

# An Approach to Design Crosscutting Framework Families

Valter Vieira de Camargo

Centro Universitário Eurípides Soares da Rocha  
Univem – Avenida Hygino Muzzi Filho, 529 – Cep  
17.525-901 – Cx. Postal 2041 – Marília – SP  
valtercamargo@hotmail.com

Paulo Cesar Masiero

Instituto de Ciências Matemáticas e de Computação  
ICMC-USP – Avenida do Trabalhador São carlense  
400 - Cep 13.560-970 – São Carlos – SP  
masiero@icmc.usp.br

## ABSTRACT

Crosscutting Frameworks are the most common type of aspect-oriented frameworks. They encapsulate only one crosscutting concern, such as persistence, distribution or synchronization. This kind of framework has been successfully used as infrastructure software providing services to higher level applications. However, as any common framework, when crosscutting frameworks are used for developing an application, usually all of their features and variabilities remain in the final architecture, even if only a subset of them was needed. This leads to a poor design with several pieces of death code, making the maintenance activities difficult. In this paper we present an approach to design a set of modules, called features, creating a family of Crosscutting Frameworks. So, during the developing of an application, just the features needed by the application are used, resulting in a clearer design, propitiating higher levels of maintainability and reusability. The approach is exemplified by means of a Family of Persistence Crosscutting Framework.

## Categories and Subject Descriptors

D.2.2 [Software Engineering]: Design Tools and Techniques – *Aspect-Oriented Framework Families*. [Software Engineering]: Reusable Software – *Aspect-Oriented Framework Families*.

## General Terms

Design, Standardization and Management

## Keywords

Aspect-Oriented Frameworks, Crosscutting Frameworks, Aspect-Oriented Frameworks Design, Architecture.

## 1. INTRODUCTION

The abstract mechanisms of Aspect-Oriented Programming (AOP) [13], such as abstract pointcuts, are essential to the implementation of crosscutting concerns in a generic way, propitiating the reuse of these concerns in other contexts. Usually, when a concern is implemented in a generic way it is called “aspect-oriented framework” (AOF) because it can be reused during the development of an application [9][15][16][17][19][19]. In a previous work, a more specific term for this kind of framework was suggested, namely “Crosscutting Framework” (CF) [5]. A CF is a framework which encapsulates only one crosscutting concern in a generic way. There is a large number of CFs proposed in literature which play the role of infrastructure software, such as persistency [16][17][19], concurrency [9] and security [15][18][19].

As in any object-oriented framework [10], a CF has variabilities which must be chosen during the reuse process. Another characteristic of CFs shared with conventional frameworks is that the more variabilities the CF has, the greater the possibilities of reusing it in other contexts. So, usually a framework implements

many variabilities to increase its reuse levels, resulting in complex architectures. Another similarity with object-oriented frameworks, which is actually a problem, is that the application developed with framework support includes the whole framework, i.e., the final architecture contains the application modules and the framework modules. This is a problem when the application to be developed requires only a subset of the framework variabilities – which is actually a common situation. Therefore, instead of using only the subset of the variabilities, all of the available variabilities are included into the final architecture of the application. Therefore, we can find several parts of the code unused, resulting in a poor design and causing maintenance, evolution and reuse problems [2][4][3][8].

Although the term “variability” is well-known in framework domain, in this work we have decided to define, in a more precise way, the terms “variability” and “feature”. For us, variability is implemented as a set of small alternative choices which determine a specific function of the framework, i.e., it turns a specific function on or off [10][3]. Usually, they are implemented by known design patterns such as Strategy, Factory Method and Template Method [11]. A feature is a “bigger” functionality of the domain which deserves to be modularized as an independent module [12]. It is possible, for example, for a feature to have its own internal variabilities. For example, in the Security domain we can have a feature called “Wrong Tentative Control” aiming at blocking a user account if he/she types the username or password erroneously. This feature can have the variabilities “control by time” – which blocks the user account if he/she gets three errors within a specific period, or “control by tentative number” – which blocks the user account if he/she got a specific number of errors independently of the period.

Our proposal is to divide the CF architecture into several “reusable features” which can be composed (or weaved) to generate a more restricted framework, i.e., a framework that only owns the features required by the application, avoiding features not used in the final architecture. Our approach can be seen as a family of frameworks, once several “members” can be built from the set of features available. For example, a family of persistence CFs includes several features that can be composed to generate a member; we can build a framework (member) with pooling, caching and connection features, or yet a member with only the pooling feature, or another member with only the caching feature, etc. The approach presented in this paper is exemplified by means of a family of Persistence CFs but the idea can be generalized to other CFs.

A characteristic of our approach is that some features can be implemented in a generic way to allow their reuse out of the original framework family. For example, the feature Pooling can be used “as is” inside the framework family in order to generate a new framework, but it can also be used in another context as long

as some abstract pointcuts are concretized. Therefore, we saw three important contributions of our approach. The first one is to improve the quality of the final application architecture as we prevent some not used features from remaining in the final architecture; the second one is to improve reuse levels since the features can also be reused out of the original family and the third one is the set of provided preliminary guidelines that can be used by other frameworks engineers.

The paper is organized as follows: the concept of CF is described in Section 2. Our persistency CF family and an example of its use are shown in Section 3. Guidelines obtained from our experience with the CF Family development are provided in Section 4. Related Work is shown in Section 5 and Final Remarks are found in Section 6.

## 2. CROSSCUTTING FRAMEWORKS

Crosscutting Frameworks are a kind of AOFs which encapsulate generic behavior of only one crosscutting concern. They have abstract composition mechanisms and functional variabilities [5]. CFs can be classified as “context-dependents” and “context-independents”. The context-independent CFs can be easily coupled to any base-code as they do not require specific details from the join points, i.e., they do not need to obtain any object or value from the base code. Usually, CFs that implements Log and Tracing concerns fit into this category. Context-dependent CFs need to obtain an object or value from the base code to work properly. An example of a CF that fits into this category is Access Control – it needs to obtain user authentication data (user name and password) to verify whether the users may access a specific system functionality. This type of CFs is more complex and must be designed with a pattern that provides several composition alternatives [6].

The reuse process of a CF has one more step compared to standard frameworks. Whilst in standard frameworks there is only the “instantiation” step, which consists of choosing the variabilities required by the application, the reuse process of a CF also includes the “composition” step, which consists in coupling the CF to a base code (an application, another CF, or even another concern implemented in a conventional way). This composition process usually consists in providing join points to abstract pointcuts.

We can also classify the CFs in “application-level CF” and “lower-level CFs”. The application-level CF is coupled directly to an application, i.e., they select join points of an application. For example, a CF of Connection acts directly in the control flow of an application because it needs to crosscut some join points to establish the connection with the database. Low-level CFs are not coupled to an application but to other CFs or even in other software modules. For example, a Pooling CF must be coupled to a join point, which can capture an object of type Connection; so it must be coupled to the Connection CF, once it is the only place where this object can be obtained.

## 3. A PERSISTENCE CF FAMILY

In this section, our persistence CFs family and an example of its use, generating a member from it, are shown. In Figure 1, a feature diagram [12] denoting the persistence CF family is shown. This family allows the creation of several different members or CFs. Each feature denoted by an empty box is a software package containing classes and aspects of a specific concern or sub-concern. Features labeled with “CF” are generic, i.e., they are CFs. Those that have no labels are features specific to the family, i.e.,

they are not generic and shall not be reused in other contexts, other than the family. Features denoted by gray boxes are only groups and do not have a software representation; they are just conceptual and represent a group of alternatives (stereotype <<exactly-one-of-feature group>>) or optional (stereotype <<zero-or-more-of-feature group>>) features. The “mutually-includes” tag means that when a specific feature is chosen, the other pointed out by the arrow is automatically included. The “requires” tag means that the selection of a feature, requires that the other feature pointed out by the arrow must be in the project before this selection.

There are two mandatory features, denoted by the <<common feature>> stereotype, which must be present in all members of the family. The first one, called “Persistence”, aims to introduce a set of persistence operations into application persistent classes. These operations can be used to store, remove, update and perform queries in the database. The second one, called “Connection”, refers to the database connection concern and identifies points in the application code where the connection must be opened and closed. This feature has variabilities, as for example the “connection type” (JDBC, ODBC or native driver) and the “database paradigm” (Relational or OO). These two variabilities must be chosen during the reuse process. The first feature (Persistence) depends on the second (Connection) to run the persistence operations; it is represented by the “mutually includes” tag. These two mandatory features are also referred to as “family main modules” or “family core”, as they must be present in all members of the family. They are also the only application-level features of the family because their coupling must be done directly to an application.

The Partial Awareness and Total Awareness features are not generic. When the former is used, the application engineer does not need to be aware of methods for storing and updating because the CF is responsible for executing them in specific points at application runtime. Nevertheless, when the Total Awareness feature is used, the application engineer must know all the persistence methods as well as the points in which these methods must be called. The Dirty Objects Controller feature is not generic either; it is an aspect that controls objects modified in the memory, for example: when this feature is used, any query to the database is preceded by a test verifying if the object was modified in the memory and needs to be updated in the database before the query is concluded. The Pooling feature is generic, i.e., it is implemented as a CF and can be used out of the family. Its objective is to improve the application performance by storing Connection objects in a repository. The three Cache features – grouped under the Cache group – are also CFs and provide three ways of improving the performance of the CF. The Policy Enforcement feature, which was implemented as a CF too, can be used to test whether some standardization policies required by the persistence CF are present in the application to which the framework will be coupled. All of the features commented in this paragraph are low-level features once their coupling must be done with the Persistence and Connection application-level features.

As previously said, one objective was to implement some features in a generic way in order to promote their reuse out of the family. However, despite of the advantages of implementing features in a generic way (using abstract methods and pointcuts, for example), this type of implementation makes their reuse inside the CF family more difficult. This happens because their coupling to the main

modules of the CF family (Persistence and Connection features) or even with other features will need some extra-code, such as: code to concretize an abstract pointcut or abstract method or code to address crosscutting precedence. However, as the probability of reusing the feature inside the family is larger than reusing it outside, it is a good design decision to provide one or more “implementation layers” containing this extra-code.

The aim of this layer is to group up the source code responsible for coupling the feature to the family core or to other features, here called “composition code”, as well as the code that defines default variabilities, called here “instantiation code”. It is possible to develop this layer in advance because the points where the low-level features must be coupled are previously known, as well as the default functional variabilities that must be used inside the family. For example, it is possible to know in advance the join points of the core family (Persistence and Connection features) where the Pooling feature must be coupled as well as the default

variability of this feature. Moreover, this layer must also have all the interference and inconsistency treatment code that the coupling can generate, for example, the code to address the precedence order of pointcuts. Clearly, the implementation layer must not be used when the feature must be coupled in some base code out of the family. In this work, this implementation layer is called “composition layer”. Therefore, the coupling process of the Pooling, Cache and PolicyEnforcement optional features to the core family does not require the software engineer to provide some code, as this is ready in the composition layer.

The composition layer of the two mandatory features (Persistent and Connection) does not have the “composition code” pre-defined, as it is impossible to anticipate the join points of the application in which these features will be coupled. However, the composition layer of the Connection includes the “instantiation code”, defining a variability default as ODBC Connection.

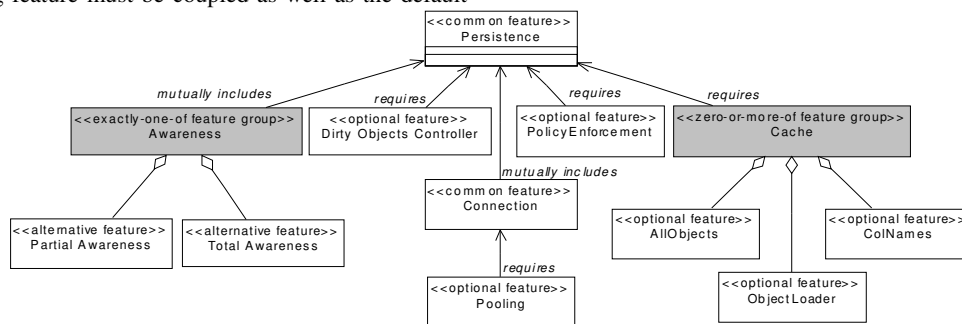


Figure 1 – Feature Diagram of the Persistence CFs Family

### 3.1 Building a Member

In this section, the CF creation process and its coupling to an application are shown. Suppose that the requirements of the application require the Persistence CF to have a connection repository and that the software engineer is responsible for programming only the removal and query methods. Besides, the connection must be performed by ODBC and the database must have MySQL. Therefore, the CF must have the Persistence and Connection features, which are mandatory, the Pooling optional feature as well as the Partial Awareness alternative feature.

An important characteristic of a CF family are the alternatives to conduct the reuse process, giving more flexibility to the development. There are two alternatives that differentiate themselves through the time in which the member is built and coupled to the application. The first one occurs initially when all the features that must be part of the CF are composed and after that the coupling to the application is performed. The second one consists initially in coupling only the family core (mandatory features) to the application and after that the remaining features are coupled to core features. These two ways are possible because the lower level features affect only the family core or some other features inside the family, but not the application itself. Thus, it is possible to couple the features at any time during the development or even when the application is running. The first alternative must be used when all of the features required by the application are previously known. However, if that is not the case, but the basic behavior of the concern is needed to carry on the development or to conduct tests, it is interesting perform the coupling only to the family core and afterwards put the other required features together. We will show only the the second type in this section.

The reuse process of the Persistence feature consists in creating a concrete aspect extending the `PersistentEntities` abstract aspect and provides application class names that must have persistent operations. This is not shown because of space limitations. The reuse process of the Connection feature also consists in creating a concrete aspect indicating the application points where the connection must be opened and closed. An example of a concrete aspect like this is shown in Figure 2. In this aspect, the join points provided to the pointcut `openConnection()` specify the points where the connection must be opened. Moreover, the join points provided to the `closeConnection()` pointcut specify the points where the connection must be closed.

After coupling the family core to the application, the Pooling feature can be coupled. Figure 4 shows a diagram representing the Pooling feature which aim is to suspend the creation of connections, preventing the creation of a new connection if there is at least one available at the repository. If there is no connection available, the control is returned to the base code enabling it possible to create a new one. However, the feature obtains this object of type `Connection` and stores it at the repository to be used later on. The pooling sub-concern is context-dependent. Therefore, it must obtain the `Connection` object from the base code to work properly. Hence, it must be coupled to the `Connection` feature, as this feature is responsible for controlling the connections with the database.

The aspect hierarchy in Figure 3 consists of the Data Catcher Pattern [6] which idea is to provide some alternatives to obtain the data, instead of providing a single way to do this. For example, the pooling concern needs to obtain the `Connection` object, and

therefore the CF must allow this object to be obtained by a number of ways to increase the reuse possibilities of this CF out of the family. Each of the aspects marked with an asterisk (\*) in

```

public aspect MyConnectionComposition
    extends ConnectionComposition {

public pointcut openConnection():
    execution (* srvCadaastro*.init(..)) ||
    execution(* srvRegisterOfEmployees.init(..));

public pointcut closeConnection():
    execution
    (* srvRegisterOfEmployees.destroy(..) ||
    execution
    (* srvRegisterOfFunction.destroy(..)); }
    
```

Figure 2 – Aspect for Coupling the Connection feature

To use this feature in its family, the software engineer only needs to add it to the project. It is possible because this feature has a composition layer. The model shown in Figure 3 uses this layer to define the composition alternative *ThisPooling\_I* as the default, which can be noted by the concrete aspect *DefaultPooling*.

In Figure 4, the *Default\_Pooling* aspect, which represents a simple composition layer, is shown. This aspect is responsible for the composition with the feature *Connection*. It must be noticed that the *connect()* method of the *OdbcConnection* class is the join point used to concretize the abstract *poolPushPop()* pointcut. This method is provided as a default join point because previous analysis has shown that this is the most appropriate join point to obtain the *Connection* object required by the *Pooling* CF. As the *Default\_Pooling* aspect extends the *ThisPooling\_I* aspect, the object obtained from the execution context is “This”. This object will be used internally by the framework to obtain the *Connection* object required by the *Pooling* feature.

In the *Default\_Pooling* aspect, some abstract methods are concretized to provide specific details of the base code. The *getMethodResponsibleForSettingTheConnection()* and *getMethodResponsibleForGettingTheConnection()* methods are used internally by the CF to obtain the *Connection* object from the execution context reflexively. If the *Pooling* feature is coupled to another base code, these methods should be concretized by the application engineer providing information on the base code. To reuse this feature in other contexts, this package should not be included in the project.

Now that the process of coupling the mandatory and optional features has been finished, it is important to notice that the *Persistency* CF coupled in the application contains only the features required by the application, avoiding that all other features shown in Figure 1 remain in the final code. So, we end up with a design clearer than if all of other features had remained in the architecture. So, the benefits resultant from the use of *Framework Technology*, such as the reuse and productivity, are balanced with the drawbacks imposed by a complex architecture.

the lower part of the diagram represents a composition alternative. More details of this pattern exceeds the scope of this paper [6].

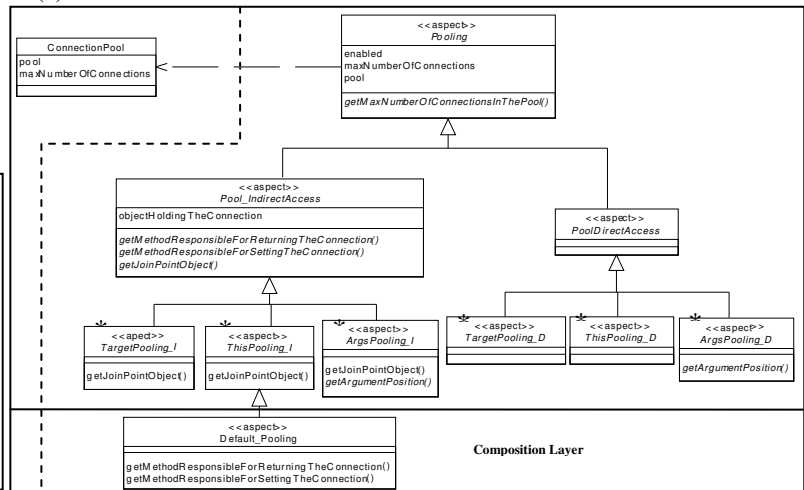


Figure 3 – CF of Pooling

Although some unused variabilities (not features) still remain in the final architecture, for example the variabilities (*Connection Type* and *Database*) of the *Connection*, the maintenance and reuse levels are not so damaged because these variabilities are small and well designed with design patterns [11]. Note that the aim of this work is to decrease the amount of unused code into the final architecture.

```

public aspect Default_Pooling extends ThisPooling_I {
    public pointcut poolPushPop():
        execution (public void OdbcConnection.connect(..));
    .. StringgetMethodResponsibleForReturningConnection() {
        return "getConnection"; }
    .. StringgetMethodResponsibleForSettingConnection() {
        return "setConnection"; }
    public int getMaxNumberOfConnectionsInThePool() {
        return 5; }}
    
```

Figure 4 – Hierarchy of DefaultPooling Aspect

## 4. PRELIMINARY GUIDELINES

The experience with the development of the *Persistence* CF shown in Figure 1 and with the development of a *Security* CF Family led us to create a set of preliminary guidelines to design CF Families:

**1. Find Domain and Extra Sub-Concerns.** Domain sub-concerns are pieces of functionality inside a concern domain. In our CF Family the *Persistence*, *Connection*, *Dirty Objects Control*, *Pooling* and *Cache* features are domain sub-concerns because they are recurring in the *Persistence* domain. Similarly, in the *Security* domain, the *Access Control*, *Authentication* and *Encryption* are domain sub-concerns. However, the *Awareness* and *Policy Enforcement* features are called “extra sub-concerns”. This type of feature supports or improves some CF characteristic and are not obtained from domain decomposition; they are obtained from technological, global constraints or contextual requirements. For example, the *Policy Enforcement* feature is a CF to verify if some politics required by the framework are present in the application. This feature is needed only because the *Persistence* CF imposes these policies.. After implementing, we

call domain sub-concerns “domain features” and extra sub-concerns “extra features”.

The aim of this step is to create a tree of sub-concerns, where each one will be implemented as a feature or as variability. The process to find domain sub-concerns consists in conducting an analysis in the concern domain and identifies its sub-concerns by decomposing each sub-concern several times. A domain sub-concern is a well-defined functionality or requirement within the concern domain. The identification of extra sub-concerns is guided by several types of requirements as previously said. For example, the Awareness feature shown in Figure 1 was created aiming at improving the application engineer productivity after we had verified that the CF reuse process could be improved. At this time, it is enough to make a tree of the sub-concerns identified in order to determine their dependencies (requires and mutually includes).

**2. Classify the Sub-Concerns.** The aim is to classify the concerns to build a feature diagram. It is common to find sub-concerns that can be classified as mandatory, alternative or optional [12]. Usually sub-concerns are classified as alternatives when they belong to the same group and just one must be chosen for each family member. The most real situation is when a new functionality is required in an evolution process and it must be added under a group that already has other features. The optional concerns are those that can be used independently of the presence of others.

**3. Analyze and Identify Sub-Concerns Level.** For each sub-concern identified it is important to define its level of granularity. We suggest classifying them into first level (application level), second level, third level and so on. For example, the sub-concerns Persistence and Connection shown in Figure 1 are classified as “first level” while the Pooling sub-concern must be classified as “second level” since it can only be coupled to first level features. If other sub-concerns were identified inside Pooling, they should be classified as “third-level” once they could only be coupled to second level sub-concerns, and so on.

**4. Determine the “Family Granularity Level”.** The family granularity level must be defined by the software engineer based on domain characteristics. This level is useful to decide how to implement a concern: as a feature or as variability. Note that the granularity level of our Persistence CF Family is two (2) since all of the features can only be coupled to the first level features: Persistence and Connection. This number suggests that all of the concerns in lower levels must be implemented as variabilities and not as features. For example, the Connection feature has sub-concerns that must be chosen by the application engineer. These sub-concerns were implemented as variabilities because its granularity level – inside the Persistence domain – is three. Two is a good number in the Persistence domain, because after that, the granularity of the sub-concerns will be too low.

**5. Generalize Concerns.** As shown in Figure 1, some sub-concerns were implemented as plain aspects and others as CFs, i.e., in a generic way. The criteria that support this decision is that sub-concerns must be implemented as CFs when they involve functionality with variabilities and possibilities to be reused in other contexts. Another reason that justifies implementing a sub-concern as a CF is when the concern is context-dependent [5]. This type of concern must be designed with a composition pattern proposed in another work [6]. The sub-concerns under the *Awareness* group can hardly be reused in other contexts because

they are specific to the family and do not have variabilities. Due to this, they were not implemented as CFs but as aspects. It is important to mention that sub-concerns can also be implemented only using objects when they are not crosscutting.

**6. Build the Composition Layers.** This step must be carried out for each generic feature of the family to provide a software module containing the “composition” and the “instantiation” code. One must analyze the family and identify the most suitable join points where the features must be coupled and implement aspects concretizing abstract pointcuts. Besides, it is also interesting to provide “instantiation code” in this module in order to define default variabilities. The `DefaultPooling` aspect shown in Figure 4 is a good example. It has as composition code, represented by the `poolPushPop()` concrete pointcut and instantiation code, represented by the concrete methods.

There are three scenarios to develop a CF Family and apply the guidelines shown in this section: i) from scratch; ii) based on previous decomposition or iii) on demand. The first one is the most difficult way because the objective is to build a complete CF Family from scratch. Usually this scenario occurs when a company wants to sell a product like this, so, it must think of all features previously. The second option consists in building the family based on some research. If the concern it is a classic concern is not difficult to find in the literature several works that has proposed such decomposition. The third approach is the most real because the software engineer only adds a new sub-concern if a new requirement is asked during the application evolution.

## 5. RELATED WORK

Various research groups have investigated the relation between AOP and Product Lines or Product Families [14][1]. None of the works mentioned is completely related to this paper, as they do not present a family of CFs or an approach to design such a family. The work more closely related to the one proposed in paper is presented by Batory *et al* [2]. It proposes a product line of frameworks, but in an object-oriented context. According to these authors, the traditional idea of dividing the framework architecture into reusable and instance specific code causes maintenance problems resultant from the complexity of the architecture. The proposal is to divide the framework architecture into parts that can be put together to generate a new framework containing only the functionalities required by the application. However they concentrate on application frameworks while here the generated frameworks are not ready to be used; they still need to be coupled to an application.

Mezini and Ostermann’s [14] approach includes a language called CAESAR [14]. In this language, crosscutting features can be defined in modules and coupled dynamically to a base code by means of pointcuts and advices. Furthermore, it is possible to generalize features to make them independent from the base code structure, enabling them to be reused in other contexts. Although the authors comment on the generalization of the features, they do not present clear guidelines of how it can be done. Another difference is that Mezini and Osterman concentrate on conventional product lines. In the approach presented by Apel *et al*. [1], aspects can be refined to increase or decrease the scope of a pointcut.

## 6. FINAL REMARKS

In this paper we show an approach to design CFs Families. The approach allows the creation of concrete and generic features that can be composed to generate a more restrict CF owning just the features required by the application to be developed. The use of this approach avoid that not used features remain in the final application architecture resulting in better maintenance levels.

The adoption of our approach inserts one more step to the development process: when someone decides to use a CF to support the development of an application, one must to build the CF before it can be used. This step is not necessary in other framework-based approaches, but the final architecture includes the whole framework

Someone could argue that CFs are small and the complexity and quantity of features they implement is low to justify the approach presented in this paper. However, CFs are usually used together with other CFs to develop an application. Actually, CFs are more useful when several of them are available. Therefore, during a development, it is common to have several CFs like persistence, distribution, concurrency, security, etc. If each of them let some not used features in the final code, we end up with a lot of them not used and, as a result, a poor design

An interesting detail is that the persistence family in Figure 1 can also be considered a unique feature. This may occur if the persistence is visualized as a unique module that can be added to the core of some other family. Inside the Security Family, developed in previous work [5], the family shown in this paper can be characterized as a unique optional feature, as it can be added and removed easily.

It is important to mention that although only the family core have been extensively tested in other contexts (out of the family), the experience of developing the persistence and also a security CF Family pointed out that the optional features implemented as CFs – *pooling*, *policyEnforcement* and *Cache* – can also be easily reused out of the family.

As future work, we intend to reuse the generic features in other system domains to validate their reusability. Nowadays, we have a tool to manage the version control of the family members [7], but we intend to extend it. We also intend to conduct a deeper study to extend the guidelines and define more precise criteria to support the decision of implementing a concern as a feature or as variability.

## 6. ACKNOWLEDGMENTS

This work is supported by CAPES and CNPq: Projeto PROSUL-Proc (Latin-AOSD) 490478/2006-9.

## REFERENCES

- [1] Apel, S., Leich, T., Saake, G. Aspectual Mixin Layers: Aspects and Features in Concert. In: Proceedings of ICSE (International Conference on Software Engineering), 2006.
- [2] Batory, D., Cardone, R., Smaragdakis, Y. Object Oriented Frameworks and Product Lines. In 1<sup>st</sup> Software Product Lines Conference (SPLC1), 2000.
- [3] Bosch, J, Mattsson, M., and Fayad, M. Framework Problems, Causes, and Solutions, CACM, 1998.
- [4] Braga R.T.V.; Masiero, P.C. A Process for Framework Construction Based on a Pattern Language. In: Proceedings of Annual International Computer Software and Application Conference (COMPSAC 2002), IEEE, Inglaterra, 2002.
- [5] Camargo, V.V., Masiero, P.C. “Aspect-Oriented Frameworks”. In: Proceedings of the 19<sup>th</sup> Brazilian Symposium on Software Engineering (SBES’2005), Uberlândia-MG, Brazil, outubro, 2005. (In Portuguese)
- [6] Camargo, V.V., Masiero, P.C. A Composition Pattern to Design Crosscutting Frameworks. In Proceedings of the ACM Annual Symposium on Applied Computing (ACM-SAC’08), Fortaleza, Brazil, 2008. (accepted for publication).
- [7] Arimoto, M., Cagnin, M, I., Camargo, V.V. Version Control in Crosscutting Framework-Based Development. In Proceedings of the ACM Annual Symposium on Applied Computing (ACM-SAC’08), Fortaleza, Brazil, 2008. (accepted for publication).
- [8] Codenie, W., Hondt, K., Steyaert, P., Vercammen, A. From Custom Applications to Domain-Specific Frameworks. Communications of the ACM, 40(10), October, 1997.
- [9] Constantinides, C. A.; Skotiniotis T.; Elrad T. Providing Dynamic Adaptability in an Aspect-Oriented Framework. In: Workshop on Advanced Separation of Concerns ECOOP 2001. Budapest, Junho 17-18, 2001.
- [10] Fayad, M. E., Schmidt, D. C., Johnson, R. Building Application Frameworks: Object-oriented Foundations of Framework Design. John Wiley & Sons, 1999.
- [11] Gamma, E., Helm, R., Johnson, R., Vlisside, J. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.
- [12] Gomaa, Hassan. (2004) Designing Software Product Lines With UML – From Use Case to Pattern-Based Software Architectures. Addison Wesley, 1a. Edição, 2004.
- [13] Kiczales, G.; Lamping, J.; Mendhekar, A.; Maeda, C.; Lopes, C.; Loingtier, J.; Irving, J. “Aspect Oriented Programming”. In: proceedings of ECOOP. pp. 220-242, 1997.
- [14] Mezini, M., Ostermann, K. Variability Management with Feature Oriented Programming and Aspects. In: SIGSOFT/FSE, ACM, 2004.
- [15] Mortensen, M., Ghosh, S. Creating Pluggable and Reusable Non-functional Aspects in AspectC++. In: Proc of the 5<sup>th</sup> Workshop on Aspects, Components and Patterns for Infrastructure Software (ACP4IS’06). Workshop of the AOSD Conference, Bonn, Alemanha, 2006.
- [16] Rashid, A., Chitchyan, R. “Persistence as an Aspect”. In: proc. of 2nd International C. on Aspect Oriented Software Development(AOSD) Boston–USA, March, 2003.
- [17] Soares, S., Borba, P., Laureano, E. Distribution and Persistence as Aspects. Software: Practice & Experience, v. 36, n. 6, p. 711-759, 2006.
- [18] Vanhaute, B., de WIN, B., Decker, B. Building Frameworks in AspectJ. In: Proc. of the 15<sup>th</sup> European Conference on Object-Oriented Programming (ECOOP), Separation of Concerns Workshop. pp. 1-6, June, 2001.
- [19] Zubairov, R., Hanenberg, S., Unland, R. Modularizing Security Related Concerns in Enterprise Applications – A Case Study with J2EE and AspectJ. In: Proceedings of NetObjectDays and GSEM, Erfurt, September, 19-22, LNI p-69, Gesellschaft für Informatik, 2005.