ACHIEVING SUPPLEMENTARY REQUIREMENTS USING ASPECT-ORIENTED DEVELOPMENT

Julie Vachon and Farida Mostefaoui
DIRO, University of Montreal
Montreal (Quebec), Canada

Keywords: Aspect-oriented development, supplementary specifications, design pattern, requirement modeling.

Abstract: The problem of attempting to work supplementary requirements (software quality attributes and constraints) only around the end of the development phase is frequent and quite risky for there is little chance the final architecture will be able to meet these quality requirements without important modifications. Supplementary specifications capture the requirements which are not defined using the use case model. They are some kind of crosscutting concerns which one would like to plug in at some later stage of the design process (e.g. after prototyping use cases). The connection with the aspect notion suggests that aspect-oriented techniques may here be called upon advantageously. This article presents an aspect-oriented methodology to support the development of supplementary specifications in UML. The use-case analysis is adapted to take care of crosscutting requirements and a pattern is proposed for the elaboration of aspect-oriented designs.

1 INTRODUCTION

Business software specification documents often contain numerous satellite requirements which complete, surround or constrain the realization of the software uses cases. These requirements may concern software qualities such as performance or reliability. They may also describe the various business rules which the application must obey. Current incremental and iterative development approaches generally focus on analysis, design and implementation of use cases i.e. main system services. It is often only by the end of the development process, indeed sometimes just a few hours before the delivery day, that developers really start worrying about those other requirements, yet recorded in a special document called Supplementary Specifications. This document intends to describe (1) quality goals, (2) design constrains and (3) various crosscutting functionalities. At this stage, it is really a matter of chance if these specifications can still be satisfied. But if the whims of fate are bad, it might become a feat to modify the system architecture and detailed design to satisfy these additional requirements unfortunately relegated to a position of secondary importance.

To overcome this problem, this paper introduces a new development approach based on the aspect-oriented paradigm and using the UML for modeling. Affording the same advantages as object-oriented disciplines, the aspect-oriented paradigm allows bringing to the highest rank all these “non use-case” concerns which crosscut the various system modules implementing main functional requirements.

By their global nature, supplementary requirements can relevantly be considered as crosscutting concerns. On one hand, they should be documented early in the analysis phase and never be neglected during development. On the other hand, since they affect many use cases, it is not a good idea to start their design first! Moreover, technical and algorithmic solutions needed to realize these requirements are often not clearly identified at the beginning of the project. However, once use cases design have mainly been taken care of, one should ideally be able to readily plug in a component implementing some given supplementary requirement without needing to modify the architecture. With the same ease, one should be able to retract or replace these components.

A software system may consist of several kinds of additional concerns: business logic, performance, data persistence, logging and debugging, authentication, security, etc. Aspect-oriented programming (AOP), as introduced in (Kiczales et al., 1997), provides explicit support for dealing with these crosscut-

Vachon J. and Mostefaoui DIRO F. (2004).
ACHIEVING SUPPLEMENTARY REQUIREMENTS USING ASPECT-ORIENTED DEVELOPMENT.
In *Proceedings of the Sixth International Conference on Enterprise Information Systems*, pages 584-587
DOI: 10.5220/0002657005840587
Copyright © SciTePress

ting concerns scattered throughout the software system and which often result in code tangling problems. Hence, our development approach is taking advantage of aspect-oriented principles and constructs to facilitate the "in-time" flexible realization of supplementary requirements, not only at the programming level, but as soon as the analysis begins.

This work-in-progress paper shows the relevance of aspect-oriented development for these "non use-case requirements". Motivations, strategies and aspect-oriented design solutions are presented. We propose a design pattern for aspect weaving and explain our methodology on a small case study.

Former Work

Various approaches have been proposed to introduce aspect-oriented concepts into UML. Some of them suggest adding aspect-oriented concepts directly in the meta-model (Suzuki and Yamamoto, 1999). Other achieve aspect-oriented modeling by introducing a new profile in UML (Aldawud et al., 2001). In (Clarke and Walker, 2001), elements are decomposed into special packages to be later weaved with others using composition relationships. Authors in (Stein et al., 2002) propose a UML based design language dedicated to the representation of AspectJ's specific constructs.

Closer to our view of modeling, Jacobson explains in (Jacobson, 2003) how aspects can be modeled as extension use cases and how the base concepts of AOP have their equivalents in use case related elements. Our goal is to go further in this direction and to show how UML can be put to work to guide and document the development of aspect-oriented systems, especially those ones having to deal with supplementary requirements. We propose a modeling approach based, on the current UML notation. Contrarily to former solutions, few extensions are here required to adapt UML to aspect-oriented analysis and design, and those introduced are very natural.

2 ASPECT-ORIENTED DEVELOPMENT OF SUPPLEMENTARY REQUIREMENTS

Use Case View

To illustrate our approach, we refer the reader to the well-known ATM (Automated Teller Machine) example. The ATM requires functionalities (logging, security, etc.) which can't be formulated as standard use cases for they are not main services but rather common sub-fonctionalities required by them. Figure 1 gives an overview of these supplementary requirements.

Table 1: Supplementary spec. for the ATM

Functionality : - Logging : Log all deposit and withdraw transactions to persistent storage on the central server. - Security : Each access to an account requires client authentication.
Reliability: - Recoverability : If there is a failure, store and forward operations in order to complete the transaction anyway and recover later on.
Performance : ...

As commonly agreed, use cases are not appropriate to capture these quality goals, design constraints or global requirements applicable to the whole system application. The Supplementary Specifications documents those requirements apart, for they would otherwise be sparsely described in uses cases.

A better idea, however, consists in turning these supplementary requirements into special use cases. As mentioned in (Malan and Bredemeyer, 2001), non-functional requirements are often "not specified in time", "compromised without attention to the trade-offs involved", and/or "specified in loose, fuzzy terms that are open to wide ranging and subjective interpretation". Transforming supplementary requirements into use cases, (thus describing the functionalities realizing them), allows bringing them back to the highest ranking (i.e. use case level) and entails developers to handle them in a more disciplined way.

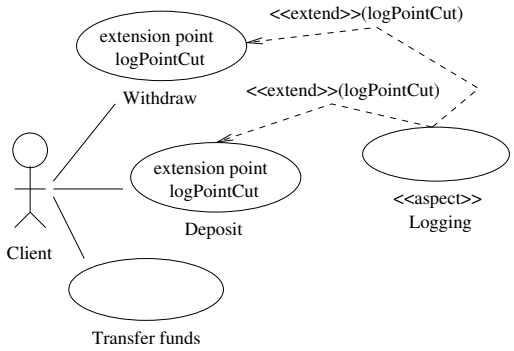

Figure 1: ATM logging expressed as a use case.

In the ATM example, the Logging supplementary requirement can be turned into a use case, as shown by Figure 1. To model the crosscutting behavior of the Logging functionality and to show its insertion into both use cases Deposit and Withdraw, the UML « `extend` » relationship is used. Hence these base use cases are implicitly modified, in a modular way, at indicated extension points, and this, without them be-

ing aware of the extension. The following correspondence therefore becomes obvious for those familiar with AOP: extended use cases can be implemented as aspects while extension points can be assimilated to pointcuts.

This way, the Logging requirement can be modeled as an independent aspect weaving into the rest of the design. A design pattern, called "Aspect-Weaving" (AW), is proposed for this modeling. It pertains to how aspects are structured and how they behave (weaving). The ATM example and its Logging requirement are modeled to illustrate aspect-oriented analysis and design based on the AW pattern.

Static View

Consider the `Logging` use case of the ATM example. Figure 2 shows how the Logging requirement can be integrated into the UML class structure of the ATM, following aspect-oriented principles.

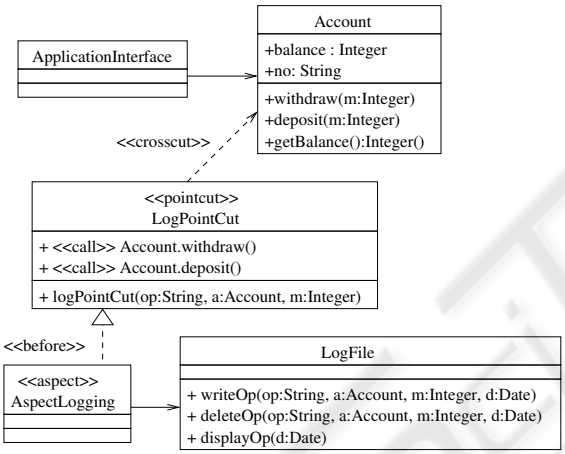

Figure 2: Class diagram for the ATM.

In this model, (1) the behavior of the `Logging` use case is captured by a class named `AspectLogging` with stereotype «`aspect` » while (2) the extension point `logPointCut` is modeled by a class interface named `LogPointCut` with stereotype «`pointcut` ». According to the use case view (Figure 1), the extension point `logPointCut` references a set of locations within the behavioral sequence of use cases `Withdraw` and `Deposit`. A location is usually related to an event (e.g. method call) in a scenario. These locations are called *joint points* and are here listed in the middle compartment of the `LogPointCut` interface. A dependency relationship, labeled « `crosscuts` », relates[1] the « `pointcut` » interface to each of the classes which may activate one of its joint points. In the ATM use case model, the `logPointCut` extension point references locations where `withdraw` and `deposit` operations are executed on accounts. To mark this dependency, the `LogPointCut` interface has a « `crosscuts` » relationship to the `Account` class implementing these operations.

The `LogPointCut` interface also contains an operation named `logPointCut` whose implementation must describe the behavior to be inserted at joint points i.e. at special locations, within the `Withdraw` and `Deposit` use cases, where operations must be logged. In our example, the `AspectLogging` class implements the `LogPointCut` interface and will thus provide the code, called an *advice*, to be executed at the given joint points. A realization relationship relates the « `aspect` » class to the « `pointcut` » interface. It has one of the following stereotypes, « `before` », « `after` », or « `around` », according to whether the advice must be executed before, after or instead of a joint point when it is reached. As specified on Figure 2, `withdraw` and `deposit` operations on `Account` objects are recorded into the `LogFile` just *before* being executed.

We generalize this modeling approach and summarize it in a design pattern called "Aspect-Weaving"(AW). Figure 3 shows the general structure of the AW pattern which has been applied to the ATM example. The next section explains how the pattern behaves.

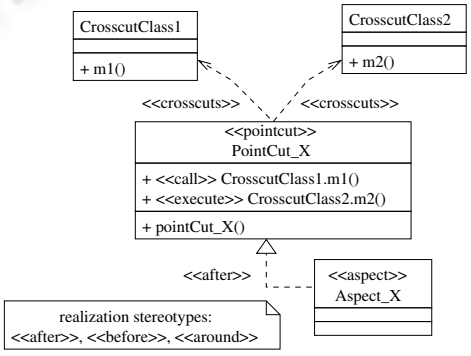

Figure 3: Structure of the Aspect-Weaving pattern.

Dynamic View

The sequence diagram of Figure 4 illustrates the weaving of the `Logging` use case into the `Withdraw` use case of the ATM example. It proposes a compiler-independent representation of this weaving into a withdraw scenario. As shown, the scenario starts when the `ApplicationInterface` object sends a withdraw request (message 1) to the `Account` object. This `withdraw` call flags a

[1]The orientation of the « `crosscuts` » dependency is significant, for it illustrates the fact that an aspect can be plugged "in or out" without modifying the specification of the crosscut classes.

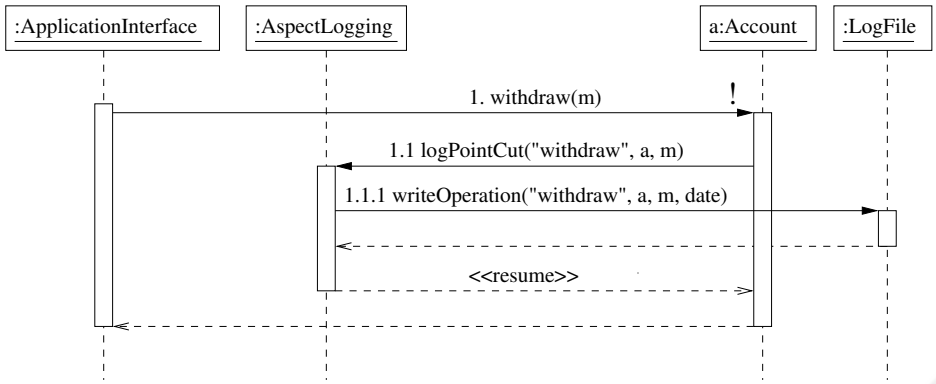

Figure 4: Sequence diagram describing a withdraw scenario of the ATM.

joint point declared in the `LogPointCut` interface (c.f. Figure 2). The joint point is represented by an exclamation mark on the sequence diagram. We therefore know that some advice call has been woven before, after and/or around the join point. As specified in the static view, the advice `logPointCut("withdraw", a, m)` implemented by `AspectLogging` is first executed (message 1.1): the withdraw operation is recorded in the `LogFile` object (message 1.1.1). The `Account` object can then resume its activities and execute its `withdraw` method. Since there is no more advice attached to the reached joint point, the procedure call gives back control to its initiator `ApplicationInterface`.

3 CONCLUSIONS

This project proposes a UML development methodology based on the aspect-oriented paradigm. It takes into account the importance of supplementary requirements by considering them as special extension use cases and by guiding their realization through aspect-oriented design.

First attempts to integrate aspects into software were made in programming languages (AspectJ, HyperJ and so on). Then, researchers took interest in ways to elaborate software designs which could easily map to aspect-oriented programs. Our belief is that aspect should be available from the very beginning, that is right from the analysis phase. Henceforth, supplementary requirements, which can often be formulated as extension use cases comparable to aspects, can be integrated into the rest of the system and their implementation can naturally be realized through aspect-oriented mechanisms. The Aspect-Weaving pattern presented in this paper is part of wider development process covering all phases from analysis to implementation.

Further work will be concerned with the modeling of a complete case study. We plan to evaluate and compare the utility and efficiency of our aspect-oriented approach regarding the development of different kinds of supplementary requirements (e.g. related to concurrency, security, etc.). The relevance of the aspect-oriented approach for software verification is also under study.

REFERENCES

Aldawud, O., Elrad, T., and Bader, A. (2001). A uml profile for aspect-oriented software development. In *OOPSLA'01 (Workshop on AOP)*.

Clarke, S. and Walker, R. (2001). Composition patterns: an approach to designing reusable aspects. In *Proc. of the 23rd Int. Conf. on Software Engineering*, pages 5–14.

Jacobson, I. (2003). Use cases and aspects – working seamlessly together. *J. of Object Technology*, 2(4):7–28.

Kiczales, G. et al. (1997). Aspect-oriented programming. In *Proc. of ECOOP'97)*, volume 1241 of *LNCS*, pages 220–242.

Malan, R. and Bredemeyer, M. (2001). Defining non-functional requirements. www.bredemeyer.com/-pdf_files/NonFunctReq.pdf.

Stein, D., Hanenberg, S., and Unland, R. (2002). A uml-based aspect-oriented design notation for aspectj. In *Proc. of the 1st Int. Conf. on AOSD*, pages 106–112.

Suzuki, J. and Yamamoto, Y. (1999). Extending uml with aspects: Aspect support in the design phase. In *ECOOP'99 (Workshop on AOP)*.