Aspectual Predesign: Extending KCPM to Map Non-Functional Requirements

Vladimir A. Shekhovtsov, Arkady V. Kostanyan

National Technical University
"Kharkiv Polytechnical Institute"
21 Frunze Str., 61002, Kharkiv, Ukraine

shekvl@kpi.kharkov.ua

Abstract: This paper presents an extension of the Klagenfurt Conceptual Predesign Model (KCPM) allowing taking into account the non-functional requirements to the system. These requirements are treated as crosscutting concerns. This process could be also seen as an addition to the AOSD paradigm that brings the gap between the Aspect-Oriented Requirement Engineering and Aspect-Oriented Modeling. The extensions of the KCPM schema and the new mapping rules are introduced for the case of asymmetric AOSD approach.

Key words: non-functional requirements, crosscutting concerns, conceptual predesign, KCPM, AOSD, mapping rules

1 Introduction

Non-functional requirements deal with issues such as performance, reliability, efficiency, usability, portability, testability, understandability and modifiability of the system. Together with functional and inverse requirements, they form an important part of the system requirements set. The problem of elicitating and engineering of these requirements is, however, quite complicated. Some papers [Chu00, Myl92] were devoted to this task, but until recently, there were no integrated approach for all the steps of the modeling process starting from such requirements. The situation was changed when the research on Aspect-Oriented Software Development (AOSD) started to appear. This approach specifically addresses crosscutting concerns of the system (that corresponds to the non-functional requirements).

Our paper presents the results of the research which goal is to allow non-functional requirements to be converted into an intermediate model residing between requirement analysis and conceptual design and to map this model into the conceptual model. The method achieving this goal will be outlined and discussed.

The remainder of this work is organized as follows. Section 2 gives brief background information. Sections 3 and 4 describe our approach. Section 5 includes the recommendations for future research. Related work is reviewed in Section 6. In Section 7, our conclusions are presented.

2 Background

In this subsection, we will briefly describe the approaches that serve as the foundation of our method. After that, we will be able to see the open issues of these approaches and formulate the statement of the problem investigated in this paper.

2.1 Aspect-Oriented Software Development (AOSD)

The starting concept for AOSD methodology is the *separation of concerns* [Dij76]. Concern represents the goal or principle that is implemented in the system component. Typical program system is the complex implementation of the several core and system concerns. Core concerns represent main goals of the system components. System concerns represent goals related to interaction of the component with other system components. For example, core concerns set for the sales management system could include product line management and order processing. The system concerns set in this case could include logging, data integrity management, and data persistence management.

Separation of concerns means that different kinds of concerns should be identified and separated for the given problem to cope with complexity of the system. This separation could be performed for different stages of the software development: requirement analysis, design and implementation. The best idea is to keep this separation through all the stages – from requirement analysis to the implementation.

Crosscutting Concerns
Sometimes we have system behavior that is presented in different (unrelated) parts of the system (for example, every account operation must be logged). The logging must be presented in all operations (in the same way), but it is not the main goal of the component and has nothing to do with account operations domain logic. Such behavior is called *crosscutting behavior*. Concerns that are related to the crosscutting behavior are called crosscutting concerns or *aspects*. Conventionally, the aspects are implemented in several unrelated system modules (e.g. in the methods of several system classes). Examples of such concerns are logging, security, persistence, load balancing, performance monitoring, caching, and thread management.

Implementation Problems
Crosscutting concerns are difficult to implement using conventional methods (including OO approach). The first problem is *code tangling*. Every operation (method) implementation contains the code implementing several unrelated concerns. As a result, the code becomes less maintainable and more complicated. The second problem is *code scattering*. It is not possible to extract the module that contains all the code for the aspect – this code is scattered among several modules of the system.

The methodology of solving such problems is called Aspect-Oriented Software Development (AOSD). The main goal of AOSD is the implementation of the tools for systematic identification, separation, presentation, and composition of crosscutting concerns. AOSD could be implemented on different stages of software development lifecycle.

Early Aspects
This is the general term for the separation of concerns on the early stages of software development (requirement analysis and design) [Ras02]. Now the research in this area is mostly at research stage.

At the requirement analysis phase, aspects correspond to the non-functional requirements to the system [Sou04]. They could be general (found in many applications) and specific to particular application only.

Some aspects could be brought to the design phase from the requirement analysis phase (this is the best way), some – identified during the design. Use cases could also be identified as aspects [Jac03]. They mostly crosscut the system components. There are several approaches of representing crosscutting behavior and aspects in UML [SHU02] (mostly using stereotypes), we will describe them in this paper in more detail.

Implementation Approaches
There are two main classes of AOSD implementation approaches: asymmetric and symmetric. Asymmetric approaches separate core and crosscutting concerns, implementing them differently. Composition rules are defined in the implementation of the crosscutting concerns. The core concern implementations are oblivious to the crosscutting concerns (this is *the obliviousness principle* [FF00]). The programmer that works on the core class knows nothing about the aspects that crosscut its code. Best-known example of asymmetric approach is Aspect-Oriented Programming (AOP).

In the symmetric approaches, all the concerns are treated equally, the composition rules are defined separately of concern implementation, and all the concern implementations are oblivious to each other. The best-known example of symmetric approach is Multidimensional Separation of Concerns (MDSOC) [Tar99]. We will not consider these approaches in our paper.

Aspect-Oriented Programming (AOP)
This approach (first presented in [Kic97]) implements clear separation between core classes and aspects. Core classes are represented by plain Java classes. Core classes and aspects are weaved together to form the final application. Here we will briefly describe five basic AOP constructs that will be used in our paper.

1. *Aspects* are modules that encapsulate crosscutting behavior. All of them are developed separately. Before running the entire application aspect composition (aspect weaving) is performed.
2. *Join points* are precisely defined places in the core classes code, e.g. method call, property access, scope of the particular class. Join points model defines the set of places where execution of base classes could be intercepted to run the crosscutting code.
3. *Pointcuts* are the rules that define the set of join points. They define the places where the aspect code crosscuts the base classes (for example, all the calls of the methods of class Account).
4. The crosscutting aspect code that runs at the join points defined by pointcut is called an *advice*. An example of the advice is the logging code that runs before all the calls of

the methods of class Account. At the weaving stage, the code of advice is injected into the code of the base classes at the places defined by pointcut.
5. The static counterpart of the advice is an *introduction*. Introductions describe members of the crosscutting aspect (its methods or fields) that are inserted into the core classes at the join points defined by pointcut. An example of introduction is the field containing the name of the log file.

The most widely known implementation of AOP is the AspectJ programming language [Asp04]. This is an extension of Java defining the set of language constructs representing crosscutting behavior.

2.2 Klagenfurt Conceptual Predesign Model (KCPM)

The traditional AOD software development process includes requirement analysis and conceptual modeling stages without any intermediate step between them. It is assumed that the domain knowledge could be easily transferred from requirement specification directly into the conceptual model. In practice, this is not always the case. Sometimes building the conceptual model directly from the requirement specification becomes difficult and error-prone task. The reason is the mismatch between the requirement specification and conceptual model. One of approaches devoted to resolving this mismatch is Klagenfurt Conceptual Predesign [MK02]. The main goal of this approach is to implement requirement gathering that could be controlled and verified by the end-user.

This goal is achieved with the help of KCPM (*Klagenfurt Conceptual Predesign Model*) – intermediate semantic model that resides between the requirement specification and the conceptual model. This model consists of semantic concepts that are more general than conceptual modeling concepts and could be more easily understood and verified by the end user. These concepts are *thing-type* (generalization of entity/class and attribute), *connection-type* (representing all kinds of relationships among the concepts), *operation* (generalization of the method), and *event*.

KCPM could be presented in tabular form (as glossaries) and as the semantic network. In this paper, we will consider only the former approach. After verification, this model could be mapped into the conceptual model. This mapping is performed according to some precisely defined rules (*mapping rules*). We will build some glossaries and mapping rules further in this paper.

The KCPM must be built from the requirement specifications (traditional approach allows only functional requirements to be used). This could be done manually by an analyst. Another approach is to extract it directly from the specifications with the help of NLP (Natural Language Processing) software. Actually, it is possible to eventually transfer the domain knowledge from the requirement specification to the conceptual model. This is the purpose of the project NIBA [Nib02] (KCPM is the part of this project).

3 The Problem Statement

Investigating the current state of affairs, we can see two open problems.

1. KCPM initially implemented the mapping for the functional requirements only [KM02]. Non-functional requirements have yet to be considered so the gap between their analysis and conceptual modeling remains open. One could hope that implementing their processing becomes an important step on the way to achieving the completeness of KCPM as a technique for gathering and mapping requirements.
2. AOSD process initially did not include the step that was similar in purpose to the conceptual predesign. As mentioned in [Sou04], "there is not a solid process for AOSD that covers the software development from requirements to design activities". One could hope that this step brings the advantages to AOSD comparable to the advantages brought to OOD by KCPM.

The purpose of this paper is to make a first step to implementing the technique that addresses both these problems.

4 Aspectual Predesign

In this paper, we will take a first look at the problem of representing and mapping non-functional requirements with the help of KCPM. To deal with these requirements, we will follow the AOSD approach, which states that they represent the crosscutting concerns (aspects) of the system (actually, in AOSD terms they are called *aspectual requirements* [RMA02]). To emphasize the nature of the requirements we are going to deal with, we call our extended predesign technique "*aspectual predesign*".

This approach could be seen from two different angles corresponding to two problems considered above:
a) as an extension to KCPM that allows mapping the aspectual (non-functional) requirements and gather information about crosscutting concerns of the system;
b) as an intermediate step of the AOSD process residing between aspect-oriented requirements engineering and aspect-oriented modeling.

Before we start to describe how the aspectual predesign extends the KCPM framework to allow dealing with non-functional requirements, several preliminary issues should be mentioned:

1. It is necessary to perform the elicitation of the non-functional requirements to obtain the natural language requirements specifications before starting the predesign. We will not deal with this issue in this paper (it is implied that these specifications have been already obtained via interviews, brain maps or other techniques).
2. It could be possible to capture the artifacts of these requirements directly from the requirements specifications via NLP methods (according to the NIBA workflow [Nib02]). This complicated problem needs investigation (see "Future research" below).

Following [MK02], we will take a closer look at two other steps of the predesign:
a) user-assisted collecting the requirements into entries of generalized KCPM schema (in the form of glossaries or semantic networks);

b) performing the mapping from the entries of this schema to the conceptual model (expressed in UML).

To be able to deal with non-functional requirements, both these steps require corrections that will be outlined below.

4.1 Extending the KCPM schema

The main problem of the aspectual predesign is related to the fact that *it is necessary to define semantic concepts that adequately represent the crosscutting concerns of the system while remaining understandable for the end-user*.

Meeting both these requirements (adequate implementation and understandability) at the same time is difficult. The main reason of this difficulty is AOSD terminology. This terminology, especially one that is established in AOP community (aspects, join points, advices, pointcuts, introductions) appears almost impossible to understand for the average user. This could be expected taking into account that:
a) AOP/AspectJ terminology was created by professional programmers whose problems are quite different from the end user problems;
b) the whole crosscutting concept is not easy to grasp even by experienced software developers and architects.

This terminological "impedance mismatch" is not limited to the end users, analysts suffer from it as well. For example, as mentioned in [SHR04], to be able to model join points via UML it was necessary to change the AspectJ-based definition of join points as "principal points in the execution of a program" [Asp04] to more general definition "hooks where enhancements may be added" (according to [Elr01]).

This paper makes only first step on the way to resolve this issue. Our approach is simplified (for example, complicated pointcut definitions were not considered). More real-life investigations involving end users are necessary in the future.

Before starting, it is necessary to have an example requirements specification. This specification will be further used in all this paper. It is very simple:

"The banking system deals with accounts and customers. All the operations with account must be logged into the file. When the customers attempt to withdraw the money from their accounts, it is necessary for them to supply a password."

Concern modeling
First, we state that the semantic concept for the concern is the thing-type. To distinguish such aspectual thing-types in a glossary, the classification column for them will contain the value "concern". The designer is responsible for supplying this value.

Fig.1 contains the fragment of thing-type glossary corresponding to our example domain. The concerns represent the characteristics of the system (security and logging).

Advice modeling
Advice roughly corresponds to the operation but is different in the way it is called (indirectly on join points it is connected with via the pointcut). In our model, we defined that:
a) the thing-type it references must map into an aspect;
b) the type column for such operation must contain the value “auto-called” (this value is preliminary, more descriptive name could be found).

UOD-area: banking				
id#	**name**	**classifi-cation**	**…**	**req. source**
D001	Security	concern		S3
D002	Logging	concern		S2
D003	Account	thing-type		S1,S2, S3
D004	Customer	thing-type		S1,S3
…				

Fig.1. Part of thing-type glossary modeling the concerns

To allow indirect calls of an advice, additional information must be supplied. This information actually belongs to the pointcut so it will be described with it.

Fig.2 contains the fragment of operation glossary corresponding to our example domain. On this figure, operations O002 and O003 correspond to the advices (“executing thing-type” columns for them refer to concerns), O001 – to the regular method.

UOD-area: banking					
o-id#	**Name**	**type**	**...**	**executing thing-type**	**…**
O001	Withdraw	manual		D004	
O002	ask for password	auto-called		D001	
O003	Log	auto-called		D002	

Fig.2. Part of operation glossary modeling the advices

Introduction modeling
There are two kinds of introductions: method introductions and field introductions. Their semantic counterparts are, respectively, operations and thing-types. The only difference between method introduction and advice operations is that for method introduction the type column will contain the value “auto-added” (also preliminary). Thing-types representing the introductory fields must be connected with the concern thing-type.

To allow indirect introductions, additional information also must be supplied via pointcut. This information is the same as was described for advices.

Join points modeling

In practice, most join points that are identifiable in the requirements fall within two categories: invoking the particular operation (in AOP, call of the particular method), and access to the particular thing-type. The latter category in AOP could have two different implementations: access to the particular field of the class and calls of all the methods of the particular class. So, *for most cases join point could be represented by either the operation or the thing-type*. Below we will see how this representation could be used.

Pointcut modeling

Pointcut is actually a connection between advice and join point so it could be represented by the connection-type. To be able to represent pointcuts, it was necessary to change the schema for the connection-type semantic concept allowing "involving thing-type" column to reference operations (advices) in addition to thing-types. To meet this goal, it was necessary to split this column into two columns: "involved type-id#" (containing the id-number of the referenced type) and "involved type" (containing either "operation" or "thing-type"). No further correction was performed.

Fig.3 contains the fragment of connection-type glossary corresponding to our example domain. Pointcut C001connects password-asking advice to the call of withdrawal operation, pointcut C002 connects the logging advice to all the operations of the Account thing-type.

UOD-area: banking									
c-id#	**name**	**…**	**Perspective**						**req. source**
			perspective#	**involved type-id#**	**involved type**	**name**			
C001	password for one operation		p001a	O002	operation	ask for password			S3
			p001b	O001	operation	withdraw			
C002	logging for all operations		p002a	O003	operation	log			S2
			p002b	D003	thing-type	Account thing-type			
…									

Fig.3. Part of connection-type glossary modeling the pointcuts

4.2 Extending the mapping process

To describe the mapping process we need to set the modeling notation to perform the mapping into. As for KCPM, this notation will be the UML. Last years, much work has been done in order to develop UML extensions (mostly using stereotypes) that allow modeling the crosscutting behavior of the system. The logical outcome of all this work would be establishing the official UML profile for AOSD. The proposition of such profile that takes into account several existing modeling notations is given in [ABE03]. In this paper we will consider mapping into more specific stereotype-based notation - Aspect-Oriented Design Model [SHU02] which contains specifications targeting AOP, and, in particular, AspectJ language.

The AODM notation includes the notion to represent aspects, advices, pointcuts, join points, and introductions. Following [MK02], let us define the rules to describe the mapping into AODM. We call them *aspectual mapping rules*. There are ten such rules; all of them will be presented in this paper.

Aspect rule (Law)
A thing-type T is mapped into an aspect A_T, if T is specified as "concern" by the designer in the classification column.

After executing this rule, all the operation referencing this thing-type (i.e. belonging to the corresponding concern) will be mapped into either advices or introduction methods according to the rules described below. All the thing-types connected with this thing-type will be mapped into introduction fields.

Advice rule (Law)
An operation O is mapped into an advice AD_O if O references the thing-type, which has been previously mapped into an aspect, and O is specified as "auto-called" in the type column.

Introduction method rule (Law)
An operation O is mapped into introduction method IM_O if O references the thing-type, which has been previously mapped into an aspect, and O is specified as "auto-added" in the type column.

Introduction field rule (Law)
A thing-type T is mapped into introduction field IF_T if T is involved in a connection-type and other involved thing-type has already been mapped into an aspect.

Pointcut rule 1 (Law)
A connection-type C is mapped into pointcut P_C if C references the operation that has already been mapped into an advice.

Pointcut rule 2 (Law)
A connection-type C is mapped into pointcut P_C if C references the operation that has already been mapped into an introduction method.

Pointcut rule 3 (Law)
A connection-type C is mapped into pointcut P_C if C it is connected by connection type to a thing-type that has already been mapped into an introduction field.

After executing any pointcut rule, other side of the connection will be mapped into the join point via one of the *join point rules*.

Join point rule 1 (Law)
An operation O is mapped into the join point J_O if O has been previously mapped into the method M_O and is involved in a connection-type, which has been previously mapped

into a pointcut. In this case, J_O represents a call to the method M_O.

Join point rule 2 (Law)
A thing-type T is mapped into the join point J_T if T has been previously mapped into the value type V_T and is involved in a connection-type which has been previously mapped into a pointcut. In this case, J_T represents an access to this value type (an attribute).

Join point rule 3 (Law)
A thing-type T is mapped into the join point J_T if T has been previously mapped into class C and is involved in a connection-type, which has been previously mapped into a pointcut. In this case, J_T represents a call to any method of C.

To map our example, it is necessary to execute the following aspectual mapping rules:
1. Aspect rule to map *Security* (D001) and *Logging* (D002) thing-types into aspects.
2. Advice rule to map *ask for password* (O002) and *log* (O003) operations into advices.
3. Pointcut rule to map C001 and C002 connection-types into pointcuts.
4. Join point rule to map *withdraw* operation (O001) and *Account* thing-type (D003) into join points.

Our simple requirement specification does not need introductions so no introduction rules were executed.

5 Related research

The most elaborated AOSD technique that covers all the stages of software development is presented in [Sou04]. This technique adapts the use-case driven approach for this purpose treating use cases as crosscutting concerns (earlier ideas by Ivar Jacobson were presented in [Jac03]). Another technique aimed at this goal is presented in [Myl92].

6 Conclusions and future work

While designing the aspectual predesign schema, our goal was to keep it as close to KCPM model as possible. As was shown, this goal was achieved quite successfully. Actually, we added *no* new semantic concepts, and the changes to existing concepts were not numerous (actually, only one structural change was necessary). This work could also be seen as a “stress test” for the flexibility and extensibility of the original KCPM approach. It is clear that the approach passed this test.

This paper attempts to establish the framework for future research. As mentioned before, all ideas presented here need extended practical validation involving real-life projects and users. Both the structure of the schema and the mapping rules are subject to change as a result of such validation. One more specific research direction is described below.

The steps of aspectual predesign that were presented in this paper could be extended to the requirement analysis phase. Actually, the whole NIBA workflow could be adjusted to take into account crosscutting concerns of the system. The ultimate goal of the whole

process is *to separate concerns based on natural language requirements specification*. It is necessary to discover the rules of expressing crosscutting in requirements descriptions, for example, “the X functionality must be implemented *for every* Y”. The possibility to efficiently capture these rules from natural language descriptions could be the target of future research.

Bibliography

[ABE03] Aldawud, O.; Bader, A.; Elrad, T.: UML Profile for Aspect-Oriented Software Development. In: AOSD Workshop on Aspect-Oriented Modeling with UML, March 2003.

[Asp04] AspectJ Team: The AspectJ Programming Guide, 2004. URL: http://aspectj.org/doc/dist/progguide/index.html

[Chu00] Chung, L; Nixon, B.; Yu, E.; Mylopoulos, J.: Non-Functional Requirements in Software Engineering. Boston: Kluwer Academic Publishers, 2000.

[Dij76] Dijkstra, E.: Discipline of Programming. Prentice-Hall, 1976.

[Elr01] Elrad, T.; Aksit, M.; Kiczales, G.; Lieberherr, K.; Ossher, H.: Discussing Aspects of Aspect-Oriented Programming. In: Communications of the ACM, Vol. 44(10), Oct. 2001, pp. 33-38.

[FF00] Filman, R.E.; Friedman, D.P.: Aspect-Oriented Programming is Quantification and Obliviousness. Proc. Workshop on Advanced Separation of Concerns, OOPSLA 2000.

[FGR03] France, R.; Georg, G.; Ray, I.: Supporting Multi-Dimensional Separation of Design Concerns. In: AOSD Workshop on Aspect-Oriented Modeling with UML, March 2003.

[FKM03] Fliedl, G.; Kop, Ch.; Mayr, H.C.: From Scenarios to KCPM Dynamic Schemas: Aspects of Automatic Mapping. In: (Düsterhöft, A.; Thalheim, B. Eds.): Natural Language Processing and Information Systems - NLDB'2003. Lecture Notes in Informatics P-29, GI-Edition, 2003, pp. 91-105.

[Jac03] Jacobson, I.: Use Cases and Aspects – Working Seamlessly Together. Journal of Object Technology. Vol. 2, No. 4, 2003. pp. 7-28.

[Kic97] Kiczales, G.; Lamping, J.; Mendhekar, A. et al: Aspect-Oriented Programming. In: Proceedings of the European Conf. on Object-Oriented Programming (ECOOP), 1997.

[KM02] Kop, Ch.; Mayr, H.C.: Mapping Functional Requirements: From Natural Language to Conceptual Schemata. In: Proc. International Conference SEA 2002, Cambridge, USA, Nov. 4-6, 2002, pp. 82-87.

[MK02] Mayr, H.C.; Kop, Ch.: A User Centered Approach to Requirements Modeling. In: M.Glinz, G. Müller-Luschnat (eds.): Proc. Modellierung 2002. Lecture Notes in Informatics P-12 (LNI), GI-Edition, 2002, pp.75-86.

[Myl92] Mylopoulos, J.; Chung, L.; Nixon, B.: Representing and Using Non-Functional Requirements: A Process-Oriented Approach. In: IEEE Transactions on Software Engineering, Vol. 18, No. 6, June, pp. 483-497.

[Nib02] Niba, L.C.: The NIBA workflow: From textual requirements specifications to UML-schemata. In: Proc. ICSSEA'2002, Paris, December 2002.

[RMA02] Rashid, A.; Moreira, A.; Araujo, J.: Modularization and composition of aspectual requirements. In: Proc. AOSD '02 (Enschede, The Netherlands, Apr. 2002), pp. 11-20.

[Ras02] Rashid, A.; Sawyer P.; Moreira A.; Araujo J.: Early Aspects: A Model for Aspect-Oriented Requirements Engineering. Proc. IEEE Joint Intl. Conf. on Requirements Engineering. IEEE CS Press. 2002. pp 199-202.

[SHU02] Stein, D.; Hanenberg, S.; Unland, R.: An UML-based Aspect-Oriented Design Notation for AspectJ. In: Proceedings of AOSD 2002 International Conference, pp. 106-112.

[SHU04] Stein, D.; Hanenberg, S.; Unland, R.: Modeling Pointcuts. In: Early Aspects 2004: Workshop at International Conference on Aspect-Oriented Software Development (AOSD 2004), Lancaster, 2004. pp. 107-113.

[Sou04] Sousa, G.; Soares, S.; Borba P.; Castro J.: Separation of Crosscutting Concerns from Requirements to Design: Adapting an Use Case Driven Approach. In: Early Aspects 2004: Workshop at AOSD 2004 International Conference, Lancaster, 2004. pp. 97-106.

[Tar99] Tarr, P.; Ossher, H.; Harrison, W.; Sutton, S.M.: N Degrees of Separation: Multi-Dimensional Separation of Concerns. Proc. ICSE'99 Intl. Conference, May, 1999.