Separation of Crosscutting Concerns from Requirements to Design: Adapting an Use Case Driven Approach

Geórgia Sousa, Sérgio Soares, Paulo Borba and Jaelson Castro
Informatics Center – Federal University of Pernambuco (UFPE)
P.O. Box 7851, CEP: 50.732-970, Recife – PE – Brazil
{gmcs, scbs, phmb, jbc}@cin.ufpe.br

Abstract

The main goal of Aspect-Oriented Software Development (AOSD) is the separation of crosscutting concerns throughout the software development process in order to improve the modularity of software system artifacts and hence its comprehensibility, maintainability and reusability. However, currently, there is not a solid process for AOSD that covers the software development from requirements to design activities. Since the aspect-oriented paradigm builds on the object-oriented paradigm, it is natural the attempt to adapt existing object-oriented software development methods, processes and techniques to be used in AOSD. In this context, this work adapts some use-case driven activities of the Unified Software Development Process in order to explicitly provide the reasoning and separation of crosscutting concerns from requirements artifacts to design artifacts. Our approach is illustrated by a case study of an Internet Banking System.

1. Introduction

The adoption of a new software development paradigm frequently progresses from techniques established in the programming, which later are incorporated in design, analysis and requirements activities. Similar to what happened with structured and object-oriented paradigms, this has been the course followed by the aspect-oriented paradigm.

At the beginning, the aspect orientation practices [1] were mainly applied at implementation activities. However, recently, the Software Engineering community has been interested in propagating them to early stages of the software life cycle. Some reasons for that are:

- obtain the benefits of the aspect orientation practices not only during the implementation, but also in the requirements, analysis and design activities;
- anticipate the reasoning about the treatment of aspects and its impact in the software development; and
- make possible the understanding of an aspect-oriented system through the requirements, analysis and design models, instead of demanding that this understanding only depends on analysis of implementation artifacts.

In this context, it has been emerged the idea of Aspect-Oriented Software Development (AOSD) to support the reasoning about aspects throughout the software development process. However, in order to do that, the software engineer should be equipped with techniques that provide means for the systematic identification, separation, representation and composition of crosscutting concerns throughout the software development.

Currently, there is not a solid process for AOSD that covers the software development from requirements to design activities. Towards this goal, this work adapts some use-case driven activities of the Unified Software Development Process (USDP) [2], explicitly providing the reasoning and separation of crosscutting concerns from requirements artifacts to design artifacts. Furthermore, since the systematic treatment of non-functional concerns is not provided in the USDP, we included in its requirements activities a method for systematically dealing with non-functional concerns: the NFR Framework [3;4;5].

The remainder of this work is organized as follows. In Section 2, we briefly present the background of our proposal. Section 3, in turn, presents our proposal, detailing the suggested adaptations and their justifications. Our approach is illustrated by a case study in Section 4. In Section 5, we review related work and finally, in Section 6, we present our conclusion and future work.

2. Background

In the following subsections, we, firstly, outline the approach on which our proposal is founded on and, later, we outline the method whose activities will be part of our proposal.

2.1 Use case driven activities in the Unified Software Development Process

In an use case driven development [2; 6], use cases not only represent system functional requirements, but also guide the development effort in producing requirements, analysis, design, implementation and test models. In the sequel, we present some use case driven activities that should be performed to produce the first three of these models in the Unified Software Development Process.

2.1.1 Requirements Activities

Figure 1 presents an outline of the main use case driven requirements activities proposed by the Unified Software Development Process.

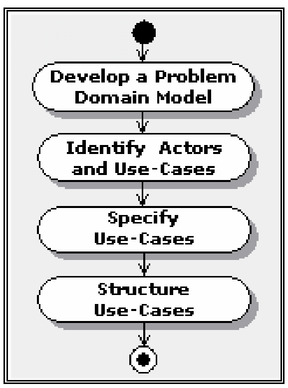

Figure 1 – Requirements activities

Firstly, we have to understand and to describe the most important business concepts and events in the problem context that the system is supposed to solve. The next activity is to identify system actors, i.e. users and other systems that will communicate with the system. The interactions that take place between an actor and the system should be identified by means of use cases. In the following activity, each use case should be specified in more detail. The last activity is to structure use cases not only to enhance its reuse and understanding, but also to prepare for the transition to the next models.

2.1.2 Analysis and Design Activities

The analysis and design activities focus on describing the internal behavior of the system that was required to realize the use cases.

The analysis model describes the system using a kind of abstraction named analysis classes. Analysis classes represent an early conceptual model for entities in the system that have responsibilities and behavior. They eventually evolve into classes and subsystems in the design model. There are three kinds of analysis classes: boundary, control and entity. Each one has its own purpose, modeling one specific role of a system component.

Figure 2 exhibits the flow of the main use case driven activities that should be performed to produce the analysis and design model.

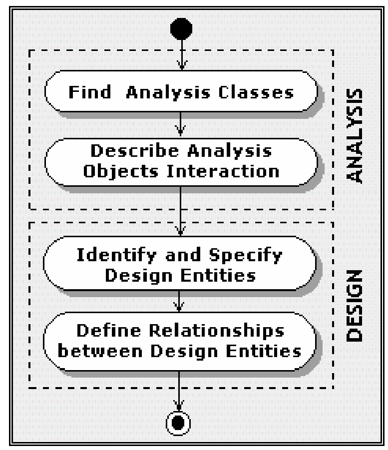

Figure 2 – Analysis and Design activities

Firstly, for each use case description, it should be identified boundary, entity and control classes and the responsibilities of each one. The following activity is to describe the use case behavior in terms of the interaction among the analysis objects. This interaction can be expressed using two types of interaction diagrams: sequence and collaboration diagrams [7]. In the sequel, design entities should be identified; depending on the type of the analysis class (boundary, entity, or control) there are specific strategies that can be used to create initial design classes (details in [2]). Attributes, operations and methods for each design class also should be specified in this activity. The last activity is establishing dependencies and associations between design classes; important inputs for this activity are the interaction diagrams and classes specifications.

2.2 NFR Framework

The NFR Framework [3;4;5] is a systematic approach to dealing with non-functional requirements (NFRs). In this approach, non-functional concerns (e.g. security, performance) are treated as goals to be achieved.

Figure 3 exhibits the sequence of activities suggested by the NFR Framework. Firstly, each non-functional concern[1] is iteratively decomposed into ones that are more specific. At some point, when the concern has been sufficiently refined, it will be possible to operationalize it, i.e. providing more concrete and precise mechanisms (e.g. operations, business rules, design decisions) to achieve it. The last step is to select among the operationalizations: accepting (✓) or rejecting (✗) each of them. During refinement and operationalization steps, contributions and possible conflicts should be established, defining the impact of the non-functional concerns to each other and identifying priorities (indicated by "!" or "!!").

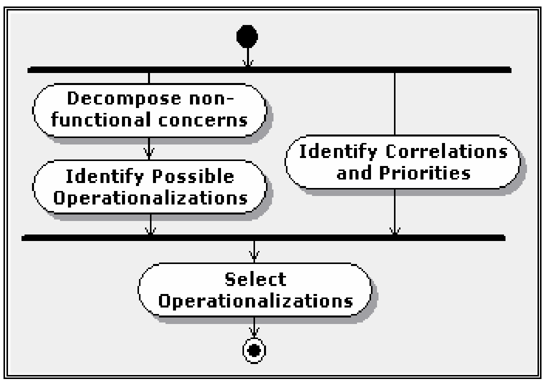

Figure 3 - NFR Framework Activities

In the NFR Framework, non-functional concerns, their interdependencies and operationalizations are graphically represented in a *Softgoal Interdependency Graph* (SIG).

3. Proposal outline

The aspect-oriented paradigm builds on the object oriented paradigm (OOP) in order to support the separation of those concerns that OOP handles poorly [8]. Then, it might be worthwhile to adapt existing object-oriented methods and techniques instead of creating a new approach for AOSD.

In this context, this paper adapts some use-case driven activities of the Unified Software Development Process [2] in requirements, analysis and design workflows. The purpose of our approach is to provide mechanisms that support the separation of crosscutting concerns in artifacts of these workflows.

In the following sections, we explain and justify each one of the adaptations that were accomplished.

[1] In the NFR Framework, non-functional concerns are named *softgoals*

3.1 Requirements activities

Figure 4 exhibits the requirements workflow emphasizing the adaptations provided by our proposal. In the following subsections, we describe each one of these adaptations and its justifications.

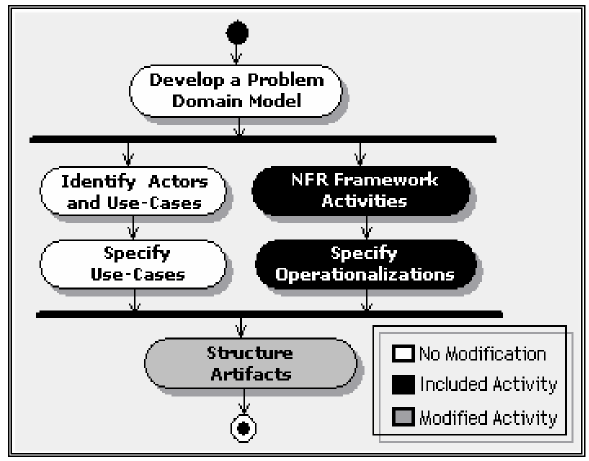

Figure 4 - Requirements Activities of Our Proposal

3.1.1 Activities focused on non-functional concerns

Since non-functional concerns are generally crosscutting, their adequate treatment is an important step in Aspect-Oriented Software Development. However, they are superficially taken into account in use case driven approaches. Then, in order to systematically deal with non-functional concerns since the early stages, we have included the NFR Framework [3;4;5] activities (presented in Figure 3) in the requirements workflow.

Throughout the NFR framework activities, each non-functional concern (e.g. security, reliability, performance) is broken down into smaller ones and then converted into *operationalizations* (i.e. operations and design decisions) that together contribute for achieving the non-functional concern. We also included an activity responsible for specifying in detail these *operationalizations*: the operations should be specified as the same way as use case are; the design decisions can be placed in special sections in the requirements document.

For each operation identified in this activity, it should be defined which use cases they applied to.

3.1.2 Activity: Structure artifacts

This activity uses the following structuring mechanisms to model shared behavior and extensions among use cases: *generalization, include-relationship and extends-relationship*.

Jacobson [9;10] advocates that the use case extension mechanism can be used to model aspects in requirements activities: an extension use case would be equivalent to an aspect and extension points would be equivalent to join points. However, we prefer to provide a new structuring mechanism to model aspects in requirements activities because:

(i) the extension mechanism can be used in situations in which the extension use case does not represent a crosscutting behaviour in the system. For example, when the extension use case represents a complex and alternative course that is specific of a base use case (e.g Figure 5); and

(ii) the way as extension points are defined hinders the reuse and comprehension of the base and the extension use case. According to UML guidelines [7], the possible extension points are specified in the base use case and there should be references to these points in the extension use case. Therefore, since there are references in the base and also in the extension use case, the composition between an extension and a base use case can not be completely considered noninvasive.

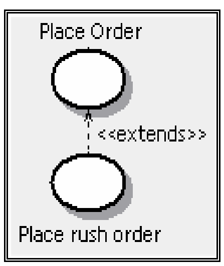

Figure 5 – Extension example (reproduced from [7])

Thus, we propose a new way to separate crosscutting behaviour in use cases: (i) the crosscutting behaviour will be placed in a use case apart, named crosscutting use case; (ii) the crosscutting use case will be connected to the use cases it affects by means of a new kind of relationship, named *crosscuts*; and (iii) information about the composition between an crosscutting use case and the use cases it affects will be described apart from both, in a composition table (see Table 1).

For each *crosscuts*-relationship, it should be specified one composition table. Besides providing better reuse and comprehensibility, the composition table simplifies determining the range of use cases that a crosscutting use case affects and how it affects each one of them.

Table 1 –Composition table for a crosscutting use case

CROSSCUTTING USE CASE: # N <NAME>			
AFFECTED USE CASE	**CONDITION (OPTIONAL)**	**COMPOSITION RULE OPERATOR**	**AFFECTED POINT**
#N<name>	condition of the extension	{overlap.after | overlap.before | override| wrap}	Step of the Scenario

The crosscutting use case should be identified at the table's top with its number and name. The first column of the composition table should list all use cases that the crosscutting use case affects. The second column describes the condition of the composition. This condition can be omitted if the composition should always be executed. The third column determines how the behavior of a crosscutting use case should be applied to the affected use case. For this purpose we use the following operators [11]: overlap (before or after); override and wrap.The last column of the composition table, in turn, specifies to which point of the affected use case the crosscutting behavior should be applied. In the requirements workflow the points are specified in terms of steps of the scenario.

We use the following heuristics to decide which relationship is more adequate to structure two use cases (use case A and use case B):

- if the execution of the use case B is an essential part to accomplish the primary purpose of the use case A (i.e. the use case A depends on the use case B to accomplish its goal), then the use case A <u>includes</u> the use case B;
- if the execution of use case B represents a complex and alternative course that is specific of the use case A, then the use case B <u>extends</u> the use case A;
- if the execution of use case B represents a course that needs to be applied in the use case A, but (i) the use case A do not depends on the execution of the use case B to accomplish its primary goal; and (ii) the use case B is not a specific course of the use case A and therefore it can be applied in others use cases; then the use case B <u>crosscuts</u> the use case A. Generally, the operationalizations of non-functional concerns have a *crosscuts*-relationship with use cases.

3.2 Analysis and Design Activities

Our proposal has made modifications in all the analysis and design activities presented in Figure 2. In the following subsections, we explain each one of these modifications and its reasons.

3.2.1 Activity: Find Analysis Classes

Finding a candidate set of analysis classes is the first activity towards a description of how the system will work. Each analysis class should represent specific behavior of an entity that collaborates to fulfill the use cases. Since, in our proposal, we consider the crosscutting behavior as a separated entity, we include a new kind of analysis class to represent it: *crosscutting class*.

For each crosscutting requirement identified in the *Structure artifacts* activity by means of the *crosscuts*-relationship, it should be created a *crosscutting class*. A crosscutting class can represent one or more analysis classes necessary to concretize a crosscutting behaviour.

3.2.2 Activity: Describe Analysis Objects Interaction

When the analysis classes have been identified and specified, it is necessary to describe for each use case how its corresponding analysis objects interact in order to realize its behavior. Generally, this interaction is represented in collaboration diagrams.

In this activity, we want to support the separation of crosscutting concerns in the interaction diagrams. Then, crosscutting objects that affect analysis objects interactions should not be placed directly in the interaction diagram. Instead, information about how a crosscutting object will affect analysis objects interactions should be described in a composition table (see Table 2).

This table provides more precise details about the composition of crosscutting concerns that the one provided by Table 1. Here we describe the composition by means of analysis objects and messages affected by a crosscutting object.

Table 2 – Composition table for a crosscutting object

CROSSCUTTING OBJECT: <NAME>			
AFFECTED USE CASE	AFFECTED ANALYSIS OBJECT	AFFECTED MESSAGE	COMPOSITION RULE OPERATOR
#N<name>	<name>	<name>	{overlap.after |overlap.before | override | wrap}

In order to fulfill this table, we start taking each composition table that was specified in the requirement model (see Table 1). Then, for each crosscutting requirement, we analyze the interaction diagrams of the use cases affected by it. So, we can specify the join points by means of messages intercepted by the crosscutting object.

It is worthwhile to mention that the composition table for a crosscutting object can generate a view of an interaction diagram aware of the crosscutting objects. This view is important for the designer/implementer of the crosscutting behavior.

3.2.3 Activity:Identify and Specify Design Elements

In general, a *crosscutting* class will generate at least one aspect in design model. However, this is not a general rule: a crosscutting class can be designed with existing mechanisms such as design patterns. The designer will choose the better solution according each particular situation.

Since aspects are characterized by adding to class(es) new behavior or new structure, we use a stereotyped class <<aspect>> to model an aspect The crosscutting behavior will be modeled as operations and new structure as properties in the aspectual class. In order to preserve the reusability and maintainability of the aspectual class, we continue using the idea of composition tables (see Table 3) to determine the join points and the composition rules.

Table 3 – Composition Table for an aspectual class

ASPECTUAL CLASS: <NAME>				
AFFECTED CLASS	CROSSCUTTING PROPERTIES			
	STATIC	DYNAMIC		
		Composition Rule Operator	Affected Point	Crosscutting Behavior
<name>	<property name>	{overlap.after |overlap.before | override| wrap}	<description>	<operation name>

By means of the composition table for crosscutting objects (Table 2) and the knowledge achieved in the previous activities, it will be possible to specify in a specific composition table (Table 3) the details about how the aspectual class will modify each affected class.

3.2.4 Activity: Define Relationships

Analyzing the composition table of an aspectual class, it is possible to determine the classes affected by it. To model the relationship between an aspect and the classes affected by it, we defined a kind of association relationship: *crosscuts.*

4. Case Study

We apply our proposal to an Internet Banking System. In the sequel, we outline how our proposal can be used throughout the requirements, analysis and design activities. In this case study, we focus only on the activities that were modified or included by our approach.

4.1 Requirements Activities

4.1.1 Activity: Specify Use Cases

At first, we have identified an actor (Bank Customer) and four use cases that this actor can accomplish (View Account Balance; View Account Statement; Transfer Funds; and Pay Bill). Simplified specifications for some of these use cases are presented in Table 4 and Table 5.

Table 4 -*View Account Statement* Use Case Specification

USE CASE # 02 - VIEW ACCOUNT STATEMENT	
MAIN SUCCESS SCENARIO	
STEP	ACTION
1	The customer informs the period
2	A list of debits and credits and the respective dates, descriptions and document numbers should be exhibited.
3	A confirming receipt is emitted

Table 5 - *Transfer Funds* Use Case Specification

USE CASE # 03 – TRANSFER FUNDS	
MAIN SUCCESS SCENARIO	
STEP	ACTION
1	The customer informs: the value of the transfer, the target account number and branch number.
2	The system validates the informed transfer data
3	The system debits the value from the customer account and credits the value in the target account
4	A confirming receipt is emitted

4.1.2 NFR Framework Activities

One of the most important non-functional concerns when building information systems to be used on the Internet is security. As we can see in Figure 6, after the successive decompositions, the following operationalizations were selected to achieve the security concern: *Limit Transaction Value, Firewall, Data Encryption, Identification, Check Internet Password, Check Customer Personal Data, Duplicate Servers, and Mirror Database.*

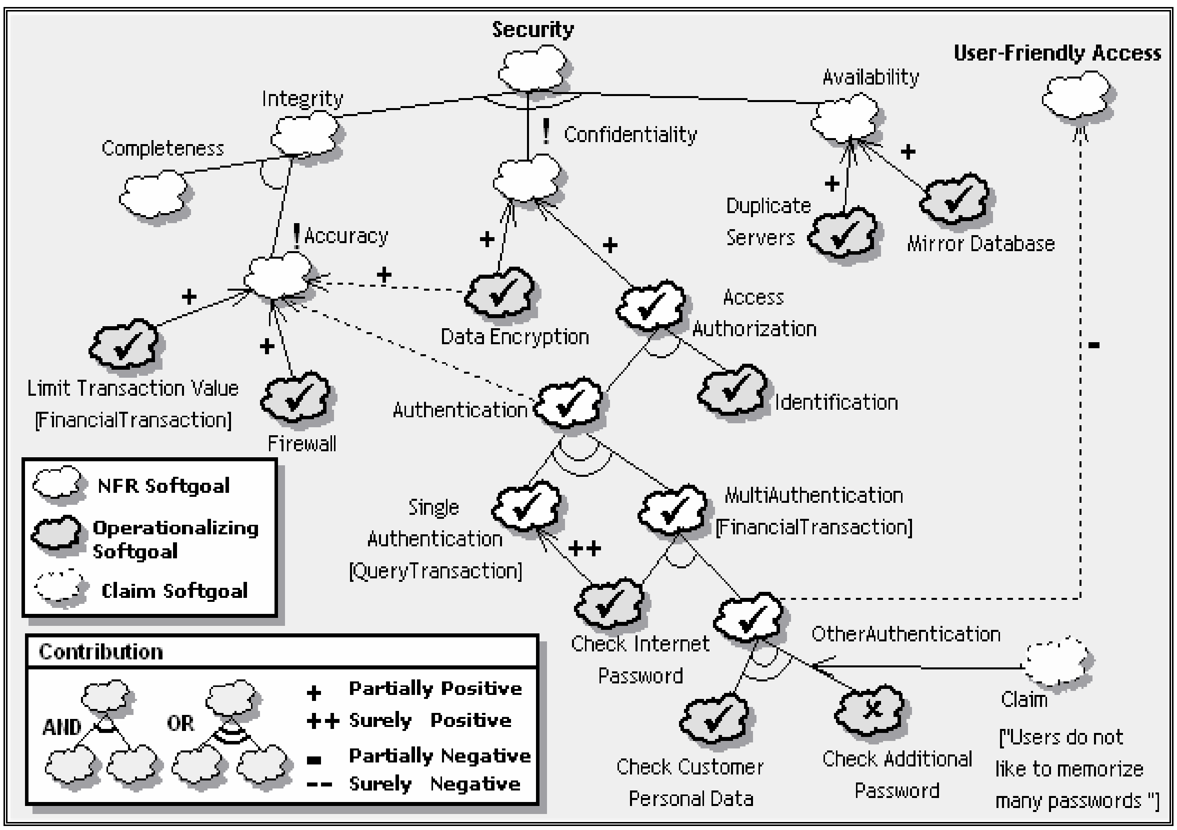

Figure 6- Softgoal Interdependency Graph for Security non-functional concern

4.1.3 Activity: Specify Operationalizations

Due to space limitation, we only show in this work the specifications for the *Check Internet Password* Operationalization (see Table 6).

Table 6 –Specification for *Check Internet Password* Operationalization

OPERATIONALIZATION # 05 – CHECK INTERNET PASSWORD	
MAIN SUCCESS SCENARIO	
STEP	ACTION
1	The actor informs the Internet Password
2	The system compares the informed Internet Password with the account's Internet Password
3	The output of the comparison is returned

4.1.4 Activity: Structure Artifacts

In this activity, we use the available mechanisms (generalizes, include, extends and *crosscuts*) to structure shared and crosscutting behavior among the requirements artifacts previously described. We use the heuristics presented in Section 3.1.2 to decide which relationship is more adequate to use.

Figure 7 shows graphically the output of this activity.

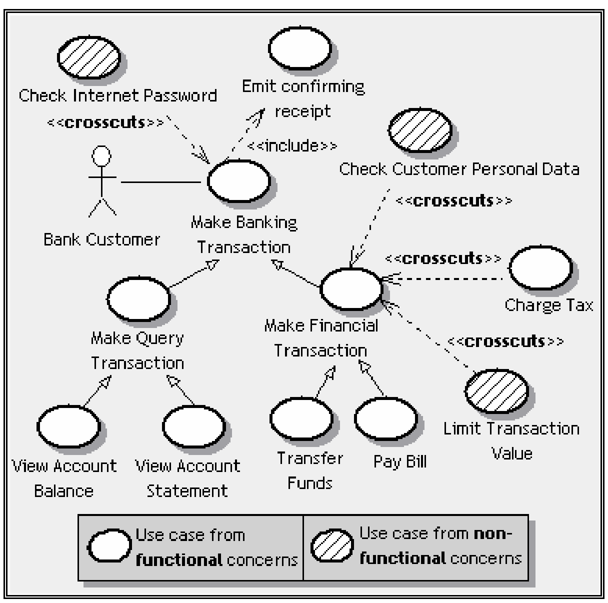

Figure 7 – Structured Use Case Diagram

First, we have analyzed the use cases. Doing this, we have identified some possible generalizations and one include-relationship between the *Emit Confirming Receipt* and the *Make Banking Transactions* use cases. Since after each financial transaction, the bank should charge a tax to the customer account, there is a *crosscuts*-relationship between the *Charge Tax* and the *Make Financial Transaction* use cases.

After that, we analyzed each operationalization and identified which use-cases each one should affect. As functional and non-functional requirements have different purposes, when an operationalization needs to be applied to a use case, we have a *crosscutting* relationship. The following crosscutting operationalizations were identified: *Check Internet Password*, *Check Customer Personal Data, Limit Transaction Value* and *Data Encryption.*

For each *crosscuts*-relationship, we should specify how to compose the crosscutting use case and the affected use cases. Table 7 and Table 8 show some of these specifications.

Table 7 – Composition for *Check Internet Password*

CROSSCUTTING REQUIREMENT: #05-CHECK INTERNET PASSWORD			
AFFECTED ARTIFACT	COND.	COMPOSITION RULE OPERATOR	AFFECTED POINT
#01 – View Account Balance	-	overlap.before	Step 1
#02 –View Account Statement	-	overlap.before	Step 2
#03 – Transfer Funds	-	overlap.after	Step 1
#04 – Pay Bill	-	overlap.after	Step 1

Table 8 – Composition for *Charge Tax*

CROSSCUTTING REQUIREMENT: #06 – CHARGE TAX			
AFFECTED ARTIFACT	COND.	COMPOSITION RULE OPERATOR	AFFECTED POINT
#03 – Transfer Funds	-	overlaps.after	Step 3
#04 – Pay Bill	-	overlaps.after	Step 3

4.2 Analysis and Design Activities

In the next subsections, we follow the analysis and design activities for the Internet Banking System including the adaptations provided by our proposal. In this section, we focus only on the *Transfer Funds* use case and in the *Check Internet Password* crosscutting use case.

4.2.1 Activity: Find Analysis Classes

In this activity we identified the following analysis classes which will be capable of performing the behavior described in the *Transfer Funds* use case:

- Boundary: Input Data User Interface (UI)
- Entity: Account
- Control: Transfer Handler; Transference Information Valuator; and Confirming Receipt Emitter.

There are also some crosscutting classes that will affect the realization of the *Transfer Funds* use case:

Check Internet password, Limit transaction value, Charge tax, Check customer personal data.

4.2.2 Activity: Describe Analysis Object Interaction

By means of an analysis of the *Transfer Funds* use case, we have obtained the collaboration diagram presented in Figure 8. This diagram should be used by the designer/implementer of this use case and therefore it is oblivious about the crosscutting objects that will affect its behavior.

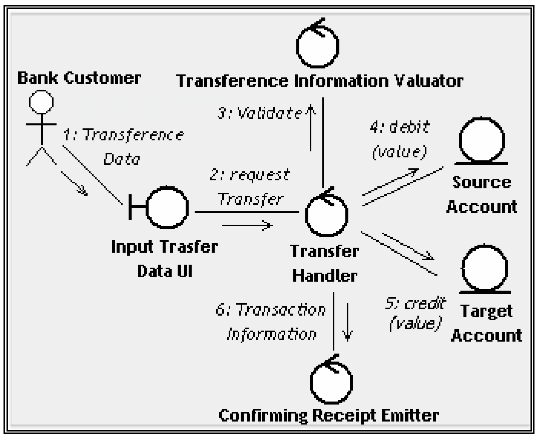

Figure 8 – A Collaboration Diagram for a realization of the *Transfer Funds* use case

Nevertheless, in the point of view of the designer/implementer of the crosscutting behavior it is interesting to visualize how crosscutting objects will affect the interaction of analysis objects. As explained in Section 3.2.2, in order to obtain this crosscutting view of the interaction diagrams, we have to determine the join points in terms of messages intercepted by the crosscutting object. Table 9 presents this specification for the *Check Internet Password* crosscutting class.

Table 9 – Composition Table for *Check Internet Password* Crosscutting Class

CROSSCUTTING CLASS: CHECK INTERNET PASSWORD			
AFFECTED ARTIFACT	AFFECTED ANALYSIS CLASS	AFFECTED MESSAGE	COMPOSITION RULE OPERATOR
#03 – Transfer Funds	Input Transfer Data UI	request transfer	overlap.before

Analyzing the composition table of each crosscutting class that affects the *Transfer Funds* use case, it can be generated a crosscutting view of Figure 8. Figure 9 presents this crosscutting view, considering only the *Check Internet Password* crosscutting class.

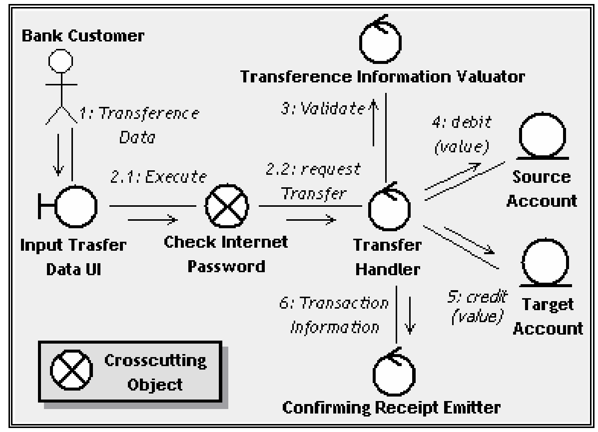

Figure 9 – Crosscutting View of the *Transfer Funds* Collaboration Diagram

4.2.3 Activity: Identify and Specify Design Entities

We decided to organize the design model in well defined layers, according to the nature of the application concerns: interface, façade, business and data. This architecture is based on the object-oriented layer architecture [12].

By means of the artifacts and the knowledge achieved in the previous activities, we identified the *InternetPasswordChecking* aspect and its properties, as exhibited in Table 10.

Table 10 – Composition Table for *Internet Password Checking* Aspectual Class

ASPECTUAL CLASS: INTERNET PASSWORD CHECKING				
AFFECTED CLASS	CROSSCUTTING PROPERTIES			
	STATIC	DYNAMIC		
		Composition Rule Operator	*Affected Point*	*Crosscutting Behavior*
Transaction Facade	-	overlap.before	doTransfer()	checkInternet Password()

4.2.4 Activity: Define Relationships

Lastly, Figure 10 presents the graphical representation of the crosscutting relationships between two of the aspects identified and the classes they affect.

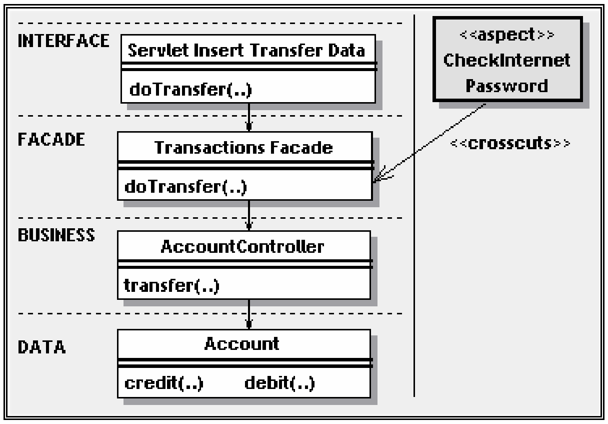

Figure 10 - Relationships between Design Entities

5. Related Work

Currently, there are few works concerned with the provision of development techniques for separation of crosscutting concerns from requirements to design. One example of this kind of work is presented by Constantinides [13]. He emphasizes the importance of identifying and modeling crosscutting concerns since the early stages of the software life cycle. Constantinides' work presents a case study to investigate the modeling of crosscutting concerns, mainly in analysis and design activities. However, although Constantinides proposes to adapt established analysis and design techniques for an aspect-oriented context, he only describes how crosscutting concerns can be visualized in sequence and classes diagrams, not suggesting techniques for developing these artifacts.

Jacobson [9;10] advocates that the use case extension mechanism has a similar purpose to aspects in AOP and that this mechanism could be used in requirements activities for AOSD. But, as we show in Section 3.1.2, there are some difficulties using this mechanism that we overcome.

Rashid et al. [14; 15] propose a generic process model for Aspect-Oriented Requirements Engineering (AORE), but do not explore the link with analysis and design activities. After that, Moreira et al. [11] and Araujo et al. [16] presents a simplified model to support the general AORE process described in [14]; these works compose non-functional and functional requirements using extensions of the use case and sequence diagrams. One of the characteristics that differs these works in AORE from our approach in the requirements activities is the treatment of non-functional concerns. They deal with non-functional concerns in a high-level of abstraction. On the other hand, we provide a systematical treatment of non-functional concerns before analyzing their crosscutting behavior: firstly refining them and later operationalizing them in more concrete and precise mechanisms. We advocate that it is more adequate to deal with operationalizations in the context of Aspect-Oriented Requirements Engineering because they better reflect how the crosscutting concerns will be treated in the latter stages [17].

In turn, one of the first proposals to extend the UML for aspect design was presented by Suzuki and Yamamoto [18]. That work extends the UML metamodel including a new kind of classifier named *aspect*. To model the aspect-class relationship, Suzuki and Yamamoto advocate the use of a kind of dependency relationship with stereotyped realization,<<realize>>, already provided by UML. However, in their work, Suzuki and Yamamoto present a notation only for inter-type declarations, not mentioning how pointcuts or advices can be modeled with the UML. Furthermore, they focus on design activities, not exploring the link with previous activities.

6. Conclusion

Scattering and tangling do not occur only in implementation artifacts. They emerge in other artifacts throughout the development process. For this reason, it is necessary to apply the separation of crosscutting concerns in all development stages. As a result, the comprehensibility, maintainability and reusability of software system artifacts are improved. Furthermore, the explicit capture of crosscutting concerns throughout all development stages can also enable developers to trace crosscutting concerns from requirements to implementation artifacts.

Nevertheless, in spite of the importance of identifying and modeling crosscutting concerns throughout the development process since the early stages, few works address this issue. This paper presents a contribution to this context by means of the adaptation of some use-case driven activities of the Unified Software Development Process [3] in requirements, analysis and design workflows.

As showed by our case study (Section 4), our proposal provides a way to separate crosscutting concerns at various levels of abstraction, from requirements to design.

This work is a first step towards a complete adaptation of the Unified Software Development Process in order to provide separation of crosscutting concerns in its workflows. Our future work will focus on improving our approach and applying it in more case studies.

7. References

[1] Kiczales, G.; Lamping, J.; Mendhekar, A.; Maeda, C.; Lopes, C.; Loingtier, J.-M. and Irwin, J. (1997) “Aspect-Oriented Programming”. In Proceedings of ECOOP ‘97, Springer-Verlag.

[2] Jacobson I., Booch G., and Rumbaugh , J. (1999) “The Unified Software Development Process”, Addison-Wesley, ISBN 0-201-57169-2.

[3] Mylopoulos, J.; Chung, L. and Nixon, B. (1992) “Representing and Using Non-Functional Requirements: A Process-Oriented Approach”. IEEE Transactions on Software Engineering, Vol. 18, No. 6, June, pp. 483-497.

[4] Chung, L; Nixon, B.; Yu, E. and Mylopoulos, J. (2000) “Non-Functional Requirements in Software Engineering”. Boston:Kluwer Academic Publishers, ISBN 0-7923-8666-3.

[5] Mylopoulos, J.; Chung, L.; Liao, S.; Wang, H. and Yu, E. (2001) “Exploring Alternatives during Requirements Analysis”. IEEE Software Jan/Feb, pp. 2-6.

[6] Jacobson, I; Christerson, M; Jonsson, P. and Overgaard, G. (1994) “Object-Oriented Software Engineering: A Use Case Driven Approach”. Addison Wesley, ISBN 0-201-54435-0.

[7] Booch, G.; Rumbaugh, J. and Jacobson, I. (1999) “The Unified Modeling Language User Guide”. Addison-Wesley.

[8] Elrad, T.; Aksit, M.; Kiczales, G.; Lieberherr, K. and Ossher, H. (2001) “Discussing Aspects of AOP”. Communications of the ACM, 44(10):33–38, October.

[9] Jacobson, I. (2003) “Use Cases and Aspects – Working Seamlessly Together”, in Journal of Object Technology, vol. 2, no. 4, July-August, pp. 7-28.

[10] Jacobson, I. (2003) “Use Cases - Yesterday, Today, and Tomorrow”. The Rational Edge, March.

[11] Moreira, A.; Araújo, J. and Brito, I. (2002) “Crosscutting Quality Attributes for Requirements Engineering”, 14th International Conference on Software Engineering and Knowledge Engineering , ACM Press, Italy, July.

[12] Ambler, S. (1998) “Building Object Applications that Work”. Cambridge University Press and Sigs Books.

[13] Constantinides, C. (2003) “A case study on making the transition from functional to fine-grained decomposition”. ECOOP 2003 Workshop on Analysis of Aspect-Oriented Software (AAOS 03), Darmstadt, July 21.

[14] Rashid, A.; Sawyer, P.; Moreira, A. and Araújo, J. (2002) "Early Aspects: a Model for Aspect-Oriented Requirements Engineering", IEEE Joint Conference on Requirements Engineering, Essen, Germany, September.

[15] Rashid, A. Moreira, A. and Araujo, J. (2003). “Modularisation and Composition of Aspectual Requirements”. AOSD 2003, ACM, pp. 11-20.

[16] Araújo, J.; Moreira, A.; Brito, I. and Rashid, A. (2002) "Aspect-Oriented Requirements with UML", Workshop: Aspect-oriented Modeling with UML, UML2002, Dresden, Germany.

[17] Sousa, G.; Silva, I. and Castro, J. (2003) “Adapting the NFR Framework to Aspect-Oriented Requirements Engineering”. XVII Brazilian Symposium on Software Engineering, Manaus, Brazil, October.

[18] Suzuki, J. and Yamamoto, Y. (1999) "Extending UML with Aspects: Aspect Support in the Design Phase", In Proceedings of the 3rd Aspect-Oriented Programming (AOP) Workshop at ECOOP'99, Springer LNCS 1743.