Abstract

The growing use of non-functional requirements in computational systems has made researchers focus their attention on methods, methodologies and processes to deal with it. The goal of this paper is to establish a well-defined model that guides the developer through the process of software development. The main idea of this model is to determine a systematic way to decompose both functional and non-functional requirements. Non-functional concepts are usually highly intermixed with functional ones. Our model, FRIDA, is based on the aspect-oriented software development, so that this intermixing is avoided. In this paradigm, the aspect is a first-class element, which is used to describe the non-functional features related to any application. In our approach, the crosscutting nature of the requirements is considered during all software lifecycle. Furthermore, each non-functional requirement is represented with one or more aspects, and functional requirements are restricted to classes hierarchies. Moreover, the UML and some extensions of it are used to determine this model, and allow the modeling of both objects and aspects involved in the system.

Keywords: Software Decomposition, Non-Functional Requirements, Aspect-Oriented Software Development.
1 Introduction

The object-oriented paradigm is dominant in the development of computational systems. The ideas involved in this model are related to building software in which the problem is decomposed into objects. Note that the object-orientation encapsulates categories of concerns. However, sometimes it is not easy to effectively model one single concern as a class, and further use this class to bind this concern to other classes. Moreover, the object-oriented model is essentially based on few modular units (classes and objects) [17, 18]. It promotes the phenomenon also called “tyranny of dominant decomposition” [17], because it supports a poor set of concerns simultaneously [9, 17].

The object-oriented paradigm allows the representation of all concerns using an only dimension – the class. This single dimension does not permit all concerns to be encapsulated [2, 17]. On the other hand, the current categories of applications (for example, distributed, fault-tolerant, Web-based, among others) are composed of different kinds of concerns (usually Non-Functional Requirements – NFRs). As observed by [22], these concerns are regarded as a multi-dimensional quality property. For instance, the NFR security [7] encompasses a great number of quality attributes such as integrity, confidentiality, reliability, and so on. Thus, when a system needs security, it is necessary to analyze and consider all the complementary or additional security properties, and how they affect other system components.

The most common way adopted by several researchers to promote the multidimensional development is to enhance the object-oriented paradigm providing the Separation of Concerns – SoC – principle through additional dimensions [14]. This principle promotes the decomposition of the software into comprehensible parts. Moreover, when SoC is applied to all software development lifecycle, the software complexity is minimized, the integration of its parts is simplified, the reuse is maximized, and the understanding and maintainability of the software becomes easier [18, 17]. It is possible to find in the literature several approaches that are based on the SoC principle, for instance [1]: Composition Filters, Adaptive Programming, Multidimensional Separation of Concerns, Subject-Oriented Programming, Meta-object Protocols and Aspect-Oriented Programming – AOP.

As the focus of our work is on NFRs decomposition, elicitation and modeling, we have chosen the AOP. This is due to a number of reasons as follows:

- traditional OO design methodologies based on classes and relationships are not adequate to express quality attributes [22];
- the goal of AO – Aspect-Oriented – technology is to provide mechanisms to identify, separate, represent and compose crosscutting concerns [6];
- AO aims at providing better means of addressing the separation of concerns [18].

We decide the modeling of NFRs during all software lifecycle, i.e. from the initial phases of the process development, so that the chances of success for the project are maximized [4, 6]. On this scenario, we propose a model called FRIDA – From Requirements to Design using Aspects – in which the key objective is to guide the developer through the basic phases of the software lifecycle.

The general lines of FRIDA concentrate on how to model NFRs and their further decomposition throughout the software development phases. Furthermore, it is used to aid in capturing the NFRs and their relationships in the earlier phases of the development. Another advantage is based on the linking between requirements and design elements. Using this linking allows the traceability of concerns throughout the whole software lifecycle, and enhances the comprehensibility of the application.

As it follows, section 2 provides an overview of the AOSD – Aspect-Oriented Software Development – terminology used as the basis for this paper. In section 3 the related works are mentioned. Section 4 describes our approach, while section 5 concludes the paper and identifies possible future directions.

2 AOSD Terminology

In the traditional models the crosscutting concerns are intertwined with the functional units. In the aspect-oriented model it is feasible to analyze, design and implement the NFRs separately from the basic system functionality [23]. It is due to the aspect-oriented promise to solve problems involved in crosscutting concerns: tangled and scattered code [14].

The AOSD has not abandoned the concepts involved in object-orientation. In this model, both objects and aspects co-exist in an integrated way, using a mechanism called aspect weaving [14, 17]. This mechanism is responsible for combining the aspect code into well-defined places, called join-points, in the functional code. Moreover, in AOSD, software properties that are not directly represented by functional units are expressed as aspects. Generally, the aspects are strongly related to quality attributes [14].
In short, the AOSD is built upon the object-oriented technology. However, it adds new concepts such as, aspect, join-point, pointcut and advice. The purpose of this section is to enumerate the key terminology established for AOSD and its relationship with this work.

Aspect is a single unit that will reflect a non-functional feature. In most cases, this unit crosscuts the functional units (procedures, functions, classes, etc.). Actually, it should be considered a crosscutting concern representation [14]. Some of the most used examples of aspects include [14, 12, 17, 23]: real-time constraints, synchronization, monitoring, remote call, communication, resource sharing, persistence, transaction control, distribution, fault-tolerance, QoS – Quality of Service, and so on.

For instance, the persistence of a set of application classes should be modeled in two ways. The first does not consider the aspect element, only the classes (Figure 1 – left side). To take into consideration the AOSD concepts, it is feasible to model the application as according to what Figure 1 depicts on the right side.

Figure 1. Classes and Aspects

Join-points are the points in which the aspects crosscut the functional concerns. Remember that it is a point where the aspect weaving inserts, into the functional code, the code related to aspects. Some of these points are relevant to the program execution, such as [14] (i) method execution and invocation; (ii) constructor execution and invocation; (iii) attributes access; (iv) exception handling and (v) classes and objects initialization.

The pointcuts are a construction that binds together join-points based on a well-defined rule. These combinations are make through operators, such as "and", "or" and "not" [23]. For instance, method invocation, instantiation, exception handling, and so on.

An advice is a code slice, similar to a method, which is executed when a join-point is reached. Actually, it determines the moment that the join-point will be executed. It is classified according to the following categories: (i) before, i.e., this advice will be executed immediately before the join-point; (ii) after, i.e., this advice will be executed immediately after the join-point and; (iii) around, i.e., this advice will be executed while the join-point is executed.

According to these concepts, it is possible to affirm that AO is an expressive style of the software development, since it provides a clear separation of concerns. Furthermore, it allows the building and organization of the code, so that all essential design decisions can be reflected in it. This clear separation of concerns is the primordial idea in application based on NFRs. For example, dependable systems can join several non-functional features, which can compromise the system functionality. So, when the dependability attribute is modeled as an aspect the intermixing of requirements is reduced.

Today, several proposals show how to apply the AO in the software development during all phases of the software development. The next section concentrate on shows the approaches related to our work.

3 Related Works

There are several proposals on how to deal with NFRs [4, 5, 6, 7, 8, 11, 15]. However, there are few proposals that address the NFRs as aspects [4, 6]. These proposals do not implement any automatic detection of NFRs, and the majority does not demonstrate how to transform the requirements into design elements. FRIDA offers automatic detection of NFRs and their further transformation into design elements. FRIDA also offers traceability, i.e. from a design element it is possible to discover the requirement and actors that generated this element in any of the previous phases.
FRIDA also addresses the following concerns: (i) how to integrate the elicited NFRs into the models used in the subsequent phases, and (ii) how it is possible to model the NFRs using “2 – dimensions” [9] (classes and aspects) of decomposition. The objective of our work is to provide a systemic way to make them.

4 An Overview of the FRIDA Model

Each kind of concern is important to one specific context. We assumption that the AO is the most appropriate way to achieve the software decomposition. The focus on our approach is to identify concerns early in the lifecycle, and advise the initial development. Many concerns span for multiple artifacts and several times for multiple phases of the lifecycle, for instance the NFRs. So, in our approach, each NFR is come across partially described in documents related to each phase of the lifecycle. FRIDA determines how the aspects will be identified and modeled during the software development process, i.e. how is possible modeling the FRs and the NFRs. Thus, our model is based on some units of decomposition (Table 1), because its objective is to guide the developer. Furthermore, it allows the most accurate and complete system specification so that it is possible to increase the quality.

Table 1. Relevant Decomposition Units in FRIDA Model

<table>
<thead>
<tr>
<th>FRIDA Phase</th>
<th>Decomposition Units</th>
</tr>
</thead>
<tbody>
<tr>
<td>Requirements identification</td>
<td>Use cases and template</td>
</tr>
<tr>
<td>NFRs Refinement</td>
<td>Checklist</td>
</tr>
<tr>
<td>Lexicon Processing</td>
<td>Lexicon – words and expressions</td>
</tr>
<tr>
<td>Conflicts Identification and Resolution</td>
<td>Conflicts matrix</td>
</tr>
<tr>
<td>Aspect Extraction</td>
<td>Aspect template</td>
</tr>
<tr>
<td>Requirements Association with Design</td>
<td>Classes</td>
</tr>
<tr>
<td>Visual Modeling of Aspects</td>
<td>Aspects</td>
</tr>
</tbody>
</table>

The general lines of this model are divided into phases, in which are described in the next sub-sections.

4.1 Requirements identification

Use cases have been recognized as a valuable source to identify and capture FRs [6, 10, 19]. Documentation of use cases can be represented using graphical or textual documents, as templates. Templates, as proposed by Coleman [10], are not related to implementation, but to goals: goals to be achieved by use case and source for requirements. The use case diagram is build, and for each use case occurrence one template will be associated and will describe the use case in a detailed way.

Figure 2 presents the template used to describe each use case. It was constructed by combining several proposals [10, 19]. This combination considers the relevant elements to AOP technique.

<table>
<thead>
<tr>
<th>Name</th>
<th>Descriptive phrase that names the use case.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Goal</td>
<td>Presents the goal of this use case.</td>
</tr>
<tr>
<td>Author</td>
<td>Person responsible to elaborate the use case.</td>
</tr>
<tr>
<td>Date</td>
<td>Use case creation date</td>
</tr>
<tr>
<td>Pre-condition</td>
<td>The state in which the system will be before the use case beginning.</td>
</tr>
<tr>
<td>Post-condition</td>
<td>The state in which the system will be after the use case has been completed.</td>
</tr>
<tr>
<td>Primary actor</td>
<td>The key actor in the use case.</td>
</tr>
<tr>
<td>Secondary actor</td>
<td>Another actor(s) that make any action.</td>
</tr>
<tr>
<td>Priority</td>
<td>Used to indicate how this use case will be delivered to customer.</td>
</tr>
<tr>
<td>Primary Pathway</td>
<td>Description of the main event flow.</td>
</tr>
<tr>
<td>Alternate</td>
<td>Summarized description of the alternate event flow.</td>
</tr>
<tr>
<td>Exceptional</td>
<td>Summarized description of the exceptional flow of this use case.</td>
</tr>
<tr>
<td>Main</td>
<td>This section describes the main activities of the scenario of this use case. Observe that the focus here is on what must be done, and not how.</td>
</tr>
<tr>
<td>Variations</td>
<td>Here the steps that modify the main sequence steps are described.</td>
</tr>
<tr>
<td>Non-Functional</td>
<td>This section is reserved for appointing the generic NFRs related to the present use case:</td>
</tr>
<tr>
<td>Performance</td>
<td>&lt;resumed description&gt;</td>
</tr>
<tr>
<td>Security</td>
<td>&lt;resumed description&gt;</td>
</tr>
<tr>
<td>Dependability</td>
<td>&lt;resumed description&gt;</td>
</tr>
</tbody>
</table>

Figure 2. FRs and NFRs Template
As observed by [7] at this moment the developer rarely knows all the involved features and solutions. The focus is on identifying only real-world entities, and the non-functional requirements should be pruned from this phase [7]. Moreover, in this template we introduce a generic way to indicate the NFRs associated with the system. For instance, reliability is a frequent requirement of distributed systems. This implies an additional modeling of faults in order to select the suitable fault-tolerance techniques. In a short description, it is not possible to enumerate these several aspects.

Note that the description of the non-functional requirements is summarized, i.e. vague and sometimes incomplete, and it needs more refinement in order to express an aspect. The next step is to identify more precisely the NFRs required by each use case. Chung et al. [8] explains that use cases and scenarios are not adequate to model NFRs, and suggest the use of documents. In our model, checklists are used to refine NFRs at early-stages of development lifecycle in order to produce documents that represent a textual description of the ideas embedded in use cases and scenarios.

4.2 NFRs Refinement

This step considers the generic NFRs, which are early available in the requirements elicitation phase (as shown in Figure 3). It operates on different levels of refinement, and tries to capture more precise information about the aspects involved. Recall that each NFR represents one or more aspects. For this purpose, checklists are provided to ensure completeness of these specific requirements.

Checklists provide a list of issues that can serve to guide the developer during the process by querying, advising and recalling previous issues about the NFR on focus [17]. For example, for each NFR being factored, the other NFRs linked to that particular use case are shown up.

Another feature associated with checklists is the NFR description, which describes a constraint in the normal or abnormal behavior of that NFR. Remember that it is necessary to consider that each description will become a requirement of the present NFR later on. Figure 3 shows part of the checklists related to the dependability context.

<table>
<thead>
<tr>
<th>R1</th>
<th>N/A2</th>
<th>P3</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dependability</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Availability</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Is there a specific percentage of time the system should be available?</td>
<td>O</td>
<td>O</td>
<td></td>
</tr>
<tr>
<td>Are there conditions in which the system will be unavailable?</td>
<td>O</td>
<td>O</td>
<td></td>
</tr>
<tr>
<td>Reliability</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Is the system fault-tolerant?</td>
<td>O</td>
<td>O</td>
<td></td>
</tr>
<tr>
<td>Does the system have exception handling or/and error recovery?</td>
<td>O</td>
<td>O</td>
<td></td>
</tr>
<tr>
<td>Is any mechanism of redundancy necessary?</td>
<td>O</td>
<td>O</td>
<td></td>
</tr>
<tr>
<td>Is quick recovery or startup capability necessary?</td>
<td>O</td>
<td>O</td>
<td></td>
</tr>
</tbody>
</table>

Figure 3. Checklists Example

Considering that a use case can encompass more than one NFR, it is advisable to assign a priority to each one for further conflict resolution. Remember that conflicts are not a concern at this phase.

We argue that the checklists approach is suitable to address the problems involved in NFRs. However, the support provided by this approach is not complete. Hence, the satisfactory way found to complement the modeling of NFRs is the use of NFRs lexicon, detailed in the next sub-section.

1 R = relevant to problem context, 2 N/A = not ascribed, 3 P = priority of the current NFR
4.3 Lexicon Processing

After describing the FRs and refining the NFRs, it is possible that some NFRs are not correctly elicited. Hence, we decided to use a lexicon, as suggested by Leite [15], and a BNF – Backus Naur Form – [16] to define it.

The lexicon is similar to a glossary. Our lexicon is concerned with keywords related to the problem domain in which these requirements are used. Actually, this lexicon is the starting point to process and analyze any use case description and other use cases related to the present one, through relationship such as <<include>> or <<uses>>, <<extends>> and generalization. Hence, we deduce that it is possible to check and validate most of the NFRs.

The basic structure of our lexicon is illustrated at Figure 4. Note that it also shows a partial BNF description for a dependability-related lexicon.

```plaintext
<NFR_Generic> ::= <performance> | <security> | <dependability>
<dependability> ::= <availability> | <reliability> | <integrity> | <confidentiality>
<availability> ::= <mod> <n <unit_t>/unit_t> unavailable | available m X n <unit_time> | n % time available
| acceptable downtime | restart time after failure
<reliability> ::= <unit reliable> <unit_rec> in <unit cap> n <unit_t> | <unit_rec> in <unit cap> in <unit_t>
<integrity> ::= <unit_unsec> <affect> | <unit.sec> <affect> <mod>
<confidentiality> ::= authorize<pos> | authentic | access control | encrypt | cipher | audit | login | valid
[unit_t] ::= ms | milliseconds | seconds | hour | day | month | year | week
[unit cap] ::= maximum | minimum | mean | largest | less than | more than
[unit_unsec] ::= denial | loss | harm | unauthorized | attack | virus | intrusion | worm | unidentified | malicious
[unit_sec] ::= protect | authorize<pos> | confidential | secret | grant | confidence | privacy | approve
<affect> ::= delete | alter | amendment
<mod> ::= method | function | class | process | data | product | system | application | information
[unit reliable] ::= error | failure | fault
[unit_rec] ::= recovery | restore | retrieve | backup
[pos] ::= e | ed | ation
```

Figure 4. BNF of Dependability Properties

The detection of NFRs starts with the present use case template and the related use cases. All words, derived words and expressions found in both of them are selected. The matching of any word defined in the lexicon with these words or expressions is selected as a candidate NFR. The developer is required to examine the candidate NFR and to decide, with the stakeholders, if it is or is not a real NFR. This issue is necessary because the decision must be a negotiation between all stakeholders. Each positive confirmation is responsible for activating the correspondent checklists. Moreover, the lexicon is not intended to be exhaustive. Actually, the lexicon describes only three areas (performance, dependability and security) in, we believe, a clear and useful way.

In our model the next step is the identification of the conflicts generated among the NFRs and the definition of an automatic way to solve them.

4.4 Conflicts Identification and Resolution

As previously stated by [7], the applications that contain NFRs can be affected by the conflicts. Moreover, according to Clarke [9] it is very important to solve conflicts in the requirements phase, since when a conflict is found in the analysis/design phases, the conflicting elements are not integrated. Clarke [9] also suggests that the conflicts shall be solved before the requirements specification is completed. Hence, in our model the identification and resolution of conflicts between requirements is as important as the elicitation of NFRs in the early stages of the software development lifecycle [5].
The main idea of the identification of conflicts is to use a knowledge base in which NFRs conflicts are stored. As some NFRs can not be precisely measured, we have decided to choose an alternative way to represent the conflicts. Following Finkelstein and Kramer [13] point of view, we choose rules on set theory as the basic mechanism to define this knowledge base. These rules define how it is possible to identify when two requirements are in conflict. Basically, the relationships between quality attributes are determined by four rules (Figure 5).

(1) alteration: \( V \to V \)
    
    alteration\((x) = x + \varepsilon \)
    
    where: \( V \) is a value set,
    
    \( x \in V \) and \( x \neq \perp \), \( \varepsilon \in V \) and \( x \neq \perp \),
    
    \(<V, +, \perp>\) is a group

(2) \( r_i \text{ alter } \Leftrightarrow \text{ alteration } (x_r) \)

    where: \( r_i \) is the requirement \( i \),
    
    \( x_r \) is the aggregate value,

    alteration is defined in (1)

(3) \( r_i \text{ impact } r_j \Leftrightarrow ((r_i \text{ alter}) \Rightarrow (r_j \text{ alter})) \)

    where: \( r_i \neq r_j \),

    alter is defined in (2)

(4) \( r_i \text{ conflicts } r_j \Leftrightarrow r_i \text{ impact } r_j \)

    where: \( r_i \neq r_j \),

    impact is defined in (3)

Figure 5. Rules on Set Theory for NFRs

Examining these rules, it is possible to determine when two attributes are conflicting if the value of all rules is true. For instance, a possible strategy for dependability is confidentiality, which causes system overhead. This attribute is clearly conflicting with performance. Actually, if the security level increases, it can have an impact on performance. However, if there is no concern about security, it is possible to maximize the system performance. The customer can decide about these trade-offs.

In conclusion, these rules can approve or refute conflicts among quality attributes by using well-specified constraints. We represent these rules by a matrix structure, as illustrated in Figure 6, also called conflicts matrix [5].

In this matrix, conflicting relationships are established. Note that only the dependability context is showed.

<table>
<thead>
<tr>
<th>Latency</th>
<th>Throughput</th>
<th>Capacity</th>
<th>Confidentiality</th>
<th>Integrity</th>
<th>Availability</th>
<th>Reliability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Latency</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Throughput</td>
<td>✔</td>
<td>✔</td>
<td></td>
<td></td>
<td></td>
<td>✔</td>
</tr>
<tr>
<td>Capacity</td>
<td>✔</td>
<td></td>
<td>✔</td>
<td></td>
<td></td>
<td>✔</td>
</tr>
<tr>
<td>Confidentiality</td>
<td>✔</td>
<td>✔</td>
<td></td>
<td>✔</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Integrity</td>
<td>✔</td>
<td>✔</td>
<td></td>
<td>✔</td>
<td></td>
<td>✔</td>
</tr>
<tr>
<td>Availability</td>
<td>✔</td>
<td></td>
<td></td>
<td>✔</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reliability</td>
<td>✔</td>
<td>✔</td>
<td></td>
<td>✔</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 6. Conflicts Matrix

The analysis of this matrix should be realized according to the following steps: (i) each row is necessarily combined with each column; (ii) applying the rules for each previous combination; and (iii) if the response of all rules is true, one conflict is identified.

Note that this matrix presents only few conflicts, but the developer can expand them. However, the new conflict addition must consider the rules (Figure 5). When a conflict is identified, we try to solve it by using priorities. A priority is assigned to each NFR. If the conflicting NFRs have the same priority, stakeholders are inquired about solutions for the conflict. Thus, the final decision on how the conflict will be solved is decided through negotiation.
4.5 Aspect Extraction

This phase deals with the identification of the crosscutting concerns considering the complete use case diagram. These crosscutting concerns can be classified into two categories:

- Global: the NFR is pertinent to the system as a whole, in other words, it is related to all use cases. For instance, the confidentiality requirements should be considered as global requirements, since they are used in all system functionalities;
- Partial: in this case the NFR is associated with some functional requirements, i.e., it belongs to some application parts.

A template which integrates the functional and non-functional aspects of the application (Figure 7), results from this phase. It follows the ideas by Brito [6] and Araújo [4].

<table>
<thead>
<tr>
<th>Name</th>
<th>Descriptive word that names the aspect</th>
</tr>
</thead>
<tbody>
<tr>
<td>Level</td>
<td>It is used to indicate the present aspect level</td>
</tr>
<tr>
<td>Description</td>
<td>The goal of the present aspect reported in this section</td>
</tr>
<tr>
<td>Priority</td>
<td>Priority is used to indicate how the aspect is important to the system under development</td>
</tr>
<tr>
<td>Use Cases list</td>
<td>This section of the template concentrates on enumerating the use cases related to this aspect</td>
</tr>
<tr>
<td>Requirements list</td>
<td>Here the requirements of the present aspect are indicated</td>
</tr>
</tbody>
</table>

![Figure 7. Aspect Template](image)

Basically, the essential features of this template are level (global/partial), priority, use cases list and requirements list. Use cases list serves to link the functional to the non-functional requirements. It constitutes a convenient way for binding, because later on it will be indispensable for relating aspects and components. The requirements list consists in enumerating the necessary requirements for the present aspect. Note that this feature is originated from the description item in the checklists.

The next step of the development of the FRIDA is to determine the design diagram. Furthermore, it establishes a link between the requirements and the design.

4.6 Requirements Association with Design

The requirements analysis phase is followed by the domain analysis (analysis, in short). The objective of the analysis phases is to refine and structure the requirements. The refinement concentrates on solving ambiguities and inconsistencies related to the requirements specification. On the other hand, structuring attempts to build the conceptual model [9]. A concept model is a set of concepts [7] in which the focus is on the emphasis of the domain concepts, and not on the software entities. Moreover, this model presents three essential elements: concept, attributes and relationships.

Following the requirements specification and analysis, it is possible to start the subsequent phase, the design [9]. In this phase, it is necessary to transform the concepts into classes. These classes are progressively enriched with details along the project. That is an essential advantage of OO. The classes encountered in the analysis are preserved for the design stage [9]. Certainly, they are enriched with behavior and new state, and still they should be complemented by auxiliary classes.

Moreover, the developer may concentrate to find behavior and to refine the relationships between classes. Furthermore, in this phase new classes may be added or redefined, since the main objective here is to provide a hierarchy of classes concerned with software solutions, such as databases, graphical interfaces, NFRs and so on. In short, the developer determines the classes that are involved in the problem solution using well-known techniques. As the classes are defined they are linked with the use case diagram elements.

Observe that several diagrams of UML – *Unified Modeling Language* – are used in the design phase, but in our work only class diagrams will be considered. Here the goal is to discover the classes and their behavior and state. The proposed requirements association is linked with the classes of the class diagram along with the use cases and/or actors. This link is similar to an html link. Actually, it establishes a reference between diagrams (and for their elements), and it binds the requirements to the aspects. Moreover, a link between diagrams is very important because it enables the traceability of any requirements, either FRs or NFRs. Then it is possible to affirm that it is bi-directional, i.e., it is feasible to find the requirements origin of the diagrams and artifacts generated in the subsequent phases and vice-versa.
Each class identified by the developer shall be bound to at least one use case. If it is not possible to establish this bind, an inconsistency exists. One class only exists if it is related to any functional requirement. For each bound class, a syntactic description is constructed, following the template shown in Figure 8.

```
<concept name = "value">
  <use cases>
    <template_id value</template_id>[<template_id value</template_id> ...]
  </use cases>
  <actors> <actor>name</actor>[<actor>name</actor> ...] </actors>
</concept name>
```

**Figure 8. Requirements and Design Link**

At this section only the FRs are bound to the use case, and thus they can have a connection represented by the corresponding UML class diagram. Note that the NFRs have not been represented visually yet. For that, we have developed the next sub-sections.

4.7 Visual Modeling of Aspects

The object-oriented analysis and design is divided into two kinds of units [9]: structural (classes, objects and attributes), and behavioral (operation, method and interface). The object-oriented paradigm does not have any structure to represent an aspect. Hence, after performing a detailed analysis of the literature [3, 20, 21, 23] we conclude that the easiest and most adequate way for aspect understanding is to represent it using UML extensions, also called stereotypes.

Aspects as well as classes are divided into state and behavior, i.e., they can have fields and methods, respectively [21, 23]. **Stereotypes** are mechanisms used to visually describe previously identified aspects (sub-section 4.5). Another feature of our model is that each requirement of the current aspect (sub-section 4.5, requirements list item) must be validated by one specific method.

In this step, it is also necessary to determine the join-points of an aspect. Remember that a join-point should be a method invocation, a constructor call, a field access, an exception handling, and so on. In FRIDA the join-point is not represented, because it will be effectively executed only when associated with a pointcut.

The **pointcuts** are established between one class and one aspect. In our model these elements are expressed as the UML Operation meta-model element. However, the behavior of a pointcut is not optional, i.e. it must be defined. The **advice** [20, 23] is an element used to specify the code that executes a pointcut. In our approach the advice element is represented as the UML Operation meta-model element, like a pointcut.

Figure 9 shows the aspect structure in a summarized way. The behavior and state are illustrated at the top of Figure 9. On the other hand, the bottom of Figure 9 depicts the pointcut and advice representations.

```
<<aspect>> aspect_name
<<aspect_fields>> aspect_field
<<aspect_methods>> aspect_method()
<<pointcut>> pointcut_name()
<<advice>> after()
<<advice>> before()
```

**Figure 9. Aspects Stereotype**

After mapping the aspects and their elements, it is necessary to establish the relationship between classes and their respective aspects. The next sub-section shows the proposed way to realize it.
4.8 Combining Components and Aspects

At this moment in the design stage, the developer acquires the diagrams that represent the aspects and the classes. However, these elements are disconnected from each other. This step shows how the composition of the components and aspects can be automatically realized.

Using the link defined in sub-section 4.6, it is possible to determine which functional requirements are bound to each class. Analyzing the template defined in section 4.5, the NFRs linked with each aspect are found. Crosscutting this information, it is feasible to identify which aspects are associated with each class. In this way, each class will be associated with one or more aspects to complete the system modeling.

Another issue observed in this step is related to the level of the aspect (determined in sub-section 4.5). When the level is global, the aspect is not associated with any classes, because it is scattered and tangled for all system units. On the other hand, when the aspect level is partial, it must be associated with all correlated classes.

Note that the process proposed in the current sub-section does not regard the composition between aspects. The next step in the development is just to identify and solve aspect-aspect compositions, and it is not within the scope of this paper.

5 Conclusions and Future Work

To offer mechanisms for the decomposition and composition of models is the responsibility of an object-oriented modeling [9]. However, there are limitations in the decomposition on this paradigm, because it shows an only dimension for decomposition – the class. Furthermore, the units of decomposition do not necessarily align with features such as persistence, synchronism, reliability, performance, security, and so on.

Moreover, the structural decomposition perceptible in a use-case driven development is not carried through the other models (analysis and design). That is, in these models the decomposition unit is a class or an interface. Therefore it is necessary to establish a link between these models (use case, analysis and design).

Our model tries to cover all software lifecycle, i.e. from requirements (considering both functional and non-functional requirements) until the design. This work shows how to discover non-functional requirements that represent software quality attributes in order to identify the separate aspects, as required by the aspect-oriented programming approach.

We propose to elicit the quality attributes that represent aspects using checklists and lexicons. Both checklists and lexicons help the developer to discover the global aspects – applied to the whole application – as well as the aspects to be applied to some parts of the application. The next step is the identification of the possible conflicts generated among the aspects and how to solve them. In FRIDA the identification and resolution of conflicts is as important as the elicitation of NFRs in the early stages of the software development lifecycle. We show the use of a conflicts matrix to automate this process whenever possible. The aspect extraction and visual modeling of aspects are very important for our model, because they allow the creation of models that can be reused in the future. Moreover, the connection between diagrams that belong to distinct phases of software development allows a high level of traceability.

At this moment, we are studying the possibility of using the sequence diagrams in our model. It is due to that some aspects require a detailed description of their execution steps. Besides, the sequence diagrams make the aspect comprehensibility and reusability easier. As the work progresses, a tool is partially developed to accomplish the software development cycle from an AOP perspective.

As a future work, we recommend a study on software architecture and the way it can be integrated to the proposed model, aiming at the definition of a complete or almost complete model. Another work is related to composition of aspects with aspects, i.e., to determine how it is possible to combine two aspects if they satisfy different FRs, or the same FR.

References


