Implementing Framework Crosscutting Extensions with EJPs and AspectJ

Uirá Kulesza[1], Roberta Coelho[1], Vander Alves[2], Alberto Costa Neto[2], Alessandro Garcia[3], Carlos Lucena[1], Arndt von Staa[1], Paulo Borba[2]

[1]Departamento de Informática – Pontifícia Universidade Católica do Rio de Janeiro (PUC-Rio)
{uira, roberta,lucena,arndt}@inf.puc-rio.br

[2] Centro de Informática – Universidade Federal de Pernambuco
{vra, acn, phmb}@cin.ufpe.br

[3]Lancaster University, Computing Department, Lancaster - United Kingdom
garciaa@comp.lancs.ac.uk

Abstract. *In a previous work, we proposed a framework extension approach based on the use of a new concept, called Extension Join Points (EJPs). EJPs enable the framework systematic extension by means of variability and integration aspects. In this paper, we show how EJPs can be implemented using the mechanisms of the AspectJ language. To evaluate the usefulness of the EJPs in the framework extension process, we have used them in the development of three OO frameworks from different domains. As a result of our case studies, we present: (i) an initial categorization of different kinds of contracts between frameworks, EJPs and aspects which can be implemented in AspectJ; and (ii) a set of lessons learned when specifying the EJPs.*

Resumo. *Uma abordagem para extensão de frameworks baseada em um novo conceito, denominado Extension Join Points (EJPs), tem sido proposta anteriormente. EJPs possibilitam a extensão sistemática de frameworks, através do uso de aspectos de variabilidade e integração. Neste artigo, nós mostramos como os EJPs podem ser implementados usando os mecanismos da linguagem AspectJ. Para avaliar a utilidade dos EJPs no processo de extensão de frameworks, nós os utilizamos no desenvolvimento de 3 frameworks OO de diferentes domínios. Como um resultado de nossos estudos de caso, nós apresentamos: (i) uma categorização inicial de diferentes tipos de contratos entre frameworks, EJPs e aspectos, os quais podem ser implementados em AspectJ; e (ii) um conjunto de lições aprendidas quando especificando os EJPs.*

1. Introduction

Object-oriented (OO) frameworks [11] represent nowadays a common and important technology to implement program families. They enable modular, large-scale reuse by encapsulating one or more recurring concerns of a given domain, and by offering different variability and configuration options to the target applications. In the framework based development, applications are implemented by reusing the architecture defined by the frameworks and by extending their respective variation points or hot-spots [11]. Hence, the adoption of the framework technology brings in general significant productivity and quality in the development of applications. Besides their advantages, some researchers [5, 8, 23, 24, 28] have recently described the inadequacy of OO mechanisms to address the modularization and composition of many framework features, such as, optional [5], alternative and crosscutting composition features [23, 24]. As a consequence, the limited modularity provided by the OO mechanisms brings difficulties to configure many framework features for specific needs, thus impeding the framework adaptation and reuse [5, 8, 23, 24, 28].

Aspect-oriented software development (AOSD) [12, 17] has been proposed as a technology which aims to offer enhanced mechanisms to modularize crosscutting concerns. Crosscutting concerns are concerns that often crosscut several modules in a software system. AOSD has been proposed as a technique for improving the separation of concerns in the construction of OO software, supporting improved reusability and ease of evolution. Recent work [2, 18, 19, 20, 21, 25, 27, 31] has explored the use of aspect-oriented (AO) techniques to enable the implementation of flexible and customizable software family architectures. In these research works, aspects are used to modularize crosscutting variable (optional or alternative) and integration features. In a previous work [19], we have proposed an approach which aims to improve the extensibility of object-oriented frameworks using aspect-oriented programming. Our approach proposes the definition of extension join points in the framework code, which can be extended by means of variability and integration aspects. These aspects are responsible to implement optional, alternative and integration features in the framework. Since the aspects can be automatically unplugged from the framework code, our approach makes easier to customize the framework to specific needs.

This paper shows and evaluates how the framework extension join points (EJPs) from our approach can be implemented in the AspectJ language. The EJPs codification in AspectJ gives us the advantages of explicitly exposing some framework join points and writing contracts that must be satisfied when extending those join points. Hence, it gives more systematization and robustness for our approach in the process of framework extension. To evaluate the usefulness of the EJPs in the framework extension process, we have used them in the development of three OO frameworks from different domains. As a result of our case studies, we present: (i) an initial categorization of different kinds of contracts between frameworks, EJPs and aspects which can be implemented in AspectJ; and (ii) a set of lessons learned when specifying the EJPs.

The remainder of this paper is organized as follows. Section 2 presents background by detailing framework modularization problems addressed by our approach and by introducing AOSD basic concepts. Next, Section 3 gives an overview of our approach for framework development with aspect-oriented programming based on the specification of EJPs. Section 4 then details how our EJPs can be implemented using AspectJ, including the specification of their contracts. This section also presents a categorization of contracts that need be defined when adopting our approach. Subsequently, Section 5 illustrates the implementation of EJPs using AspectJ for two different case studies. Section 6 presents the lessons learned from our case studies. Related work is discussed in Section 7. Finally, Section 8 summarizes our contributions and provides directions for future work.

2. Background

This section briefly revisits research work that describes the inadequacy of object-oriented mechanisms to modularize specific framework features. We also present the basic concepts of AOSD and discuss emerging aspect-oriented design approaches.

2.1 Issues in Modularizing Framework Features

Despite the well-known benefits of OO frameworks in implementing program families, recent research has exposed the inadequacy of framework technology in modularizing features with particular properties, such as optional [5] and crosscutting composition [23, 24] features. These issues hinder the framework instantiation process to meet specific user needs. As a result, framework reuse can become unmanageable or even impracticable. Next, we describe these two problems of framework feature modularization.

Modularizing Optional Framework Features. Batory et al [2] address the issues of the framework technique in modularizing optional features. An optional feature is a framework functionality that is not used in every framework instance. According to such research, developers typically deal with this problem either by implementing the optional feature in the code of concrete classes during the framework instantiation process, or by creating two different frameworks, one addressing the optional feature and the other one without it. As a result, many framework modules are replicated just for the sake of exposing optional features, thus leading to "overfeatured" frameworks [8], in which several instance-specific functionalities can be present.

By analyzing a number of available frameworks (such as JUnit and JHotDraw), we note that the most widespread practice in implementing framework optional features is the use of inheritance mechanisms to define additional behavior in the framework classes. In the JUnit framework, for example, inheritance relationships are used to define a specific kind of test case as well as additional and optional extensions to test cases and suites.

Crosscutting Feature Compositions in Frameworks Integration. Mattsson et al [23, 24] have analyzed the issues in integrating OO frameworks and proposed several OO solutions. Their research relates the composition of two frameworks to the composition of a new set of features (represented as a framework) in the structure of another framework. For example, suppose we need to extend the JUnit framework to send specific failures that occur to software developers. A specific test failure report could be send by e-mail to different software developers, every time a specific and critical failure happens. Imagine we have available an e-mail framework to support our implementation. The problem here is how we could implement this functionality in the JUnit framework. It involves the integration of the JUnit and the e-mail framework. This composition could be characterized as crosscutting since we are interested to send a failure report by e-mail during the execution of the tests.

Based on a case study [20] with feature compositions involving four OO frameworks of varying complexity and addressing concerns from distinct horizontal and vertical domains [10], we have concluded that the framework integration solutions presented by Mattson et al [23, 24] are invasive and bring several difficulties to the implementation, understanding, and maintenance of the framework composition code. Our analysis has shown that 6 out of 9 solutions described by those authors have poor modularity and a crosscutting nature, requiring invasive internal changes in the framework code.

2.2 Aspect-Oriented Software Development

Aspect-oriented software development (AOSD) [12, 17] is an evolving approach aiming at modularizing concerns, which existing paradigms are not able to capture explicitly. It encourages modular descriptions of complex software by providing support for cleanly separating the basic system functionality from its crosscutting concerns. Crosscutting concerns are concerns that often crosscut several modules in a software system. AOSD supports the modularization of crosscutting concerns by providing abstractions to extract these concerns and later compose them back when producing the overall system. AOSD proposes the notion of aspect as a new abstraction and provides new mechanisms for composing aspects and components (classes, methods, etc.) together at specific join points.

AspectJ [3] is an aspect-oriented extension to the Java programming language. The aspect abstraction in AspectJ is composed of inter-type declarations, pointcuts and advices. Pointcuts have a name and are collections of join points. Join points are well-defined points in the dynamic execution of system components. Examples of join points are method calls and method executions. Advice is a special method-like construct attached to pointcuts. Advices are dynamic crosscutting features since they affect the dynamic behavior of components. Inter-type declarations specify new attributes or methods to be introduced in specific classes. In this work we will focus on the use of aspect-oriented abstractions to modularize framework extensions which implement optional, alternative or crosscutting composition features.

2.2.1 Obliviousness and Crosscutting Interfaces (XPIs)

Filman and Friedman [13] have identified two properties, *quantification* and *obliviousness*, which they believe are fundamental for aspect-oriented programming. The *Quantification* property refers to the desire of programmers to write programming statements with the following form: "*In programs* P, *whenever condition* C *arises*, *perform action* A". The AspectJ programming language, for example, supports this property by means of the pointcut, join point and advice mechanisms described above. Obliviousness establishes that programmers of the base code – the classes which will be affected by the aspects – do not need to be aware of the aspects which will affect it. It means that programmers do not need to prepare the base code to be affected by the aspects. The following sentence from the authors synthesizes both properties [13]: "*AOP can be understood as the desire to make quantified statements about the behavior of programs, and to have these quantifications hold over programs written by oblivious programmers.*"

In a recent study, Sullivan et al [30] have compared the obliviousness methodology with a new approach to AO development based on *design rules* [4]. In their approach, the authors propose the specification of interfaces between the base code and the aspects, which determine the anticipated definition of join points from the base code before its implementation. These join points are used subsequently in the implementation of the system aspects. The design rule based approach [30] addresses the decoupling of the base and aspect code by offering a clear specification of the interaction and contracts between them and by allowing their parallel development. In the study, the authors have also observed how their approach helps to reduce or eliminate several disadvantages of the obliviousness approach, such as, the codification of complex and fragile pointcuts expressions and the tight coupling of the aspects to changeable and complex details from the base code.

Griswold et al [16] have recently shown how the interfaces between the base code and the aspects, called crosscutting interfaces (XPIs) and previously proposed by the design rules based approach, can be partially implemented in AspectJ. The XPIs are used to abstract a crosscutting behavior existing in the base code. The implementation of XPIs in AspectJ is composed of: (i) *a syntactic part* – which allows to expose specific join points by specifying pointcuts in aspects; and (ii) *a semantic part* – which details the meaning of the exposed join points and it can also define constraints (such as, pre- and post-conditions) that must be satisfied when extending those join points. This semantic part can be partially implemented with enforcement aspects (implemented with `declare error` and `declare warning` AspectJ constructs) [9] or by defining contract aspects which guarantee specific constraints are satisfied before and after the advices execution.

The definition of XPIs has inspired the central idea of our approach to extend object-oriented frameworks by exposing a set of extension join points (EJPs) present in their implementation. Next section gives an overview of the approach. Section 4 details our study of implementation of framework EJPs in AspectJ.

3. An Approach to Extending OO Frameworks with Aspects

This section gives an overview of our framework development approach [19]. Section 4 details our study of realization of the approach using AspectJ.

3.1 Extension Join Points (EJPs)

In our approach, an OO framework specifies and implements not only its common and variable behavior using OO classes, but it also exposes a set of extension join points (EJPs) which can be used to also extend its functionality. Similar to XPIs [16, 30], EJPs establish a contract between the framework classes and a set of aspects extending the framework functionality. Unlike XPIs, however, EJPs aims at increasing the framework variability and integrability. Accordingly, we propose to use the XPI concept in the framework development context, in which EJPs serve three different purposes:

(i) to expose a set of framework events that can be used to notify or to facilitate a crosscutting integration with other software elements (such as, frameworks or components);

(ii) to offer predefined execution points spread and tangled in the framework into which the implementation of optional features can be included;

(iii) to expose a set of join points in the framework classes that can have alternative implementations of a crosscutting variable functionality.

In this context, EJPs document crosscutting extension points for software developers that are going to instantiate and evolve the framework. They can also be viewed as a set of constraints imposed on the whole space of available join points in the framework design, thereby promoting safe extension and reuse. A key characteristic of EJPs is that framework developers and users do not need to learn totally new abstractions to use them, as they can mostly be implemented using the mechanisms of AOP languages (Section 4).

3.2 Framework Core and Extension Aspects

Our approach promotes framework development as a composition of a core structure and a set of extensions. A framework extension can define one of the following: (i) the implementation of optional or alternative framework features; or (ii) the integration with an additional component or framework. The composition between the framework core and the framework extensions is accomplished by different types of *extension aspects*, each one defining a crosscutting composition with the framework by means of its exposed EJPs. We next describe the main concepts of our approach:

(i) *framework core* implements the mandatory functionality of a software family. Similar to a traditional OO framework, this core structure contains the frozen-spots that represent the common features of the software family and hot-spot classes that represent non-crosscutting variabilities from the domain addressed;

(ii) *variability aspects* implement optional or alternative features existing in the framework core. These elements extend the framework EJPs with any additional crosscutting behavior;

(iii) *integration aspects* define crosscutting compositions between the framework core and other existing extensions, such as an API or an OO framework. These elements also rely on the EJPs specification to define their implementation.

The design of an OO framework with aspects following our approach is shown in Figure 1. According to this figure, both variability and integration aspects intercept only join points matched by pointcuts in the EJPs provided by the framework; further, such aspects must comply with all the constraints defined by the EJPs. This brings systematization to the framework extension and composition with other artifacts, providing a number of benefits [19], such as enhanced understandability and evolution of the framework core, safe framework reuse, and pluggable/unpluggable crosscutting framework extensions.

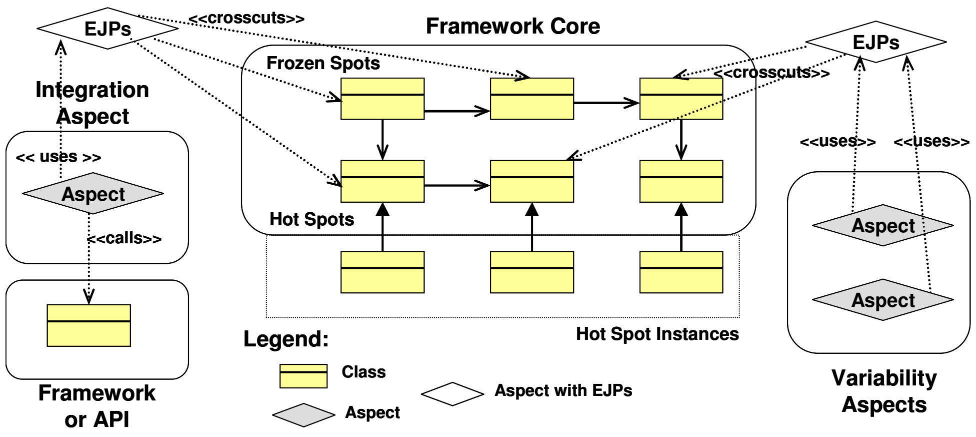

Figure 1. Elements of our Framework Development Approach

4. Implementing Extension Join Points with AspectJ

In this section, we explore the use of AspectJ language to specify the framework extension join points. The EJP codification in AspectJ language brings the following advantages to the framework extension process: (i) it enables the developer to expose a set of join points that are spread in the framework in a single aspect, that can be used to extend the framework functionality with integration and variability aspects; and (ii) it allows the representation of many constraints – that must be satisfied when extending those join points – in a way that they will not just be stated but they will be enforced during compilation and runtime. Next sections detail how we have implemented our EJPs in AspectJ.

4.1 EJPs Structure

The way we codified the EJP in AspectJ-style was inspired in the way Griswold et al [16] codified the XPIs. Each EJP is represented by an aspect comprising a set of pointcut descriptors that represents the set of extension join points of a framework. The EJP constraints which regulate the relationships between the framework, EJPs and extension aspects (mentioned in Section 3.1) are represented, in our approach, by separate aspects. However, we have defined a different methodology from the proposed by Griswold et al [16] to specify these constraints. We have classified them in the following categories: (i) *framework internal contracts* - contracts between the framework and its EJPs – and (ii) *framework extension contracts* - contracts between the EJPs and its extension aspects. The next section describes in detail the kinds of contracts defined in our categorization. Table 1 presents the main elements which comprises an EJP in AspectJ.

4.2 EJPs Contracts

During the definition of the EJPs` contracts, we first categorized the kinds of contracts that should exist between the elements of our approach (Figure 1); we next evaluated different ways to specify them in AspectJ. In the following, we detail our categorization of contracts and the guidelines on their implementation.

The *framework internal contracts* define constraints whose purpose is to assure that framework refactorings and evolution do not affect the functionality of its extension aspects. They are classified in the following categories: (i) *structural* – which aims to guarantee the framework implements specific interfaces defined by the EJPs; and (ii) *behavioral* – which assures the framework EJPs comprises all and only the framework events (or states) that the EJP is intended to expose.

The *framework extension contracts* are used to assure that each extension aspect respects constraints and invariants of the framework. The following categories were defined: (i) *structural* – these contracts assure that aspects only extend the framework join points exposed by the EJPs; (ii) *behavioral* – specify the framework classes' methods that can be invoked by the extension aspects; and (iii) *invariants* – define specific pre- and pos-conditions that must be preserved before and after the execution of extension aspect advices.

Tables 2 and 3 present the EJP contracts categorization. They also show the different mechanisms of AspectJ that we have used to implement them. AspectJ offers several mechanisms that can be used to specify our different contracts. When choosing mechanisms for each contract type, we prefer static mechanisms to dynamic ones, since only the former can be verified in compilation time, which is the case of the `declare parents`, `declare error` and `declare warning` statements. Some kinds of contracts, however, depend on dynamic information to be implemented. For these specific cases (such as verification of framework invariants), we have used the `adviceexecution` pointcut designator of AspectJ, which allows to intercept the execution of advices. Next section details the specification of EJPs for our case studies, including the implementation of their respective contracts.

Element Name	Purpose
Name	Specifies the name of the EJP, and is represented by the aspect's name in AspectJ.
Scope	Defines all the framework elements that are "encapsulated" by the EJP. It is represented by an AspectJ pointcut descriptor using the within designator including all the packages that comprises the framework (a scope example can be seen in Figure 2).
Crosscutting Extension Points	Quantifies the framework join points that represent relevant events or transition states occurring during the execution of the framework functionalities.
Accessors	Defines a set of pointcuts whose goal is to act for an aspect like accessor methods acts for a class. They expose EJP-specific information, which is useful for the definition of EJP contracts, such as: • EJP main purpose: each EJP should have a main purpose which can be, for example, to expose a specific event or an abstract state of the system. • All exposed join points They are defined as protected because they should be used only by EJP contracts.
Framework Internal Contracts	These contracts constrain the framework developer to expose in the EJP all the events that are expected to be exposed and to implement (in the framework) any interface, which is necessary for the exposure of such events.
Framework Extension Contracts	These contracts regulate the interaction between extension aspects and EJPs. The internal and extension contracts are defined in a separate aspect, in AspectJ.

Table 1. EJP Main Elements

Contract Type	AspectJ Implementation
Structural	Specification of interfaces that must be implemented by framework classes. The obligation to implement these interfaces is assigned by the EJPs using the `declare parents` inter-type construction of AspectJ. The interfaces are also declared inside the aspects that represent the EJPs.
Behavioral	Implementation of enforcement policies guaranteeing that the extension join points are called only and in all appropriate places inside the framework. This contract can be specified using `declare warning` and `declare error` AspectJ statements.

Table 2. Framework Internal Contracts

Contract Type	AspectJ Implementation
Structural	This contract can not be implemented in AspectJ, due to a current limitation of the language which does not allow the developer to restrict specific join points to be affected. Hence, to assure that extension aspects can only extend the EJPs, the developers must follow the programming practice of using only pointcuts specified in the EJP aspects.
Behavioral	This kind of contract restricts the framework classes' methods that can be accessed inside the extension aspects. There are two different ways to specify it: (i) using `declare warning` and `declare error` AspectJ statements, which allow the static verification of policies; and (ii) by defining advices which intercept every advice execution that realizes calls to the framework classes' methods. The `adviceexecution()` pointcut designator is used to intercept the advices execution.
Invariants	This contract defines pre- and pos- conditions that must be assured before and after the advice execution. These contracts are also defined using `adviceexecution()` pointcut designator to intercept the advices execution.

Table 3. Framework Extension Contracts

5. Case Studies

We have conducted three different case studies in which we analyze the use and suitability of AspectJ language to codify our framework EJPs. We selected frameworks from different domains and codified their EJPs and extension aspects using AspectJ language. Due to space limitation, the following sections briefly describe the implementation of EJPs for two case studies. For a complete description of the implementation of EJPs and extensions aspects for these case studies, please refer to [18]. Section 6 discusses lessons learned and guidelines derived from our case studies.

5.1 JUnit

The main purpose of the JUnit framework is to allow the design, implementation and execution of the unit tests in Java applications. According to the JUnit framework, each unit test is responsible for exercising one class method in order to assure that it performs as expected. The JUnit main functionalities are: the definition of test cases or suites to be executed; the execution of a selected test case or suite; and the collection and presentation of the test results. However, different extensions can be implemented to add new functionalities into the JUnit framework core. Some examples of simple extensions are the following:

(i) enable JUnit to execute each test suite in a separate thread, and wait until all tests finish. In order to implement this extension we need to observe the event when the test suite starts running, the event when each test method runs, and the event when the test suite stops running.

(ii) enable JUnit to run each test repeatedly. In order to implement this extension we need to observe the event when each test method runs.

These extensions need to observe JUnit internal events, which are spread over JUnit classes. In other words, such extensions are not well modularized in the object-oriented design. In our approach, an EJP was used to expose such *key events* that are not adequately captured by the OO design and that are useful for crosscutting compositions scenarios. Figure 2 presents an EJP, called `TestExecutionEvents`, which exposes a set of join points in the JUnit framework. Some of these join points were discovered by checking them against these anticipated crosscutting extension scenarios. Based on this first set of discovered join points, we could foresee other relevant events that may be of interest when extending JUnit.

The `TestExecutionEvents` EJP facilitates the definition of JUnit framework crosscutting extensions, since we can implement the extension aspects by reusing join points exposed by it. If necessary, extension aspects can also define more specific EJP-based pointcuts. Therefore, it is possible to codify aspects that affect only specific test cases or suites defined to test an application. In order to do it, it is only necessary to append a sub-expression to the

EJP pointcuts when defining an advice (e.g. `<EJP_pointcut> && within(<AppTestCase>)`). Besides the public pointcut descriptors, the EJP also contains a set of protected pointcuts which represents the EJP scope and the EJP accessors detailed in Section 4.1.

```
public aspect TestExecutionEvents {
       //Needed by: RepeatAllTests extension
       public pointcut testExecution(Test test):
              target(test) && call (void Test.run(TestResult));
       //Needed by: ActiveTestSuite extension
       public pointcut testSuiteExecution (TestSuite ts,TestResult rs):
              target(ts) && call (void TestSuite.run(TestResult)) && args(rs);
       //Needed by: ActiveTestSuite extension
       public pointcut testExecutionFromSuite(TestSuite ts,Test t,TestResult rs):
              target(ts) && call (void TestSuite.runTest(Test, TestResult)) &&
              args(test, result);
       //It is not already used any anticipated extension
       public pointcut testCaseExecution (TestCase tc, TestResult rs):
              target(tc) && call (void TestCase.run(TestResult)) && args(rs);
       //AUXILIARY METHODS:
       protected pointcut EJPMethodsScope():
                     withincode (void TestSuite.runTest(Test, TestResult)) ||
                     withincode (void TestCase.runTest()) ||
                     withincode (void TestSuite.run(TestResult));
                     withincode (void Test.run(TestResult));
       // Framework Scope
       protected pointcut FWScope(): within(junit..*);
        //The main propose of this EJP is to expose all the points in the
       // framework that result in a test execution.
       protected pointcut MainPurpose(): call (void TestResult.run(Test));
}
```

Figure 2. The AspectJ code of one EJP for JUnit framework

```
public aspect TestExecutionEventsContracts {
    //Behavioral Internal Contract
    declare error:
            (!TestExecutionEvents.EJPMethodsScope() &&
              TestExecutionEvents.MainPurpose() ):
             "Contract violation: Test execution should occur "+
              "through one of the methods: Test.run(), TestSuite.run(),"+
              "TestSuite.runTest(),TestCase.run(), TestCase.runTest()";
    //Behavioral Extension Contract
    public pointcut variabilityaspects(): within(variabilityaspects..*);
    before() : cflow ( adviceexecution() && !variabilityaspects() ) &&
           ( call(* *(..)) && TestExecutionEvents.FWScope() ){
           throw new RuntimeException("Contract Violation: no aspects, except" +
           " variability aspects, can access the elements of JUnit framework.");
     }
     ...
}
```

Figure 3. Corresponding contract of TestExecutionEvents EJP.

As discussed in the previous sections, each EJP contains a set of contracts regulating the internal and extension constraints. Figure 3 illustrates the `TestExecutionEventsContracts` aspect. This aspect contains one internal contract constraining the designer to assure that the pointcut descriptors (PCD) defined in the EJP comprises all and only the join points that results in test method executions. In other words, if any method not specified in an EJP pointcut (`!TestExecutionEvents.EJPMethodsScope()`) tries to call a unit test (`TestExecutionEvents.MainPurpose()`) a contract violation will be signed at compilation time.

The extension contract illustrated in Figure 3, assures that: no aspect, except the variability ones, can directly or indirectly, call a method, create an instance, or access an attribute of an element defined inside JUnit framework. The `adviceexecution()` matches the join points representing the execution of any advice. The expression `adviceexecution() && !variabilityaspects()`, defined in line 11, matches join points that occur during the execution of an advice and that are not defined inside a variability aspect – we defined, in line 10, that every variability aspect will be stored on packages matching the pattern `variabilityaspects..*`. This expression surrounded by `cflow` designator, matches the advice execution of non-variability aspects, or any method in the control flow of the advices defined in such aspects. Finally, the expression `call(* *(..)) && TestExecutionEvents.FWScope()` matches any method call, instance creation, or access an attribute of an element defined inside JUnit framework. Figure 4 shows the `TestExecutionEvents` EJP, which crosscuts JUnit elements and is used by a set of extension aspects.

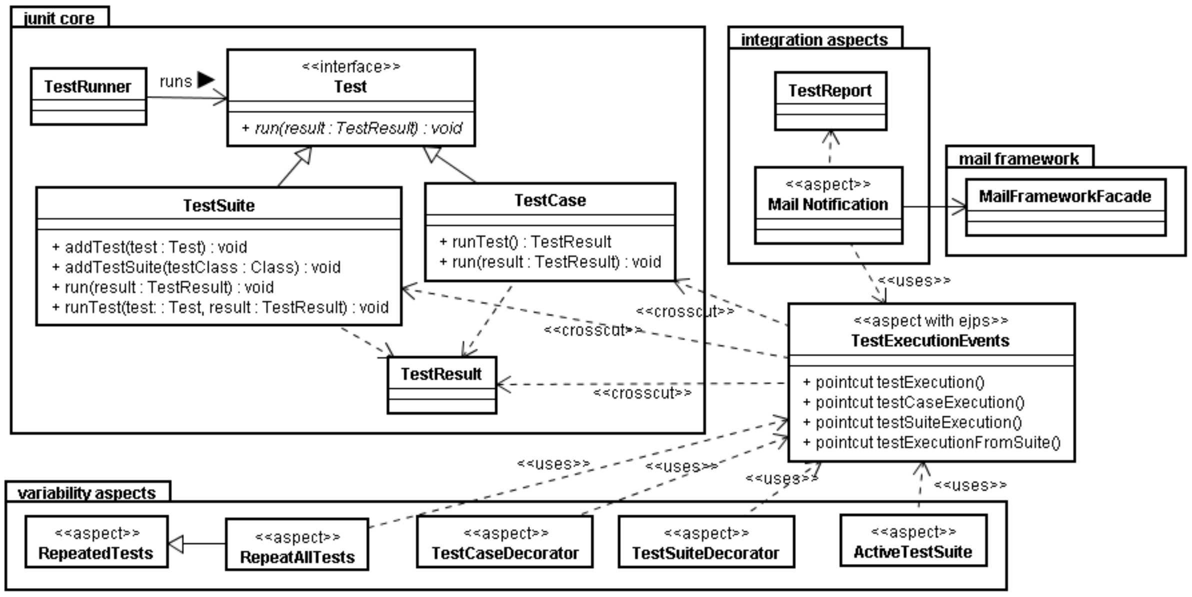

Figure 4. Overview of JUnit Framework and some crosscutting extensions.

5.2 J2ME Game Software Product Line

In this case study, we implemented variant features of an industrial J2ME game Software Product Line[1] based on EJPs. J2ME games are mainstream mobile applications of considerable complexity [2]. Their overall structure and behavior are defined by a framework known in this domain as the *game engine*. Essentially, this is a state machine whose state change is driven by elapsed time and user input through the device keypad. State changes affect the state of various drawing objects (*game actors*) and how they interact. Then, these objects are drawn again after such state changes. Typical hot-spots of this framework include some abstract classes defining basic drawing capability for game actors.

The case study implementation exposed game engine EJPs in order to allow the composition of crosscutting extensions in its basic functionality. Some interesting EJPs are the following: (i) how images are initialized and used; (ii) drawing of specific images; and (iii) game startup and changing screens. We have chosen these EJPs because they represent relevant events that can be of interest when extending the game engine core workflow. The resulting SPL architecture is shown in Figure 5. Package `rain.core` denotes the SPL core,

[1] Access to the game SPL instances was provided by Meantime Mobile Creations/CESAR.

i.e. the game engine. Package `extension join points` encapsulate all the EJPs, which are used by variability aspects and integration aspects in corresponding packages to implement crosscutting extensions.

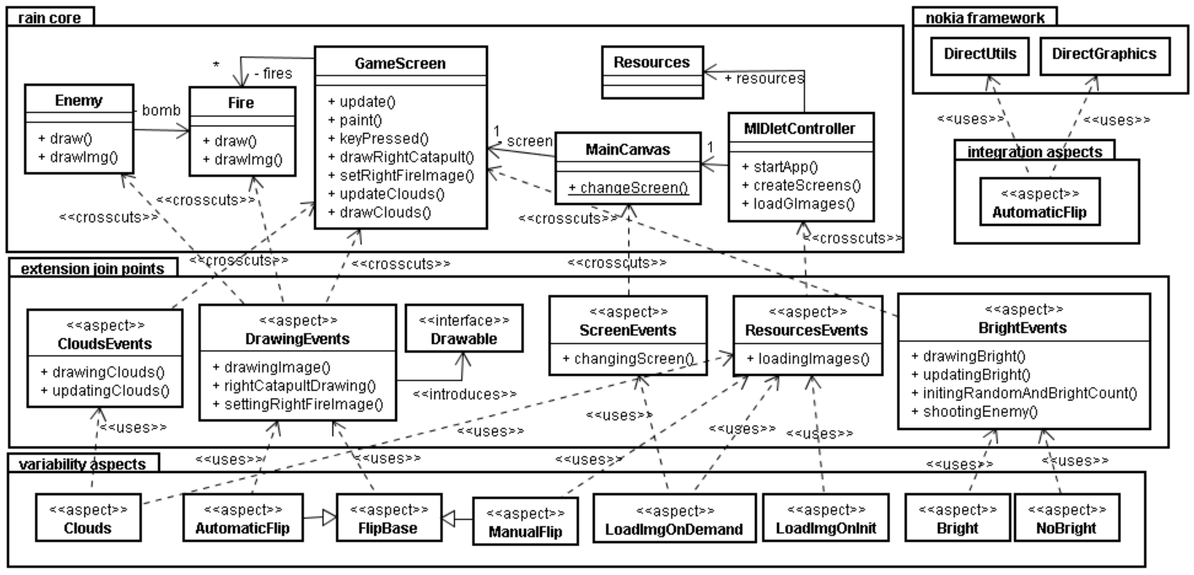

Figure 5. Architecture of the J2ME Game Product Line.

For example, the `DrawingEvents` and the `ResourceEvents` EJPs were composed with variability aspects to implement the alternative features for drawing some images. Specific images may be drawn at various locations and, under certain circumstances, may be transformed (rotated, flipped), which may be accomplished either manually (`ManualFlip` variability aspect) by using fresh new images or automatically (`AutomaticFlip` variability aspect) by transforming the original ones by calling device proprietary drawing API. In fact, this latter aspect also behaves as an integration aspect, due to the interaction with the proprietary API (another framework). Therefore, the fact that such aspect is in two packages is merely a logical, but not a physical view. By exposing these EJPs and composing them with variability and integration aspects, we provide modular implementation for the variant features the aspects represent.

In particular, the `FlipBase` aspect depends on the `DrawingEvents` EJP, which specifies all relevant events needed by such aspect, namely the drawing of images of game objects (Figure 5). We sketch this EJP in Figure 6:

```
public abstract aspect DrawingEvents {
 /* The purpose of the drawingImage PCD is to expose all and only
 drawing requests of images associated to game items that move around the
 game screen. All such requests must be implemented by call to a method
 matching the PCD. We require aspects advising this PCD to access only
 some framework objects through the Drawable or Graphics types. */
       public interface Drawable {
              public void drawImg(Graphics g, int ofsX);
              ...
       }
       declare parents: Enemy implements Drawable;
       declare parents: Fire implements Drawable;
       public pointcut drawingImage(Drawable d, int offSetX, Graphics g) :
              execution(public void Drawable.drawImg(Graphics, int))
               && this (d) && args(g, offSetX);
        ...
}
```

Figure 6. Structure of the DrawingEvents EJP.

The comment in the EJP is a semantic specification of the framework internal contract and the framework extension contract. In the former, the core must signal its intent of drawing image of game items by calling specific methods of the `Drawable` interface, which it must implement (`declare parents` constructs), thus forcing the contract; in the latter, the variability aspect should access framework context only through the EJP; further, such aspect cannot access internal framework details. This constraint can be checked with the `declare warning` construct in the aspect in Figure 7.

```
public aspect DrawingExternalContractChecker {
   // Framework Scope - Calls Not Allowed
   public pointcut FWScopeNotAllowed():
       call ( * !(Drawable+||Graphics).*(..) ) && call (* raincore.*.*(..));

   public pointcut aspectsPackages(): within(variabilityaspects..*);

   declare warning: FWScopeNotAllowed() && aspectsPackages():
             "Extension aspects are accessing internal framework details";
}
```

Figure 7. Contract checking for EJP.

The `FWScopeNotAllowed()` pointcut denotes calls to framework internal types, where we assume that the `Drawable` interface and the `Graphics` class should be visible to the variability aspects. The `aspectsPackages()` denotes calls within such aspects.

Figure 5 also shows other EJPs in the SPL, representing additional crosscutting extensions; it further illustrates that one EJP may be used by more than on variability or integrability aspects, and, conversely, that each such aspect may extend more than one EJP.

6. Discussion and Lessons Learned

This section provides further discussion of issues and lessons we have learned in the evaluation of our approach and the use of AspectJ to implement our EJPs.

6.1 EJP-based Approach Analysis

EJPs stability. EJPs specify not only a set of extension join points in which frameworks can be extended, but they also represent the interfaces between the framework classes and extension aspects. In this sense, they have the same purpose of the XPIs, proposed by Griswold et al [16]. Hence, the implementation of EJPs gives us the benefit to evolve the framework classes without break the aspects that extend its functionality. However, to achieve this benefit is important that joins points exposed by EJPs and their respective contracts remain working when refactoring the framework classes. EJPs can also evolve to accommodate new requirements required by the extension aspects, such as, the exposition of new framework join points or the exposition of additional arguments in the existing join points exposed. All the contracts defined by the EJPs must be revalidated and if necessary rewritten during the refactoring and evolution of EJPs due to change in the framework classes or new demands in the extension aspects.

EJPs modeling. EJPs can also be considered framework hot-spots [11]. They represent flexible points in the execution of specific framework scenarios that can have a crosscutting extension inserted. We have encountered in our case studies that although the modeling of EJPs is dependent on the framework domain, they in general represent relevant events or transition states occurring during the execution of the framework functionalities. Since the EJPs are modeled to accommodate the insertion of optional, alternative and integration features in the framework, the early identification of these elements in the domain analysis [10] also helps the EJPs modeling.

EJPs as Architectural Enforcement. In our approach, framework classes and extension aspects are constrained to interact in a manner that respects EJPs' contracts. Since EJPs provide a way for the definition of inter-module interactions, it can also be useful for the enforcement of architectural properties in general. Architectural properties, like the extension join points, are not localized in a single system module, they must be observed in many of them. Using EJPs contracts to enforce architectural policies can bring more robustness to aspect oriented systems.

EJPs Specialization. EJPs expose framework join points in which can be inserted new crosscutting behaviors by means of the extension aspects. Since many of these join points can be some of the hook methods of framework hot-spots, EJPs can also be specialized to affect only specific hot-spots instances. In JUnit case study (Section 5.1), for example, we have presented an example of EJP pointcut which can be customized to affect only specific instances of test cases and suites. In that case, the pointcut defined in the EJP need be redefined by the developer implementing the extension aspect that will use it.

6.2 EJPs Implementation in AspectJ

Contracts Implementation in AspectJ. The AspectJ available mechanisms allowed the implementation of four kinds of EJP contracts defined in our category (Section 4.2). Three different mechanisms were used to implement them: (i) the `declare error` and `declare warning` statements to enforce policies between the framework, EJP and aspects; (ii) the `declare parents` statement that guarantees framework classes implement interfaces defined by the EJPs; and (iii) `adviceexecution` pointcut designator which allows to intercept advices and define specific contracts to be validated before and after their executions. There are specific constraints that cannot be checked with aspects; for example, fields introduced into core classes by means of inter-type declaration should not be accessed by the core (thus resulting in a dependency of the core into the aspects). This cannot be checked statically by aspects, thus requiring an enhanced analysis tool.

Join Point Encapsulation. The only kind of contract not implemented in AspectJ language was the extension contract which determines that aspects can only extend the framework join points exposed by the EJPs. Currently, there is no existing mechanism in AspectJ to restrict advices to extend specific join points. The programming practice to allow developers to only reuse the join points exposed in the EJPs was used in our case studies to guarantee this kind of contract. Larochele et al [22] have proposed a mechanism, called join point encapsulation, which aims to prevent selected join points from being modified by aspects. They extend the AspectJ language to support their `restrict` statement whose implementation allows to prevent the access to specific join points. Since this mechanism was implemented only to previous versions of AspectJ, we did not have the chance to experiment it in our case studies.

Annotation-based Pointcuts. AspectJ [3] has recently incorporated mechanisms to specify join points using Java annotations. EJPs can benefit from this mechanism by allowing inserting the annotations directly in the framework classes' join points being affected. This implementation decision can give more stability (Section 6.1) to the EJPs, since signature-based pointcuts are subject to changes when the framework needs to be constantly refactored. In other words, annotation-based pointcuts now available in AspectJ can become the EJPs more robust in scenarios where the use of the signature-based model generates pointcuts complex and difficult to maintain.

Interface-based Contracts. EJP can define interfaces and specify that types in the framework implement these interfaces by means of the `declare parents` static crosscutting construct available in AspectJ. This structural internal contract is important for promoting higher abstraction for the extension aspects, since these will intercept events on generic rather than specific types, thus leading to reduced coupling with the framework and to higher reuse of extension aspects.

7. Related Work

Our concept of EJPs is inspired by Sullivan et al's work [30] on specification of crosscutting interfaces (XPIs). XPIs abstract crosscutting behavior, isolating aspect design from base code design and vice-versa. Continuing this work, Griswold et al show how to represent XPIs as syntactic constructs [16]. EJPs play a similar role to XPIs, but specifically in the context of framework development, by exposing a set of framework events for notification and crosscutting composition, and by offering predefined execution points for the implementation of optional and alternative features. In the specification of the semantic part of EJPs, however, we have defined a different methodology to specify the constraints which regulate the relationships between the framework, EJPs and extension aspects.

Open Modules [1] introduces a strong form of encapsulating join points occurring inside a module. It permits defining an interface composed by set of pointcuts that can be advised by clients. Any other join point that occurs inside the module is protected from external advising. It permits evolution of a module implementation without considering the aspects advising exported pointcuts, since no changes are made on the interface. It's possible because the aspects are coupled with the module exclusively by the module's interface. However, Open Models has a limitation on the pointcuts that can be written on the interfaces. These poincuts can only intercept join points occurring inside the module, making impossible writing an interface that crosscuts more than one module. Our approach doesn't have this limitation, since an EJP can declare pointcuts (extension points) involving join points occurring in any number of classes. We use contracts based on AspectJ's inter-type constructions (declare error and warning) to control coupling between framework core, EJPs and extension aspects.

Feature oriented approaches (FOAs) have been proposed [29] to deal with the encapsulation of program features that can be used to extend the functionality of existing base program. Batory et al [5] argue the advantages that feature-oriented approaches have over OO frameworks to design and implement product-lines. Mezini and Ostermann [25] have identified that FOAs are only capable of modularizing hierarchical features, providing no support for the specification of crosscutting features. These researchers propose CaesarJ [26], an AO language that combines ideas from both AspectJ and FOAs, to provide a better support to manage variability in product-lines. Our approach is directly related those authors work, since we believe that the design of product-line architectures may benefit from the composition and extension of different frameworks using integration and variability aspects. Additionally, we propose the definition of EJPs as a form of reducing and exposing coupling between the framework core and its extensions, witch are implemented using aspects.

Zhang and Jacobsen [30] propose the Horizontal Decomposition method (HD), a set of principles guiding the definition of functionally coherent core architecture and customizations of it. They suggest dividing the middleware in core and aspects that customize the core with orthogonal functionality. HD adopts obliviousness as a principle, suggesting that framework core should be unaware of the aspects. Our approach suggests that is necessary to use some mechanism to control and expose coupling between framework core and its extension, witch we called EJPs.

Mortensen and Ghosh [27] investigate how AOP helps using and extending object-oriented frameworks in VLSI CAD applications. They suggests that in general the code necessary for integrating the framework into the application is crosscutting, and shows that using AOP it was possible to better modularize that code, reducing the number of lines of code and also improving the application structure. They propose using AOP for constructing a reusable library of framework-based aspects useful in a family of framework-based applications. Our approach suggests using aspects not only for integrating frameworks into applications, but also for composing independent frameworks. Another difference is that we advocate using aspects inside the frameworks to capture crosscutting concerns and expose these extension points.

8. Conclusions and Future Work

In a previous work, we proposed a framework extension approach based on the use of Extension Join Points (EJPs). EJPs enable the framework systematic extension by means of variability and integration aspects. In this paper, we have shown how EJPs can be implemented using the mechanisms of the AspectJ language. Our EJPs were implemented by exposing specific framework join points using AspectJ pointcuts and by defining a set of contracts specified using different static and dynamic AspectJ mechanisms. These contracts play a fundamental role in our approach because they help to govern the relationships between the framework and extension aspects by ensuring that important constraints respected.

As future work, we intend to continue the evaluation of the approach in the development and refactoring of object-oriented frameworks. We also plan to realize quantitative studies [6, 15] to compare the approach with the use of OO techniques with respect to traditional software metrics. In order to enable the adoption of our approach, we intend to derive a more systematic implementation method which offers more detailed steps and guidelines to the design and implementation of extensible OO frameworks with aspects. Finally, we plan to explore the extension of current domain analysis and design methods [10] to support the early modeling of extension join points and framework extension aspects. This also involves to investigate the suitability of UML-based notations to represent the EJPs, such as the aSideML crosscutting interfaces [7].

Acknowledgments. We would like to thank the members of Software Productivity Group at Federal University of Pernambuco for valuable suggestions for improving this paper. This research was partially sponsored by FAPERJ (grant No. E-26/151.493/2005 and No. E-26/100.061/06), CNPq (grants No. 552068/2002-0, 481575/2004-9 and 141247/2003-7), MCT/FINEP/CT-INFO (grant No. 01/2005 0105089400), and European Commission Grant IST-2-004349: European Network of Excellence on AOSD (AOSD-Europe).

References

[1] J. Aldrich, “Open Modules: Modular Reasoning about Advice,” Proceedings of ECOOP’05, LNCS 3586, Springer 2005, pp. 144–168.

[2] V. Alves, P. Matos, L. Cole, P. Borba, G. Ramalho. “Extracting and Evolving Mobile Games Product Lines”. Proceedings of SPLC'05, LNCS 3714, pp. 70-81, September 2005.

[3] AspectJ Team. The AspectJ Programming Guide. http://eclipse.org/aspectj/.

[4] C. Baldwin, K. Clark. Design Rules: The Power of Modularity. MIT Press, Cambridge, MA, 2000.

[5] D. Batory, R. Cardone, and Y. Smaragdakis, Object-Oriented Frameworks and Product-Lines. 1st Software Product-Line Conference (SPLC), pp. 227-248, Denver, August 1999.

[6] N. Cacho, et al. Composing Design Patterns: A Scalability Study of Aspect-Oriented Programming. Proceedings of AOSD'06, Bonn, Germany,2006.

[7] C. Chavez, A. Garcia, U. Kulesza, C. Sant'Anna, C. Lucena. Taming Heterogeneous Aspects with Crosscutting Interfaces. Journal of the Brazilian Computer Society, 2006 (to appear).

[8] W. Codenie, et al. "From Custom Applications to Domain-Specific Frameworks", Communications of the ACM, 40(10),October1997.

[9] A. Colyer, et al. Eclipse AspectJ: Aspect-Oriented Programming with AspectJ and the Eclipse AspectJ Development Tools, Addison-Wesley, 2004.

[10] K. Czarnecki, U. Eisenecker. Generative Programming: Methods, Tools, and Applications, Addison-Wesley,2000.

[11] M. Fayad, D. Schmidt, R. Johnson. Building Application Frameworks: Object-Oriented Foundations of Framework Design. John Wiley & Sons, September 1999.

[12] R. Filman, T. Elrad, S. Clarke, M. Aksit. Aspect-Oriented Software Development. Addison-Wesley, 2005.

[13] R. Filman, D. Friedman. Aspect-oriented programming is Quantification and Obliviousness. In [12], pp. 21–35. Addison-Wesley, 2005.

[14] E. Gamma, et al. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.

[15] A. Garcia, C. Sant'Anna, E. Figueiredo, U. Kulesza, C. Lucena, A. Staa. Modularizing Design Patterns with Aspects: A Quantitative Study. Proc. 4th Intl. Conference on Aspect-Oriented Software Development, Chicago USA, March 2005.

[16] W. Griswold, et al, "Modular Software Design with Crosscutting Interfaces", IEEE Software, Special Issue on Aspect-Oriented Programming, January 2006.

[17] G. Kiczales, et al. Aspect-Oriented Programming. Proc. of`ECOOP'97, Finland, 1997.

[18] U. Kulesza, et al. "Implementing Framework Crosscutting Extensions with EJPs and AspectJ", Technical Report, PUC-Rio, Brazil, August 2006.

[19] U. Kulesza, V. Alves, A. Garcia, C. Lucena, P. Borba. Improving Extensibility of Object-Oriented Frameworks with Aspect-Oriented Programming, Proceedings of ICSR'2006, Springer Verlag, LNCS 4038, pp. 231-245, Torino, Italy, June 2006.

[20] U. Kulesza, A. Garcia, C. Lucena. "Composing Object-Oriented Frameworks with Aspect-Oriented Programming", Technical Report, PUC-Rio, Brazil, April 2006.

[21] U. Kulesza, A. Garcia, C. Lucena, A. von Staa. "Integrating Generative and Aspect-Oriented Technologies", Proceedings of SBES'2004, pp. 130-146, Brasilia, Brazil, October 2006.

[22] D. Larochelle, et al., "Join Point Encapsulation," Proc. Workshop Software Eng. Properties of Languages for Aspect Technologies (SPLAT), 2003.

[23] M. Mattson, J. Bosch, M. Fayad. Framework Integration: Problems, Causes, Solutions. Communications of the ACM, 42(10):80–87, October 1999.

[24] M. Mattsson, J. Bosch. Framework Composition: Problems, Causes, and Solutions. In [7], 1999, pp. 467-487.

[25] M. Mezini, K. Ostermann: "Variability Management with Feature-Oriented Programming and Aspects". Proceedings of FSE'2004, pp.127-136, 2004.

[26] M. Mezini, K. Ostermann. "Conquering Aspects with Caesar". Proc. of AOSD'2003, pp. 90-99, March 17-21, 2003, Boston, Massachusetts, USA.

[27] M. Mortensen, S. Ghosh. Using Aspects with Object-Oriented Frameworks, Proceedings of AOSD'2006, Industry Track, Bonn, Germany, March 20-24, 2006.

[28] D. Riehle, T. Gross. "Role Model Based Framework Design and Integration". Proceedings of OOPSLA'1998, pp. 117-133, Vancouver, BC, Canada, October 18-22, 1998.

[29] Y. Smaragdakis, D. Batory. Mixin Layers: An Object-Oriented Implementation Technique for Refinements and Collaboration-Based Designs, ACM TOSEM, 11(2): 215-255 (2002).

[30] K. Sullivan, et al. Information Hiding Interfaces for Aspect-Oriented Design, Proceedings of ESEC/FSE´2005, pp.166-175, Lisbon, Portugal, September 5-9, 2005.

[31] C. Zhang, H. Jacobsen. "Resolving Feature Convolution in Middleware Systems". Proceedings of OOPSLA'2004, pp.188-205, October 24-28, 2004, Vancouver, BC, Canada.