to inform the name of this class would be to implement the hook method retuning the `FrameworkClass2.class` subtype (i.e. `AppClass2. class`). However, using our design strategy this can be avoided.

In our strategy, the values of the attributes of an annotation are values (types) which are the return types of hook methods. So, there is no need to implement complex hook methods, the only thing to do is to declare annotations. For example, in the `FrameworkClass1Metadata` annotation, there is an attribute called `frameworkClass2()`. This attribute defines

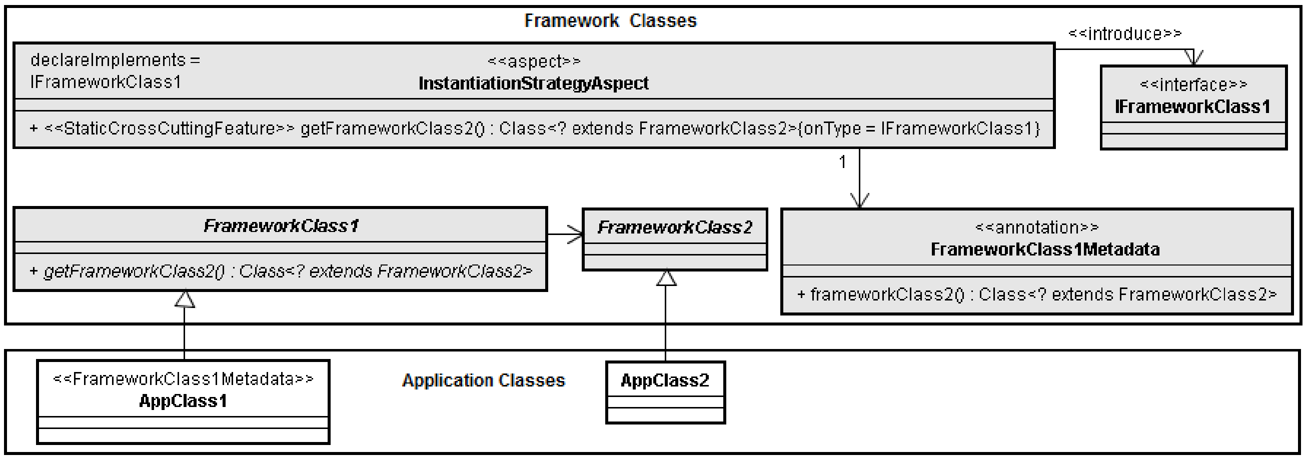

Framework Classes
<<introduce>>
declareImplements = IFrameworkClass1
<<aspect>>
InstantiationStrategyAspect
+ <<StaticCrossCuttingFeature>> getFrameworkClass2() : Class<? extends FrameworkClass2>{onType = IFrameworkClass1}
<<interface>>
IFrameworkClass1
1
FrameworkClass1
+ getFrameworkClass2() : Class<? extends FrameworkClass2>
FrameworkClass2
<<annotation>>
FrameworkClass1Metadata
+ frameworkClass2() : Class<? extends FrameworkClass2>
<<FrameworkClass1Metadata>>
AppClass1
Application Classes
AppClass2

Figure 3. Strategy of instantiation of frameworks with Aspects, Interfaces and Annotations.

```
01 package appclasses;
02
03 @FrameworkClass1Metadata(
04    frameworkClass2 = AppClass2.class
05 )
06 public class AppClass1 extends FrameworkClass1 {
07    ...
08 }
```

Figure 4. AppClass1 class.

an expression in which the symbol ? (question mark) acts as a parameter and indicates that any `Class` type that match this symbol will extend `FrameworkClass2`, which is another framework class. Thus, during the instantiation process the application engineer must supply an application class to match this symbol. In this example, the `AppClass1` is annotated with `FrameworkClass1Metadata`. Using this annotation the value for the attribute `frameworkClass2` must be `AppClass2`. This can be seen in Figure 4 (lines 03-05).

The implementation of the `FrameworkClass1 Metadata` annotation can be seen in Figure 5a). In Figure 5b), it is also possible to see the `IFrameworkClass1` interface. It is an empty interface, which plays the role of Container Introduction [6], i.e. an interface in which the `InstantiationStrategyAspect` aspect introduces concrete implementation for the hook methods defined in `FrameworkClass1`.

```
01 package instantiationstrategy;
02 …
08 @Target(ElementType.TYPE)
09 @Retention(RetentionPolicy.RUNTIME)
10 public @interface FrameworkClass1Metadata {
11
12    Class<? extends FrameworkClass2> frameworkClass2();
13 }
a)
```

```
01 package instantiationstrategy;
02
03 public interface IFramework1Class {}
b)
```

Figure 5. FrameworkClass1Metadata and IFrameworkClass1 interface.

The `InstantiationStrategyAspect`, shown in Figure 6, uses intertype declarations to make all classes annotated with `FrameworkClass1Metadata` to implement the `IFrameworkClass1` interface (lines 05-06). This aspect also uses intertype declarations to introduce into this interface a concrete implementation for the `getFrameworkClass2()` method, which returns a `.class` from the class which extends `FrameworkClass2` (lines 08-13). The return type of this method is the attribute value of the `FrameworkClass1Metadata` type (Figure 5a.).

```
01 package instantiationstrategy;
02
03 public aspect InstantiationStrategyAspect {
04
05    declare parents: (@FrameworkClass1Metadata *)
06           implements IFrameworkClass1;
07
08    public Class<? extends FrameworkClass2>
09          IFrameworkClass1.getFrameworkClass2() {
10          FrameworkClass1Metadata metadata = getClass().
11                getAnnotation(FrameworkClass1Metadata.class);
12          return metadata.frameworkClass2();
13    }
14 }
```

Figure 6. InstantiationStrategyAspect aspect.

V. Preliminary Assessment

We applied the proposed strategy to a GRENJ-based application. GRENJ [4] is a white-box framework built on top of a pattern language called GRN [2] and has been implemented by using the Java language. In short, GRENJ enables the creation of applications to manage resources (e.g. rental, trading and maintenance). The case study consists in a system for DVD rental whose requirements are the following: the rental shop rents DVDs of movies which may have one or more copies; the DVDs are rented to the customers; it is also necessary to store the information related to the rental; besides, a client may rent more than one DVD in the same transaction.

In the diagram of the Figure 7, the classes of the framework are the shaded areas (e.g. `Resource` and `SimpleType`), while the application classes are the ones with white background (e.g. `Category` and `Style`). Analyzing this diagram, we can notice that the `Resource`, `SimpleType`, `ResourceRental` and `TransactionItem` classes possess a set of abstract methods which need to be implemented by classes that extend them. Therefore, an application engineer needs to provide concrete implementation for methods like `typeClasses()` and `typeFieldsInitialize()` in the classes `Film`, `Category` and `Style` which, respectively, extend `Resource` and `SimpleType`.

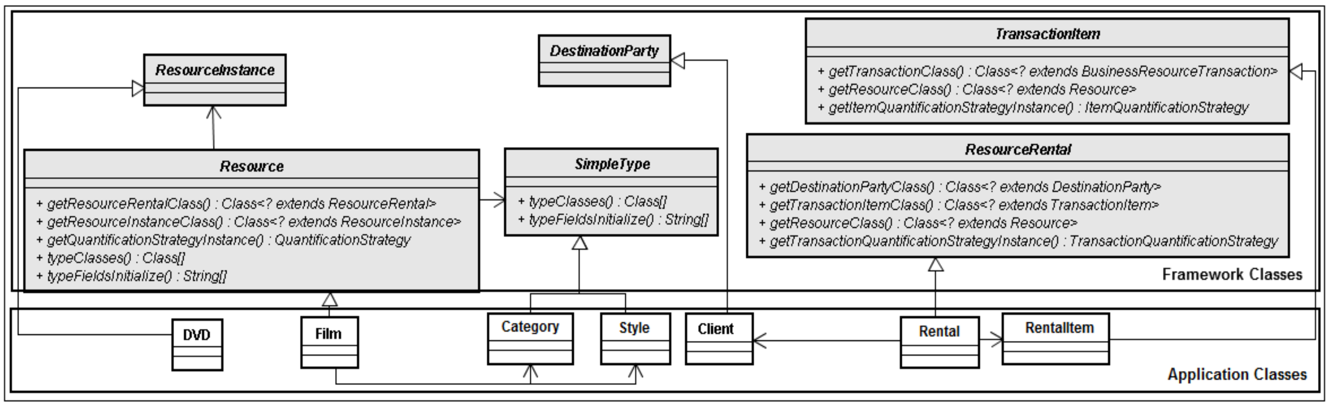

ResourceInstance
DestinationParty
TransactionItem
+ getTransactionClass() : Class<? extends BusinessResourceTransaction>
+ getResourceClass() : Class<? extends Resource>
+ getItemQuantificationStrategyInstance() : ItemQuantificationStrategy
ResourceRental
+ getDestinationPartyClass() : Class<? extends DestinationParty>
+ getTransactionItemClass() : Class<? extends TransactionItem>
+ getResourceClass() : Class<? extends Resource>
+ getTransactionQuantificationStrategyInstance() : TransactionQuantificationStrategy
Resource
+ getResourceRentalClass() : Class<? extends ResourceRental>
+ getResourceInstanceClass() : Class<? extends ResourceInstance>
+ getQuantificationStrategyInstance() : QuantificationStrategy
+ typeClasses() : Class[]
+ typeFieldsInitialize() : String[]
SimpleType
+ typeClasses() : Class[]
+ typeFieldsInitialize() : String[]
Framework Classes
DVD
Film
Category
Style
Client
Rental
RentalItem
Application Classes

Figure 7. Class diagram of the DVD rental shop system.

Due to space limitations, we illustrate the framework instantiation strategy proposed in this paper applying it to two hook methods: `typeClasses()` and `typeFieldInitialize()`, which need to become concrete within the `Film` class that extends `Resource`. This is depicted in the diagram of Figure 8. The `ResourceMetadata` annotation, the `IResource` interface and the `ResourceInstantiationAspect` aspect together compose the implementation of the proposed strategy in the GRENJ framework.

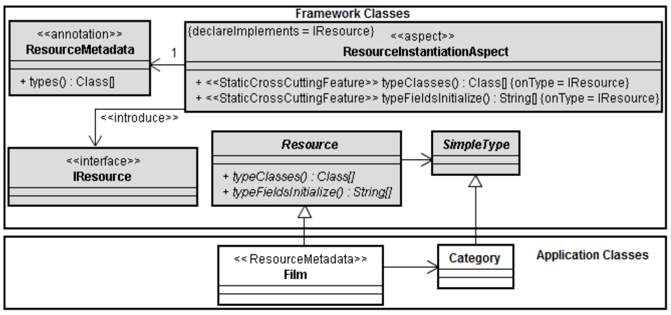
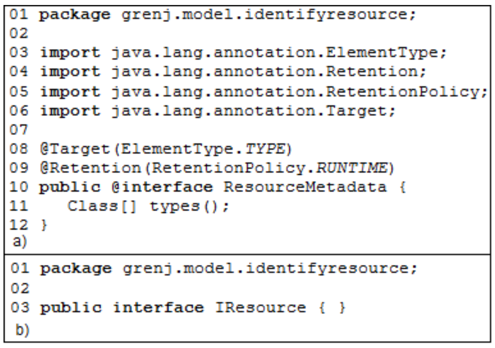

Framework Classes
<<annotation>>
ResourceMetadata
+ types() : Class[]
1
{declareImplements = IResource}
<<aspect>>
ResourceInstantiationAspect
+ <<StaticCrossCuttingFeature>> typeClasses() : Class[] {onType = IResource}
+ <<StaticCrossCuttingFeature>> typeFieldsInitialize() : String[] {onType = IResource}
<<introduce>>
<<interface>>
IResource
Resource
+ typeClasses() : Class[]
+ typeFieldsInitialize() : String[]
SimpleType
<<ResourceMetadata>>
Film
Category
Application Classes

Figure 8. Use of the mechanisms of the proposed strategy.

The `Film` class is marked with the stereotype `<<ResourceMetadata>>`, indicating that the `ResourceMetadata` annotation is declared in this class. In this annotation, the `types()` attribute (of the type `Class[]` – see line 11 of Figure 9a) is used to extract the returning types of the hook methods defined in the `Resource` class. The `IResource` interface is an empty interface which plays the role of *Container Introduction*, *i.e.*, an interface in which the `ResourceInstantiationAspect` aspect introduces concrete implementation for the hook methods defined in `Resource`. Such hook methods must be implemented by the classes that extend `Resource`. Figure 9b) presents the implementation of the `IResource` interface.

In order for the application engineer not to have to provide concrete implementation for the `typeClasses()` and `typeFieldsInitialize()` hook methods, both defined in the `Resource` class, we applied the elements of our strategy to facilitate the instantiation of the application class `Film` (Figure 8), which extends `Resource`. The `ResourceInstantiation Aspect` aspect, whose code is shown in Figure 10a), makes use of intertype declarations to introduce concrete implementations of `typeFieldsInitialize()` and `typeClasses()` into `IRe source` (lines 07-14 and 16-20 in Figure 10a). These methods return, respectively, an array of `Class` objects and an array of `String` objects. These return types correspond to the `types` attribute of the `ResourceMetadata` annotation, which is obtained by the `ResourceInstantiationAspect` aspect. Line 5 in Figure 10a) defines that `ResourceMetadata` annotation implements the `IResource` interface. Once this is done, the classes which have been marked with this annotation implement the methods that the aspect introduced into `IResource`, i.e. `typeClasses()` and `typeFieldsInit ialize()`.

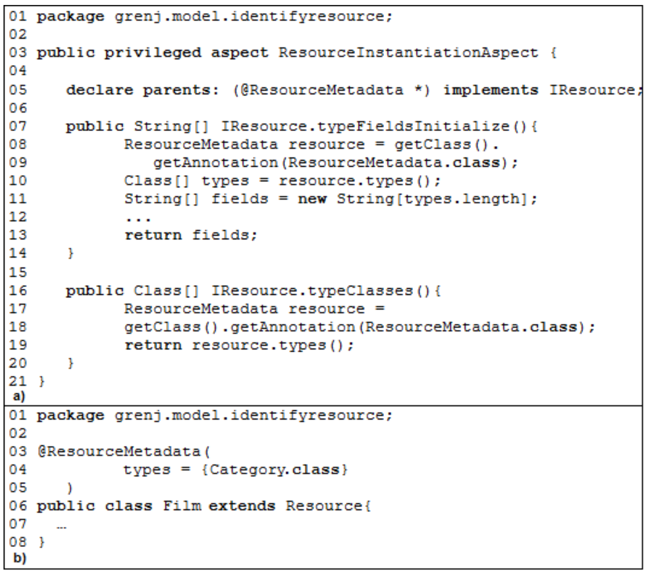

```
01 package grenj.model.identifyresource;
02
03 import java.lang.annotation.ElementType;
04 import java.lang.annotation.Retention;
05 import java.lang.annotation.RetentionPolicy;
06 import java.lang.annotation.Target;
07
08 @Target(ElementType.TYPE)
09 @Retention(RetentionPolicy.RUNTIME)
10 public @interface ResourceMetadata {
11    Class[] types();
12 }
```
a)

```
01 package grenj.model.identifyresource;
02
03 public interface IResource { }
```
b)

Figure 9. Implementation of the IResource interface and the ResourceMetadata annotation

Every class marked with the `ResourceMetadata` annotation automatically contains a concrete implementation of the `typeFieldsInitialize()` and `typeClasses()` methods, which are hook methods of the GRENJ's `Resource` class. In the example presented in this section (depicted in Figure 8), this class is `Film`, whose implementation is shown in Figure 10b). In lines 3-5 the `ResourceMetadata` annotation is declared, indicating the types associated with the `Film` class, which corresponds to the `Category` type.

The use of the strategy in applications instantiated with GRENJ reduced the number of methods that need to be overridden by the application engineer. This is possible because we can use an annotation to abstract the returning types of framework hook methods from a class that have common returning types. For example, the annotation attribute `types` from `ResourceMetadata` (Figure 8) abstracts the information necessary to implement the `typeClasses()` and `typeFieldInitialize()` methods from `Resource` class (the framework class). Table I summarizes the numbers of methods that needed to be implemented in application classes that extend GRENJ's classes with and without the use of the proposed instantiation strategy. Using the conventional way, the instantiation process of the DVD rental application requires concrete implementation for 33 hook methods. In the instantiation of the same application using the proposed strategy, it is necessary to provide concrete implementation for 17 methods. These results show that the use of the instantiation strategy reduced in 48.4% the number of methods that need to be overridden by the application engineer.



```
package grenj.model.identifyresource;

public privileged aspect ResourceInstantiationAspect {

   declare parents: (@ResourceMetadata *) implements IResource;

   public String[] IResource.typeFieldsInitialize(){
         ResourceMetadata resource = getClass().
            getAnnotation(ResourceMetadata.class);
         Class[] types = resource.types();
         String[] fields = new String[types.length];
         ...
         return fields;
   }

   public Class[] IResource.typeClasses(){
         ResourceMetadata resource =
         getClass().getAnnotation(ResourceMetadata.class);
         return resource.types();
   }
}
```

a)

```
package grenj.model.identifyresource;

@ResourceMetadata(
         types = {Category.class}
   )
public class Film extends Resource{
   …
}
```

b)

Figure 10. Implementation of the Film class and the ResourceInstantiationAspect aspect.

Furthermore, the use of annotations eliminated the need of providing concrete implementations for complex hook methods that involves knowledge about details of the framework architecture. So, it improves the framework reusability. For example, considering the implementation of the hook method presented in Figure 2, our strategy requires the application engineer to only annotate the class to have a concrete implementation for this hook method instead of implementing it manually. However, this effort reduction did not account for changes required in the application classes (e.g. annotating the code), which will be further considered in a more comprehensive assessment of the proposed instantiation strategy.

TABLE I. COMPARISON BETWEEN PROPOSED DESIGN STRATEGY X TRADITIONAL INSTANTIATION

Application Classes	GRENJ conventional instantiation	GRENJ proposed design strategy	%
Film	10	5	- 50%
Category	7	5	- 28.5%
Style	4	2	- 50%
Rental	6	2	- 66.6%
RentalItem	6	3	- 50%
Total:	**33**	**17**	**- 48.4%**

A limitation of the proposed strategy is that its adoption increases the number of units of the framework, once it is necessary to add three units related to the strategy for each class of the framework that contains abstract methods. Moreover, in order to implement our design strategy, the developer must be aware of the framework internals to be able to define the correct aspects and annotations. Another fact concerns the use of annotations, which requires the application engineer to know the mapping between them and the framework's classes. In the same way that we need of a cookbook to guide the instantiation of white-box frameworks without our strategy, to use the strategy in white-box frameworks it is necessary the elaboration of an instantiation cookbook containing the description of the mapping between annotations and framework's classes and how annotations should be used.

VI. RELATED WORK

Antkiewicz and Czarnecki [1] proposed the concept of *Framework-Specific Modeling Language* - FSML, a special category of modeling languages which are defined based on object-oriented application frameworks. A FSML is used to express models that show how abstractions provided by the framework are used in the code of applications derived from it. The use of FSML assists in the instantiation process of applications based on the frameworks by providing the mapping between variabilities of the framework, which permits the identification of mandatory, optional and alternative elements. FSMLs may be used to help in the process of creating mechanisms for framework instantiation, as the one presented in this paper, once the mapping of the framework variabilities has been described using an FSML. This information may be used for creating and grouping the annotations, interfaces and aspects of the instantiation strategy related to each framework variability.

Santos et al. [16] proposed a new approach to develop Framework-Specific Modeling Languages (FSML), which permits the elimination of the need to implement code generators. Their approach allows the generation of a complete application from the application model, which is done based on a domain specific language. This approach is related to our work as it allows the instantiation of an application from an application model. However, to use it, it is necessary to first elaborate the complete model of the domain. Our approach has a more restricted focus, which is the instantiation of white-box frameworks in which a strategy to automate the realization of hook methods is proposed.

Kulesza *et al.* [11] presented a generative technique in which the code that specializes abstract aspects is generated from instances of feature models. Our approach also focuses on automating the code implementation in specializations, on the level of framework classes, in which the proposed instantiation strategy is used to automate the implementation of hook methods of the frameworks. Unlike Kulesza *et al.*'s work [11], the instantiation strategy does not use feature models to automate the implementation. This is due to the fact that our strategy is applicable to instantiate white-box frameworks.

VII. CONCLUSIONS

In this work, a strategy to facilitate the instantiation process of frameworks was presented. Our design strategy uses dependence injection [14] in a generic way to automate the implementation of framework hook methods in order to reduce the application engineer effort. Its adoption reduces the complexity of the information that needs to be provided to instantiate the framework, as well as the number of hook methods that the application engineer needs to implement. This is done with annotations, interfaces and aspects. Reducing the number of hook methods that need to be implemented in turn reduces the likelihood of making mistakes during the framework instantiation process. The strategy proposed in this study is applicable to white-box frameworks which have been developed with programming languages that support aspect-oriented programming, annotations and interface definitions.

The presented strategy was applied in the instantiation of an application derived from the GRENJ framework [4]. As a result, it was possible to observe a reduction of 48.4% in the number of hook methods that needed to be implemented by the application engineer in comparison with the conventional process. This result may vary according to the framework to which the strategy is applied. The bigger the number of hook methods present in a framework, the smaller the number of methods that the application engineer will need to implement in comparison with conventional framework instantiations.

Based on the analysis of the instantiation strategy and its application in the case study, we noticed that from this strategy, it is possible to extract a pattern for the instantiation of white-box frameworks in which there is: (i) an empty interface which acts as a *Container Introduction* [6]; (ii) an annotation, which is used to establish a mapping between the application classes and the return types of the hook methods associated to them; and (iii) an aspect that introduces operations in the empty interface and makes the classes labeled as annotations implement it.

The adoption of our strategy is adequate to medium and large frameworks, which have a high number of hook methods whose realization is complex. For frameworks considered small, which contain few hook methods and that can be easily implemented, we believe the mechanism of instantiation is not a feasible alternative given that its use increases the number of modular units of the framework, which may affect its maintainability.

There are basically two scenarios in which our approach can be applied to. The first one is the reengineering of an existing legacy framework to a new version which is adherent to our design. As frameworks evolve over time, the number of hook methods can increase as well as their complexity. So, using our approach, the instantiation process can demand less time and effort. Another one is the design of new frameworks. As a team can know our design strategy in advance, it can be adopted since the beginning in order reach a good design and avoid the problems previously mentioned.

A limitation of the preliminary assessment presented in Section V is the absence of an evaluation of the approach's impact on the framework maintainability, in particular due to the increase of framework units. Another limitation of the instantiation strategy is that in order to use it is necessary to have knowledge about the mapping between the framework's classes and the annotations related to each one of these classes. This problem may be mitigated with the elaboration of an instantiation manual which contains the description of this mapping and the way to use the annotations in application classes. Our future research shall address a comparative study on the use of the instantiation strategy proposed in object-oriented and aspect-oriented frameworks. Another suggestion of future research is the study on the impact of using *Feature-Oriented Programming* in the instantiation process of frameworks. Also, there is the proposal of making experiments in order to precisely assess the effectiveness of the instantiation strategy presented in this paper. Creation of mechanisms that permit automatic generation of modular units of the instantiation strategy in frameworks is yet another issue for future research.

Acknowledgments

We would like to thank the financial support received from FAPESP (grant 2008/07833-0) and CNPq (grant 152693/2011-4).

References

[1] M. Antkiewicz, K. Czarnecki, K. 2006. Framework-Specific Modeling Languages with Round-Trip Engineering. In ACM/IEEE 9th International Conference on Model Driven Engineering Languages and Systems. Springer-Verlag, pp 692-706.

[2] R. T. V. Braga, F. S. R. Germano, P. C. Masiero, P. C. 1999. *A Pattern Language for Business Resource Management. Proceedings of the Annual Conference on Pattern Languages of Programs*, 6:1–33.

[3] V. V. de Camargo, P. C. Masiero. 2008. A pattern to design crosscutting frameworks. In *Proceedings of the 2008 ACM symposium on Applied computing, pp.* 759-764.

[4] V. H. S. Durelli, R. S. Durelli, R. T. V. Braga, S. S. Borges. 2010. A Domain Specific Language for Lessening the Effort Needed to Instantiate Applications Using GRENJ Framework. In *Information Systems Brazilian Symposium*, pp. 31-40.

[5] J. Evermann, 2007. A Meta-Level Specification and Profile for AspectJ in UML. Victoria University Wellington, New Zealand. AOSD - Aspect Oriented Software Development.

[6] S. Hanenberg, A. Schmidmeier. 2003. Idioms for Building Software Frameworks in AspectJ. In: 2nd AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software.

[7] Hibernate. 2010. Hibernate Framework. https://www.hibernate.org.

[8] JHotDraw. 2010. JHotDraw Framework. available in: http://www.jhotdraw.org/.

[9] R. E. Johnson. 1997. Frameworks = (components + patterns). *Communicatins of ACM* 40, 10 (Oct. 1997), pp. 39-42.

[10] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. M. Loingtier, and J. Irwin, 1997. Aspect-oriented programming. In Proceedings ECOOP'97, LNCS 1241, pages 220–242. Springer.

[11] U. Kulesza, C. Lucena, P. S. C. Alencar, A. Garcia, 2006. Customizing Aspect-Oriented Variabilities Using Generative Techniques.In Proceedings of the 18th International Conference on Software Engineering and Knowledge Engineering.

[12] Oracle. 2011. The Java Tutorials: Annotations. http://download.oracle.com/javase/tutorial/java/javaOO/annotations.html

[13] R. A. Rashid, R. Chitchyan. 2003. Persistence as an aspect. In *Proceedings of the 2nd international Conference on Aspect-Oriented Software Development* , ACM Press, pp. 120-129.

[14] D. R. Prasanna. 2009. *Dependency Injection* (1st ed.). Manning Publications Co., Greenwich, CT, USA.

[15] R. Ré. 2002. A Process for Framework Development from Reverse Engineering of Web-based Information Systems: Application to the Online Auction Domain. Instituto de Ciências Matemáticas e Computação da USP - São Carlos, Brasil (*in portuguese*).

[16] A. L. Santos, K. Koskimies, A. Lopes. 2008. Automated Domain-Specific Modeling Languages for Generating Framework-Based Applications. In *Proceedings of the 12th International Software Product Line Conference*, ACM Press, pp. 149-158

[17] Spring. 2010. Spring Framework. http://www.springsource.org.

[18] A. Yassin, M. E. Fayad. 2000. Domain-Specif Application Frameworks: Frameworks Experience by Industry, John Wiley & Sons.