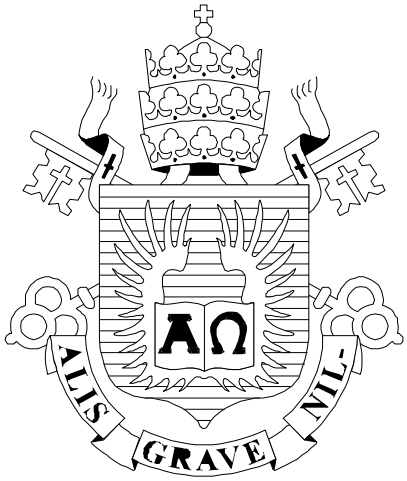

PUC

ISSN 0103-9741

Monografias em Ciência da Computação
n° 16/06

Composing Object-Oriented Frameworks with Aspect-Oriented Programming

Uirá Kulesza
Alessandro Fabricio Garcia
Carlos José Pereira de Lucena

Departamento de Informática

PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO
RUA MARQUÊS DE SÃO VICENTE, 225 - CEP 22453-900
RIO DE JANEIRO - BRASIL

Monografias em Ciência da Computação, No. 16/06 ISSN: 0103-9741
Editor: Prof. Carlos José Pereira de Lucena April,2006

Composing Object-Oriented Frameworks with Aspect-Oriented Programming*

Uirá Kulesza, Alessandro Fabricio Garcia[1], Carlos José Pereira de Lucena

[1]Computing Department – Lancaster University (UK)

uira@les.inf.puc-rio.br, garciaa@comp.lancs.ac.uk, lucena@inf.puc-rio.br

Abstract. The composition of object-oriented (OO) frameworks are often required in application development in order to meet the pressing needs of reuse-in-the-large and time to market. However, framework composition has been often shown as a challenge in real software development due to recurring integration activities involving heterogeneous framework features, such as domain entities, overlapping functionalities, and diverging control flows. One of the underlying problems is that the composition strategy cannot be implemented in a modular manner based on traditional OO mechanisms. Invasive changes are often required in the frameworks being composed. In this context, this paper presents a systematic investigation on the use of aspect-oriented programming (AOP) for the modular composition of OO frameworks. We have revisited several object-oriented solutions previously proposed to the composition of OO frameworks. After that, we have evaluated whether the use of AOP can improve each of these existing OO solutions. Our comparative analysis was based on a set of modularity properties, and through a case study involving the composition of four OO frameworks with different characteristics and from distinct domains. The outcomes of this first systematic investigation have pointed out that the aspect-oriented solutions present several benefits in relation to the original OO solutions in terms of modularization of the composition code.

Keywords: Object-Oriented Frameworks, Aspect Oriented Programming, Software Composition.

Resumo. Este artigo apresenta uma investigação sistemática da adoção de programação orientada a aspectos (POA) na composição modular de frameworks orientados a objetos (OO). Nós revisitamos diversas soluções OO propostas anteriormente para a composição de frameworks e avaliamos o potencial de POA na modularização de cada destas soluções. Nossa análise comparativa foi baseada em um conjunto de propriedades modulares e realizada através de um estudo de caso envolvendo a composição de quatro frameworks OO com diferentes características e de domínios distintos. Como resultado de nossa investigação sistemática, nós concluímos que o uso de técnicas orientadas a aspectos oferece uma melhor modularização do código de composição para muitos dos problemas de composição de frameworks OO comumente encontrados.

Palavras-chave: Frameworks Orientados a Objetos, Programação Orientada a Aspectos, Composição de Software.

---

* This work has been sponsored by the Ministério de Ciência e Tecnologia da Presidência da República Federativa do Brasil.

In charge for publications:

Rosane Teles Lins Castilho
Assessoria de Biblioteca, Documentação e Informação
PUC-Rio Departamento de Informática
Rua Marquês de São Vicente, 225 - Gávea
22453-900 Rio de Janeiro RJ Brasil
Tel. +55 21 3114-1516 Fax: +55 21 3114-1530
E-mail: bib-di@inf.puc-rio.br
Web site: http://bib-di.inf.puc-rio.br/techreports/

Table of Contents

1 Introduction

Object-oriented (OO) frameworks [7] 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. Over the last years, an increasing number of open-source OO frameworks have become available to software developers [8, 13, 28]. However, with the complexity of modern software, it is unlikely that a single framework is enough to cover all the concerns of relevance for a given application. As a result, whereas framework-based application development originally included a single framework [7], increasingly often multiple OO frameworks are used in realistic development scenarios [21]. The key problem is that their modular composition is not a trivial task since frameworks are designed to be extended, but not to be combined [27, 21]. It is a very common policy given that frameworks' developers simply cannot foresee all the ways in which they will be composed.

In order to take the advantage of reuse-in-the-large, software engineers have typically to cope with various intricate composition issues, such as diverging control flows and integration of entity functionalities [20, 21]. However, the reuse process is hindered if the adopted composition strategy is not modular itself. The intrusiveness of the composition code in the implementation of the framework modules potentially leads to several undesirable consequences, including invasive changes, ripple effects in the variable and frozen parts of the frameworks, difficulty to understand and maintain separately the frameworks and their composition, and unplugabillity of the integration code [20, 21].

There are several OO solutions to mastering framework integration issues, ranging from wrappers and adapters to more holistic change strategies. Mattsson et al [20, 21] have systematically presented and analyzed these existing OO solutions for each challenge related to composition of OO frameworks. The solutions proposed by the authors represent the state-of-art of available solutions [22]. The problem is that most of the existing OO techniques for framework composition involve some form of crosscutting between the framework classes and the integration code. The implementation of composition strategies often crosscut the functionality of the framework classes, preventing the requirement of modular composition and reuse.

In this context, it is important to systematically verify the suitability of aspect-oriented programming (AOP) to enable modular composition of OO frameworks. Some authors have examined the influence of AOP on different software composability scenarios, such as COTS [16] and design patterns [12, 4]. However, there is no systematic study that investigates the impact of AOP on framework composability. This paper revisits and analyzes existing OO techniques against a set of modularity properties. Our investigation is focused on three recurring challenges for framework composition: (i) composition of framework control, (ii) framework gap, and (iii) composition of entity functionality. We also present aspect-oriented (AO) solutions based on AspectJ language [17] for these three challenges, and compare them with the OO approaches. A number of lessons learned on the design and implementation of the AO solutions are also discussed. Our investigation is based on a case study with compositions involving four OO frameworks of different complexities, including the use of mainstream technologies such as Swing [8] and Hibernate [13], and addressing concerns from distinct horizontal and vertical domains [5].

The remainder of this paper is organized as follows. Section 2 presents the frameworks used in our case study, and an overview of the Mattsson et al's study. Also, an analysis of the modularity and crosscutting properties of the authors' OO solutions is described in Section 2. Section 3 presents the AO solutions proposed to enhance the modularity of the composition code relative to each of the integration challenges explored in our case study. Section 4 analyzes the AO solutions developed to redesign and reimplement the original OO solutions, and also presents some lessons learned related to design and implementation issues. Section 5 discusses related work. Finally, Section 6 offers the conclusions and directions for future work.

2 Composition of Object-Oriented Frameworks

This section provides a review of the Mattsson et al's study [20, 21] using a set of four frameworks as case studies. Section 2.1 presents the examples of OO frameworks, which are used throughout the paper to illustrate the composition scenarios, emerging problems, and solutions encountered when composing OO frameworks. Section 2.2 describes the problems and solutions presented by Mattsson and colleagues. Finally, Section 2.3 analyzes those existing OO solutions against some modularity and crosscutting properties.

2.1 A Suite of OO Frameworks: Our Case Study

This section presents four examples of OO frameworks which were used in our case study. They encompass distinct vertical and horizontal domains: (i) measurement support for product quality control (Figure 1), (ii) GUI infrastructure (Figure 2), (iii) statistical analysis (Figure 3), and (iv) persistence (Figure 4). We have chosen these frameworks because the three first ones were also previously used in the Mattsson et al's study [20, 21]. They also involve different features and combinations that cover the main composition problems to be examined in this paper. All the frameworks were implemented using the Java programming language. The implementation of the measurement framework was completely based on available design and code documentation [3]. The GUI and persistence frameworks were offered a simplified implementation based on mainstream technologies, such as, the Swing API [8] and the Hibernate framework [13]. We have then decided to implement the aspect-oriented solutions (Section 3) using AspectJ [17], which is a Java extension and the most popular aspect-oriented programming language.

Figure 1 shows the structure of the object-oriented framework for measurement systems. It was previously described in [3]. It captures the main entities and functionalities of a measurement process used for quality control on a certain kind of products. The framework allows for the categorization of those products in acceptable or unacceptable based on specific quality criteria. This framework implements a typical measurement cycle that is formed by the following steps: (i) data collection phase – a trigger (`Trigger` class) indicates that a product is entering the system and sensors measure relevant properties of the incoming item (`PhysicalSensor`, `Sensor`, and `UpdateStrategy` classes); (ii) analysis phase – the collected data by the sensors are converted to a common representation and after that they are compared to ideal expected values (`MeasurementValue`, `MeasurementItem`, and `CalculationStrategy` classes). Based on this comparison, the measured items are then classified in quality categories; and (iii) actuation phase – an action is performed over it according to the item classification (`Actuator`, PhysicalActuator and `ActuationStrategy` classes). Several kinds of ac-

tion can be executed, such as removing the element from the production line, or only label it to be processed later. Figure 1 illustrates the main classes responsible to implement the measurement cycle. The classes which specify variation points or hot-spots [7] are indicated in grey color. Examples of such classes are: physical sensors and actuators, and strategies of actuation and sensoring.

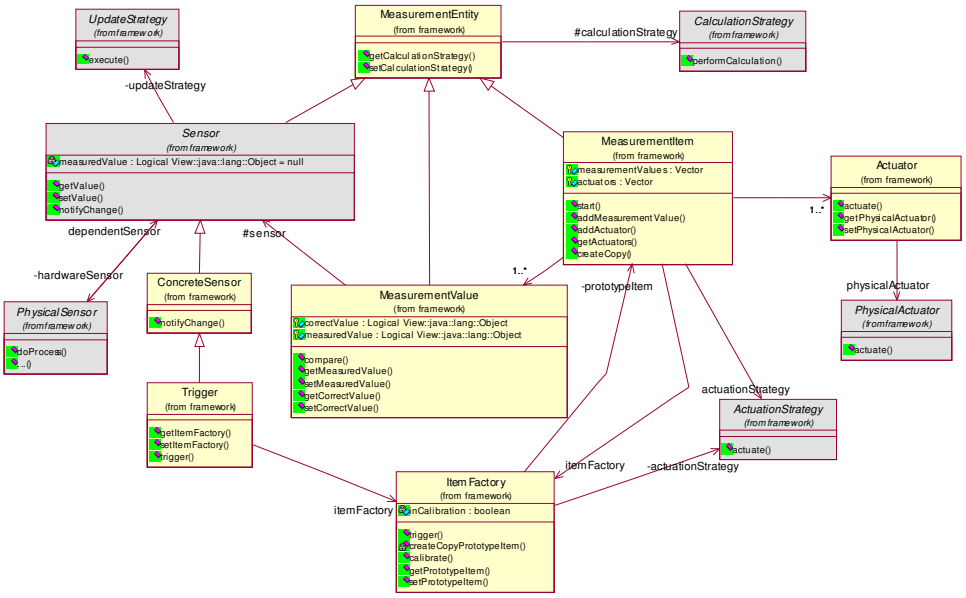

Figure 1: The Measurement Framework.

Figure 2 presents the class diagram of a GUI framework based on the Java Swing architecture [8]. It supports the presentation of application data in different UI Swing tables. Two abstract classes (`DataTableModel` and `ObjectTableModel`) allow to specify different representations of data table based on an array or on a list of objects. These classes inherit from the `AbstractTableModel` class of the Java Swing API. Application data are presented visually using the `JTable` Swing class. Finally, the framework user must extend the `GUIFramework` class by defining an implementation to its `createTables()` abstract method. A concrete implementation of this method must create and return a list of `JTable` configured with their respective `TableModel` objects which together will show application data. All the information written in `TableModel` objects are automatically presented in its respective `JTable` object based on a notification protocol implemented in the Swing API [8].

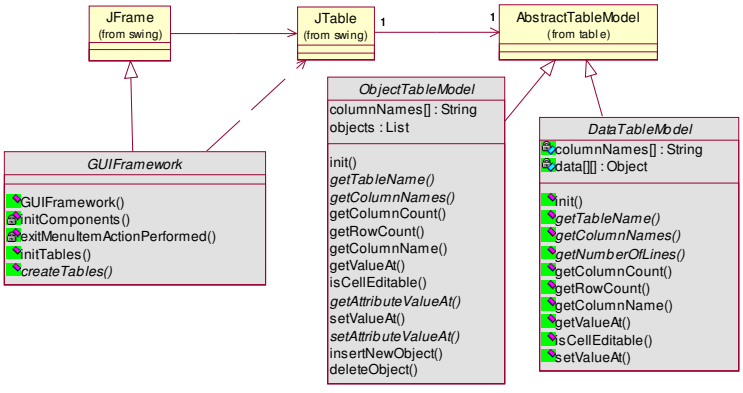

Figure 2: The GUI Framework.

Figure 3 presents the structure of a small statistical analysis framework. It supports the statistical analysis of data gathered in an application. This framework differs from

the frameworks described previously, because it represents a "called" framework instead of a "calling" framework, according to the classification proposed by [27]. A calling framework, such as the Measurement and GUI frameworks, is responsible to control and invoke all the other parts of the application. On the other hand, a called framework, such as the statistical analysis framework, is a passive entity which can be called by other parts of the application. The statistical analysis framework contains: (i) a façade class (`StatisticalFrameworkFacade`) which provides statistical services to be invoked by applications; and (ii) two abstract classes, `StatisticAlgorithm` and `Statistic`, representing, respectively, variation points to implement a statistical algorithm and the resulted statistic from an algorithm processing.

Figure 4 shows the main classes of our representative persistence framework based on Hibernate [13] and Spring [28]. It is also a called framework. The `PersistenceFrameworkFacade` class exposes the persistency services provided by the framework. It aggregates a set of concrete Data Access Objects [1], which implements the persistence methods considering specific kinds of objects. The `EntityDAO` abstract class specifies the signature of the persistence methods to be implemented by concrete DAOs subclasses. This class inherits from the `HibernateDaoSupport` class in order to reuse the database services provided by the Hibernate framework, such as, simple database access, connection management and transaction demarcation.

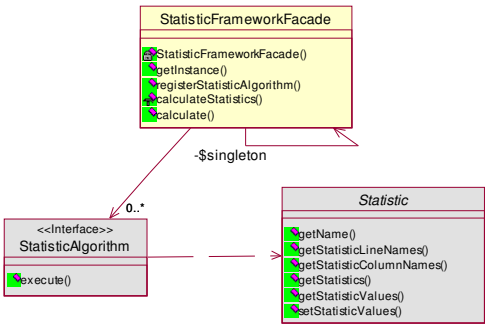

Figure 3: The Statistical Framework

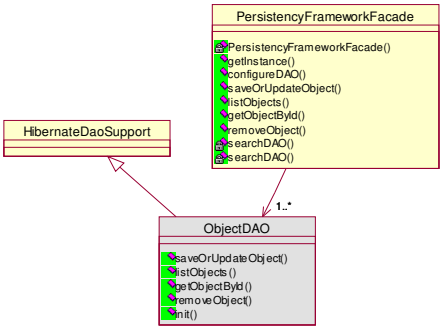

Figure 4: The Persistence Framework

2.2 Framework Composition: Problems and OO Solutions

Mattsson et al [20, 21] discuss five problems relative to pair-wise compositions of object-oriented frameworks. In this paper, we will focus on three of them: (i) composition of framework control, (ii) framework gap, and (iii) composition of entity functionality. Those authors also describe three different OO based solutions for each framework composition problem; each solution can be individually applied or in conjunction to a second one to overcome a specific problem. Next subsections will briefly present the 3 investigated problems and the respective OO solutions. Refer to [20, 21] for further details about them. Section 3 will propose aspect-oriented (AO) solutions to deal with the composition problems, and compare them against the traditional OO solutions presented in this section.

2.2.1 Control Composition of Calling Frameworks

The composition of framework control occurs when two calling frameworks are being instantiated and combined in an application context. Since each calling framework expects to assume the control of the application, their composition needs to consider the conflicts resulting from their concurrent execution, such as race conditions and syn-

chronization of common objects. It may alternatively be required to define a unique control loop for the involved frameworks. Their composition is even more challenging in the presence of event notifications between the two composed frameworks. An example is the composition between the two calling frameworks for measurement and GUI support (Section 2.1) in order to present information about the measurement process visually in GUI components.

The OO solutions proposed to address the framework control problem are:

• *Concurrency*. This solution proposes to assign a separate thread of control for each framework, instead of composing their control flow. It can only be used when there is no need of event notification between the frameworks. In some applications, it may be necessary to define synchronization code to the application-specific objects accessed by the frameworks. Lea [18] describes several object-oriented Java solutions which can be used to framework concurrency control;

• *Wrapping*. This solution complements the first one when event notifications are required between the involved frameworks. It consists in encapsulating each framework with a wrapper. Each wrapper is responsible for intercepting input and output events of the framework. Thus, all the framework classes which communicate with external ones need to be modified in order to notify the wrapper. In this way, the wrappers allow for the framework integration based on the manipulation of their external events;

• *Removal and Rewriting*. This solution can be used when it is necessary some kind of internal events notification between the frameworks. It implies a modification of each framework code in order to define a unique control loop that addresses the requirements of both the framework composition and the application. Thus, access to the framework source code is required.

2.2.2 Framework Gap

The framework gap problem occurs when two (or more) frameworks need to be composed to address the application requirements, but their simple, direct composition is not enough to cover all the requirements completely. The framework composition needs to be extended in some way to address a functionality in the application domain. For example, the composition between the Measurement and GUI frameworks could be insufficient to address the additional functionality of showing statistic data related to the measurement process. Thus, the statistical analysis framework could be used to achieve that purpose.

The OO solutions proposed to address the framework gap problem are:

• *Wrapping and Extending*. This solution deals with the framework gap problem when called frameworks are being used. It proposes to introduce a wrapper element that aggregates the frameworks and the additional extension required for closing the gap. This wrapper is also responsible to provide a new interface to the resulting services of the composition;

• *OO Mediator*. This solution is proposed to address the framework gap when using calling frameworks. The mediator is used to control the interactions between the frameworks and additional extensions that are being composed. The interactions between the frameworks and additional extensions usually require the modification of internal framework classes to notify other ones of events of interest;

• *Redesign and Extend*. This solution consists in modifying the source code of frameworks and extensions in order to integrate them in only one framework. It could be adopted when the composition of frameworks and extensions intends to be reused in development of future applications.
2.2.3

2.2.3 Composition of Entity Functionality

This framework composition problem is related to the composition of representation of a domain entity provided by a domain-specific framework with a different functionality provided by another framework. A typical example of this problem happens when it is necessary to use a persistence framework to store the information of domain-specific framework entities (such as the processed items) through the Measurement framework (Section 2.1). In this case, since the Measurement framework does not provide extension points to add the persistence functionality, it can be difficult to compose both framework functionalities.

The OO solutions proposed to address the composition of entity functionality are:

• *Aggregation or Multiple Inheritance*. The first solution proposed by Mattsson et al is to use the mechanisms of aggregation or multiple inheritance to compose the framework entities with the additional functionality provided by another framework. The main drawback of these solutions is that state changes in the domain-specific part of the composed classes will not automatically influence the modules implementing the additional functionalities. Each composed class aggregates (or inherits from) the domain-specific entity class and the additional functionality class. The problem is that a composed class cannot keep control of every change happening in the entity class by means of other classes in the domain-specific framework;

• *Observer*. This solution is proposed to deal with the drawback of the previous solution. The Observer design pattern [9] must be used to provide the notification behavior between the framework and the composed class. The framework entity classes play the subject role of the Observer pattern and their respective composed class must play the observer role to receive notifications of changes in the entity classes.

2.3 Analysis of the OO Solutions

This section discusses the advantages and limitations of the OO solutions proposed by Mattsson et al. Table 1 details the properties identified for the 9 OO solutions investigated for framework composition. We have used three modularity properties to analyze the OO solutions. These modularity properties are adapted from a comparison criteria defined by Hannemann and Kiczales [12]. Such criteria were originally created for comparing OO and AO solutions of GoF design patterns [9].

The modularity properties are the following: (i) locality – indicates if the composition code is completely separated in modules; (ii) composition transparency – indicates if it is possible to reason about different framework compositions independently; (iii) pluggability – determines how easy is to add or remove the composition code. Two additional properties are also used to identify the crosscutting nature of each technique: (i) tangling – indicates if the composition code is tangled with the framework code into a specific framework module, and (ii) scattering – defines if the composition code of the OO solution is also spread over several classes of the framework(s). We discuss below the main advantages and drawbacks of the OO techniques.

Crosscutting Nature. Table 1 shows that 6 of the 9 OO solutions have both scattering and tangling properties. It means that the majority of the OO solutions require the modification of many and different framework classes. In many cases, the composition code spreads over several framework classes and also tangles with framework code. This reflects the crosscutting nature of framework composition code. Thus, as a result, we have that the design and implementation of many OO composition solutions are not completely modularized, and often affected by replication of design and code elements.

Invasive Changes. Because of the crosscutting nature of the framework composition code, many of the OO solutions require invasive changes to the internal code of frameworks. For example, the Mediator solution proposed to solve the framework gap problem when using calling frameworks requires the intrusive introduction of composition code in the framework classes in order to provide the necessary interactions to integrate the frameworks. The invasive changes occur in many cases because the composition requires the propagation of internal events between the frameworks. The need for invasive changes is reflected in Table 1 by the "no" mark under the locality property. It also shows that 6 of the 9 OO solutions do not succeed to separate the composition code in modules.

	Modularity Properties				
OO Solution (Composition Problem Id)*	Locality	Composition Transparency	(Un) Pluggability	Scattering	Tangling
1. Concurrency (I) #	no	no	no	Yes	yes
2. Wrapping (I)	no	no	no	Yes	yes
3. Remove and rewrite (I)	no	no	no	Yes	yes
4.Wrapping and Extending(II)	yes	yes	yes	No	no
5. OO Mediator (II)	no	no	no	Yes	yes
6. Redesign and extend (II)	no	no	no	Yes	yes
7. Aggregation (III)	yes	yes	yes	No	no
8. Multiple Inheritance (III)	yes	yes	yes	No	no
9. Observer Pattern (III)	no	no	no	Yes	yes

Table 1. Properties of the Object-Oriented Solutions

* *The "Composition Problem ID refers to the identifier of each framework composition problem, as follows: (I) Composition of Framework Control; (II) Framework Gap; and (III) Composition of Entity Functionality.*

The properties in the Concurrency OO solution refers to the definition of synchronization code when it is necessary in the framework composition.

(Un)plugability. Also, many OO solutions cannot be plugged or unplugged. It means developers cannot add or remove them easily with the purpose of redefining or adapting the composition code. Table 1 shows that 6 of the 9 OO solutions cannot be easily plugged or unplugged. Examples of such unpluggable solutions are the solutions that require the internal modification of frameworks to provide the composition, such as: (i) the Wrapping solution to the framework control problem; (ii) the OO Mediator solution to the framework gap problem; and (iii) the Observer solution to the composition of entity functionality.

Increasing Complexity. The complexity of understanding and modifying the OO solutions increases when developers need to deal simultaneously with different framework compositions. The composition transparency property, presented in the Table 1, represents the possibility to reason about different framework composition solutions used simultaneously. Table 1 shows that 6 of the 9 OO solutions cannot be composed transparently. It happens because many of the OO solutions are not able to separately modularize the code relative to each framework composition. As a conse-

quence, it is difficult to reason independently about each OO composition solution adopted.

3 Composing OO Frameworks with Aspects

This section describes the AO solutions proposed as a more modularized alternatives than the original OO solutions (Section 2.2). The solutions are organized based on the OO composition problems addressed. In some cases, AOP is applied in order to complement the original solution, while in other ones it is used to replace the OO implementation. Also, whenever appropriate, we refer to other research works that have already explored a specific solution although in different contexts. For some of the solutions, we present examples of the composition design and code in the context of the frameworks presented in Section 2.1. In our study, aspect-oriented programming was mainly analyzed as an implementation technology that allows for the modularization of the composition code related to each OO solution. AspectJ was the approach chosen to implement the new AO solutions. Each of new AO solutions proposed was implemented by composing two frameworks.

3.1 Composition of Framework Control

The composition problem of framework control happens because both frameworks expect to provide the main thread control of the application. Three OO solutions are proposed to deal with this problem (Section 2.2.1): (i) concurrency; (ii) wrapping; and (iii) remove and rewrite. In the following, we analyze the impact of AOP in the modularization of each one of the proposed solutions.

Concurrency + Synchronization Aspects[1]. AOP can be used as a complementary technology to the concurrency solution. Synchronization aspects can be defined to control the concurrent execution of common application code of both frameworks. Examples of concurrency control which can be implemented in AspectJ are method execution synchronization and block of conflicting execution flows [26]. The use of AOP to specify concurrency concerns has been addressed by some research works [19, 26]. Since only the synchronization code needs to be codified to this solution, we can say that AOP is used here to implement all the framework composition code. The synchronization aspects can also be used to complement the following solutions based on Observer aspects.

Wrapping >> Observer Aspects[2]. In the original OO solution, each framework is encapsulated by a wrapper. After that, every input and output method call in the frameworks can be intercepted in order to notify the other one. This solution offers the restriction that internal framework events cannot be intercepted by the wrappers. Using AOP this solution can be replaced with Observer aspects to also intercept framework internal events whenever necessary. However, better than defining a complete wrapper for each framework, it can be specified a set of aspects which allow intercepting different framework events, including internal events, and notify the interested entities.

Remove and Rewrite >> Observer Aspects. The "Remove and Rewrite" OO solution is presented by Mattsson et al to deal with the Wrapping solution restriction of not

[1] The "+" symbol indicates a complementary solution to the original solution proposed.

[2] The ">>" symbol indicates a substitutive solution which replaces the original solution proposed.

intercepting framework internal events. The AO solution based on a set of Observer aspects can also be adopted to replace this solution. The AO solution is less effort-consuming than the original OO solution, because it is not necessary to codify internal changes in the framework classes.

Next we present an example of an AspectJ solution as a set of Observer composition aspects. This AO solution addresses the problems of both the "wrapping" and the "remove and rewrite" solutions. Synchronization aspects can be used to complement this solution whenever it is necessary to have concurrency control during the access to framework classes. Our example presents the composition between the Measurement and the GUI frameworks (Section 2.1). The GUI framework will be used to show details about the items that are being processed in the Measurement framework. This section uses an example of an instance of the Measurement framework which implements a beer can system. This application aims at removing dirty beer cans from an input stream. This framework instance is described in [3]. We also will use it to illustrate the AO solutions for framework composition in Sections 3.2 and 3.3.

The implementation of the composition between the Measurement and GUI frameworks defines aspects which are responsible for intercepting the execution of methods occurring in the Measurement framework and capturing information to be visualized in the GUI framework. The information captured by the composition aspect is bypassed to the `TableModel` objects of the GUI framework. The notification protocol between the `TableModel` and `JTable` objects already implemented[3] in the GUI framework guarantees the visualization of the data.

Figure 5 shows the composition aspects and the respective framework classes which it affects. The `MeasurementGUIComposition` aspect defines the common code which is reused always it is necessary a composition between the Measurement and GUI frameworks. It is responsible to intercept events in the `Trigger` class and `PhysicalSensor` and `ActuationStrategy` subclasses in order to update information about the items already processed by the Measurement framework instance. The `BeerCanComposition` subaspect defines the variable code of the composition dependent on the frameworks instances, such as, the method of initialization of both framework instances and the update of `TableModel` objects related to a specific Measurement framework instance. An example of a table model of the Beer Can instance is the `BeerCanTableModel` class, which presents details about the attributes of beer can already processed.

Figure 6 and 7 shows the partial AspectJ code of the composition aspects. The `MeasurementGUIComposition` aspect defines the following functionalities: (i) a set of abstract methods and pointcuts which guarantees the initialization of the GUI framework when the Measurement framework is created (lines 5-9); (ii) it saves a reference to all the `TableModel` objects created in the GUI framework in order to notify them about data update (lines 11-17); and, finally (iii) it intercepts execution of methods in the Measurement framework to capture information to be written in the `TableModel` objects, such as, the beginning of an item processing (lines 21-34), the sensor activation (lines 18-19) and the finalization of an item processing in the ActuationStrategy objects (lines 36-46). The `MeasurementGUIComposition` aspect also maintains internally

3 It is important to emphasize that the Measurement and GUI framework run in different threads. An additional aspect was created to init a new thread, every time the processing of a new item is initiated in the Measurement framework. It is done by intercepting the `doProcess()` method of the `HWBCTrigger` class, which represents a `PhysicalSensor` subclass that inits the measurement process. The GUI framework is instantiated by creating its own thread in the `main()` method of the `GUIMeasurementApplication` class.

data about the processed items with their respective initial and final processing time. This data is passed to the `ProcessedItemTableModel` object of the GUI framework.

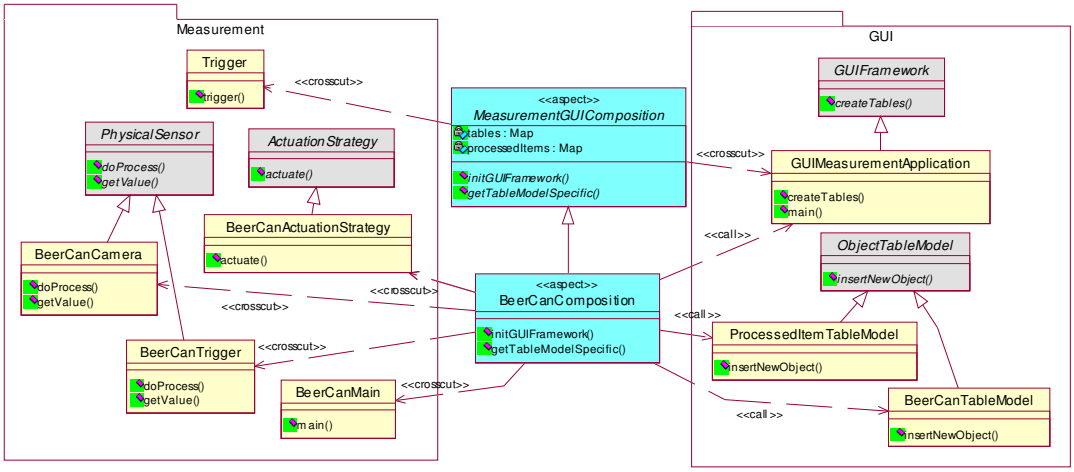

Figure 5. The Composition of the Measurement and GUI Frameworks

```
01 public abstract aspect MeasurementGUIComposition {
02   private Map tables = null;
03   private Map processedItems = new HashMap();
04
05   public abstract pointcut measurementInitialization();
06   before(): measurementInitialization() {
07       initGUIFramework();
08   }
09   public abstract void initGUIFramework();
10   ...
11   public pointcut tableModelInitialization():
12               execution(public Map GUIFramework+.createTables());
13   after () returning(Map tables): tableModelInitialization(){
14      this.tables = tables;
15      this.initStandardTableModels();
16      this.initSpecificTableModels();
17   }
18   public pointcut sensorActivation(PhysicalSensor physicalSensor):
19      execution(public void PhysicalSensor+.doProcess()) &&target(physicalSensor);
20
21   public pointcut itemProcessingInitialization(Trigger triggerSW):
22               execution(public void Trigger.trigger()) && target(triggerSW);
23
24   before(Trigger triggerSW): itemProcessingInitialization(triggerSW){
25       this.createProcessedItem();
26       long threadId = Thread.currentThread().hashCode();
27       ProcessedItem currentItem =
28              (ProcessedItem) this.processedItems.get(new Long(threadId));
29       AbstractTableModel tableModel =
30              (AbstractTableModel) this.getTableModel(triggerSW);
31       if (currentItem != null && tableModelGeneral != null){
32            tableModelGeneral.insertNewObject(currentItem);
33       }
34   }
35   ...
36   public pointcut itemProcessingFinalization(ActuationStrategy actuator):
37     execution(public void ActuationStrategy+.actuate(..)) && target (actuator);
38
39   after(ActuationStrategy actuator): itemProcessingFinalization(actuator){
40      ...
41      ProcessedItem currentItem =
42         (ProcessedItem) this.itemsProcessingTime.get(new Long(threadId));
43      currentItem.setEndTime(new Date());
44      // Updates the respective table model
45      ...
46   }
47 }
```

Figure 6. MeasurementGUIComposition Aspect

The `BeerCanComposition` subaspect specifies the pointcut which represents the initialization of the Measurement framework instance and implements the `initGUIFramework()` method by calling the specific method which initializes the GUI framework. It also implements the `getSpecificTableModel()` method which returns the `TableModel` associated with a specific kind of an object. This method is called by `getTableModel()` in the `MeasurementGUIComposition` aspect. The BeerCanComposition subaspect can also define additional advices and pointcuts which are in charge of updating `TableModel` objects of a framework instance. Figure 2 shows, for example, the definition of an advice associated with the `sensorExecution()` pointcut that updates the `BeerCanTableModel` object based on information captured by the `BeerCanCamera` class, a specific `PhysicalSensor` subclass (lines 10-17).

```
01 public aspect BeerCanComposition extends MeasurementGUIComposition {
02   public pointcut measurementInitialization():
03       execution(public static void BeerCanMain.main(..));
04   public void initGUIFramework(){
05       GUIMeasurementApplication.main(null);
06   }
07   public AbstractTableModel getSpecificTableModel(Object object, Map tables){
08       ...
09   }
10   after (PhysicalSensor physicalSensor): sensorActivation(physicalSensor){
11       AbstractTableModel tableModel =
12               (AbstractTableModel)this.getTableModel(physicalSensor);
13       Object object = physicalSensor.getValue();
14       if (object != null && tableModel != null) {
15          tableModel.insertNewObject(object);
16       }
17   }
18 }
```

Figure 7. `BeerCanComposition` Subaspect

3.2 Framework Gap

Three OO solutions are proposed to solve the manifestation of framework gaps (Section 2.2.2): (i) wrapping and extending; (ii) OO mediator; and (iii) redesign and extend. Below we discuss the potential of AOP to improve each of these solutions. After that, we present an example which extends the composition aspects of the Measurement and GUI frameworks (presented in Section 3.1) to calculate statistic information about the measurement process using the Statistical framework. The composition aspects are adapted to work as a mediator between the three frameworks.

Wrapping and Extending + AO Observer. This solution is proposed to solve the framework gap problem when using called frameworks (Section 2.1). The AO solution complements the original OO solution by allowing a better integration between the frameworks being composed and the extension designed to address the framework gap. An aspect can be used to define different kinds of interaction between frameworks and the extension, including the handling and notification of internal events.

OO Mediator >> AO Mediator. An OO solution based on the use of a mediator software is proposed to address the problem of framework gap when using calling frameworks. AOP can improve this solution by defining the mediating module as an aspect. The mediator aspect defines the integration code between the frameworks and additional extensions to address the application requirements. The AO solution modularizes the code relative to the composition of frameworks and extensions in the mediator aspect. During the independent evolution of each framework and extension, changes can be introduced in a way that is modular to their composition. Localized changes can be done to the mediator aspect.

Redesign and Extend. We believe that for many cases the AO Observer and Mediator solutions presented previously are adequate solutions to solve the framework gap problem because they do not require a high coupling between frameworks and extension components. Such AO solutions can often avoid the need for changing the involved framework implementations. The alternative of redesigning and extending should be used only in cases where: (i) there is a strong and stable connection between the frameworks and extension domains which motivates the development of a unique framework; (ii) the new framework will be used in the development of many future applications; and (iii) it is difficult to compose the framework and extensions using the AO observer and mediator solutions.

In order to illustrate the aspect-oriented mediator solution to address the problem of framework gap, we assume the need to extend the composition between the Measurement and GUI frameworks (Section 3.1) with statistical data related to the measurement process. The following statistical functionalities are implemented to fill the framework gap: (i) calculate the amount of items processed by the measurement framework; (ii) calculate the best, the worst, and the average time to process each item; and (iii) calculate the percentage of acceptable and unacceptable products. The statistical analysis framework (Section 2.1) can be used to address these functionalities.

Figure 8 shows the design of the composition of the Measurement, GUI and Statistical frameworks. The `MeasurementGUIComposition` aspect now specifies the common composition code between all instances of the Measurement, GUI and Statistical frameworks. It uses the Statistical framework to address the new statistical functionalities. The following new classes were also created in order to overcome the framework gap: (i) two concrete algorithms (`ProcessingTimeAlgorithm`, `BeerCanQualityAlgorithm`) and statistics (`ProcessingTimeStatistic`, `BeerCanQualityStatistic`) to calculate the times for processing the items and the percentage of acceptable and unacceptable products. These classes represent the instantiation of the Statistical framework to address the framework gap; and (ii) the `ProcessingTimeTableModel` and `BeerCanQualityTableModel` classes which represent the statistical data to be shown in the GUI framework.

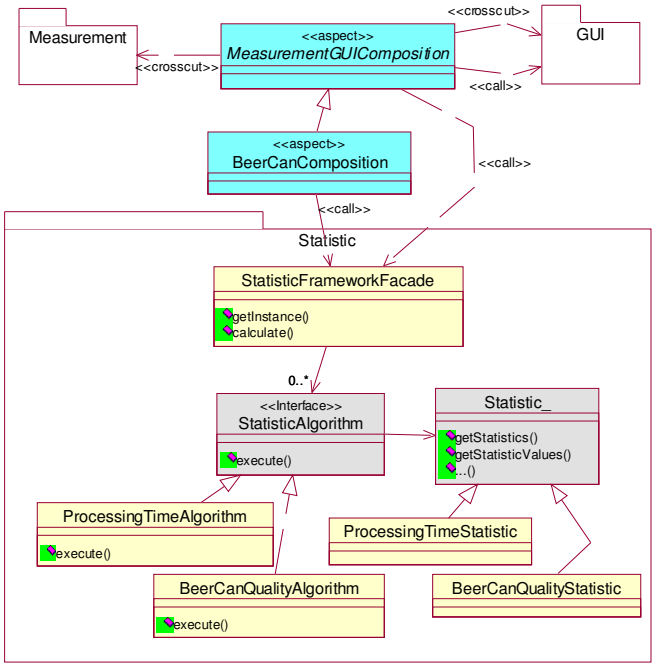

Figure 8. The Measurement, GUI and Statistical Framework Composition

```
01 public abstract aspect MeasurementGUIComposition {
02   private Thread statisticalService = null;
03   ...
04   public abstract pointcut measurementInitialization();
05   before(): measurementInitialization() {
06      this.configureStatisticFramework();
07   }
08   public void configureStatisticFramework(){
09      StatisticFrameworkFacade statisticFramework =
10                     StatisticFrameworkFacade.getInstance();
11      StatisticAlgorithm algorithm = new ProcessingTimeAlgorithm();
12      statisticFramework.registerAlgorithm("Measurement", algorithm);
13      this.registerSpecificStatisticAlgorithm(statisticFramework);
14      this.initStatisticalService(10000);
15   }
16   public abstract void registerSpecificStatisticAlgorithm(
17                            StatisticFrameworkFacade facade);
18   public void initStandardTableModels(){
19      ...
20      DataTableModel dataTableModel = new ProcessingTimeTableModel();
21      table = new JTable(dataTableModel);
22      this.tables.put("Processing Time Statistics", table);
23   }
24   public void abstract initSpecificTableModels();
24   private void initStatisticalService(long delay){
25      if (statisticalService == null){
26        statisticalService = new Thread() {
27           public void run() {
28             while (true){
29                 try {Thread.sleep(delay);
30                 }catch(InterruptedException e){}
31                 calculateStatistics();
32             }
33           }
34         };
35        statisticalService.start();
36}
37   }
38   private void calculateStatistics(){
39      Map statistics = this.calculateGeneralStatistics();
40      this.updateGeneralStatistics(statistics);
41      this.updateSpecificStatistics(statistics);
42   }
43   private Map calculateGeneralStatistics(){
44      StatisticFrameworkFacade statisticFramework =
45                          StatisticFrameworkFacade.getInstance();
46      Map statistics = statisticFramework.calculateStatistics(
47                               "Measurement", processedItems.values());
48      return statistics;
49   }
50   private void updateGeneralStatistics(Map statistics){
51      // Update some statistics in their respective table models
52      ...
53   }
54   public abstract void updateSpecificStatistics(Map statistics);
55 }
```

Figure 9. `MeasurementGUIComposition` Composition Aspect

Figure 8 also shows the `BeerCanComposition` subaspect that was specified to: (i) allow the register of specific algorithms and statistics of an instance of the Measurement framework, such as `BeerCanQualityAlgorithm` and `BeerCanQualityStatistic`; and (ii) to define the creation and manipulation of specific statistic table models (such as, `BeerCanQualityTableModel` class).

Figure 9 shows the partial code of the `MeasurementGUIComposition` aspect to implement the statistical functionalities. It has now the following additional responsibilities: (i) it configures the Statistical framework by registering specific algorithms to be executed (lines 5-15); (ii) it creates the statistical `JTable` objects and respective `TableModel` objects to visually present the statistical data (lines 18-23); and finally, (iii) it initializes a thread which from time to time calculates the new statistic data by invoking the Statistical framework services and updates the `TableModel` objects with this

new statistical information (lines 24-53). Many abstract methods of the `Measurement-GUIComposition` aspect are called by template methods in order to make it possible to its subaspects register new statistic algorithms to be executed (lines 16-17), create new statistic table models (line 24) and update the table models with new statistic data calculated (line 54). The `BeerCanComposition` subaspect (Figure 7) specifies implementations for all these methods.

3.3 Composition of Entity Functionality

Mattsson et al propose three solutions to cope with compositions of entity functionalities (Section 2.2.3): (i) aggregation; (ii) multiple inheritance; and (iii) Observer design pattern. Both aggregation and multiple inheritance solutions present restrictions related to the difficulty to manage state updates in the composed classes resulting from the use of those mechanisms [20, 21]. The Observer-based solution deals with this restriction of the other solutions, but it requires many invasive changes in the framework classes. Below we present an alternative solution which avoids the drawbacks of OO solutions.

Entity-Functionality "Glue" Aspect. Our proposed AO solution to overcome compositions of entity functionality is based on an aspect to advise specific joinpoints in the domain-specific framework classes. This aspect also specifies the execution of the additional functionality by invoking the second framework at the occurrence of those joinpoints. Thus, the aspect works as a glue code between the frameworks.

As an example of composition of entity functionality, we present the composition of the Persistence framework (Section 2.1) with the Measurement, GUI and Statistical framework composition presented in previous Sections 3.1 and 3.2. Figure 10 shows this composition. The `PersistenceComposition` and `BeerCanPersistenceComposition` aspects work as a "glue" between the GUI, the Statistical and the Persistence frameworks. This latter stores the domain-specific data using different DAOs (`ProcessedItemDAO` and `ProcessingTimeStatisticDAO` classes).

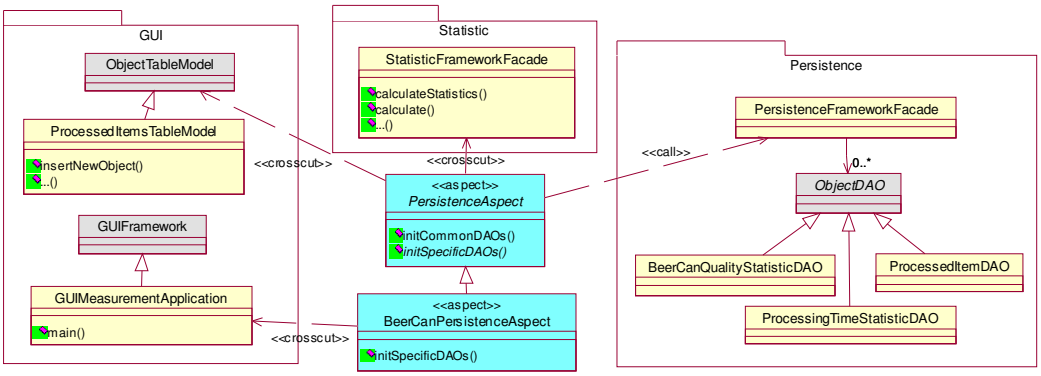

Figure 10. Composition of the GUI, Statistical and Persistence Frameworks

Figure 11 shows the source code of the persistence composition aspects. The `PersistenceComposition` aspect specifies an abstract pointcut that represents the creation of an instance of the GUI framework and an associated advice which initializes the Persistence framework (lines 4-9). It also creates and configures in the Persistence framework a set of default DAOs to be used in all applications instantiated from the resulted framework composition (lines 21-27). Finally, it intercepts (i) the `insertNewObject()` method of the `ObjectTableModel` subclasses to persist information pre-

sented by the GUI framework (lines 10-15). and (ii) the `calculateStatistics()` method of the `StatisticFrameworkFacade` class in order to persist statistic data calculated by the Statistic framework (lines 16-20). The `BeerCanPersistence` Composition subaspect specifies the pointcut of the GUI framework instance initialization (lines 2-3) and defines, if they exist, new concrete DAOs (lines 5-9).

```
01 public abstract aspect PersistenceAspect {
02   PersistencyFrameworkFacade persistenceFramework = null;
03
04   public abstract pointcut initializePersistenceService();
05   after (): initializePersistenceService(){
06      this.persistenceFramework = PersistencyFrameworkFacade.getInstance();
07      this.initCommonDAOs();
08      this.initSpecificDAOs();
09   }
10   public pointcut processedItemsTableModel(Object object):
11               execution(public void ObjectTableModel.insertNewObject(Object))
12               && args(object);
13   after(Object object): processedItemsTableModel(object){
14      this.persistenceFramework.saveOrUpdateObject(object);
15   }
16   public pointcut statisticalDataPersistence():
17      execution(public Statistic StatisticAlgorithm+.execute(..));
18   after () returning(Statistic statistic): statisticalDataPersistence(){
19      this.persistenceFramework.saveOrUpdateObject(statistic);
20   }
21   public void initCommonDAOs(){
22      this.persistenceFramework.configureDAO(
23            ProcessingTimeStatistic.class.getName(),
24            new ProcessingTimeStatisticDAO());
25      ...
26   }
27   public abstract void initSpecificDAOs();
28   ...
29 }

01 public aspect BeerCanPersistenceAspect extends PersistenceAspect {
02   public pointcut initializePersistenceService():
03       execution(public static void GUIMeasurementApplication.main(..));
04
05   public void initSpecificDAOs(){
06       this.persistenceFramework.configureDAO(
07            BeerCanQualityStatistic.class.getName(),
08            new BeerCanQualityStatisticDAO());
09   }
10 }
```

Figure 11. Code of the GUI, Statistical and Persistence Framework Composition Aspects

4 Analysis of the AO Solutions

This section presents a detailed analysis of our case study (Section 2.1). Section 4.1 presents compares the AO and OO solutions with respect to the modularity properties (Section 2.3). Section 4.2 discusses several issues related to the design and implementation of framework composition with aspects.

4.1 Modularity Analysis of the AO Solutions

For every framework composition problem addressed, we observed many improvements when using the AO solutions. These improvements can be observed mainly in terms of modularization of the composition code. Table 2 summarizes the improvements of each AO solution. Similar to the analysis of the OO solutions (Section 2.3), we have also used the three modularity properties: locality, composition transparency, and unpluggability. The complementary property, presented in the Table 2, indicates if the AO solution is used in conjunction with the original OO solution proposed.

The total of 5 new AO solutions were characterized as an alternative to the OO solutions previously proposed. AOP was used as a complementary solution in the implementation of 2 AO solutions. All these 5 AO solutions brought benefits in relation to the OO solutions (see Table 2). As we mentioned before, many of the AO solutions have been explored by the AOSD community, although not in the context of framework composition.

	Modularity Properties			
AO Solution - Composition Problem ID *	Locality	Composition Transparency	Unpluggability	Complementary
1. Concurrency + Synchronization Aspects (I)	yes	yes	yes	yes
2. AO Observer (I)	yes	yes	yes	no
3. OO Wrapper + AO Observer (II)	yes	yes	yes	yes
4. AO Mediator (II)	yes	yes	yes	no
5. "Glue" Aspect (III)	yes	yes	yes	no

Table 2. Properties of the Aspect-Oriented Solutions

** The Composition Problem ID refers to the identifier of each framework composition problem, as follows: (I) Composition of Framework Control;(II) Framework Gap; (III) Composition of Entity Functionality.*

Our study shows mainly that AOP can be used as an effective technology to completely modularize the framework composition code. In our study, all the AO solutions achieved this objective. In general, each framework composition problem presents the need to integrate some of the frameworks features, such as: control flows, entities, or functionalities. The majority of the OO solutions need to define invasive changes in the framework code in order to implement the required composition code. Using aspects, we can define the specific points in the execution of a framework that could be linked to the functionality of another framework (or legacy component) in order to address their composition. The presented AO solutions also avoid the code scattering and tangling presented by many OO solutions (Section 2.3).

The complete modularization of the framework composition code in the AO solutions also brings the benefit of plugging and unplugging whenever necessary specific compositions. Since the integration code is completely codified in aspects, developers can easily add or remove the composition code during the implementation or evolution of software architectures based on their specific design decisions. Thus, this (un)pluggability characteristic gives to the developers more flexibility when deciding for the use of specific framework compositions.

Also, the complete modularization of framework composition code helps in the documentation of the adopted solutions. In fact, the aspects used in each solution can be viewed as a direct documentation of the design decisions considered by the system developers to implement the framework composition. As a consequence, this improved documentation brings benefits to the understanding and maintenance of the code relative to the framework composition. In software applications, where developers need to specify different and large amount of framework compositions this improved documentation is even more important.

4.2 Design and Implementation Issues

In this section we discuss several issues related to the design and implementation of framework composition with aspects. The discussed issues are related to the reusability and modularization of the composition aspects and also to guidelines for making it easier the composition between frameworks.

4.2.1 Common and Variable Composition Code

The implementation of the composition aspects in our study brought the need to separate common and variable code related to the framework composition. The common composition code implements the functionalities that will be used in every application instantiated from the resulted framework composition. For example, in the Measurement and GUI framework composition (Section 3.1), the `MeasurementGUIComposition` aspect specifies the interception of several internal classes (`Trigger`, `ConcreteSensor`, `ActuationStrategy`) of the Measurement framework in order to present information about the measurement process in `TableModel` objects of the GUI framework. The variable composition code implements the specific functionality which is dependent on the instantiation of the framework composition. An example in the context of the Measurement and GUI framework composition was the definition of the initialization methods of each framework instance, which are implemented in the `BeerCanComposition` subaspect.

The separation between the common and variable code allows to reuse many lines of code when instantiating different applications from the resulted framework composition. It also gives flexibility to concretize specific part of the composition code when it is necessary. In certain cases, developers do not need to specify a variable composition code or do not intend to reuse the framework composition in different applications; they could codify all the composition code in only one aspect without the need to distinguish the common and variable code. Aspect-oriented refactoring of the composition aspects can be done whenever there is a demand for generalizing the composition code.

4.2.2 Modularization of the Composition Aspects

Another issue related to the design and implementation of the composition aspects is how to modularize them in order to define flexible different combinations of the frameworks composition. In the solution described in Section 3.2, for example, the statistical functionality was added to the implementation of the composition aspect responsible to integrate the Measurement and GUI frameworks. An alternative design could separate the statistical functionality in a different composition aspect which intercepts directly events of interest in the Measurement and GUI frameworks. It makes possible to plug and unplug the statistical functionality. In this design, two different hierarchies of aspects must be created: (i) one to manage only the Measurement and GUI framework composition, and (ii) the other one to specify the Measurement, Statistical and GUI framework composition that addresses only statistical functionalities. Since the aspect hierarchies must intercept a common set of join points in the Measurement framework, a different aspect could be implemented to expose the set of shared pointcuts[4].

[4] Next subsection 4.2.3 shows that this aspect can work as the syntactic part of a crosscutting programming interface (XPI) [29] of the Measurement framework.

Another example is related to the composition of the persistence functionality with the Measurement and GUI framework composition. We have many options to compose the persistence framework, such as: (i) to add the calls to the persistence framework directly in the composition aspects presented in Section 3.2; (ii) to create a new composition aspect which integrates the Measurement and Persistence frameworks; and (iii) to create a new composition aspect which only integrates the GUI and Persistence framework. The first option does not allow plugging and unplugging automatically the persistence functionality from the other framework compositions, but it avoids the implementation of new aspects. The second option could be suggested in cases you have the combination of Measurement and Persistence framework without the GUI functionality. Finally, the third option, it is recommended in cases you can have the independent composition of the GUI and Persistence framework without the presence of Measurement framework.

Thus, the separated implementation of different composition aspects can make it possible to choose different combinations of frameworks to be integrated. A careful design must be done by modeling a number of composition aspects which allow to instantiate automatically different framework compositions of interest.

4.2.3 Exposing Framework Pointcuts

The composition of frameworks requires in many situations the interception of some events in the internal framework classes. In the composition with the Measurement framework, for example, the execution of triggers, sensors and actuators were intercepted by the composition aspects in order to capture some information and call additional functionalities from the other frameworks. We believe it is fundamental to make it visible relevant joinpoints that are part of the internal execution of the framework. The goal is to make it easier the design and implementation of a composition aspect.

A traditional OO framework documents its extension points, such as abstract classes and interfaces, in order to enable framework users to instantiate their applications. We advocate the additional documentation of relevant framework join points to enable developers to compose it with additional frameworks or extensions. We believe that crosscutting programming interface (XPI) [29, 11] is an appropriate approach to document the framework joinpoints since it can establish a contract between the "base" code provided by the framework and the composition aspects. Besides to allow the specification of the *extension join points* of a framework, they also determine a set of constraints, pre- and pos-conditions which must be satisfied when composing or extending the framework functionality. Thus, the use of XPIs to document and externalize the framework extension join points can help: (i) to expose only those details of the framework code which are interesting to compose with other extensions; and (ii) to analyze the impact of a framework internal change (such as, a refactoring) in their respective composition aspects.

According to its authors [29, 11], an XPI has: (i) a syntactic part – which allows to expose specific join points; 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. The syntactic part of an XPI can be expressed in a programming language, such as AspectJ; and (ii) the semantic part can be specified as a combination of natural and formal languages. In Section 4.2.2, an aspect was created to specify a set of common pointcuts of the Measurement framework. These pointcuts are reused by different aspects that composes separately the Measurement and GUI frameworks and the Measurement, Statistical

and GUI frameworks. The aspect which exposes the shared pointcuts can be seen as the syntactic part of an XPI. All the composition aspects with the Measurement framework are defined in terms of the join points exposed by that aspect. This XPI must also expose the design rules which specify constraints on behavior composed across the extension join points of the Measurement framework. Examples of such constraints are: (i) to prohibit changes to the trace of all the framework joinpoints exposed to avoid causing disturbance in the measurement process; and (ii) to create separate threads to execute the additional compositions in order to not affect the performance of the Measurement framework.

4.2.4 Framework Composition Causes

In their original study, Mattson et al [20, 21] identify the main causes responsible for the framework composition problems, which are: (i) the framework cohesive behavior; (ii) the domain coverage; (iii) the design intention; and (iv) the access to the framework source code. Below we discuss the contribution that aspect-oriented programming can bring to deal with these causes. The framework cohesive behavior refers to the inherent interaction between the framework classes to implement a software family architecture. Mattson et al argue that many of the composition problems with the framework are caused by that cohesive behavior. It happens because external classes which extend or are composed with the framework must not only implement an adequate domain behavior but they also must have a correct cohesive behavior. We believe the documentation of the framework extension join points, using the XPI approach, for example, can make visible internal extension points of the framework. It can also provide rules that prevent the external classes to break the framework cohesive behavior.

The domain coverage cause is related to how a framework composition covers a specific problem domain. During the framework composition, the developer can have to deal with three situations: (i) no domain overlapping between the frameworks; (ii) little overlapping; and (ii) considerable overlapping. In this work, we have proposed the use of aspects to enable the composition of frameworks in order to address a framework gap. In this case, there is no domain overlapping between the frameworks, but it can be necessary to intercept internal events happening in some of them in order to compose them to cover the problem domain. The problems related to the little and considerable overlaps between frameworks are not considered in this work. Section 5 presents related work which aims to address these problems.

Object-oriented frameworks are in general designed to be reused through extension and not through composition. In contrast, called frameworks are clearly easier to compose than calling frameworks, because they provide an interface which exposes their services and they can play a passive role in the application. Mattsson et al argue that it is fundamental for a framework to explicit which ways of reuse it provides, if only by extension or also by composition. They present the lack of explicit design intentions as one of the causes of many framework composition problems. In our work, we also advocate the importance of making explicit the framework design intention to facilitate not only its extension but also its composition. Besides, we believe the exposition and documentation of extension join points can improve the framework reuse in the following scenarios: (i) to implement new and not anticipated crosscutting extensions in the framework (such as, optional features); (ii) to compose easier the framework with other ones. Finally, we encourage the implementation of called frameworks, since their composition is supposed to be easier than calling frameworks.

The last main cause presented by Mattson et al is the lack of access to the source code of the framework. As we saw in Section 2.3, for many OO framework composition solutions proposed by the authors, it is necessary to codify invasive changes inside the framework classes. The authors also argue that developers can have access to the framework code but it is very difficult to understand its internal structure to compose it with another framework. Again here, we believe the documentation of the extension join points provide the developers with the relevant points of crosscutting composition and do not expose a large amount of classes and behaviors which they do not need to understand or are not interested.

5 Related Work

Sullivan and Notkin [31] explore the difficulties related to the development and evolution of integrated systems. They propose a mediator-based design approach to deal with these integration difficulties. Their approach addresses the integration of objects by modularizing the behavioral relationships that integrate them. The abstraction used to specify these behavioral relationships is called mediators. Also, a new language mechanism, called abstract behavioral type (ABT), is proposed to support the specification of mediators. In a more recent work, Sullivan et al [29] investigate the usefulness of AspectJ to the modularization of their mediators. They argue that the main limitation of AspectJ is related with its model of aspect instances. Although it is possible to specify a mediator as an AspectJ aspect to provide the composition code between different objects, it also requires costly work-arounds to manage their respective associations. The main restriction of AspectJ presented by the authors to modularize the composition code was totally valid for our study; we needed to create several list objects in the implementation of many aspects to manage the dependence between the framework objects. Two recent research work [23, 24] address the restriction of the AspectJ language by offering more flexible aspect instances model.

As we mentioned before, the AspectJ implementations of the Adapter, Mediator and Observer design patterns [9] presented by Hannemann and Kiczales [12] can be used to compose the object-oriented frameworks. These design patterns represent classical solutions to the integration of components/objects. Also, modularity properties used by the authors to compare Java and AspectJ implementation of design patterns were used to analyze the characteristics of OO and AO solutions to compose frameworks.

DeLine [6] discusses the problems related to the integration of software components into a system. He shows that the component integration requires an appropriate interaction with the systems' other components. The author defines the term component packaging as the predefined commitments about how a component interacts with other components in the system. DeLine indicates that when a component implements a useful functionality but it presents an inappropriate packaging, a problem of packaging mismatch is characterized. A catalog of techniques for resolving packaging mismatch is offered by the author [6]. DeLine proposes an approach, called Flexible Packaging, to deal with the packaging mismatch. It promotes the code separation of component functionality from the component packaging (interaction). The approach allows reusing separately the component functionality and component packaging in different contexts. There are many similarities between the catalog (problems and solutions) presented by DeLine to integrate components and the composition aspects used as base in this study to compose OO frameworks. The Flexible Packaging approach can be viewed as an aspect-oriented approach that promotes the separation of component

integration from its base functionality. Thus, DeLine's work has also focused on the separation of composition code, although using a different kind of aspect. As a future work, we intend to explore the use of mature aspect-oriented approaches (e.g., AspectJ) to implement the solutions proposed by DeLine.

Feature oriented approaches (FOAs) have been proposed [25] to deal with the encapsulation of program features that can be used to extend the functionality of existing base program. Batory et al [2] argue the advantages that feature-oriented approaches have in respect to object-oriented frameworks to design and implement product-lines. Mezini and Ostermann [22] have identified that FOAs are only capable to modularize hierarchical features. They do not support the specification of crosscutting features. They propose CaesarJ [22], an AO language that combines ideas from both AspectJ and FOAs, to provide a better support to manage variability in product-lines. The work of those authors has a direct relation to our work, since we believe that the design of product-line architectures can require the composition of different frameworks using aspects. As an ongoing work, we intend to investigate and compare the use of these two different approaches to manage variability in a software product-line in order to better understand their respective benefits and limitations. The feature-oriented mechanisms of CaesarJ, for example, seem to be an appropriate approach to manage different classes and aspects which modularize the same feature in the framework composition.

Some other authors have also discussed the problems and solutions related to the composition of frameworks or components. Hölzle [14] discusses the problems related to the composition of independently-developed components. The author also proclaims in that work the advantages that an existing aspect-oriented approach (subject-oriented programming) could bring to the component and framework integration. Sparks et al [27] discuss object-oriented solutions to deal with the framework gap problem.

6 Conclusions and Ongoing Work

Most of the existing solutions for framework compositions involve some form of crosscutting between the framework classes and the composition strategy. This study is a first systematic assessment about the benefits and drawbacks on the use of the AOP technology to implement modular composition of OO frameworks. A significant catalog of OO solutions that address framework composition was revisited and analyzed against a set of modularity properties. The study also demonstrates how aspect- and object-oriented techniques must be used as complementary technologies. Aspect-oriented solutions were designed for a significant case study involving the integration of four OO frameworks. The study encompassed three main composition challenges, namely composition of framework control, framework gap, and composition of entity functionality.

Our study concluded that AOP is a promising technique to support the modular composition of different frameworks addressing both vertical and horizontal domains. However, to facilitate the composition process, it is desirable that framework designs be planned for potential compositions by exposing and documenting internal framework events as possible integration points. As ongoing work, we plan to evaluate the implementation of the AO framework composition solutions using other AOP approaches, such as, EoS [22] and CaesarJ [23]. Our aim is to verify how these approaches can help to simplify the AspectJ implementations. Finally, we also intend to explore

the use of XPI [11, 29] in the design and implementation of other object-oriented frameworks in order to expose their extension join points.

References

[1] ALUR, D.; MALKS, D.; CRUPI, J. Core J2EE Patterns: Best Practices and Design Strategies. Prentice Hall PTR, 2nd edition, 2003.

[2] BATORY, D.; CARDONE, R.; SMARAGDAKIS, Y. Object-Oriented Frameworks and Product-Lines. 1st Software Product-Line Conference (SPLC), Denver, August 1999.

[3] BOSCH, J. Design of an Object-Oriented Framework for Measurement Systems. In Domain-Specific Application Frameworks, M.Fayad et al (eds), John Wiley, pp.177-205, 1999.

[4] CACHO, N.; SANT`ANNA, C.; FIGUEIREDO, E.; GARCIA, A.; BATISTA, T.; LUCENA, C. Composing Design Patterns: A Scalability Study of Aspect-Oriented Programming. Proc. 5th Conference on Aspect-Oriented Software Development (AOSD'06), Bonn, Germany, March 2006.

[5] CZARNECK, K.; EISENECKER, U. Generative Programming: Methods, Tools, and Applications, Addison-Wesley, 2000.

[6] DE LINE, R. Avoiding packaging mismatch with Flexible Packaging. IEEE Transactions on Software Engineering 27(2):124-143, February 2001.

[7] FAYAD, M.; SCHMIDT, D.; JOHNSON, R. Building Application Frameworks: Object-Oriented Foundations of Framework Design. John Wiley & Sons, September 1999.

[8] FOWLER, A. A Swing Architecture Overview, Sun Developer Network, December 2005. URL: [http://java.sun.com/products/jfc/tsc/articles/architecture/].

[9] GAMMA, E.; et al. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading, 1995.

[10] GARCIA, A.; SANT'ANNA, C.; FIGUEIREDO, E.; KULESZA, U.; LUCENA, C; VON STAA, A. Modularizing Design Patterns with Aspects: A Quantitative Study. Proc. of the 4th Conference on Aspect-Oriented Software Development (AOSD'05), Chicago, USA, March 2005.

[11] GRISWOLD, W.; et al, "Modular Software Design with Crosscutting Interfaces", IEEE Software, Special Issue on Aspect-Oriented Programming, Jan/Feb 2006.

[12] HANNEMAN, J.; KICZALES, G. Design Pattern Implementation in Java and AspectJ. Proceedings of OOPSLA'02, November 2002, pp. 161-173.

[13] Hibernate Relational Persistence For Idiomatic Java, http://www.hibernate.org/.

[14] HÖLZLE, U. Integrating Independently-Developed Components in Object-Oriented Languages. Proceedings ECOOP '93, LNCS, 1993.

[15] KICZALES, G.; et al. Aspect-Oriented Programming. Proc. of`ECOOP'97, LNCS 1241, Springer-Verlag, Finland, June 1997.

[16] KVALE, A.; LI, J. ; CONRADI, R. A Case Study on Building COTS-Based System using Aspect-Oriented Programming. Proceedings of SAC'2005, pp. 1491-1498.

[17] KICZALES, G.; et al, "Getting Started with AspectJ," Comm. ACM, vol. 44, pp. 59--65, 2001.

[18] LEA, D. Concurrent Programming in Java: Design Principles and Patterns, 2nd Edition, Addison Wesley Professional.

[19] LOPES, C. D: A Language Framework for Distributed Programming. PhD Thesis, Northeastern University, 1997.

[20] MATTSON, M.; BOSCH, J.; FAYAD, M. Framework Integration: Problems, Causes, Solutions. Communications of the ACM, 42(10):80–87, October 1999.

[21] MATTSON, M.; BOSCH, J. Framework Composition: Problems, Causes, and Solutions. In "Building Application Frameworks: Object Oriented Foundations of Framework Design" Eds: M. E. Fayad et al, Wiley & Sons, ISBN 0-471-24875-4, 1999, pp. 467-487.
[22] MEZINI, M.; Ostermann, K. "Variability management with feature-oriented programming and aspects". Proceedings of Foundation on Software Engineering (FSE'2004), SIGSOFT, pp. 127-136, 2004.
[23] RAJAN, H.; SULLIVAN, K. Eos: Instance-level aspects for integrated system design. In Proe. of ESEC/FSE, pp.297-306, 2003.
[24] SAKURAI, K. ; et al. "Association Aspects", Proceedings of the AOSD´2004, pp. 16-25, Lancaster, UK, 2004
[25] SMARAGDAKIS, Y.; BATORY, D. Mixin Layers: An Object-Oriented Implementation Technique for Refinements and Collaboration-Based Designs, ACM Transactions on Software Engineering and Methodology, April 2002.
[26] SOARES, S. An Aspect-Oriented Implementation Method. Doctoral Thesis, Federal Univ. of Pernambuco, Oct 2004.
[27] SPARKS, S.; BENNER, K.; FARIS, C. "Managing Object-Oriented Framework Reuse", IEEE Computer, pp. 53-61, September 1996.
[28] Spring Framework, http://www.springframework.org/.
[29] SULLIVAN, K.; et al. "Information Hiding Interfaces for Aspect-Oriented Design", In the Proceedings of ESEC/FSE´2005, 5-9 Sept 2005, Lisbon, Portugal
[30] SULLIVAN, K.; et al. Non-Modularity in Aspect-Oriented Languages: Integration as a Crosscutting Concern for AspectJ. In Proceedings AOSD'2002, pp. 19-27, 2002.
[31] SULLIVAN, K.; NOTKIN, D. Reconciling Environment Integration and Software Evolution. ACM TOSEM, 1(3):229-268, July 1992.