Early Aspects: Aspect-Oriented Requirements Engineering and Architecture Design

---

Workshop Proceedings

21 March, 2004

Lancaster, UK

http://trese.cs.utwente.nl/workshops/early-aspects-2004/

Bedir Tekinerdoğan, Ana Moreira, João Araújo, Paul Clements

(Eds.)

http://early-aspects.net/

Workshop Organisers

- Bedir Tekinerdoğan, University of Twente, The Netherlands (Contact)
- Ana Moreira, Universidade Nova de Lisboa, Portugal
- João Araújo, Universidade Nova de Lisboa, Portugal
- Paul Clements, Software Engineering Institute, Carnegie Mellon University, USA

Program Committee

- Mehmet Aksit, University of Twente, The Netherlands
- Krzysztof Czarnecki, University of Waterloo, Canada
- Charles Haley, Open University, UK
- Kim Mens,Universite catholique de Louvain, Belgium
- Awais Rashid, Lancaster University, UK
- Stan Sutton, NFA Consulting, USA

Technically Sponsored by:

In conjunction with 3rd International Conference on Aspect-Oriented Software Development
March 22-26, 2004,
Lancaster, UK

BCS SPECIALIST GROUP

BCS

The Requirements Engineering Specialist Group
of the British Computer Society (BCS RESG)

TABLE OF CONTENTS

Presented Papers

Other Papers

Early Aspects: Aspect-Oriented Requirements Engineering and Architecture Design

Workshop Report,

Lancaster, 21 March, 2004

Bedir Tekinerdoğan
University of Twente
Dept. of Computer Science,
Software Engineering,
TRESE, The Netherlands
bedir@cs.utwente.nl

Ana Moreira
Universidade Nova de Lisboa, Dept. Informática, Faculdade de Ciências e Tecnologia, Portugal,
amm@di.fct.unl.pt

João Araújo
Universidade Nova de Lisboa, Dept. Informática, Faculdade de Ciências e Tecnologia, Portugal
ja@di.fct.unl.pt

Paul Clements
Software Engineering Institute, Carnegie Mellon University, USA
clements@sei.cmu.edu

Abstract

This paper reports on the third Early Aspects: Aspect-Oriented Requirements Engineering and Architecture Design Workshop, which has been held in Lancaster, UK, on March 21, 2004. The workshop included a presentation session and working sessions in which the particular topics on early aspects were discussed. The primary goal of the workshop was to focus on challenges to defining methodical software development processes for aspects from early on in the software life cycle and explore the potential of proposed methods and techniques to scale up to industrial applications.

1. Introduction

Conventional aspect-oriented software development (AOSD) approaches have mainly focused on identifying the aspects at the programming level and less attention has been taken on the impact of crosscutting concerns at the early phases of the software development. Current requirements engineering and architecture design approaches, on the other hand, have not explicitly addressed the crosscutting nature of some requirements. The combination of these two issues –the importance of crosscutting concerns at programming level and the impact in the whole system of the decisions made during the early development phases – led to the creation of the Early Aspects research topic in 2002 (www.early-aspect.net). Early aspects are defined as concerns in the early life cycle phases which cannot be localized and tend to be scattered over multiple early phase modules.

Obviously, the early software development phases, including requirements analysis, domain analysis and architecture design, actually set the early design decisions and as such impact the whole system. Therefore, if early aspects are not handled properly, they will, similarly to aspects at programming level, lead to serious maintenance and evolution problems.

This paper reports on the results of the workshop on *Early Aspects: Aspect-Oriented Requirements Engineering and Architecture Design* which was held on March 21, 2004 in Lancaster, UK. This workshop aimed at supporting the cross-fertilization of ideas in requirements engineering, software architecture design and aspect-oriented software development. It continued the work started at the first and second editions of this workshop held in conjunction with AOSD'2002 and AOSD'2003, respectively.

The outline of the paper is as follows. Section 2 sets Early Aspect within the context of AOSD. Section 3 provides an overview of the topics covered by the workshop. Section 4 presents the

workshop papers. Section 5 shows the workshop program. Section 6 talks about the workshop discussions and results. Section 7 lists the workshop participants. Finally Section 8 presents the conclusions of the workshop.

2. Early Aspects in the context of AOSD

Early Aspects focus on aspects at a higher abstraction level than programming or even design. To make the explicit distinction we categorize aspects as *early aspects*, and *intermediate aspects*, and *late aspects* (see Figure 1).

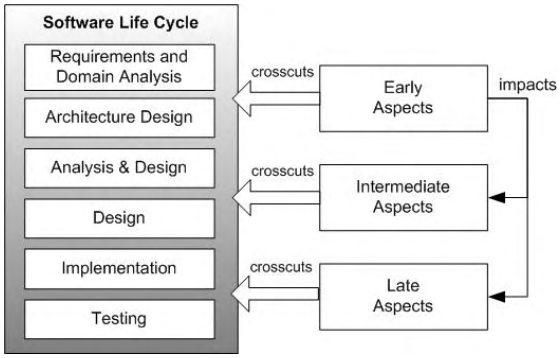


Figure 1. *Relation between Early Aspects, Intermediate Aspects and Late Aspects*

The impact of early aspects on the software development life cycle manifests itself in two ways. First of all, early aspects are crosscutting concerns that are identified in the early phases of the software development life cycle, including requirements analysis, domain analysis and architecture design. Secondly, early aspects also impact aspects in the subsequent phases. Many early aspects identified in the early phases will ripple through the other phases as well. Other early aspects might be specific to the early phases, and crosscut only the specific modules at the early phases. Likewise, it may well be that new aspects will appear as we progress in the software life cycle.

3. Topics

Topics of interest for the workshop included in particular: aspect-oriented requirements engineering, aspect-oriented domain engineering, mapping between aspect-oriented requirements, domain analysis and architecture, aspect-oriented architecture design, tool support and automation for aspect-orientation. A set of open questions for each of these topics, is listed below.

Aspect-oriented requirements engineering

- How to identify aspects at the requirements level?
- How to model aspects at the requirements level?
- How to integrate and compose aspects with other modelling mechanisms, such as goals, viewpoints and use cases, and establish trade-offs?
- How to trace requirements level aspects through later development stages and during re-engineering?
- How to validate aspects identified at the requirements level?

Aspect-Oriented domain engineering

- What are the criteria for domain aspect decomposition?
- How can we derive aspects from domain knowledge?
- How can we abstract and generalize domain aspects for reuse?
- What are the composition relations between domain aspects?
- How to represent domain aspects?

Mapping between aspect-oriented requirements, domain analysis and architecture

- Should the mapping be formal or informal?
- To what is a requirements concern mapped onto?
- What are the language' features required to support a mapping?
- What is the benefit ratio of mapping/coding? What are the pros and cons of mapping in the first place?

Aspect-oriented architecture design

- How to support evolution in the architecture using aspects?
- How to reason about architectures and aspects to know that the architecture is a good one (trade-offs between aspects)?
- How to model the architecture to take aspects into account?
- When designing an architecture, how and when to identify aspects?
- How to set the scope for a software product line architecture using aspects

- Aspects in the Model-Driven Architecture approach

Tool support and automation for aspect-orientation

- Which tools are there to support aspect-oriented development?
- Formalisms and notations for specifying aspects
- What formalisms can be used at early software development stages?

4. Workshop Papers

The workshop received 17 submissions. Table 1 contains a list of the 14 papers accepted for the workshop.

Table 1. List of accepted papers

Title of the paper	Authors
Finding Aspects in Requirements with Theme/Doc	E. Baniassad, S. Clarke
Identifying Aspects using Architectural Reasoning	L. Bass, M. Klein, L. Northrop
Facets of Concerns	C. Bogdan
Integrating the NFR framework in a RE model	I. Brito, A. Moreira
Tracing Aspects in Goal driven Requirements of Process Control Systems	I. El-Maddah, T. Maibaum
Aspect-Orientation from Design to Code	I. Groher, T. Baumgarth
Problems, Subproblems and Concerns	M. Jackson
Aspect-Oriented Context Modeling for Embedded Systems	T. Kishi, N. Noda
Generating Aspect-Oriented Architectures	U. Kulesza, A. Garcia, C. Lucena
Concerned about Separation	H. Mili, A. Elkharraz, H. Mcheick
Refining Feature Driven Development - A Methodology for Early Aspects	J. Pang, L. Blair
On imperfection in information as an "early" crosscutting concern and its mapping to aspect-oriented design	M. Sicilia, E. Garcia
Separation of Crosscutting Concerns from Requirements to Design: Adapting the Use Case Driven Approach	G. Sousa, S. Soares, P. Borba, J. Castro
Modeling Pointcuts	D. Stein, S. Hanenberg, R. Unland

We value the interaction between the participants and the results of the working groups. For this reason used the morning session for a limited number of short presentations and the afternoon was reserved for discussions and overall conclusions. Table 3 contains the 6 papers we have chosen for presentation.

5. Program

The program of the workshop is illustrated in Table 3. The program consisted of two sessions:

1. **Presentation Session**, in which selected papers were presented.
2. **Discussion Session**, in which selected topics on early aspects were discussed.

Table 3. Program of the workshop

8:45-9:00	**Introduction to workshop**
09:00-10:30	**Presentation Session** 1. Finding Aspects in Requirements with Theme/Doc, E. Baniassad, S. Clarke 2. Integrating the NFR framework in a RE model, I. Brito, A. Moreira 3. Tracing aspects in goal driven requirements of process control systems I. El-Maddah, T. Maibaum 4. Generating Aspect-Oriented Agent Architectures, U. Kulezsa, A. Garcia, C. Lucena 5. Identifying Aspects Using Architectural Reasoning, L. Bass, M. Klein, L. Northrop 6. Problems, Subproblems and Concerns, M. Jackson
10:30-11:00	**Morning break**
11:00-14:30	**Discussion Session I** – *Key Problems and Motivations*
12:30-14:00	***Lunch***
14:00-14:30	**Plenary Session:** *Presenting Fundamental Problems + Plenary discussions*
14:00-15:30	**Discussion Session II** - Setting the Research Agenda
15:30-16:00	*Afternoon break*
16:00-17:00	**Discussion Session II Cnt'd** - Setting the Research Agenda
17:00-17:30	***Plenary Session:*** *Presenting the Research Agenda for next years*

5.1 Presentation Sessions

The presentation session consisted of six paper presentations, each of which was presented in 15 minutes. The presented papers and the short

description of these, taken from the original papers, is as follows:

Problems, Subproblems and Concerns
M. Jackson
This position paper sketches how problems may be understood from a perspective based on problem frames. Problem analysis from this perspective reveals structural issues in a clearer light. It leads to a need for composition, both in the problem world and in the solution world. The goals of aspect technology would be clarified by such analysis, and the aspect technology may in turn offer some power in understanding and implementing the compositions.

Finding Aspects in Requirements with Theme/Doc
E. Baniassad, S. Clarke
To identify aspects early in the software lifecycle developers need support for aspect identification and analysis in requirements documentation. To address this, we have devised the Theme/Doc approach for viewing the relationships between behaviours in a requirements document to identify and isolate aspects in the requirements. This paper describes the approach, and illustrates it with a case study and analysis.

Integrating the NFR Framework in a RE Model
I. Brito, A. Moreira
This paper presents a model to handle advanced separation of concerns during requirements engineering. It builds on our work already produced on advanced separation of concerns for requirements engineering by adding two main ideas: (i) the integration of catalogues to help identifying and specifying concerns and (ii) improve the composition rules by informally defining some new operators.

Tracing Aspects in Goal Driven Requirements of Process Control Systems
El-Maddah, T. Maibaum
Goal driven requirements analysis methods are well suited to trace the different aspects of software applications to the early design level. This paper illustrates how to use the GOPCSD tool in developing aspect-based process control applications. The GOPCSD tool adopts goal driven requirements analysis concepts and has been adapted from the KAOS method to address process control systems.

Generating Aspect-Oriented Agent Architectures
U. Kulezsa, A. Garcia, C. Lucena
In this paper we define a domain specific language (DSL) that permits us to model orthogonal and crosscutting agent features. The agent features are then expressed in an aspect-oriented architecture. The implementation of the generative approach encompasses: (i) XML technologies to specify the DSL; (ii) Java and AspectJ programming languages to implement a concrete version of our aspect-oriented agent architecture; and (iii) a code generator, implemented as an Eclipse plugin.

Identifying Aspects Using Architectural Reasoning
L. Bass, M. Klein, L. Northrop
Architectural aspects are candidate aspects to be carried through detailed design and implementation. We set the stage by introducing some new terminology. We begin with a small set of quality requirements for an example system, present a software architecture that satisfies those requirements, and highlight the architectural tactics at work in that architecture. We then identify architectural aspects and their constituent architectural advice, pointcuts, and join points.

5.2 Discussion Sessions

We have deliberatively adopted a very short presentation session to provide more opportunity for discussion.

For the discussion sessions the participants were separated in four sub-groups.

- *Requirements Engineering*, which focused on aspects in requirements engineering
- *Domain Engineering/Application Domain*, focusing on aspects in domain engineering
- *Software Architecture Design*, focusing on aspects in architecture design
- *Specification of Early Aspects*, focusing on defining appropriate notations for early aspects.

Note that these sub-groups were not totally independent and there is some overlap. We did not consider this as a problem, nor did we experienced it later on as a problem. Each sub-group provided some interesting results that we will describe in Section 6.

The discussion session was split in two sub-sessions. The first section, *Key Problems and Motivations* focused on defining the context of the selected topics and the identification of the important problems. The session, *Setting the Research Agenda*, focused on defining the important research problems derived from the first problem.

The two sub-sessions were separated by an intermediate plenary session. The reason for this was to provide a chance to pollinate the ideas and to get feedback from the participants of the other groups.

The tasks for the first session *Key Problems and Motivations* were the following:

- Define a **Mind Map**, providing a conceptual representation of the selected topic based on the input from the participants in the group.
- Identify the Fundamental Problems from the Mind Map.

A mindmap is based on the concept of visual thinking of cognitive science. It is a way of organizing and sharing knowledge. Mindmaps are often developed by a brainstorm session and aim to reflect the common ideas on the domain.

A mindmapping activity starts with writing the main topic into the center of the map, and later on main ideas are linked to the main topic. Each main idea on its turn can have branches describing the detail ideas.

Within the first session each sub-group defined a mindmap of the selected topics. The result of the mindmaps are shown in Figure 2, Figure 5, and Figure 6.

In the second session, *Setting the Research Agenda*, the mindmaps were used to identify the most fundamental problems. The problems were described in the following format:

- **Problem Description,** describing the problem shortly
- **Motivation**, motivating the problem
- **Example**, describing an example

Given the mindmaps and the problem descriptions we aimed at providing a concrete research agenda. The following sections will elaborate on the results of the subgroups.

6. Results of Discussion Sessions

In the following we will describe the results of the workshop for each sub group.

6.1 Requirements Engineering

The mindmap for the topic Requirements Engineering is shown in Figure 52. There are many issues to be discussed and investigated: from identification to validation of aspectual requirements, from composition to traceability of aspectual requirements.

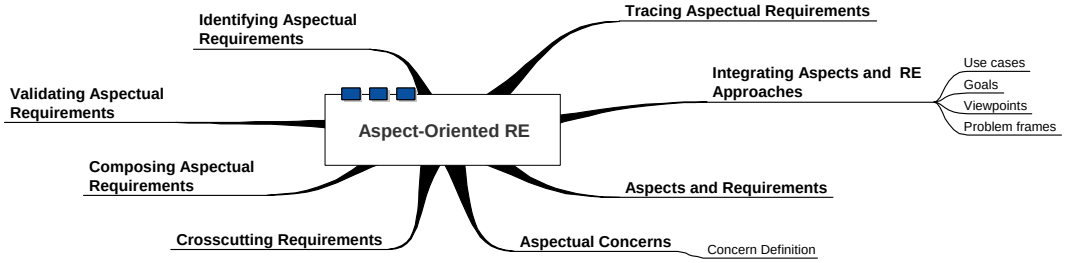


Figure 2. Mindmap for Aspect-Oriented Requirements Engineering

From the list of topics identified in our mindmap the majority of group members decided that we should first concentrate the discussions on understanding the basic concepts behind aspect-oriented requirements engineering: concerns, crosscutting, and aspectual requirements.

Unfortunately, the discussion could not progress to the other topics of the mindmap, because of lack of time.

6.1.1 Problems identified

The following set of questions was intensively discussed, and some answers were proposed.

What is the definition of a concern?

- Is a concern a requirement?
- Is a concern what a stakeholder thinks is important?
- Is a concern anything the developer must consider for the system to be successful?
- Is a concern a user expectation?
- Is a concern anything a developer must consider for the system to be successful?
- Is a concern ANYTHING? (!).

Not having an agreed definition provokes in general communication problems. However, such a definition cannot be rigid, i.e., it is important to have an agreed and flexible one. Moreover, to have a definition is helpful to identify and relate concerns. Our agreed possible definitions are listed below:

- Concern is a desired observable property;
- Concern is a feature;
- Concern is some responsibility;
- Concern is a sub-problem.

What is crosscutting?

- Crosscutting concerns are domain specific concepts that do not fit into an object abstraction.
- A concern is crosscutting when it, or part of it, contributes to multiple concerns.
- Crosscutting arises when two abstractions relate.

What is an aspect?

- It is an artifact to address a requirement.
- It is a situation where N concerns interact.
- It is a crosscutting concern.
- It is a requirement that comes from different points of view.

Why is the lack of a common terminology a problem?

- Because we need to know what we are talking about;
- We need to know how to prepare for design;
- Different versions of “aspect” may be useful in different domains.

6.1.2 Research agenda for the topics discussed

The following points were identified as interesting research topics for the near future:

- Decomposition of concerns;
- Traceability and completeness;
- Appropriate concern representations;
- How to represent specific kinds of subject matter: is there a specialization needed?
- Composition of crosscutting concerns;
- Resolution of conflicts that emerge during composition.

6.2 Domain Engineering and Aspects

(Contribution from the subgroup consisting of: Hafedh Mili, Robert Laney, Bashar Nuseibah, Jianxiong Pang, and Peter Sawyer)

Aspect-oriented software development (AOSD) methods propose new software modularization boundaries that provide additional opportunities for reuse and easier maintenance. Domain engineering is concerned with the development of software artifacts that are reusable across applications within the same application domain. Our group looked at the potential synergies between domain engineering and AOSD. We argue that AOSD is an enabling technology for domain-engineering and propose a number of research directions for the field.

6.2.1 AOSD: an enabling technology for domain engineering

Domain engineering may be defined as the process of developing software artifacts that are reusable across an application *domain.* Domain engineering differs from application engineering in terms of

intent, *process*, and *product*. The **intent** is to develop reusable components. The **product** of domain engineering is a set of development artifacts that are (should be) reusable by design. Those artifacts may not be concrete enough to be used within an application as is; they often require more or less well-defined tailoring mechanisms to make them usable for a specific application. A recurring challenge in domain engineering is to develop general components that are as concrete as the technology of the day permits [Mili et al., 2001].

Figure 3 shows the trade-off between concreteness (usability) and generality (usefulness). New software development techniques enable us to trade less and less applicability for a given "concreteness", and vice-versa. Indeed, reusable components tend to consist of a fixed part, that can be reused as-is, and a variable part, which is often to be supplied by the component user. New modularisation and abstraction techniques typically help us reduce the size of the variable part. For example, by separating interface from implementation, objects enable us to write client code that is not dependent on service implementation.

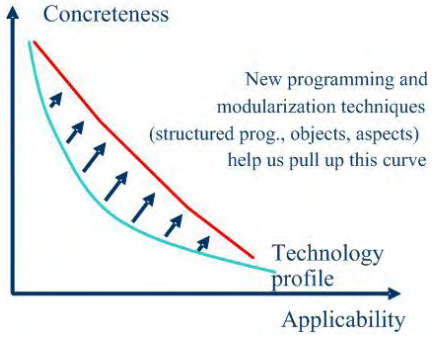


Figure 3. *Trade-off between generality (usefulness) and concreteness (usability)*

Consider the case of two modules M1 and M2 where M1 implements "Grading and logging" and M2 implements "Grading and tracing". Because logging and tracing may crosscut a number classes, those classes will be reusable. If, on the other hand, we are able to separate M1 into two artifacts, one for "Grading" and one for "logging", and M2 into two artifacts, one for "Grading" and one for "tracing", we realize that M1 and M2 will share the "Grading" artifact. Figure 4 illustrates this point. We thus argue that AOSD is an enabling technology for domain engineering.

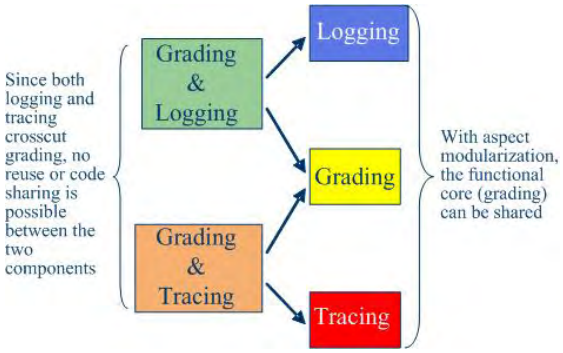


Figure 4. *AOSD supports greater reuse thanks to finer-grained variabilities*

6.2.2 AOSD for domain engineering: a research agenda

Having identified AOSD as an enabling technology for domain engineering (DE), DE will naturally benefit from general advances in aspect-oriented theory and tools and techniques. However, we see more interesting synergies. One promising area of research appeared to us to deal with what we might call *domain engineering of aspects*, described below.

A number of participants have recognized that while user requirements are good sources of concerns, a lot of concerns that pertain to a software system are often implicit. Such is the case with most non-functional requirements, including security, logging, error handling, and the like. Such concerns are business domain independent, and it pays to identify them and to build, if possible, the corresponding artifacts. Domain engineers can reuse such concerns and the corresponding artifacts in their domain models and components, and focus instead on the *functional* concerns of the domain. We discuss the relevant issues in some detail.

6.2.3 Identify common concerns

The goal here is twofold:

1) identify or *catalogue* of common non-functional concerns, and
2) identify the common interactions between such concerns, where applicable.

Neither task is trivial. When building the catalog of concerns, we should strive for completeness, but more importantly, for clear boundaries between the various concerns. For example, traceability and

logging are related. A financial decision support system (e.g. mortage application handler) needs traceability for legal reasons to be able to justify the system's decision. Traceability can be achieved in part through logging, but logging may provide useless output (of non decision-making functions) and miss out on some important decision critical information. The distinction has to be drawn and clarified. *Feature models* in the *Feature-Oriented Domain Analysis* (FODA) [Kang et al., 1990] do a good job of organizing "features" (or concerns[1]) in feature trees, embodying feature containment (the parent feature includes its children features, as in "security" implies "authentication" and "encryption"), feature selection (two alternative features, as in "logging" mapping to either "local logging" or "remote logging"), and dependencies across feature tree branches. An example of the latter is the case where our example grading system needs to run in either local mode or client-server mode: remote logging may only be possible with the client server architecture.

There could be other dependencies besides the "excludes" (between "local logging" and "remote logging") and "requires" relationships ("remote logging" requires "client-server architecture).

6.2.4 Identify relationships between concerns and development artifacts

Some of our concerns may map nicely to identifiable development artifacts while others won't. If we are building an on-line auctioning system for eGolf, and eGold is concerned about throughput (e.g. 10,000 transactions/second for any given item), there is no single artefact, regardless of shape, aspect or color, that will address this concern. However, before we give up on a concern completely, we have to look at its subconcerns: it may be the case that *sub-concerns* may map more easily to artifacts.

"Mappable" concerns may map to classes or methods, and yet others may map to subjects, aspects — in the AspectJ sense [Kiczales et al., 1997] — or composition filters [Aksit & Bergman, 1992]. One the many challenges here is to recognize that not all concerns are aspectual — they don't all look like nails to the aspect hammer. Domain engineers should explore alternative implementations of concerns.

6.2.5 Identify interaction patterns between the artifacts and see how they map back to interactions between concerns

In section 2.2, we identified some dependencies between concerns. Some of these dependencies may translate into interaction patterns between the corresponding artifacts. For example, if a concern *C1* implies another concern *C2*, it means that we cannot incorporate (include, compose or weave) the aspect that embodies *C1*—call it *A1*. Without incorporating the aspect that embodies *C2*—call it *A2*. More complex interactions may occur. We may refer to this case as *essential interaction* between aspects.

There may also be cases where the concerns do not interact, but where the corresponding aspects do. This is a case of *accidental interaction*, which would normally be symptomatic of poor aspect (artifact) design or of a shortcoming of the implementation technolog;y. Either way, such interaction patterns have to be identified, and potential resolutions developed.

6.2.6 Discussion

This is a preliminary investigation into the possible synergies between AOSD and domain engineering that focussed on the reuse of non-functional concerns across application domains. Clearly, domain engineering will also benefit from advances in aspect-oriented implemenation techniques as well as advances in concern identification and aspect modeling to handle domain-specific functional requirements.

6.3 Software Architecture

The mindmap for the topic software architecture is shown in Figure 5.

Software architectures include the early design decisions and embody the overall structures that impact the quality of the whole system. It is generally accepted that architecture design should support the required software system qualities. As shown in Figure 5, *Quality Concerns* forms obviously a key issue in Software Architecture.

[1] For simplicity, we equate feature with concern.

For ensuring the quality factors the common assumption is that identifying the fundamental concerns for architecture design is necessary (*Decomposition)*. A number of approaches have been introduced to derive the fundamental architectural abstractions. The abstractions can be derived from the solution domain or the requirements (*Functional Concerns*).

Although the architecture design approaches vary in deriving architectural abstractions they share the common idea that architectural abstractions should represent the relevant concerns of the system. This implies that for identifying the right architectural abstractions, a thorough understanding and an appropriate application of the separation of principle is necessary.

A solid architecture design heavily depends on and represent solution domain knowledge (*Architectural Knowledge)*. This is derived from the domain knowledge and provided by the architectural patterns.

Composing the architectural concerns is one of the challenging tasks (*Composition)*. In general, architectural concerns can be combined in different ways and it is important to derive the appropriate composition with respect to the quality requirements.

Once the architectural abstractions are identified it is necessary that the architecture is appropriately specified (*Specification*). This can be done visually or textually as in the case of architecture description languages.

The conventional approaches have mainly focused on separating the concerns that fit nicely into architectural components. Unfortunately, less focus has been given on concerns that cannot be captured in single components and tend to crosscut several components.

Crosscutting concerns need to be identified, specified and evaluated at the architecture design level. Software architectures are generally documented using architectural views. The key question here is how to specify aspects in the views. Another issue to investigate is how the crosscutting relates to the views. Can aspects be totally captured in single views, if not how to cope with the crosscutting over different views?

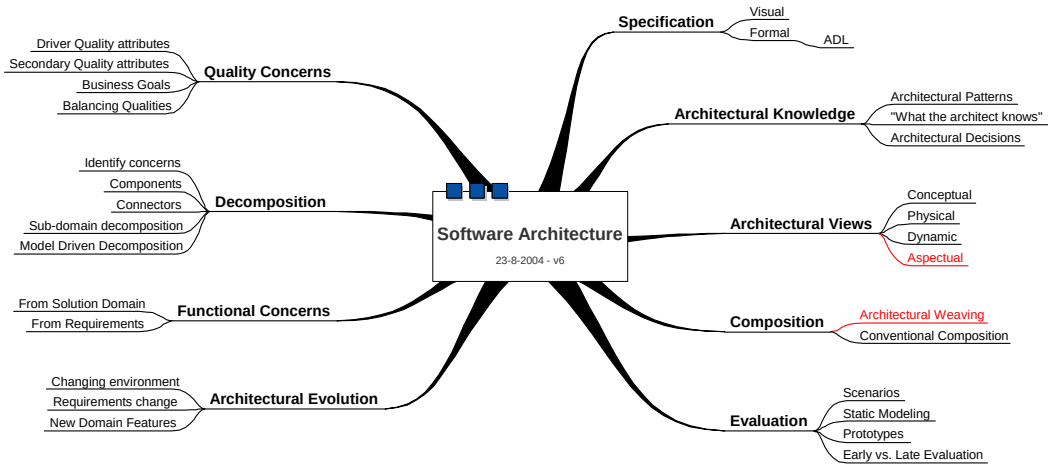


Figure 5. *Mindmap of software architecture topic*

6.4 Specification of Early Aspects

The mindmap for *Specification of Early Aspects* is shown in Figure 6. It should be noted that specification of aspects plays also a role in the other subgroups. During the presentation of the ideas we could also identify some recurring ideas. Specification of early aspects refers to the specification of aspects during the architecture design, requirements analysis and domain analysis phases. Before specifying early aspects it is necessary to identify the aspects first, which is done in the requirements analysis and architecture design phases.

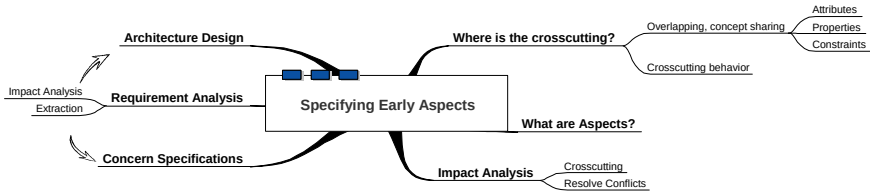


Figure 6. *Mindmap for Specification of Early Aspects*

7. Participants in the Workshop

The participants in this edition of the Early Aspects workshop is listed in Table 2.

Table 2. Participants to the workshop

Participant	Affiliation
1. João Araújo	Univ. Nova de Lisboa, Portugal
2. Elisa Baniassad	Trinity College, Ireland
3. Isabel Brito	Inst. Politécnico de Beja, Portugal
4. Gary Chastek	Software Engineering Institute, US
5. Siobhan Clarke	Trinity College, Ireland
6. Islam El-Maddah	King's College, UK
7. Iris Groher	Siemens AG, Germany
8. Charles Haley	Open University, UK
9. Michael Jackson	Open University, UK
10. Shmuel Katz	Israel Inst. of Technology, Israel
11. Uirá Kulesza	PUC-Rio, Brazil
12. Robin Laney	Open University, UK
13. Marius Marin	Delft Univ. Technology, Holland
14. Hafedh Mili	University of Quebec, Canada
15. Ana Moreira	Univ. Nova de Lisboa, Portugal
16. Bashar Nuseibeh	Open University, UK
17. Jianxiong Pang	Lancaster University, UK
18. Awais Rashid	Lancaster University, UK
19. Pete Sawyer	Lancaster University, UK
20. Miguel-Angel Sicilia	University of Alcalá, Spain
21. Sergio Soares	Federal Univ. Pernambuco, Brazil
22. Daniel Speicher	University of Bonn, Germany
23. Dominik Stein	University of Essen, Germany
24. Stan Sutton	T. J. Watson Research Center, US
25. Bedir Tekinerdogan	Univ. of Twente, Holland

8. Conclusions

The results of the workshop show that the early aspects topic is still in its infancy but progressing. The goal of the workshop was primarily not to find solutions but first to identify the right questions to shape the research of the early aspects topics. During the workshop a number of key research areas have been identified to scope and consolidate the area of early aspects.

Currently we can state that the scope of the early aspects domain is defined to a large extent. This workshop showed that several ideas are recurring with respect to the previous early aspects workshops. In particular there seems now to be an agreement that *early aspects* refers to the aspects that can be identified during the requirements analysis, domain analysis and architecture design phases. This means that aspects during the detailed design are not counted as early aspects. In this report we have termed the aspects at the detailed design as *intermediate-aspects.* Aspects which refer to the crosscutting concerns at the implementation phase, testing and maintenance phases are termed as *late aspects*.

Since early aspects refers to the crosscutting concerns during the requirements analysis, domain analysis and architecture design phases, the research is also focused on these three phases. So far, in general, the research on early aspects appeared to proceed separately and independently in each of these phases. It appears, however, that the early aspects in the three phases are not independent and directly impact each other. A concern such as *synchronization* that is identified during the requirements analysis phases requires the modeling of it during the architecture design. During the domain analysis some aspects might be identified which were overlooked during the requirements analysis phases. There is certainly a relation among the concerns but so far the parity and the semantics of the relations among the concerns in the early phases are not completely clear yet and more research is required to crystallize the concepts.

This workshop and the previous workshop have shown that early aspects exist and that they need to be handled with care to provide better maintainable software. In the future, we expect that the questions addressed in this workshop will be solved gradually.

9. Acknowledgements

We would like to thank the participants to the workshops and the reviewers of the workshop papers.

10. References

[Aksit et al., 1992] M. Aksit, L. Bergmans, and L. Vural, “An object-oriented language-database integration model: the composition filters approach”, *in Proc. of ECOOP 92*, Springer Verlag 1992.

[Domain, 1999] *Domain engineering: A model-based approach*, technical report, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 1999. http://www.sei.cmu.edu/domain-engineering.

[Harrison & Ossher, 1993] W. Harrison & H.Ossher, “Subject-oriented programming: a critique of pure objects,” *in Proc. OOPSLA’93*, pp. 411-428.

[Kang *et al.* 1990] Kang, K., S. Cohen, J. Hess, W. Novak, and A. Peterson, "Feature-Oriented Domain Analysis (FODA): Feasibility Study" CMU/SEI-90-TR-021. www.sei.cmu.edu/publications/documents/90.reports/ 90.tr.021.html

[Kiczales et al., 1997] G. Kiczales, J. Lamping, C. Lopez, “Aspect-Oriented Programming,” in *Proc. ECOOP’*97.

[Mili et al., 2001] H. Mili, A. Mili, S. Yacoub & E. Addy, *Reuse-Based Software Engineering*, Addison-Wesley, 2001

[Mili et al., 2004] H. Mili & A. Elkharraz, “Aspect composition: problems and some solutions,” LATECE Technical Report LAT-3-4-04, 12 pages, March 2004.

Finding Aspects In Requirements with Theme/Doc

Elisa Baniassad and Siobhán Clarke
Department of Computer Science
Trinity College, Dublin 2, Ireland
{Elisa.Baniassad, Siobhan.Clarke}@cs.tcd.ie

Abstract

Aspects are behaviours that are tangled and scattered across a system. In requirements documentation, aspects manifest themselves as descriptions of behaviours that are intertwined and interdependent. Some aspects may be obvious, as specifications of typical crosscutting behaviour. Others may be more subtle, making them hard to identify. In either case, it is difficult to analyse requirements to locate all points in the system where the aspects should be applied. To identify aspects early in the software lifecycle developers need support for aspect identification and analysis in requirements documentation. To address this, we have devised the Theme/Doc approach for viewing the relationships between behaviours in a requirements document to identify and isolate aspects in the requirements. This paper describes the approach, and illustrates it with a case study and analysis.

1. Introduction

Conceptually, an aspect is an element of functionality that is woven throughout other system behaviours. At the source level, aspects are tangled and scattered code. At the requirements level, aspects are tangled and scattered descriptions of functionality. In a requirements document, tangled functionality can only be described in relation to other functionality, whereas scattered functionality is described throughout the requirements document.

Tangling and scattering at the requirements level makes it difficult for developers to reason about whether they have encountered aspects. It is difficult to mentally note whether a behaviour that is scattered across a document is actually dependent on (tangled with) other behaviours, or just badly encapsulated in the requirements set. Additionally, it is often difficult for a developer to keep in mind which behaviours are tangled.

These difficulties place a barrier between a developer and full adoption of aspect-orientation, because they make it difficult for developers to conceive of whether they have "crosscutting behaviours" in their set of requirements.

Using intuition or even domain knowledge is not necessarily sufficient for identifying the potentially broad range of aspects within a reasonable amount of time. For instance, developers might start by looking in their documentation for typical aspect-style behaviour, such as logging, tracing, or debugging functionality, or non-functional requirements, but this likely does not cover the full range of potential aspects.

We assert that developers need support for viewing and manipulating their requirements to expose how elements of functionality relate to one another.

To address this need, we propose the Theme/Doc approach, which provides views of requirements specification text, exposing the relationship between behaviours in a system. These views assist a developer in determining which elements of functionality are "aspects" and which are "base". Theme/Doc views also provide a feature-oriented view of a requirements set. All views can be mapped to Theme/UML models. Theme/Doc and Theme/UML together comprise the *theme approach*.

1.1. Theme/Doc

Theme/Doc is based on the notion of a *theme*, which represents a feature of a system. Multiple themes can be combined to form a functioning whole according to a multi-dimensional model [10]. There are two kinds of themes: *base themes*, which may share some structure and behaviour with other base themes, while modelling these from their own perspective, and *crosscutting themes* which have behaviour that overlays the functionality of the base themes. Crosscutting themes are *aspects* [3].

The Theme/Doc tool operates on the basic assumption that if two behaviours are described in the same requirement,[1] then they are related. Behaviours can relate in three

[1] We currently take a requirement as being a sentence in a requirements document, or a single requirement in a set of requirements. Because of the lexical nature of the tool, however, the granularity and format of a

ways: they can be erroneously or coincidentally related, meaning that the requirement could be re-written so that they were no longer coupled, they can be hierarchically related, in that one behaviour is a sub-behaviour of another, or they can be related by crosscutting, meaning that the requirement describes one behaviour as an aspect of another. The Theme/Doc tool provides views that expose which behaviours are co-located in requirements. These views assist the developer in determining what kind of relationships exist between behaviours, and whether those behaviours are base or aspects.

In Section 2 we outline how to apply the Theme/Doc approach and tool. We then present a case study (Section 3) in which we apply the approach on a larger example. Next we raise issues for discussion (Secion 4), and review related work (Section 5). Finally, we conclude (Section 6).

2. A Small Example

In this section we work through the example of a small Expression evaluation system to illustrate the basic points of how to use Theme/Doc to support the identification of aspects in a set of requirements.

2.1. Expression System Requirements

1. evaluation capability which determines the result of evaluating expression.
2. display capability which depicts expression textually.
3. check-syntax capability which determines whether expression are syntactically correct.
4. log capability that logs the evaluation display and check-syntax activities.
5. an expression is grammar-defined as a variableexpression or a numberexpression or a plusoperator or a minusoperator or a unaryplusop or a unaryminusop.
6. a plusoperator is grammar-defined as an expression and a plus and an expression.
7. a minusoperator is grammar-defined as an expression and a minus and an expression.
8. a unaryplusop is grammar-defined as a plus and an expression.
9. a unaryminusop is grammar-defined as a minus and an expression.
10. a variableexpression is grammar-defined as a letter and an expression.
11. a numberexpression is grammar-defined as a number and an expression.

requirement can be set arbitrarily.

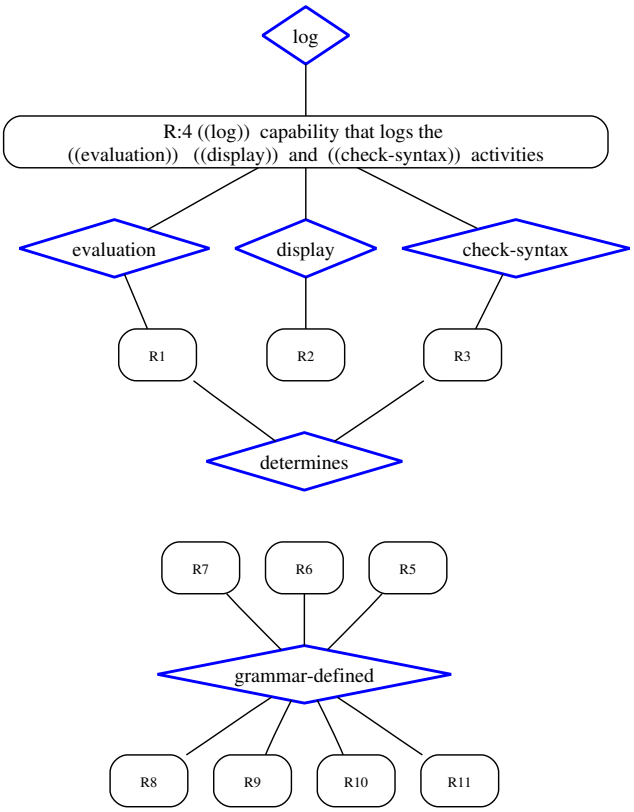


Figure 1. Action View

2.2. Identify Actions and Entities

The tool takes the requirements, as written above, as input. Behaviours in requirements are identified by a set of keywords provided by the developer to the Theme/Doc tool. These behaviours are referred to as "actions". Strictly speaking, any lexical string, including a non-functional requirement, state, etc. can be made a "key-action" if it was considered a candidate theme. However, we have found that using actions is a good starting point for finding themes, and that requirements that seem not to contain actions can often be refined to include actions. The developer also provides a set of key-entities as input. Entities can also encompass resources. It uses these inputs to generate the Theme/Doc views.

For this set of requirements, we identify six actions: evaluation, display, determine, check-syntax (the whole concept, not just the verb "check"), log, and grammar-defined. We also identify nine entities: expression, variable-expression, number-expression, plus-operator, minus-operator, unary-plus-operator, unary-minus-operator, plus and minus.

2.3. Categorize Actions into Themes

Theme allows the individual design of different system features. In Object-orientation, not all the nouns in a requirements document are designed as objects or classes. Similarly, in Theme, not all actions are designed as separate features of the system: some actions are sub-behaviours of other actions. In this step, we set out to identify the features, or themes of the system, by identifying which actions are major-enough to be modeled separately (once we get to the point of modeling). For this, we use a Theme/Doc view called the *action view*.

Figure 1 shows the action view for the Expression system. An action view consists of two elements: actions, shown as diamonds, and requirements, shown as rounded boxes. If an action is mentioned in a requirement, there is a line linking the action to the requirement.

In this view, we can either see requirements as labels (their requirement number), or we can enlarge them to see their content, as we have for R4. The action view is non-hierarchical, so even though it looks as though some actions are "higher" than others, this is just a coincidence of layout.

We use this view to determine whether actions should be themes, or just behaviour (perhaps methods) within themes. Deciding which actions are not major enough to be a theme is a highly intuitive process. We scan these actions and question whether it makes sense to have each of them as a feature in our system. If they are not feature-worthy, we demote them from our action view. The remaining "major actions" will be our themes. The "log" feature makes sense as a theme: it is something that we would, perhaps, like to turn on or off, or at least model separately from the other actions we see in the view. A "check-syntax" feature makes sense for the same reason, as does a "display" feature, an "evaluation" feature, and a feature centrally responsible for defining the grammar (the "grammar-defined" theme). The "determines" action, however, does not seem to be as strong a potential theme as the rest. It is involved in two requirements, but in a relatively minor way and it seems hard to imagine it as a collection of classes. It is more likely a sub-behaviour: a method, rather than a feature in and of itself. For this reason, we decide to demote "determines" from our action view.

2.4. Identify Crosscutting Themes

We use the *major action view* to help us determine which themes are base, and which are aspects. This view is made up only of the major actions from the previous action view. It is identical to the view shown in Figure 1 except the "determines" node is removed.

Our focus in using this view is on the requirements that are shared by more than one theme. If a requirement is shared by two or more themes, we must decide which theme should provide that functionality. Shared requirements flag for us that we may have identified an aspect in our system, since they imply that two themes cannot operate without behaviourally relying on one another.

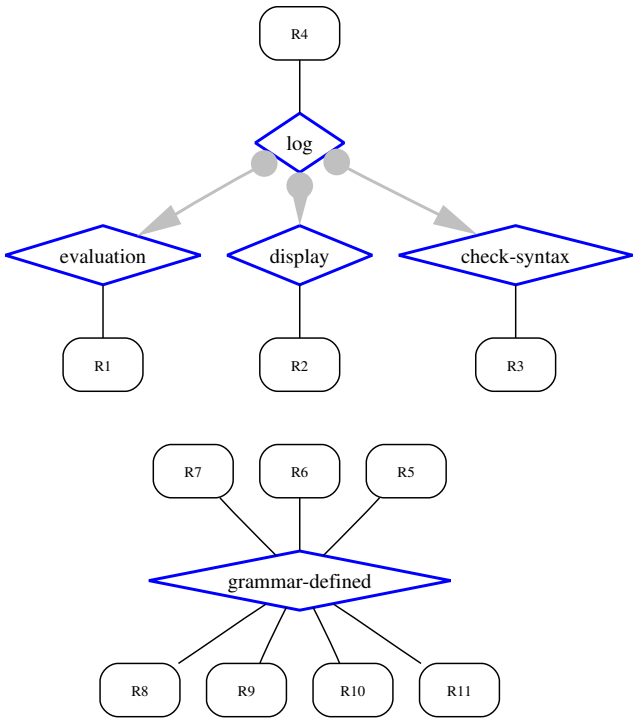


Figure 2. Clipped Action View

As we can see, some requirements are already associated with only one theme. It is straightforward to assume that whatever functionality should be associated with those themes. So, when designing the check-syntax theme we know it should implement the functionality described in R3.

We begin our investigation by inspecting R4 (shown in Figure 1) which is shared between several themes. We want to ensure that we have not encountered a vaguely written requirement that is masquerading as a shared one. So, we check to see whether the requirement can be re-written into several requirements that each refers to only one theme. However, we can see that in this case the only possible re-writing would be to break it into three sentences that each mention log, and another of the themes (a log capability that logs the evaluation activity; a log capability that logs the display capability, etc). There is no re-writing that gets a 1-1 relationship between themes and requirements: the log feature must be overlaid on the evaluation, display, and check-syntax features. This means that the log theme is an aspect.

We denote that the log theme crosscuts the other three associating the shared requirement, R4, with it. This association clips the links from R4 to the other three themes. In its place a grey arrow indicating a crosscutting relationship is placed from the aspect theme to the base themes (Figure 2). It is the developer's job to ensure that this 1-1 relationship is achieved. If a particular shared requirement is too ambiguous to make the association clear, then it is up to the developer to revisit the requirements set, and resolve the ambiguity.

There are no other shared requirements, so we can now move on to examining our themes individually, and planning for design. The product of this process is referred to as the *clipped action view*. The grey arrows in this view indicate the crosscutting-hierarchy.

2.5. View Individual Themes

Themes (at this point a "major action" is the same as a theme) can be viewed individually as well as grouped in the action views. An individual theme view shows the requirements associated with the major action, as well as minor actions mentioned in the requirements. Key entities are also shown individually in this view, as boxes. These views are used to check that the associations were made correctly when manipulating the major action view to form the clipped view. They can also be used to determine how themes should be modeled using Theme/UML [1].

3. Case Study

The goals of this case study were to test the Theme approach on a larger example, and perform preliminary assessment of it in terms of effectiveness for finding aspects, support for assessment of requirements coverage, and scalability. We will first give a general description of the location aware game that was the basis for the case study, and then provide results and analysis.

3.1. Location Aware Game

The set of requirements used in this case study are those for a location aware game called the Crystal Game, which was developed in an independent research project. The game has 89 requirements, so is roughly eight times larger than the example provided in Section 2. The object of the game is to collect crystals that are deposited throughout the location. As a player moves around the game space, their hand-held device will alert them when they have encountered a crystal. Computer-generated characters also take part in the game. When a player encounters one of them, they will interact and perhaps duel. When a player encounters another player, they will duel, and the loser will turn all

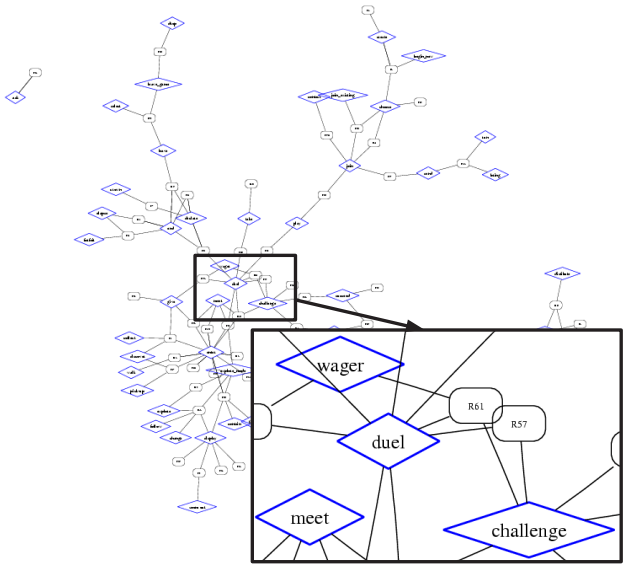


Figure 3. Game Action View: All Actions

of their crystals over to the winner. The game ends after a specified time period. The winner is decided by how many crystals each player has. There are other constraints and requirements in this game which will be of interest and will be described in later sections.

3.2. Results

In this section we review the steps we took to apply the Theme approach to the Crystal Game requirements.

3.2.1. Finding Themes

We identified 59 actions in the game requirements, and generated an action view to examine their relationships. Based on intuition and some cursory analysis of the view, we determined that all of these actions should not be modeled as separate themes. Instead, we examined the view to determine the relationships between the actions, to decide how to group the actions into larger themes.

This was a mainly analyticial process, but it was supported by the action view. Because actions that share requirements are displayed close to one another in the view, we were able to examine closely located actions to assess whether they should be grouped into a common theme.

We used the view shown in Figure 3 to perform such an assessment. This figure shows the initial action view for the game, with the centre portion of the view enlarged. The enlarged view shows four actions, *duel, wager, challenge* and *meet*. We examined the requirements they shared, considered the meaning of the actions, and determined that *duel,*

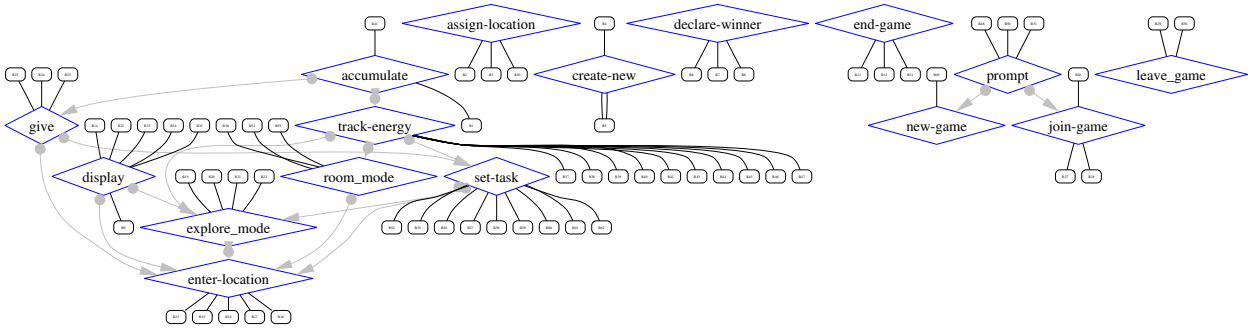


Figure 4. Clipped Action View of Major Game Actions

wager, and *challenge* should all be grouped under the general heading of *duel*, since players challenge one another to duel, and wager crystals on the outcome of a duel. In that case, we classified *duel* as being more major than *wager* and *challenge*, which we saw as sub-actions of *duel*. The *meet* action was connected to *duel* because when players encounter one another they duel. We examined requirements shared by *meeting* and *duelling* and determined that since they were not synonymous, they should not be grouped into one theme. Later, we determined that *duel* and its sub-actions should be grouped under the more major action, *set-task*. In the end, we arrived at the view shown in Figure 4, which displays the 16 major actions which became our themes. Of those actions, five are independent, while others share requirements, and hence crosscut one another in some way.

The clipping functionality of the tool helped us investigate the major action view to determine which themes are crosscutting and which are base. In the case of the *prompt* theme this was straightforward. The *prompt* theme (shown to the right of Figure 4) shared requirements with two other themes, *new-game* and *join-game*. By examining the shared requirements it could be seen that the prompting behaviour crosscut these two themes.

As is visible on the left side of Figure 4, there are several related themes. To determine which of those was crosscutting, we began by assessing the requirements between the *explore-mode* and *enter-location* themes. We determined that *explore-mode* crosscut *enter-location*. By continuing to examine themes that shared requirements with *enter-location* we further determined that *room-mode* was crosscutting, as was *give, accumulate, set-task*, and *display*. We then examined the remaining shared requirements, and encountered themes that crosscut other crosscutting themes. For instance, the *track-energy* theme was determined to crosscut *set-task, room-mode* and *explore-mode*, all of which crosscut *enter-location*. There are five themes that crosscut other crosscutting themes: *display, set-task, give, track-energy* and *accumulate*.

3.2.2. Determining Composition Order of Themes

We used the crosscutting relationships shown in the clipped action view (Figure 4) to determine the order of binding. In this view, the themes are positioned hierarchically, based on whether they crosscut one another. The grey arrows indicate which themes crosscut other themes. We can see, for instance, that there are no grey arrows extending from *enter-location*, which indicates that it is base functionality. To determine the binding order, we begin with the lowest themes in the crosscutting-hierarchy, and work to the highest, incrementally binding in one crosscutting theme at a time. To determine what is first in the binding we identify the themes that crosscut only that theme (*room-mode* and *explore-mode*), and placed those first, and second in the binding order. The final bindings were done with the "more" crosscutting themes: *display, give* and *track-energy*. The very last binding is of *accumulate*, since it crosscuts the *give* and *track-energy* themes. It is not intended that establishing the aspect-base associations at the Theme/Doc level guarantees that conflict-free theme composition at the modeling stage. Theme/Doc is only intended to be used to assist visualization of the relationships between elements of functionality described in the requirements as opposed to a design/model checker.

3.3. Analysis

In this section we discuss how the results of the application of the Theme approach reflect on its effectiveness at support for aspect identification, requirements coverage, and on its scalability.

3.3.1. Effectiveness of Support for Aspect Identification

Through the application of the Theme approach, we were able to identify eight aspects: *explore-mode, room-mode, accumulate, track-energy, give, set-task, display* and *prompt*. Had we carefully read the requirements document

we may have identified seven of these behaviours as aspect behaviours since they provide tracking or logging style functionality. However, it is unlikely that we would have identified the *give* functionality as an aspect because mentions of the *give* action are spread throughout the document, and it might have been difficult to recall that the same abstract behaviour is occurring with relation to different system features. Also, since in the document text it is described as a consequence of other actions, such as meeting, and duelling, it is possible that we would have automatically thought of *give* as a method in those actions. It wouldn't have been until we were modelling or implementing it that we would have noticed its crosscutting nature.

We also found our approach effective support for determining the binding order for multiple crosscutting themes. This may be otherwise difficult to determine.

3.3.2. Requirements Coverage

We were initially concerned that it may be difficult to assess whether all the requirements have been associated with a theme. We noted that the action view can be used to monitor requirements coverage, because if a requirement is not associated with a theme it is orphaned in the view. By orphaned, we mean that only the sentence record for the requirement appears, without being linked to a diamond-shaped action. We found that a requirement could be orphaned in two ways. Orphans can appear in the initial action view if the requirement contains no key actions. This may happen if it refers to another action, but does not mention it explicitly. By inspecting the requirement we can identify the requirement's original location in the text, and can read the requirement in context to determine to which action it refers. The other way orphans can appear is when forming the major action view. As major actions are identified, they are added to a new list of keywords. The minor actions that have been grouped under the major action will be annotated so that they will be linked to the major action in the view. Minor actions that are not grouped with major actions will disapear, and their requirements will appear to be orphaned. We systematically visited the orphaned requirements to determine whether any of their minor actions should be promoted to major, or whether to group those requirements under other major actions.

3.3.3. Scalability of Action Views

When applying Theme/Doc, actions are classified into two types: major and minor. Major actions become themes, while minor actions were slotted to become methods within a theme. This approach has essentially provided two "zoom-levels" of action view: a developer can zoom-in to see all the actions, or can zoom-out and just see the major actions. This approach worked very well with the small Expression example, and was very useful for the Crystal Game.

However, were we to scale the requirements further, it would be necessary to apply other approaches, since it would not be feasible to fit an entire major action view for a very large system onto a screen or a page. In this case, query functionality is needed to form sub-views that could be examined separately from the entire action view. Additionally, it would likely be useful, in a larger system, to provide more degrees of zooming so at some level the entirety of the system could be seen in one view.

4. Discussion

In this section, we provide discussion of issues we noted while performing our case study.

4.1. Synonyms

Synonyms are handled through a synonym dictionary which, for the sake of the action view, automatically augments the requirements text so that the correct associations will be made. This is more complicated when two words are the same but have different meanings in terms of the system. For instance, the term *give* was used in the Crystal Game not only for giving crystals, but also for giving audio and visual signals to players. The action view helped identify instances where this occured, because the common action brought together other actions which, upon analysis, should not be linked. For instance, the common term *give* brought closer together *accumulate* and *prompt*. We could intuit from having read the requirements document that these two actions should be unrelated. When inspecting the relationships around the *give* action, it was clear that the term was being used in different senses. We then used the annotation feature of the tool to replace the audio sense with the term *give-audio*. These annotations are not shown in the theme view.

4.2. Ambiguities Found in Requirements

We found that the Theme approach helped us identify ambiguities in the original requirements specification. While refining the 59 actions into the 16 themes we found that there were subtle ambiguities in the initial requirements document. For instance, we found that it was not explicitly mentioned how crystals were collected by a player, unless the crystal was actually given to them by another player or a game character. One requirement mentioned picking up crystals ("a player explores the world and picks up crystals"), and another mentioned the accumulation of crystals ("a player collects crystals by discovery in a location, or when a player or character gives one to them.") Though it

was implied, there was no specific description of a player actually picking up a crystal when they discover it in a location. This subtle ambiguity was discovered when we saw that the *pick-up* action, and the *collect* action were not located close to one another in the view shown in Figure 3, though we knew intuitively that they should be related. This was highlighted because the *collect* action was closely placed to the *give* action.

4.3. Evolution of Requirements

Neither our example nor our case study considered what would happen were the requirements to change over time. There are two situations in which requirements can change: during the requirements gathering and analysis stage, or after modeling has begun. In the former situation, the evolution can be handled by re-generating the views for the set of requirements, and deciding which themes the new requirements should fall under. A developer would follow the same process as outlined for the original requirements set for the new or changed set of requirements. This is also a possibility in the latter situation, after modeling has begun.

5. Related Work

There have been several efforts in capturing and relating aspect-oriented requirements [9, 11, 4, 8, 7, 6, 2]. Here we consider the two which relate most closely to the Theme approach.

Rashid et al [8] provide the AORE (Aspect-Oriented Requirements Engineering) model and ARCaDe (Aspectual Requirements Composition and Decision support) approach and tool for describing components and requirements-level aspects. Examples of these aspects are compatibility, availability, or security. This work builds on the ViewPoints model [5], which is intended to support the integration of heterogeneous requirements specified from multiple perspectives. An early stage in the AORE model is the identification and specification of concerns. The approach to this differs from the Theme approach to concern identification in that it relies on the domain knowledge of the developer to identify possible non-functional requirements to be taken into account when implementing a particular requirement. Those concerns are not explicitly mentioned in the requirements specification; it is up to the developer to ascertain their relevance on their own. We see this as a complemetary approach to our own. Such domain knowledge will always play a large part in system design. The Theme/Doc approach aims to support the analysis of relationships between behaviours described in requirements specifications. It is possible that the Theme/Doc approach to aspect identification could be used during the concern identification phase of AORE, or could support AORE's extension to include functional as well as non-functional requirements.

Katera and Katz [7] propose architectural views of aspects as a means for reasoning about the relationships among aspects in a system. They describe aspects as cross-cutting augmentations to an existing design. In particular, they allow for specification of the overlap between aspects through the concept of a *sub-aspect* that provides the overlapping functionality, and they make relationships between aspects explicit. A UML approach is given to support these views which differs from the Theme/UML approach: it provides additional architectural support for aspect modelling to that provided by Theme/UML, and it uses aspect mappings rather than multi-dimensional composition style semantics. Theme/Doc could be integrated into this approach since the relationships exposed between behaviours in a set of requirements could be used to establish the behaviours between aspects and sub-aspects in this approach, as well as support the identification of functionality shared between components.

6. Conclusions

In order to identify aspects in a set of requirements, we need to see how behaviours described in the requirements relate to one another. In this paper we have presented Theme/Doc, which provides views of requirements specifications that are intended to expose relationships between behaviours in requirements. Our case study showed that this approach is effective in helping to identify aspects in requirements, and helped us identify functionality that would enhance the scalability of the approach.

7. Acknowledgements

We would like to thank Conor Ryan, Alan Gray, David McKitterick, Karl Quinn and Tonya McMorrow from the original Crystal Game development team, on which our case study was based, and Mary Lee for her work with early versions of Theme/Doc as applied to the Crystal Game.

References

[1] E. Baniassad and S. Clarke. Theme: An approach for aspect-oriented analysis and design. In *International Conference on Software Engineering (to appear)*, 2004.

[2] J. Castro, M. Kolp, and J. Mylopoulos. Towards requirements-driven information systems engineering: The tropos project, 2002.

[3] S. Clarke and R. J. Walker. Composition patterns: An approach to designing reusable aspects. In *International Conference on Software Engineering*, pages 5–14, 2001.

[4] R. Darimont and A. van Lamsweerde. Formal refinement patterns for goal-driven requirements elaboration. In *Foundations of Software Engineering*, pages 179–190, 1996.

[5] A. Finkelstein. The viewpoints faq. *BCS/IEE Software Engineering Journal*, 11(1), 1996.

[6] J. Grundy. Aspect-oriented requirements engineering for component based software systems. In *4th IEEE International Symposium on Requirements Engineering*, pages 84–91.

[7] M. Katera and S. Katz. Architectural views of aspects. In *Proceedings of the International Conference on Aspect-oriented Software Development*, pages 1–10, 2003.

[8] A. Rashid, A. Moreira, and J. Araujo. Modularisation and composition of aspectual requirements. In *Proceedings of the International Conference on Aspect-oriented Software Development*, pages 11–20, 2003.

[9] S. Sutton. Early stage concern modeling. In *Early Aspects Workshop, Held with AOSD*, 2002.

[10] P. Tarr, H. Ossher, W. H. Harrison, and S. S. Jr. N degrees of separation: Multi-dimensional separation of concerns. In *Proceedings of the International Conference on Software Engineering*, pages 107–119. IEEE Computer Society Press, 1999.

[11] X. Wang and Y. Lesperance. Agent-oriented requirements engineering using congolog and i*. In *Submitted to AOIS-2001, Bi-Conference Workshop at Agents 2001 and CAiSE'01.*, 2001.

Problems, Subproblems and Concerns

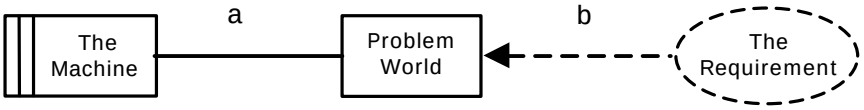
Position Paper submitted to
Early Aspects 2004

Michael Jackson

Abstract. Inevitably, aspect-oriented programming has focused on solutions; 'early aspects' aims to focus on problems. This position paper sketches how problems may be understood from a perspective based on problem frames. Problem analysis from this perspective reveals structural issues in a clearer light. It leads to a need for composition, both in the problem world and in the solution world. The goals of aspect technology would be clarified by such analysis, and the aspect technology may in turn offer some power in understanding and implementing the compositions.

1. The Machine and the Problem World

The goal of a software development is to produce a system to satisfy some r*equirement*. The system may be regarded as having two fundamentally distinct parts. The *machine* consists of the software to be built, executed by one or more general-purpose computers. The *problem world* is where the problem is located: it is here that the machine's behaviour will take effect and the success of the system must be evaluated. Essentially, the problem world is given[1]; the machine is what we must construct. This view [Jac00] can be represented in a simple diagram:



The interface *a* consists of a set of shared phenomena (for example, events and states) by which the machine can detect the behaviour and state of the problem world and can affect the problem world through its control of some of those shared phenomena. The requirement is a condition on the behaviour and states of the problem world whose satisfaction the machine must ensure. The requirement is expressed in terms of a set of phenomena *b*. In general these phenomena *b* are distinct from the phenomena of the interface *a*: the requirement, therefore, can not be expressed solely in terms of the behaviour and properties of the machine. It follows that satisfaction of the requirement depends not only on the specification of the machine's behaviour at *a*, but also on the causal properties of the problem world by which the machine's behaviour at *a* can be guaranteed to produce the required effects in terms of the problem world phenomena *b*.

The early development stages[2] must therefore include explicit articulation of the requirement, detailed specification of the appropriate machine behaviour at interface *a*, and identification and description of those properties of the problem world on which the specified machine will rely to ensure satisfaction of the requirement.

[1] That is, the causal and structural properties of the problem world are given, and some of its behavioural properties. It is usually the purpose of the machine to alter the problem world's behaviour within the given constraints.

[2] That is, the *conceptually* early stages. No position is being taken here on the *temporal* ordering of development activities.

2. Problem Decomposition

Since any realistic system will be too large and complex to handle as a whole, developers need ways of structuring that will allow them to master its size and complexity. Three interrelated structures are immediately identifiable:

- The requirement may be decomposed into *subrequirements*. For example, the requirement "provide lift service" may be initially decomposed into "service user requests" and "maintain safe operation".
- The problem world may be decomposed into *domains*. For example, the problem world of the lift control system may be decomposed into "users", "buttons", "lift mechanism" and "doors".
- The machine may be structured into *submachines*. For example, there may be one submachine devoted to scheduling lift service, another devoted to detecting malfunctions of the lift winding gear, and a third devoted to emergency action.

These decompositions may be understood as facets of the decomposition of the whole problem into *subproblems*. A subproblem is itself a problem, in the sense that it has a machine, a problem world and a requirement. The problem world for a subproblem is some projection of the problem world of the whole problem. For example, the problem world for the "maintain safe operation" suproblem may have only the "lift mechanism" and "doors" domains, and only certain phenomena and properties of those domains: the "users" and "buttons" may be irrelevant.

3. Subproblem Concerns

A central advantage of this kind of decomposition is that subproblems can be restricted to known problem classes, each characterised by a *problem frame*. Object patterns [Gam94] capture classes of design solutions; problem frames may be thought of as patterns in the problem space. Because each subproblem is of a familiar and documented class, the *concerns* it raises can become, over time, fully documented and well known to all competent developers. These concerns may be associated with particular types of problem domain, of requirement, or of relationship between the machine and the problem domains.

When the concerns associated with a problem are well known and documented, development can become more reliable. Many of the failures described in the literature on risks in computer-based systems—for example, in the work of Neumann[Neu95]—stem from neglect of concerns that should have been immediately recognised. Two examples of such concerns are:

- Initialisation. When software execution begins it is—typically—necessary to ensure that the machine and the problem world are in corresponding initial states. A failure of this kind led to friendly fire deaths in the Afghanistan war [Was02][3].
- Identities. When a problem domain contains multiple instances of an entity type, care is needed to ensure that the machine interacts on each occasion with the appropriate instance. A failure of this kind either led to, or contributed to, at least one air crash in which many people died [Neu95 pp44-45]. Failure to address an identities concern is so common in certain engineering fields that its symptom—'cross-wiring'—has an established name.

By structuring in terms of subproblems of known types, with known concerns, the developer can more readily apply the corpus of existing knowledge to their analysis and solution.

[3] I owe this illustration to Steve Ferg.

4. Subproblem Composition

Decomposition alone is never enough: it is always necessary to recombine the decomposed parts. At first sight it may seem that subproblem requirements can be combined by logical conjunction, and subproblem machines by concurrent execution: subproblem domain projections need no composition because they are projections of an already composed physical reality, and software implementation demands no more than a mechanism for appropriate distribution of shared events.

However, this optimistic view is far too simple. Wherever two subproblems have problem domain phenomena in common there is a potential interaction that must be appropriately handled in the composition. Three examples of these *composition concerns* are:

- Interference. This is a well-known concern. Wherever there are two machines, one controlling phenomena in a common domain and another monitoring the domain state—that is, a *writer* and a *reader*—some form of mutual exclusion may be necessary.
- Interleaving. Consider a one-way traffic light system in which the schedule of light phases can be edited at a console by a privileged operator: the two subproblems, editing and using the schedule, must be interleaved. In addition to avoiding interference it is also necessary to ensure that the changeover between two valid schedules does not give rise to an invalid sequence—for example, one in which a green light for northbound traffic is followed immediately by a green light for southbound traffic.
- Conflict. Two subproblems may require contradictory effects in the domain. For example, the requirements "service user requests" and "maintain safe operation" may be in conflict when an equipment fault is detected. The requirement to service user requests demands that the motor be turned on in order to move the lift car in response to a request; but the requirement to maintain safe operation may demand that the motor be switched off and the emergency brake applied to prevent further movement of the lift car.

The composition task, then, may demand introduction of an appropriate communication mechanism, or the choice and enforcement of requirement precedence to resolve conflict, or even the recognition and analysis of the composition itself as an additional subproblem.

5. Why Defer Composition?

At first sight, the composition task may seem to be a self-inflicted wound. Why adopt an analysis and decomposition technique that gives rise to an apparently gratuitous additional difficulty? The answer, of course, is that the difficulty of composition is neither gratuitous nor additional. The difficulty is always there: the issue is simply whether the composition concerns should be dealt with earlier or later—before or after the subproblems have been identified and analysed. Here we advocate that composition should be deferred until the subproblems to be composed have been thoroughly understood.

Premature composition is the source of much unnecessary complexity in software development, because it forces the treatment of each subproblem—whether separately recognised or not—to address both the subproblem's own concerns and the composition concerns in which it participates. The developer's view of the subproblem is polluted and confused by the composition concerns, and in this way important specific subproblem concerns can more easily be left unrecognised and neglected.

Deferring composition, by contrast, reveals the subproblems in their pure forms, in which they correspond more exactly to the patterns defined in their problem frames. Documented relevant knowledge of the concerns associated with each class of subproblem can therefore be more reliably applied. Deferring composition also allows the composition

concerns themselves to be addressed in a context in which the composition task itself is well defined because the components to be composed—the subproblem requirements, machines, and problem domains—have been adequately captured and analysed.

The argument underlying this approach has a clear relevance to the consideration of aspect orientation in general and of early aspects in particular. It is no distortion to say that a fundamental motivation of aspect orientation at any stage of development is to separate consideration of the parts from consideration of how they should be, and can be, composed. This motivation is given particularly clear and salient expression by the use of problem frames for requirements and problem analysis.

References

[Gam94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides; *Design Patterns: Elements of Object-Oriented Software*; Addison-Wesley, 1994.

[Jac00] Michael Jackson; *Problem Frames: Analysing and Structuring Software Development Problems;* Addison-Wesley, 2000.

[Neu95] Peter G Neumann; *Computer-Related Risks*; Addison-Wesley 1995.

[Was02] Staff writer Vernon Loeb; 'Friendly Fire' Deaths Traced to Dead Battery: Taliban Targeted, but U.S. Forces Killed; Washington Post, Page A21, March 24, 2002.

Integrating the NFR framework in a RE model

Isabel Brito
Departamento de Engenharia,
Escola Superior de Tecnologia e Gestão
Instituto Politécnico de Beja,
7800-050 Beja, Portugal
+351 284311540
isabel.sofia@estig.ipbeja.pt

Ana Moreira
Departamento de Informática,
Faculdade de Ciências e Tecnologia,
Universidade Nova de Lisboa,
2829-516 Caparica, Portugal
+351 212948536
amm@di.fct.unl.pt

Abstract

In this paper we build on our work already produced on advanced separation of concerns for requirements engineering by adding two main ideas: (i) the integration of catalogues to help identifying and specifying concerns and (ii) improve the composition rules by informally defining some new operators.

1. Introduction

Separation of concerns refers to the ability of identifying, encapsulating and manipulating parts of software that are crucial to a particular purpose [5]. Separation of concerns reduces software complexity and enhances understandability and traceability throughout the development process. It minimizes the impact of change promoting evolution.

Non-functional concerns, such as response time, accuracy, security and reliability, are properties that affect a system as a whole. Existing approaches to deal with non-functional concerns have some limitations due to the diverse nature of these concerns. Moreover, most approaches handle non-functional concerns separately from the functional requirements of a system. This shows evidence that the integration is difficult to achieve and usually is accomplished only at the later stages of the software development process.

Another problem is that the current approaches fail in addressing the crosscutting nature of some concerns, i.e. it is difficult to represent clearly how these concerns can affect several requirements simultaneously. Since this is not supported from the requirements stage to the implementation stage, some of the software engineering principles, such as abstraction, localization and modularisation, can be compromised. Furthermore, the resulting system is more difficult to maintain and evolve.

This paper builds on our work already produced on advanced separation of concerns for requirements engineering. In [2], and improved version of [3], we have presented an approach for requirements engineering to identify, specify and integrate concerns, including crosscutting concerns. In those papers, we proposed the concepts of match point, dominant concern and composition rule. The novelty introduced here is, on the one hand, the use of catalogues, such as the NFR framework [4] to help identify and specify concerns and, on the other hand, a refinement of the composition rules by using some new operators inspired in the LOTOS operators [1].

The rest of this paper is organised as follows. Section 2 gives an overview of our model, and discussing its main activities. Section 3 applies the approach to a subway system. Finally, Section 4 draws some conclusions and points out directions for future work.

2. A model for separation of concerns at RE

A generic model to handle separation of concerns during requirements engineering is depicted in Figure 1. It is composed of four main tasks, as: identify concerns, specify concerns, identify crosscutting concerns and compose concerns.

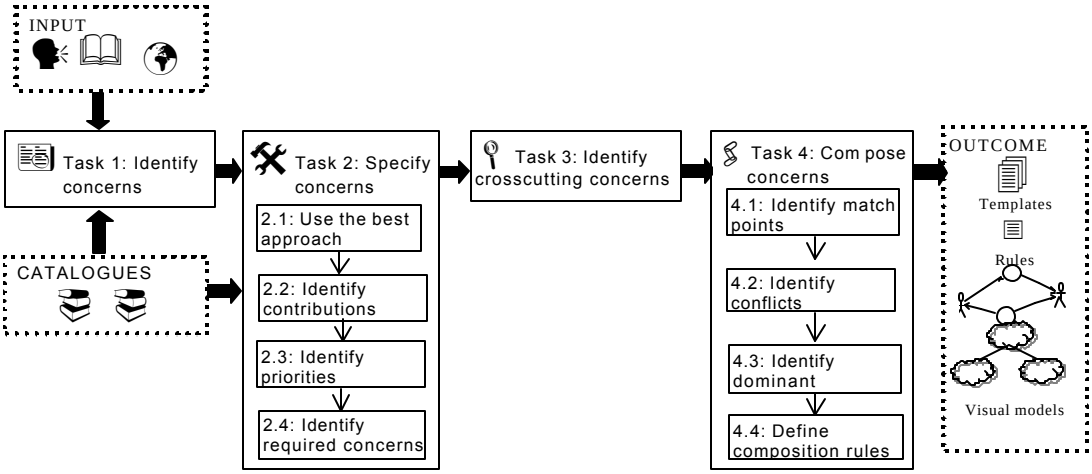


Figure 1: A model for separation of concerns at RE

Despite the direction of the thick black arrows, indicating forward development, an iterative and incremental process should guide the model.

Task1: Identify concerns. A concern is any matter of interest in a system, i.e., it is a goal or a set of properties that the system must satisfy. The identification of concerns is accomplished by undertaking a complete and exhaustive analysis of the documentation and any other information provided by the system' stakeholders. (Stakeholders are all the people that have a direct or indirect influence in the system under study. They may be users, clients, managers, developers, etc.)

We believe that requirements elicitation can be accomplished by using any of the existing techniques. However, so far we have only studied the use of viewpoints [8] and UML [7]. While some techniques emphasize the main functions that the future system should implement (e.g. use cases [10]), others emphasize constraints and certain properties that affect the whole system (e.g. the NFR framework [4]). To alleviate the onus of the identification of the broadly scoped requirements we propose the use of existing catalogues, such as the NFR framework [4].

Each concern should be registered in a template as depicted in Table 1.

Table 1: A template to describe concerns

Name	The name of the concern.
Source	Source of information, e.g. stakeholders, documents, domain, catalogues and business process.
Stakeholders	People that need the concern in order to use the system.
Description	Explains the intended behaviour of the concern.
Classification	Helps the selection of the most appropriate approach to specify the concern. For example: functional, non-functional, goals.
Contribution	Represents how a concern can be affected by other concern. This contribution can be positive (+) or negative (-)
Priority	Expresses the importance of the concern for the stakeholders. It can take the values: *Very Important*, *Important*, *Medium*, *Low* and *Very Low*.
Required concerns	List of concerns needed by this concern.

The outcome of Task 1 is the completion of the rows *Name*, *Source*, *Stakeholders* and *Description*. The remaining rows will be filled during Task 2.

Task 2: Specify concerns. This task is divided into four subtasks: apply the approach that better specifies each concern; identify contributions between concerns so that conflicts can be detected; identify priorities; the list of concerns needed by which concern to accomplish its behaviour. Besides completing the remaining unfilled rows in each template, these subtasks will produce auxiliary documentation generated by the chosen specification techniques (e.g. use cases, viewpoints, interdependency graphs).

Task 2.1: Specify concerns using the best approach. We may use any of the existing techniques for requirements specification. In certain cases a choice may have been done during Task 1, especially if particular techniques have been used to help the elicitation process. If so, we should build any models or other documentation proposed by those techniques. Based on this we can fill the *Classification* row in Table 1.

Task 2.2: Identify contributions between concerns. A contribution relationship between two concerns defines the way in which one concern affects the other. This contribution can be collaborative (or positive) and is

represented by "+", or damage (or negative) and is represented by "-". This information is added to the *Contribution* row in Table 1. It is based on this information that we will be able to detect conflicts between concerns.

Task 2.3: Identify priorities. For each concern we need to identify its degree of importance in the system. This is done with the stakeholders' help. This information is added to row *Priority* in Table 1.

Task 2.4: Identify required concerns. This provides the list of concerns that the concern under study concern needs to accomplish its role. If this concern does not require any other concern the keyword <*none*> is used.

Task 3: **Identify crosscutting concerns**. A concern is crosscutting if it is -required by more than one other concern. This task is accomplished by taking into account the information in rows *Required concerns* and *Descriptions* of the template describing that concern.

Task 4: Compose concerns. The goal of this task is to compose all the concerns so that the developer can get a grasp of the whole system. In this paper we will focus our attention to the composition of those concerns that are crosscutting, as the non-crosscutting ones bring no new problems. To guide the composition, we propose four subtasks: identify match points, identify conflicts, identify the dominant concern, and define the composition rules.

Task 4.1: Identify match points. Based on row *Required concerns* in Table 1 we identify the points where the composition will take place, i.e. the match points. A match point tells us which crosscutting concerns should be composed with a given concern. This can be better illustrated with a bi-dimensional table of Stakeholders X Concerns (see Table 2). Each cell is filled with the list of crosscutting concerns (denoted CC_i) that affect a given concern and represents a match point (denoted MP_i). One composition rule must be defined for each match point.

Table 2: Match points identification

Concerns / **Stakeholders**	$Concern_1$	…	$Concern_m$
$Stakeholder_1$	CC_1, CC_2 **(MP_A)**		CC_2, CC_5 **(MP_J)**
$Stakeholder_2$	CC_1, CC_3 **(MP_B)**		
…	…		
$Stakeholder_n$			CC_3, CC_5 **(MP_Z)**

Task 4.2: Identify conflicts. This subtask supports the identification of conflicting situations between concerns. For a given match point we need to analyse if any of the involved crosscutting concerns contribute negatively to any other (based on the *Contribution* row in Table 1). Concerns contributing positively to other concerns raise no problems.

Task 4.3: Identify dominant concern. This subtask helps solving the conflicts identified in the previous subtask and is based on *Priority* row of the template. If the priority attributed to each concern is different, the problem is not too difficult to solve, and the dominant concern is that with higher priority. However, if at least two crosscutting concerns have the same priority, a trade-off must be negotiated with the user. To guide the negotiation among users we propose the identification of the dominant crosscutting concern for a given match point. Therefore, if two or more crosscutting concerns exist in a given match point, with negative contribution and the same priority, we start by analysing two concerns first to identify the dominant, then take the dominant and analyse it with a third concern and so forth until we have taken into consideration all the crosscutting concerns. The result is the concern with higher priority between them all. Next we need to identify the second dominant crosscutting concern among the remaining concerns, and so on until we have a dependency hierarchy between all the concerns.

The identification of the dominant concern is important to guide the composition rule.

Task 4.4: Define composition rules. The composition rule defines the order in which the concerns will be applied in a particular match point. It takes the form: `<concern> <operator> <concern>`. The operators we propose are inspired on the LOTOS specification language [1]. In the descriptions below C_i denotes i^{th} Concern:

- Enabling (denoted by $C_1 >> C_2$): This is a sequential composition and means that the behaviour of C_2 begins if and only if C_1 terminates successfully.
- Disabling (denoted by $C_1 [> C_2$): This is the disabling composition and means that C2 interrupts the behaviour of C_1 when it starts its own behaviour. This allows the representation of interruptions.
- Pure interleaving (denoted by $C_1 ||| C_2$): This is a parallel operator and means that C_1 evolves separately from C_2. It represents concurrent composition without interaction.
- Full synchronization (denoted by $C_1 \| C_2$): This is another parallel operator and means that the behaviour of C_1 must be synchronized with the behaviour of C_2. It represents concurrent composition with interaction.

A composition rule can be defined based on simpler composition rules, separated by one of the above operators. We will use brackets, "(" and ")", to attribute priorities to the operators.

3. Applying the approach to a case study

The case study we chose is based on the Washington subway system:

"To use the subway, a client has to own a card that must have been credited with some amount of money. A card is bought and credited in special buying machines available in any subway station. A client uses this card in an entering machine to initiate her/his trip. When s/he reaches the destination, the card is used in an exit machine that debits it with an amount that depends on the distance travelled. If the card has not enough credits the gates will not open unless the client adds more money to the card. The client

can ask for a refund of the amount in the card by giving it back to a buying machine."

Task 1: Identify concerns. Based on the requirements given above, we could think that the system has to offer the following concerns: BuyCard, LoadCard, RefundCard, EnterSubway and ExitSubway.

Different types of methods are used to specify functional requirements. Use case driven approaches describe "the ways in which a user uses a system" that is why use case diagram is often used for capturing functional requirements [10]. The concerns listed above are not too difficult to identify, if we think in terms of uses cases, for example. Other concerns can be identified based on the NFR catalogue [4]. For each entry in the catalogue, we must decide whether it would be useful in our system or not. For example, (i) Response Time, as the system needs to react in a short amount of time to avoid delaying passengers; (ii) Accuracy, as only right amounts should be debited from, or credited to a card; (iii) Multi-access, so that several passengers can use the system concurrently; (iv) Availability, as the system and the machines must be available when the subway is open; (v) Security, as card information must be protected against illicit actions.

For each of these concerns we fill rows *Name*, *Source*, *Stakeholders* and *Description* in a template. Here we show only the templates that will help us illustrating concerns that are crosscutting and a situation of possible conflict. The concerns chosen are Enter Subway (Table 3), Response Time (Table 4) and Availability (Table 5).

Task2: Specify concerns. Concerns are specified using the following three subtasks.

Task 2.1: Specify concerns using the best approach. Apply the techniques that best suite the specification of the concerns encountered. In Task 1 we have already made a choice when identifying concerns. Therefore, we should specify the first five concerns using use cases (through scenarios, for example) and the rest of the concerns using Softgoal Interdependency Graphs (SIG) [4]. SIG is a hierarchy graph of softgoals (i. e. non-functional concerns) that shows the interdependencies between them.

Let us considerer the simple situation where only the actor Client is handled. The corresponding use case diagram is illustrated in Figure 2, where the use cases EnterSubway, ExitSubway and LoadCard have been refined to factorize the common functionality ValidateCard.

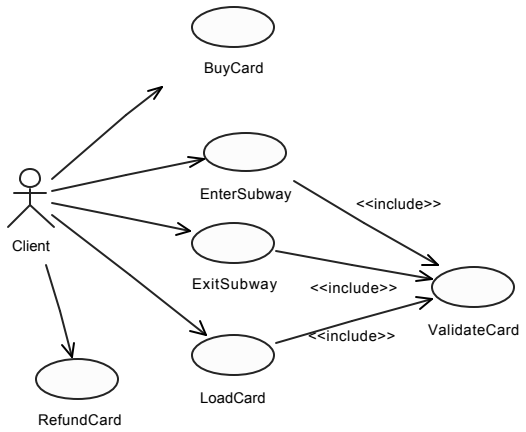


Figure 2. The use case diagram of the Subway System

Nodes of a SIG are (non-functional) concerns, and are represented by clouds, and the lines represent decompositions (of a given concern into its sub-concerns). When all sub-concerns of a given concern are needed to achieve that concern, an AND relationship is defined with an arc connecting the lines (see Figure 3). (A situation where not all the sub-concerns are necessary to achieve the concern, an OR relationship, is defined with two arcs linking the decomposition lines.)

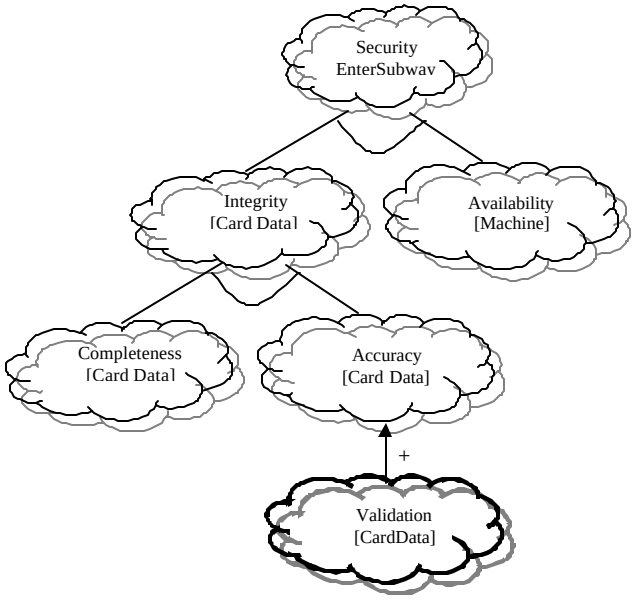


Figure 3: Security specification for EnterSubway

In this case Integrity and Availability are both needed to achieve Security, and therefore an AND relationship is required. Furthermore, Accuracy and Completeness are needed to achieve Integrity. Notice that a subject matter is added under the concern name. A subject matter is the topic addressed by the non-functional concern. This means that in Figure 3, Security is being handled for EnterSubway, Integrity, Completeness and Accuracy must be guaranteed

for Card Data and Availability needs to be guaranteed for Machine. Additionally, Card Data Accuracy may be satisfied with a Validation Card Data procedure, i. e. a possible design solution to satisfy the concern. This is known as operationalization [4] and is represented with a thick dark cloud and an arrow with a positive sign. Other operationalizations can be proposed for the other concerns.

Task 2.2: Identify contributions between concerns. For each concern we must identify its contribution to other concerns and fill the *Contribution* row in the template. Based on the NFR catalogue, we could, for example, say that Availability has a collaborative (+) contribution with Multi-access and a damage (-) contribution with Response Time (See Table 5).

Task 2.3: Identify priorities. In this task we identify the importance of each concern to the system. This information is added to the *Priority* row.

Task 2.4: Identify required concerns. Lists all the other concerns required to accomplish the behaviour of the concern being specified. This information is added to the *Required concerns* row.

At the end of this task, the template of each concern is complete.

Table 3: Enter Subway template

Name	EnterSubway
Source	Stakeholders, set of initial requirements, knowledge of the system
Stakeholders	Client, owner of the system
Description	To initiate the trip the client inserts his/her card in an entering machine. The system checks the card and registers an entrance.
Classification	Functional
Contribution	<none>
Priority	Very Important
Required concerns	Response Time, Accuracy, Security, Availability, ValidateCard.

Table 4: Response Time template

Name	Response Time
Source	Stakeholders, set of initial requirements, NFR catalogue, knowledge of the system
Stakeholders	Client, owner of the system
Description	The system has to react in time in specific situations, for instance to enter the subway, exit the subway.
Classification	Non-functional
Contribution	(-) Security, (-) Multi-access
Priority	Very Important
Required concerns	<none>

Table 5: Availability template.

Name	Availability
Source	Stakeholders, set of initial requirements, NFR catalogue
Stakeholders	Client, owner of the system
Description	All the machines of the system need to be accessible from 6am to 2am. For instance, to enter subway, exit subway, buy card, load card and refund card.
Classification	Non-functional
Contribution	(+) Multi-access, (-) Response Time
Priority	Very Important
Required concerns	<none>

Task 3: Identify crosscutting concerns. Looking at rows *Required concerns* and *Description* we can identify crosscutting concerns. For example, Response Time is crosscutting because it is required in EnterSubway and ExitSubway. ValidateCard, on the other hand, is also crosscutting, as it is required in EnterSubway, LoadCard, and ExitSubway (see Figure 2). Other crosscutting concerns are: Availability, Multi-access, Security and Accuracy.

Task 4: Compose concerns. The goal here is to compose all concerns to obtain the whole system. During this process conflicts may be identified.

Task 4.1: Identify match points. This is accomplished by building the match points table (see Table 6).

Table 6. Identification of match points (**AC**: Accuracy; **MA**: Multi-access; **RT**: Response Time; **S**: Security; **AV**: Availability, **V**:ValidateCard)

Concern / Stakeholder	EnterSubway	BuyCard	ValidateCard	…
Client	RT, S, AC, AV, V **(MP_A)**	S, AC, AV **(MP_B)**	AC	

Task 4.2: Identify conflicts. When the same cell lists more than one crosscutting concern we must verify the contribution between them. For example, in our case study the MP_A match point has five concerns. As we identified in Task 2.2, Availability and Response Time, for example, contribute negatively to each other, i. e. they constrain each other's behaviour, and they may generate a conflict.

Task 4.3: Identify dominant concern. To solve the possible conflicts identified above, we need to verify the priority of each concern involved. Let us only deal with Availability and Response Time. As both have the same priority (Very Important) we need to negotiate a trade-off with the system's stakeholders. Supposing that the priority of Response Time is decreased, Availability becomes the dominant concern in MP_A. However, as Availability is a sub-concern of Security, we need to make clear if Security is also dominant with respect to Response Time. What makes sense in this situation, and because Security is composed of two sub-concerns (Availability and Integrity), is to guarantee Availability first and only at the end guarantee Integrity. On the other hand, Response Time

concern is needed during, i.e. it wraps, Accuracy and EnterSubway concerns.

Task 4.4: Define composition rules. A composition rule for MP_A could be defined using the operators described above as follows:

```
( Security.Availability
  >> ( (ValidateCard >> EnterSubway)
       ||
        ResponseTime
       ||
        Security.Integrity.Accuracy
      ) >> Security.Integrity
) [> ErrorHandling
```

Noticed that this expression should not be looked at as a formal LOTOS behavioural expression, since at this level of abstraction we cannot really talk about the "behaviour" of a concern. Our main goal is basically to give the idea of the order in which the concerns should be satisfied. With this in mind, we could then informally say that after the successfully satisfaction of Availability, Response Time, EnterSubway and Accuracy must synchronize and be satisfied in parallel. Notice that EnterSubway can only be satisfied after the successful satisfaction of ValidateCard. Only after this will Integrity be satisfied. If something goes wrong during the composition process, the system must be able to handle the errors raised. Here we represent this situation by defining the concern ErrorHandling that has the capability to interrupt any of the concerns involved by the disable operator. (Notice that we are using the "." notation to represent a sub-concern of a given concern, e.g. Security.Integrity.)

4. Conclusions and future work

This paper presents a model to handle advanced separation of concerns during requirements engineering. It extends the work presented in [2] and [3] in two ways. First, it proposes the use of catalogues to help in the identification and specification of concerns. The catalogue we have mainly used is the NFR framework. Second, it explores a refinement of the composition rules, by using a set of operators based on the LOTOS major operators.

There are still several problems we need to address in our approach. The major problem we need to investigate is related to the composition rules. LOTOS is a formal specification language with a well-defined semantics. In this paper we started exploring the idea of LOTOS-like operators to specify composition rules. However, at the moment we cannot rigorously talk about a concern's behaviour, in order to then be able to write a valid behavioural expression. To solve this problem we need to (i) work on the decomposition of concerns in terms of responsibilities in order to be able to define composition rules at a finer level of granularity and (ii) study the operationalizations of non-functional concerns so that only non-functional concerns that are mapped onto functions or aspects are used in the composition rules. There will be others that will map onto a design decision, for example, and that will not be considered in a LOTOS-like behavioural expression [8]. The decomposition of concerns will help us study the level of granularity at which a conflicting situation can be handled. Another problem we need to further study is the ErrorHandling concern. We believe it is a necessary, but we know that it will be very difficult to specify.

We are also planning to define a visual integrated notation. Finally, we would like to investigate the possibility of extending a LOTOS tool, Light, for example, to support the validation of the composition rules.

5. References

[1] Brinksma E. (ed): *Information Processing Systems -- Open Systems Interconnection-- LOTOS -- A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour*, ISO 8807, 1988.

[2] Brito, I. and Moreira A. "Advanced Separation of Concerns for Requirements Engineering". VIII Jornadas de Ingeniería del Software e Base de Datos, JISBD 2003. Alicante, España. 2003.

[3] Brito, I. and Moreira A. "Towards a Composition Process for Aspect-Oriented Requirements". Early Aspects Workshop at AOSD Conference. Boston, USA. 2003.

[4] Chung, L., Nixon, B., Yu, E. and Mylopoulos, J. Non-Functional Requirements in Software Engineering, Kluwer Academic Publishers, 2000.

[5] Dijkstra, E.W., *A Discipline of Programming*, Prentice-Hall, 1976

[6] IBM Research, MDSOC: Software Engineering Using Hyperspaces, http://www.research.ibm.com/hyperspace/

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

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

[9] Workshop on Multi-Dimensional Separation of Concerns, International Conference on Software Engineering, ICSE 2000, http://www.research.ibm.com/hyperspace/workshops/icse2000

[10] Schneider, G. and Winters, J., Applying Use Cases – A Practical Guide, Addison-Wesley, 1998.

Tracing Aspects in Goal driven Requirements of Process Control Systems

Islam A. M. El-Maddah and Tom S. E. Maibaum
King's College London
Department of Computer Science
{Elmaddah, Tom}@dcs.kcl.ac.uk

Abstract

Aspects, such as safety, security and productivity play an important role in developing process control systems. These aspects need to be specified at requirements analysis level. Goal driven requirements analysis methods are well suited to trace the different aspects of software applications to the early design level. This paper illustrates how to use the GOPCSD tool in developing aspect-based process control applications. The GOPCSD tool adopts goal driven requirements analysis concepts and has been adapted from the KAOS method to address process control systems. The various aspects of the process control application are modeled using high-level goals within the requirements goal-model. Although these goals are concerned with "soft" issues, the GOPCSD tool guides the user to specialize them to operational/functional sub-goals, assigned to (combinations of) appropriate agents. This can prepare the stage for the consistency and completeness checks offered in the GOPCSD tool to bring the different aspects together. The user will be able to reason about the why of the requirements units, as well as validate the overall operation of the process control application. Finally, the tool will automatically translate the corrected requirements into a B specification, which traces the early requirements aspects to the initial high-level design.

1. Introduction

The software industry has insinuated itself into many applications, reaching a considerable level of maturity in many areas. Consequently, the struggle in the software sector is turning towards the quality of the produced applications. Apart from product operation, issues such as how much safety, usability, security, mobility, and reusability the software products offer are gaining more attention from the client's perspective. These quality aspects make one application preferable to another and, hence, encourage the client to choose it. As noted in [11], more effort should be paid to the quality aspects of requirements for the application. Hence, the software provider needs to pay equal attention to the quality of achieving client's functions and the functions, themselves. This indicates the value of integrating quality attributes and operation of the software applications as early as at the requirements and user needs levels.

Delaying the aspect requirements later to the design and implementation levels, without explicitly demanding them at the requirements analysis level, may result in potential conflicts with the original requirements, as well as operational elements of the design. In particular, this later stage of the development lifecycle may present difficulties in changing/altering the original requirements, which may in turn result in making the client unhappy. In addition, agreeing with the client only on the functional requirements may be likened to contracting one application and developing a different one. In fact, the functional requirements should be regarded as what the system should perform, whereas the quality aspects should affect the manner of this performance. For example, in a lift system, safety aspects can restrict the movement of the cabinet if the door is not closed. Thus, the specification of the functions and the restrictions enforced by qualitative aspects should be better specified at the same level. So, not only should the aspects not die, as in [23], at the implementation stages, destroying traceability, but aspects should also be alive from the beginning, at the user needs and system requirements stage.

In the field of process control systems, quality aspects include safety, usability, security and economic. Neglecting one of these aspects in control applications would result in unsafe, insecure, or very expensive solutions, which will usually be rejected by the client. Although two control applications can perform the same functions, they may still differ in how much safety they offer, how easy it is to use them, or how much power they consume. This can make one of them more successful than the other.

There has been considerable interst in dealing with aspects at the level of programming languages by supplying explicit language constructs to implement aspects: AOP [14], AspectJ [30], PROSE [27], [3], [5], ARCaDe [28], [29]. However, at the requirements and specification level, aspects are represented using appropriate language constructs, or are implicit in specifications of the safety, livens and fairness

properties, as in B [1, 4], VDM [13], SMV [24] RSDS [18] and SCR [12].

In [7], aspects within requirements goals were monitored at runtime; the implementation code had two main parts: one to fulfill the operational goals of the system and the other to monitor the aspects and their requirements. In Planguage (a programming planning language), Gilb in [11] has suggested some quantitative scales to be used to measure the quality aspects.

To represent the aspects in the requirements level, the requirements model has to be able to trace these aspects from the level of client needs down to the early specification level, and then to the implementation level [6]. Moreover, the developed method should provide reasoning about the why of the various parts of the requirements, to validate the aspect-based requirements model as in KAOS [15]. In addition, an aspect-based development should provide an opportunity to evolve the requirements or change them later in response to the results of changing client needs and/or the results of analysis.

One of the requirements approaches that devotes a high level of attention to traceability and understandability is goal driven methods, such as KAOS [15] and TROPOS [25]. In these methods, the requirements are represented in terms of hierarchies of goals called goal-models. Each goal represents an identified requirement, possibly an aspect, like safety, security or function. The main goal of each goal-model specifies the overall application and usually is refined to sub goals that represent different parts, modes, and aspects of the applications. The refinement process ends at terminal level, where each terminal goal is simple enough to be assigned to an agent to carry it out *as an operation*. These operations may be assigned to agents such as software components, humans or hardware devices. Since all higher-level goals are eventually realized as a combination of terminal goals operationalised by (combinations of) agents, higher-level goals representing aspects are transformed to operational goals realized by agents. This transformation of aspects to combinations of operational goals realized by combinations of agents is the key to aspect-based design provided by (some) goal oriented requirements methods like KAOS and GOPCSD.

As a result, the aspects can be traced down from a high-level goal to the terminal goal level, where they can be translated to early design form, as a B specification [1], which is acknowledged as a suitable formal tool for designing reactive systems. This traceability from requirements to early design offered by the goal-models ensures Aspect Oriented Software Development (AOSD) will fulfill its goal of being a software engineering method, as noted in [5].

Although the B method suits the design of process control systems, the formal logic and sophisticated mathematics of its formal language have hindered both its usability and understandability by users, such as process systems engineers, and restricted its use to a small part of the software engineering community. This demonstrates some interference of concerns between the systems engineer, as the client, and the software engineer, as the service provider. More informality should be allowed to the client to be able to express these quality aspects and the requirements method should not depend on deep knowledge of some specific design and implementation method and its possibly alien and unfamiliar formalisms and notations. Hence, some automated mapping should translate the agreed requirements to a formal design model, as noted in [21]. Thus, we were motivated to develop the GOPCSD (Goal Oriented Process Control Systems Design) tool [10] that adapts the goal driven requirements analysis method of KAOS [15, 16], but after some significant adaptations to fit the nature of process control systems.

This paper consists of six sections. In section one, we introduced the research area. In section two, we briefly describe the GOPCSD tool, especially in relation to developing aspect-based requirements. In section three, we illustrate how to create aspect- based goal-models in GOPCSD, using some process control examples. Then, in section four, we show how to bring the different aspects together using the checks and tests offered in the GOPCSD tool. In section five, we use the animation utility offered by the tool to validate the corrected requirements after bringing the aspects together. Finally, in section six, we draw conclusions and suggest future directions for further work.

2. The GOPCSD Tool

The GOPCSD tool [10] has been developed to manage rationally the gap between the process control systems engineer's perspective, as the client, and the formal specification level, supplied normally by a software engineer. To accomplish this; we have developed an integrated requirements development environment, where the process control requirements can be constructed using a provided library, structured in terms of hierarchies of goals, and then checked, corrected, validated and finally automatically translated to a B formal specification. In the following sub sections, we briefly describe the requirements elements to represent process control applications in

the GOPCSD tool and the development phases needed to complete the generation of a formal specification in B corresponding to the corrected requirements.

2.1 Requirements Elements

The goal-oriented requirements of any process control system will be represented within the GOPCSD tool by the following elements:

2.1.1. The Components. In process control systems, components represent the physical parts of the applications, such as valves, robots, and deposit belts. The detailed specifications of each component, including its variables, agents and goal-models, are stored in the GOPCSD library. The systems engineer can create/edit the component details using the GOPCSD library manager.

2.1.2. The Variables. Variables are considered as an essential part of formalizing the user requirements. In the GOPCSD tool, the application's global state is usually described by a set of variables. Each of these variables has one of three types: input, output or intermediate. In the GOPCSD tool, the variables are associated with the high-level goal-model templates or the components, each of which the user can import from the library; however, the tool user can also create, edit, and delete variables from the application design space.

2.1.3. The Agents. Agents are the objects that control the application parts and its local environment. Some of the agents can be part of the application to be built, like software interface programs for hardware parts, or, alternatively, they can be existing programs or hardware devices that will be responsible for accomplishing defined goals to fulfil the overall application operation. Agents can have one of the following three types: device, software and human. The main source of agents is via the user's importing of components from the library. But, if it is required to declare agents apart from those associated with the components, the user can create, edit, and delete them from within the GOPCSD tool environment.

2.1.4. The Goal-models. Goal-models constitute the main segments of the structured requirements; they represent the user requirements as a hierarchy of goals. Each goal-model starts with a main goal that has general scope; this main goal is usually refined to a number of sub-goals describing sub-parts, different aspects, or operation-modes of the application. Each of the goals within the goal model represents a qualitative or operational requirement; the tool supports informal descriptions of goals as well as formal descriptions based on temporal logic [22].

2.2 Development Phases

The GOPCSD tool covers the early development stages. It refines and formalises the abstract user's needs to functional and formal specifications. After the user corrects and validates the requirements, the tool automatically generates a B specification. The tool has three development phases as follows:

2.2.1. Phase One. In this phase, the GOPCSD tool guides the user to construct the requirements and structure them in terms of goal-model hierarchies. The tool offers the user two options to choose from or to integrate in constructing the requirements: to import from the provided library (this library is dynamic in the sense that the user can create components and templates and store them in the library for future use), or to create the requirements entities from scratch using the tool's utilities. After importing the components and templates or creating them, the next stage is to refine the incomplete goals and to combine the separate high level goal-models, representing abstract operational and qualitative goals, with operational goal models corresponding to components, to arrive at a complete model where each terminal goal is assigned to an agent as an operational goal. When the user manages to construct a single goal-model specifying the entire application, the second phase of correct and debugging the goal-model can start.

2.2.2. Phase Two. This phase checks the completeness and consistency of the requirements. The tool provides various checks to remove the major part of the requirements bugs. In addition, an animation utility is offered to validate the requirements by executing the goal-model corresponding to the application. Thus, the application performance can be portrayed for the client before generating formal specifications and implementations. The tool enables the user to stimulate the system by changing the values of input variables and mimicking faults and delays in the different components.

2.2.3. Phase Three. This is the third and final phase of generating formal specifications. After the goal-model has been checked and modified, the tool can

automatically generate general "use case"-like operations that each has an associated pre- and post-condition and a responsible actor (one of the application's agents) to perform the operation. The GOPCSD tool also generates a B formal specification. The generated B machines will be documented using the informal description of the requirements elements supplied by the systems engineer.

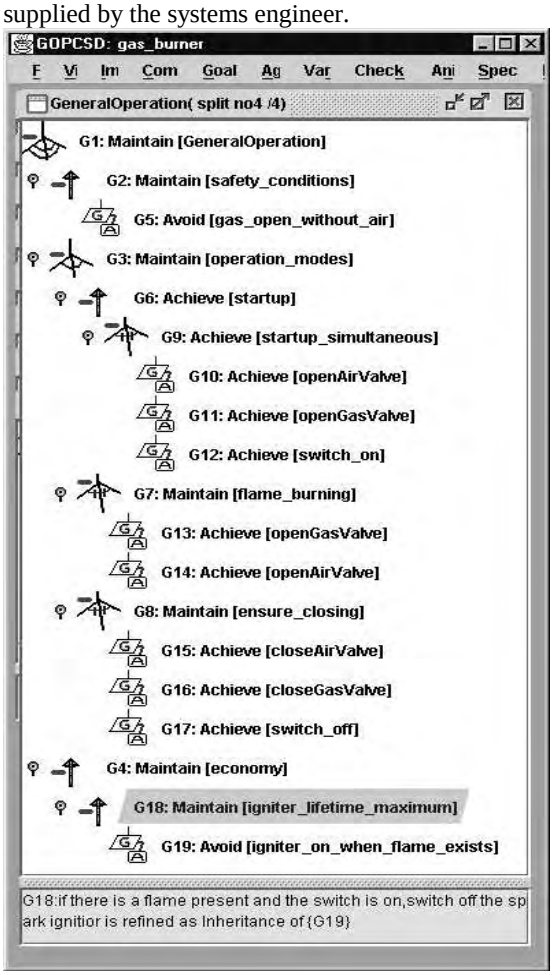


Fig. 1, *the goal model of the gas burner system*

2.3 The GOPCSD Tool Support for AOSD

The various aspects of the process control application requirements will be represented as high-level conjunctive goals. Aspect keywords, such as safety, productivity, etc., will be written explicitly within the informal description of the high-level goals. The sub goals of one aspectual goal will inherit the aspect from their parent goal. The non-aspectual high-level goals will be concerned with the normal operation of the system, and will usually be grouped under one high-level goal (see goals G3 in fig. 1 and G5 in fig. 3). Each high-level aspectual goal will be refined to a level in which the goal representing the aspect can be formulated as a combination of operations that demands an operational scenario to be undertaken by a combination of agents.

After completing the construction of the goal-model, the different aspects of the requirements will be brought together through the consistency and completeness checks. The tool provides utilities for reasoning about why and how for the different goals, including aspect related ones, of the goal-model. Thus, an early correction path is achieved as well as an informal means for better understanding of the constructed requirements model.

To validate the constructed requirements model, the GOPCSD tool provides an animation utility, which can reason about the why of taking the current actions in terms of aspect goals, cycle by cycle, during the symbolic execution. Finally, the tool automatically generates a B specification corresponding to the corrected requirements, which is documented (using the informal description segments of the goals) to be able to trace the aspects down to the implementation level.

3. Creating the Aspect-based Requirements

To achieve the single requirements model, which is the aim of the GOPCSD method, from the multiple aspect requirements, the construction of the requirements specifications may use one of two variations. In the first one, the specification has to consider the different aspects all at once. This (eventually) entails that the defined operations involving pre- and post-conditions must combine to affect the different aspects. For example, in a production cell, the robot arms should be moved in a safe way, while also ensuring that the service time for each metal blank will be as small as possible.

In another possible variation, a separate requirements view concerning each aspect can be constructed; and, then, the requirements from the different views can be examined to ensure they do not prescribe any incomplete nor inconsistent behavior, ensuring the views are integrated. For example, regarding the robot arm movement, there will be different requirements from the point of view of safety, productivity and operation. Then, using conflict and completeness analyses, some situations will be highlighted, where goals relating to different aspects will compete. Thus, by modifying the various aspect related goals, after the problems pointed out by

consistency and completeness checks, the requirements model will automatically adapt itself to weave the various aspects together.

The second approach has the promise of less effort being expended for the design of each aspect view (due to simpler models of each aspect), as well as similar effort being required to test the completeness and consistency of the requirements overall. Hence, as noted in [2] to decrease the effort required to put the aspects together, we adopt this second approach to group the requirements by aspects at the high-levels within the goal-model. The requirements model will be then elicited by refining the initial user needs into detailed system requirements.

In one case study, we examined a gas burner system, as studied before in [17]. The burner system has safety requirements (to control the gas concentration in the area around the burner to avoid explosion) and economy requirements (to increase the lifetime of the igniter). The GOPCSD tool guides the user to elicit the requirements in a goal-model, which has three main goals G2, G3 and G4 concerning safety, operation and economy, respectively, as shown in fig. 1. Each of these goals has been refined to sub-goals concerning local parts or stages of the burner operation. Since the economy is considered as a soft goal (difficult to be formalized or measured), we specialize the economy goal (G4) to a sub-goal (G18), which maximizes the igniter lifetime. Similarly, because it is difficult to formalize and represent all factors affecting the lifetime of the igniter, we specialize goal G18 to G19, where we formulate a goal of avoiding switching the igniter on if the flame already exists. What can be done to effectively manage the lifetime of a physical component is exactly the kind of knowledge that we would expect a process systems engineer specialized in burner systems to possess.

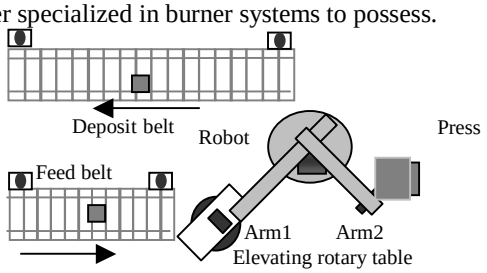


Fig. 2, *the production cell*

For safety requirements, which are again considered as a soft goal (G2), we specialize to an instance of the safety goal that concerns itself with the gas concentration around the burner (G5). In goal G5, it was then easy to formalize the requirement about the concentration of the gas being controlled by avoiding opening of the gas valve while the air valve is closed. Again, this is likely to be the kind of specialist knowledge possessed by the process systems engineer. Thus, we were able to formalize some issues related to soft concepts/"non-functional requirements" by using the specialization sub-goaling mechanism of GOPCSD. All other sub-goaling mechanisms in GOPCSD require a logical equivalence between the goal and its set of sub-goals.

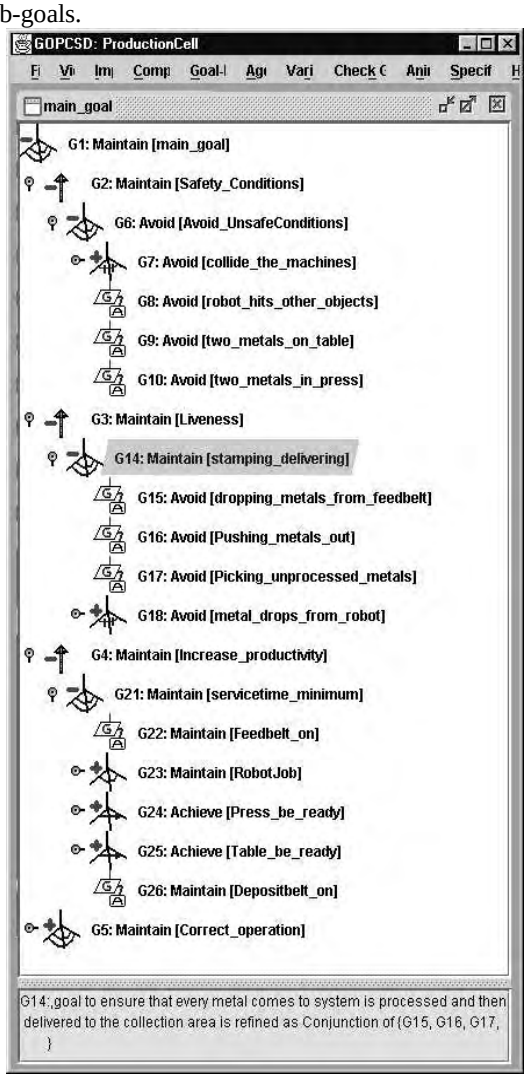


Fig. 3, *the refined goal-model of the production cell*

In another case study, we examined a production cell [20], as shown in fig. 2. The production cell has a safety goal (to avoid collision between the different machine parts), a productivity goal (to increase the throughput of the system) and a liveness goal (to insure every blank metal coming through the feed belt will be stamped and delivered to the deposit belt). We model each of these aspects by one high-level goal, as shown

in fig.3. Goals G2, G3, G4 and G5 represent the safety, liveness, productivity (throughput) and operational goals, respectively. Each of these goals is refined, separately, as shown in fig. 3.

Although maintaining safety conditions for the production cell is considered as a soft goal that has many factors, some of which are difficult to determine and/or control, we specialize the sub goal G6 (concerned with the avoidance of dangerous and controllable situations) from the safety goal G2. Goal G6 is designed to avoid some predicted circumstances that may arise during run-time and are detectable and avoidable by the control system. Again, the knowledge of the process systems engineer is crucial in identifying the right specialization to apply. Hence, goal G6 is refined as a number of conjunctive goals that are designed to avoid the machines colliding with each other (G7), avoid hitting any objects by the robot arms (G8), avoid having two metals on the table tray at the same time (G9) or inside the press at the same time (G10). Goal G7, avoiding collision between the different machines can be refined as -goal G11-avoiding the table hitting robot arm1, and -goal G12-avoiding the press hitting robot arm1 and -goal G13-arm2. Each of these sub-goals is tracing the safety aspect and can be formally specified and assigned to an agent, such as a robot arm or a table motor, as a set of operational goals.

Like safety, the productivity issue has a soft reprehensive goal G4. The productivity of the cell is affected by various factors. Some of these factors are unmanageable, whereas others are uncontrollable by the designed application. Thus, we create a more rigid/harder goal (G21) that resembles the accessible factors the control application can operationally perform to decrease the service-time for each individual blank metal. So, we specialize G4 into an instance G21 concerning with the reduction of the service time as a functional factor to increase the productivity. Goal G21 can be refined in terms of sub goals reducing the service time in the local parts of the production cell. A possible refinement could be as follows: G22, maintain a continuous stream of incoming blank metals; G23, keep the robot ready to serve either the press or the table; G24, keep the table ready to receive new blank metals; G25, keep the press ready to process a new metal; and G26, keep delivering the processed metals. Although these sub-goals are tracing a non-functional aspect, each of them is considered on its own as a high-level functional goal that can be refined systemically into operational sub-goals.

Similarly, the liveness goal G3 can be refined into operational goals, as can the operational goal G5.

Note that again we have used a combination of specialization of operational/non functional goals to functional ones, and the usual refinement of operational goals, to achieve the completion of the overall goal model for the application

Designing each aspect view separately, as in [2], can be considered an advantage because it is much easier to concentrate on the different views separately rather than a single combined view. Furthermore, allowing different views for each aspect encourages a group of systems engineers to develop the requirements model in parallel. In addition, the aspect goals and sub goal-models can then be reused from similar applications and/or previous versions of the same application.

4. Putting the Various Aspects together

Bringing the different aspects together could be achieved dynamically at run-time or in advance. As in [27], to dynamically weave the different aspects, the aspects should be traced into the executable segments of the program. Thereafter at runtime, the segment that traces the most important aspect to fulfill amongst the activated segments should be executed. We could equally apply this approach to goal driven requirements by ranking the goals, according to the aspects, which they are tracing.

An alternative direction to put the aspects together instead at runtime entails some analysis to be carried out to put the requirements composed from the different aspects together before the final implementation stage. This analysis should examine the various situations that can occur during run-time.

Although it is difficult to examine all possibilities that may occur during run-time, especially at requirements stages, the second approach can still be considered better than the first, from the validation point of view, since the decisions about what to do when requirements of different aspects conflict were validated at the requirements stage and not left for resolution when the run-time system was being created.

In particular, this latter approach should be the right one if the aspects do not have absolute rankings over each other for all run-time configurations, as noted in [29]. For example, in some circumstances productivity decision should come before operational ones, while in other circumstances the opposite situation may occur.

Thus, we adopt the idea of performing some analysis at the requirements stage to predict the

situations when the different aspects can undermine each other. The GOPCSD tool allows the user to perform consistency (goal-conflict analysis) and completeness checks for this purpose.

Conflict happens within goal driven requirements when different goals are simultaneously fired and attempt to control the same parts of the components of the system. To be able to remove such cases, the requirements need to undergo consistency checks to ensure such conflicts will not happen during run-time. The GOPCSD tool provides consistency checking [8] based on testing the various runtime configurations, to detect whether the active goals under each of these configurations prescribe inconsistent behaviors, as in [16].

For example, having performed consistency check on the complete goal-model of the production cell (shown in fig. 3), it shows that goals G40 and G11 are conflicting. Goal G11 restricts the rotation of the robot if its first arm is extended until the rotary table moves down to avoid hitting it, whereas goal G40 (grandchild goal of goal G23, to increase the productivity of the cell by moving the robot if it is free to wait for the next expected job) prescribes the rotation of the Robot towards the press to be ready to pick up the metal, which is being pressed at that moment. Knowing that goal G40 has a productivity aspect (increasing the throughput) and goal G11 has a safety aspect, we should modify the pre-condition of the productivity goal G40 to make sure the two goals will not be active at the same time. The pre-condition of goal G40 can be modified to be mutually exclusive with respect to the pre-condition of goal G11. Thus, goal G40 will be restricted to be active only when the robot arms are retracted. Thus, one of the safety aspect goals (G11) affected the way the productivity goals work. Similarly, other conflicting goals could be changed, paying attention to the importance of each aspect for each individual conflict.

The incompleteness and indeterminism of the requirements may lead to situations where the system prescribes strange or unplanned run-time behaviors. For example, the control system may toggle between two states (or possibly more) attempting to address different aspects. Although the control system actions are consistent, they might exhibit strange behaviors because of the indeterminism and incompleteness concealed within the unchecked requirements. Hence, the GOPCSD tool offers a completeness check based on the completeness of software requirements specification (SRS) [6], which discovers the cases where the requirements prescribe indeterminism or absence of control. Consequently, the tool provides suggestions for the user to resolve such cases.

Another example to show how completeness analysis can be employed to merge the various aspects is provided, as follows. According to normal requirements specifications of the gas burner system, if the gas valve is open, it closes whenever the air valve is closed to maintain safety conditions, goal G5. On the other hand, when the user switches the burner on, the gas valve opens (goal G11: start the system up when requested) and the control system should ensure the two valves are open to maintain the flame burning (G13 keeps the gas valve open to maintain the flame burning). These three goals can prescribe a behavior of indeterminacy, when air valve is closed and the switch is open. *(switch_switch_state = ON and air_valve_state = CLOSED).* This results in a state, which toggles between opening and closing the gas valve. It is essential to remove such a situation and similar situations. The solution is to modify the pre-conditions of these goals to make sure they are disjoint, and then the behavior will be deterministic. This may be achieved by enabling the safety goal G5 to close the valve only when the switch is off. This is probably unacceptable because the unsafe situation can occur during starting up, when the switch is on. The other choice, which is more logical, is that the gas valve will not be open unless the air valve is already open, i.e. to restrict goals G11 and G13 by adding *(air_valve_state=OPEN)* in their pre-conditions. Having changed the pre-conditions of goals G11 and G13, the safety and operational goals are merged together, without conflicts. (Of course, having changed the goal-model, conflict and completeness checks should again be applied.)

The systems engineer will almost certainly be in a better position to take responsibility for making these decisions before the GOPCSD tool generates B formal specifications and leaving the stage prepared for a software engineer to get involved in developing the control application, within the B toolkit [4] or other similar environments.

5. Validating the Requirements

An important step is to validate the requirements, especially after modifying them to combine the different aspects. The GOPCSD tool provides an animation utility, which symbolically executes the goal-model. The traces of execution cycle by cycle contain the applied events (changes in the input variables) and the list of the activated goals within the goal-model, which trace the different aspects from the high-level

abstract goals to the terminal functional goals. Thus, the user can be acquainted with the current state of the system and reason about the why for system actions. Because each goal is tracing one aspect, the user can be familiar with how the various aspects will work together to fulfill user needs. This increases the understandability of the requirements model. The generated formal specification usually does not achieve such a high level of understandability (a main drawback of formal methods), at least by the normal systems engineer.

In addition, the animation utility allows the user to change the system state and observe the activated goals. This can be effectively used to check whether and how the process control system can recover from unsafe/insecure states. Serving as an early conceptual prototype, the animation utility can guide the systems engineer to fine tune and enhance the requirements.

6. Conclusions

The requirements stage of software development is acknowledged to be the main factor that defines the relative success of software application projects [26, 31]. Thus, the quality aspects should certainly be represented in the requirements stage. This motivated us to design the GOPCSD tool to trace the client's needs related to aspects down to the formal specification level.

The GOPCSD tool adopts the main concept of goal driven requirements analysis of KAOS. Within GOPCSD, the user can create aspect requirements, separately as sub goal-models, and then to combine them using completeness and consistency checks. The corrected requirements will be automatically translated into a B specification by the tool.

The GOPCSD tool does not restrict the user to begin the development lifecycle with a perfect requirements design. It guides its user to reach this stage after a feedback loop of checking and enhancing the requirements. This issue can be clearer in medium- and large- scale applications, where constructing a complete and consistent requirements model at the beginning is usually difficult to achieve.

The GOPCSD tool enables the user to express the productivity (throughput), liveness, safety, economy and operational aspects as conjunctive high-level goals. The requirements completeness and consistency checks can be employed to combine the different aspects in the same manner they are used to remove the requirements bugs. This will not entail more effort on the user side. The user may possibly have an easier and simpler view of each aspect rather than a single view that combines the aspects. The tool guides the user to develop the application faster in this respect, as recommended in [2]; it provides the integration of the different views using the conflict, completeness and, possibly, the animation utlities. This reduces the effort required by the systems engineer as well as not requiring a high-level of expertise in the process control field.

The way the different aspects are separated makes it easier for the user to decide which goal to weaken or strengthen when resolving conlficts or incompleteness; in addition, it helps to increase the modifabilty, traceability, and augmentability [6] of the entire requirements model of the process control application.

Further research should be directed towards evaluating the various aspects while symbolically executing the requirements, as noted by defining quality metrics in [11] and incorporating them via the animation utility. This can guide the client/implementer to have an early measure for how much each aspect is achieved in particular requirements specification solutions.

Another direction for future work is to see how we can remove the discovered conflict between the aspect related goals by prioritizing them, as noted in [16]. When more than one goal is fired simultaneously, the one with highest priority will be allowed to perform its action. This direction reduces the effort required by the user to modify the pre-conditions. However, further effort may be directed towards alerting the user to what will happen during run-time and also enabling him/her to overwrite the default priority system in some configurations. Thus, this can maintain the aspect sub goal-model ready to be reused in other versions of the application.

References

1 J. R. Abrial, "The B Book: Assigning Programs to Meaning", Cambridge University Press, 1995.

2 M. Aksit, Separation and Composition of Concerns in the Object-Oriented Model, ACM Computing Surveys, volume 28, p 148, 1996 Position paper for the ECOOP `96 adaptability in OO software development workshop, 1996

3 M. Aksit and B. Tekinerdogan, Aspects-oriented Programming using Composition-Filters, in Object-Oriented Technology, S. Demeyer and J. Bosch (Eds.), ECOOP' 98 Workshop Reader, Springer Verlag, pp. 435, 1999.

4 B Core UK limited (1998), B Toolkit http://www.b-core.com/btoolkit.html

5 S. Clarke and R. Walker, Towards a Standard Design Language for AOSD, Proceedings of the 1st

international conference on Aspect-oriented software development, 2001
6 Alan M. Davis, Software Requirements: Objects, Functions and States, Prentice Hall PTR, 1993
7 A. Dingwall-Smith and A. Finkelstein, "Monitoring Goals with Aspects," University College London, Dept. of Computer Science August 2003.
8 S. Easterbrook, “Resolving Requirement Conflicts with Computer-Supported Negotiation”, In requirement engineering: social and technical issues, M. Jirotka and J. Goguen (Eds.) Academic Press, 1994, pp 41-65.
9 I. A. El-Maddah and T. S. E. Maibaum, Goal-oriented Requirements Analysis of Process Control Systems, Proc. First ACM & IEEE International confernce on Formal Methods and Models for codesign (MemoCode), France, 2003
10 I. A. El-Maddah and T. S. E. Maibaum, Goal-oriented Process Control System Design Tool, Tool Exhibition of FM03, Italy, 2003
11 T. Gilb, Competitive Engineering, a Handbook for Systems and Software Engineering Management using Planguage, 2003, Addison-Wesley
12 C. Heitmeyer, J. Kirby, Jr., B. Labaw, and R. Bharadwaj, SCR*: A toolset for specifying and analysing software requirements. In Proc. Computer-Aided Verification, 10th Annual Conf. (CAV' 98), Vancouver, Canada, 1998
13 C. B. Jones, “Systematic Software Development using VDM”, Englewood Cliffs, NJ: Prentice-Hall, 1990.
14 G. Kiczales, J. Lamping, A. Menhdhekar, Chris Maeda, C. Lopes, Jean-Marc Loingtier and J. Irwin, Aspect-Oriented Programming, Proceedings European Conference on Object-Oriented Programming, volume 1241, Springer-Verlag, pp 220-42, 1997
15 A. van Lamsweerde, A. Dardenne, B. Delcourt, and F. Dubisy, “The KAOS Project: Knowledge acquisition in automated specifications of software”, proceeding AAAI Spring Symposium series, Track: “Design of composite systems”, Stanford University, March 1991, pp 59-62.
16 A. van Lamsweerde, R. Darimont and E. Letier “Managing Conflicts in Goal-Driven Requirement Engineering”, IEEE Transactions on Software Engineering, Special Issue on Managing Inconsistency in Software Development, Nov. 1998.
17 K. Lano and A. Sanchez, Design of Real-time Control Systems for event driven Operations, Formal Method Europe, LNCS vol. 1313, Springer-Verlage, Berlin, Germany, 1997
18 K. Lano, K. Androutsopoulos, and D. Clark, Structuring and Design of Reactive Systems using RSDS and B, FASE, ETAPS 2000
19 N. G. Leveson, M. P.E. Heimdahl, H. Hildreth, and J. Reese. Requirements Specification for Process Control Systems. IEEE Transactions on Software Engineering, Vol. SE-20, No. 9, pp. 684-707 (September 1994).
20 C. Lewerentz and T. Lindner, Case Study “Production Cell” a comparative study in formal software development, FZI Karlsruhe, 1995
21 T. S. Maibaum (2000) ”Mathematical Foundations of Software Engineering: a Roadmap”, future of Software engineering, Limerick, Ireland
22 Z. Manna and A. Pnueli, “The Temporal Logic of Reactive and Concurrent systems”, Springer-Verlag, 1992.
23 F. Maththijs, W. Joosen, B. Vanhaute, B. Robben and P. Verbaeten, Aspects should not die, position paper at the ECOOP'97 workshop on Aspect-Oriented Programming, 1997
24 K. L. McMillan Symbolic Model Checking: An approach to State Explosion Problem, Kluwer Academic, 1993
25 J. Mylopoulos and J. Castro, Tropos: A Framework for Requirements-Driven Software Development In J. Brinkkemper and A. Solvberg (eds.), Information Systems Engineering: State of the Art and Research Themes, Lecture Notes in Computer Science, Springer-Verlag, p. 261-273, June 2000
26 B. Nuseibeh and S. Easterbrook, Requirements Engineering: a Roadmap, Future of Software Engineering, Limerick, Ireland, 2000
27 A. Popovici and T. Gross and G. Alonso, "Dynamic weaving for aspect oriented programming", Proceedings of the 1st International Conference on Aspect-Oriented Software Development, 2002
28 A. Rashid, A. Moreira, and J. Araujo, Modularisation and Composition of Aspectual Requirements. 2nd International Conference on Aspect-Oriented Software Development. ACM, 2003, pp 11-20
29 A. Rashid, P. Sawyer, A. Moreira and J. Araujo, Early Aspects: A Model for Aspect-Oriented Requirements Engineering. IEEE Joint International Conference on Requirements Engineering. IEEE Computer Society Press, 2002, pp 199-202
30 Xerox Corporation. The AspectJ Programming Guide. Online documentation, 2001, http://www.aspectj.org/
31 P. Zave and M. Jackson, “Four Dark Corners of Requirements Engineering“, ACM Trans. Software Eng. And Methodology, 6(1). 1997. p 85

Generating Aspect-Oriented Agent Architectures

Uirá Kulesza Alessandro Garcia Carlos Lucena

Software Engineering Laboratory – SoC+Agents Group
Computer Science Department
Pontifícal Catholic University of Rio de Janeiro - PUC-Rio - Brazil
{uira, afgarcia, lucena}@inf.puc-rio.br

Abstract

Multi-agent systems (MASs) encompass multiple features which tend to be scattered in the software artifacts produced through the system modeling, architecture specification and implementation. This paper presents the definition of a generative approach to support uniformly the modularization of MAS features since early development stages. The proposed approach explores the MAS domain to enable the code generation of agent architectures. We have defined a domain specific language (DSL) that permits us to model orthogonal and crosscutting agent features, such as the agent knowledge, autonomy, interaction, adaptation, learning, and roles. The agent features are then expressed in an aspect-oriented architecture. The implementation of the generative approach encompasses: (i) XML technologies to specify the DSL; (ii) Java and AspectJ programming languages to implement a concrete version of our aspect-oriented agent architecture; and (iii) a code generator, implemented as an Eclipse plugin, which maps abstractions in the DSL to specific components and aspects of the agent architecture.

1. Introduction

Multi-agent systems (MASs) encompass multiple features which tend to be scattered in the software artifacts produced through the system modeling, architecture specification and implementation. With MASs growing in size and complexity, the explicit separation of agent features are still deep concerns to MAS engineers [10, 17, 27]. MASs can be seen as a specific domain that can be explored to improve our capacity to develop that kind of systems. In fact, in last years many MASs development approaches [10, 17, 27] have already explored MAS domain to create facilities to develop these systems. However, these approaches have focused on orthogonal features, such as, agent types, goals, beliefs, and plans.

Generative Programming [6] has been proposed recently as an approach based on domain engineering. It addresses the study and definition of methods and tools to enable the automatic production of software from a high-level specification. This paper presents the development of a generative approach that explores features of the MAS domain to enable the code generation of agent architectures. The proposed approach supports the modeling and implementation of orthogonal (non-crosscutting) and crosscutting agent features since preliminary development phases in a uniform way.

Our generative approach for MASs defines a domain-specific language, called Agent-DSL, in the problem space. Agent-DSL permits us to model agent features, such as, knowledge, interaction, adaptation, autonomy and roles. In the solution space, we have specified an aspect-oriented agent architecture. It is composed of interacting *aspectual components*, which are the modularity units to design crosscutting features in the architectural definition of an agent.

We have used different technologies to accomplish our generative approach for MASs. First, in the problem space, feature diagrams were initially specified to establish the relevant concepts and features encountered in the definition of an agent. Thereafter, we used this information to represent the elements of the Agent-DSL using XML-Schema [28]. Second, in the solution space, we designed the agent architecture using aspect-oriented abstractions that model each of the features encountered in Agent-DSL. After that, we have used Java and AspectJ programming languages to implement specific object-oriented components and aspectual components of the architecture.

In addition, we have defined Eclipse Modeling Framework (EMF) code templates [3] that are used to generate components and aspects to a specific agent.

Finally, in the configuration knowledge of the generative approach, a code generator has been implemented as an Eclipse plugin. It is responsible for mapping abstractions in the Agent-DSL to components and aspects of the agent architecture.

The remainder of this paper is organized as follows. Section 2 describes the process of domain analysis and design for the development of the generative approach. Section 3 presents the domain implementation to accomplish our generative approach for MASs. Section 4 discusses some lessons learned and presents our conclusions. Section 5 points to directions for future work.

2. Defining a Generative Approach for MASs

Multi-Agent Systems (MASs) are a horizontal domain that involves a number of orthogonal and crosscutting features. An orthogonal (non-crosscutting) feature is easily represented by a single modular abstraction through different development phases. Crosscutting features naturally cut across different modular abstractions in the software artifacts produced in distinct stages of the software lifecycle. Some examples of crosscutting agent features are interaction, autonomy, adaptation, learning and collaboration [11, 12, 13, 23].

The MAS domain can be explored to improve the quality and productivity on the development of these systems. In this context, the purpose of our generative approach is threefold: (i) support uniformly the crosscutting and non-crosscutting features of software agents since early development stages, (ii) abstract the common and variable features, and (iii) to enable the code generation of aspect-oriented agent architectures.

Figure 1 depicts our generative approach that is composed of:

(i) a domain-specific language (DSL), called Agent-DSL, used to collect and model both orthogonal and crosscutting features of software agents;

(ii) an aspect-oriented architecture designed to model a family of software agents. This architecture is centered on the definition of *aspectual components* to modularize the crosscutting agent features at the architectural level of abstraction;

(iii) a code generator that maps abstractions of the Agent-DSL to specific compositions of objects and aspects in the agent architecture.

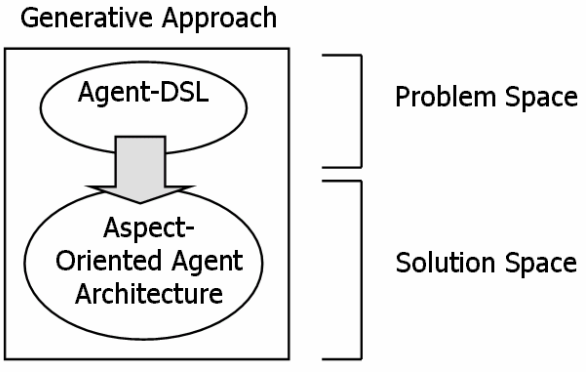


Figure 1. Generative Approach for MASs

The definition of our generative approach involved two initial phases that were concerned with the gathering of a deep understanding of the MAS domain: (i) the domain analysis, and (ii) the domain design. Following the initial phases, the implementation of the generative approach involved three additional phases: (iii) the implementation of our domain specific language, (iv) the implementation of the aspect-oriented agent architecture, and (5) the implementation of the code generator. Sections 2.1 and 2.2 present respectively the initial phases, and Section 3 presents the implementation phases.

2.1. Domain Analysis

The domain analysis was supported by our extensive work on the development of several multi-agent systems [11, 12, 13, 14], and on a survey of different modeling languages, MAS architectures and platforms [26]. To proceed with the definition of the Agent-DSL, we initially captured and analysed the different features associated with the agent definition [10, 11, 13, 17, 26, 27]. We also investigated possible relationships between these features. Feature models [20] were used to represent the features and their respective relations. This section presents a subset of the feature models produced.

Figure 2 depicts a feature model which defines the essential features associated with the *agent* concept. It is composed of its *knowledge* and its basic properties, which we termed "*agenthood*". The knowledge feature encompasses *beliefs*, *goals* and *plans*. Agent beliefs describe information about the agent itself and about the external environment with which the agent interacts. To achieve a goal, an agent executes a specific plan. During the execution of a plan, the agent manipulates its beliefs. The agenthood feature is composed of three subfeatures: *interaction*, *adaptation*, and *autonomy*. Figure 2 presents the feature diagram of the agent knowledge and agenthood.

Figure 2. The Knowledge and Agenthood Features

The interaction feature (Figure 3) is the agent capacity to communicate with the environment. The agent can receive or send *messages* to the environment by means of its *sensors* and *effectors*, respectively. External messages are translated to the agent ontology using specific *parsers* in its sensors. Effector parsers translate internal messages to a specific external representation, such as FIPA ACL [9].

Figure 3. The Interaction Feature

The adaptation feature, depicted in Figure 4, is formed by *belief adaptation* and *plan adaptation*. Belief adaptation is responsible for interpreting received messages from the environment and manipulating its beliefs based on the message contexts. Plan adaptation determines the plan the agent must execute whenever a new goal needs to be achieved.

Figure 5 presents the main features associated with the autonomy property. The purpose of the agent autonomy is to instantiate and manage the agent goals. It deals with three types of goals: *reactive goals*, *proactive goals*, and *decision goals*. Reactive goals are instantiated when the agent receives an external request from other agents or environment components. Proactive goals are instantiated due to internal events that occurs, such as, the end of a plan execution or the achievement of a specific agent state. Finally, the decision goals are instantiated due to external or internal events and are used to decide if special reactive or proactive goals could be instantiated. The autonomy property is also responsible for monitoring the adopted *concurrency strategy*. It supports the goal achievement by implementing a mechanism for executing concurrently agent plans.

Figure 4. The Adaptation Feature

Figure 5. The Autonomy Feature

In addition to the agent knowledge and the agenthood features, an agent can incorporate additional properties. Additional features include *roles*, *mobility*, and *learning*. The initial version of our Agent-DSL provides support for the role feature. A role gives to the agent extra capacities of knowledge, interaction, adaptation and autonomy. Each agent can play different roles during its execution. A role is played by the agent

in a specific context, for instance, the need to collaborate with other agents.

Following the feature definition, we analyzed each of the agent features in order to find out which of them had orthogonal or crosscutting nature. The first step of our analysis was to capture additional relationships which were not present in our initial feature model. The crosscutting features were identified based on the degree in which a given feature was related to other features not present in the same feature hierarchy. For example, the autonomy feature is related to the interaction feature and the knowledge feature. Orthogonal features included the knowledge elements, i.e. agent types, goals, plans, and beliefs. Crosscutting features included the agenthood features and the additional properties.

2.2. Domain Design

In the domain design, we have worked on the definition of an aspect-oriented agent architecture that considers the orthogonal and crosscutting features of software agents (Section 2.1). The aspect-oriented agent architecture is a refinement of a previous work [11, 13]. It is composed of two kinds of components: (i) a central component that modularizes the orthogonal features associated with the agent knowledge, and (ii) the *aspectual components* that separate the crosscutting agent features from each other and from the knowledge component.

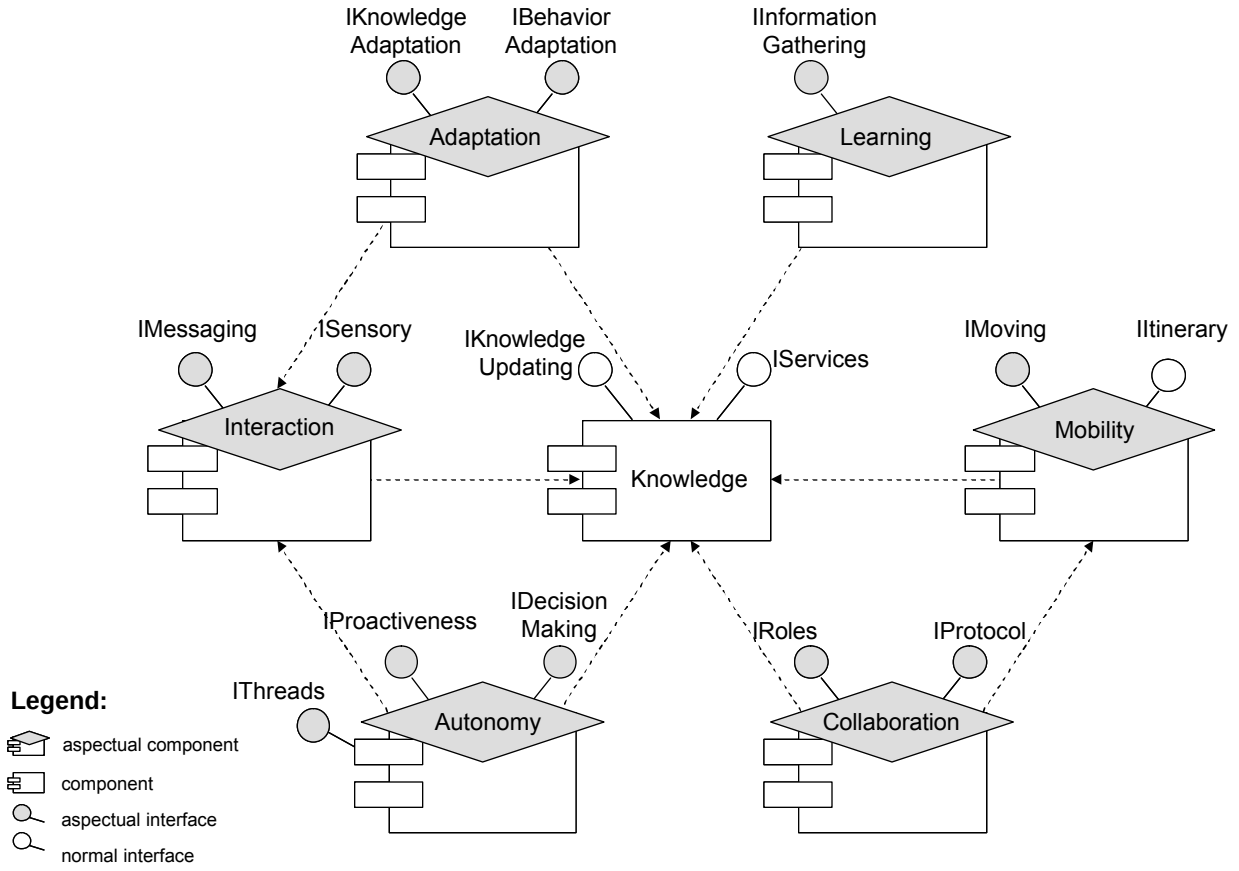


Figure 6. The Aspect-Oriented Agent Architecture

Figure 6 illustrates the architectural components and their relationships. We have used the aSide modeling language to represent the software architecture [5]. Each component has one or more interfaces. They have two kinds of interfaces: *normal interfaces* (colored in white) and *crosscutting interfaces* (colored in grey). The normal interfaces provide services to the other components. The crosscutting interfaces specify how an aspectual component crosscut other architectural components.

Note that each of the aspectual components is related to more than one architectural component, representing their crosscuting nature. The knowledge component contains only the orthogonal knowledge features. The knowledge component is refined as a set

of classes, representing the agent feature and its knowledge features (beliefs, goals, plans). To define specific agents, we model subclasses of the Agent class, as well as, we implement new classes to represent the belief, goals and plans of these agents.

Each of the agenthood features and additional agent properties are modeled as aspectual components due to their crosscutting nature. An aspectual component is refined as a set of aspects and auxiliary classes, which are also part of the crosscutting feature. The aspectual components crosscut elements of the Knowledge component and other aspectual components. For instance, the Interaction component introduces into the Knowledge component some operations to receive or send messages (IMessaging Interface), and pointcuts to sense events from environment objects (ISensory Interface).

The Autonomy component crosscuts the Interaction component because it needs to create goals according to messages received from the external environment. It also crosscuts the Knowledge component since goals may have to be created whenever some pieces of the agent knowledge change. The crosscutting interface IGoalCreation specifies how the Autonomy component crosscut these components in order to instantiate the agent goals. The crosscutting interface IThread describes how the execution autonomy crosscuts the Knowledge component to implement the concurrency agent strategy.

The Adaptation component crosscuts the Interaction component and the Knowledge component. It is connected with the former since adaptation of beliefs (Section 2.1) may be required upon the reception of external messages. The connection with the later is because adaptation of plans is necessary whenever a new agent goal needs to be achieved. The IKnowledgeAdaptation and IBehaviorAdaptation interfaces specify respectively how the Adaptation component affects the other architectural components.

Figure 6 also presents how the additional components (Mobility, Collaboration and Learning) crosscut each other and the agenthood components. It is worth to highlight that Figure 6 only presents the mandatory relationships between the architectural components, i.e. the relationships which are always present in agent architectures independently from specific applications. However, as the agent complexity increases, a specific component can crosscut other agent components. For instance, the Collaboration component can crosscut also the Interaction component when more sophisticated coordination strategies are required in a MAS. The Learning component also usually crosscuts the Collaboration and Interaction components.

3. Implementing the Generative Approach

We have implemented a first version of the generative approach for MASs using the following technologies: (i) XML-Schema [28] was used to specify the semantic of the Agent-DSL based on the feature models (Section 2.1); (ii) AspectJ and Java programming languages were used to implement components and aspects of the generic agent architecture; we have also defined EMF/JET code templates [3] which were useful to generate components and aspects to a specific agent; (iii) finally, a code generator supports the architecture configuration and has been implemented as an Eclipse plugin - it is responsible for mapping abstractions in the Agent-DSL into components and aspects of the agent architecture.

The following subsections describe in more detail the use of the technologies mentioned above to implement the generative approach.

3.1. Agent-DSL

The Agent-DSL was implemented using XML Schema [28] technology. The feature models were translated to XML Schema complex types. Many Agent-DSL complex types are composed of other complex types of the DSL. Below we present partially the XML Schema of the Agent-DSL.

```
…
<!-- Role Type -->
<xs:complexType name="RoleType">
  <xs:sequence>
    <xs:element name="name" type="name" minOccurs="1"
       maxOccurs="1"/>
    <xs:element name="description" type="description"
       minOccurs="0" maxOccurs="1"/>
   <xs:element name="belief" type="BeliefType"
       minOccurs="1" maxOccurs="unbounded"/>
   <xs:element name="goal" type="GoalType"
       minOccurs="1" maxOccurs="unbounded"/>
   <xs:element name="plan" type="PlanType"
       minOccurs="1" maxOccurs="unbounded"/>
    <xs:element name="interaction"
       type="InteractionType" minOccurs="1"
       maxOccurs="1"/>
    <xs:element name="adaptation"
       type="AdaptationType" minOccurs="1"
       maxOccurs="1"/>
    <xs:element name="autonomy" type="AutonomyType"
       minOccurs="1" maxOccurs="1"/>
   </xs:sequence>
</xs:complexType>

<!-- Agent Type -->
<xs:complexType name="AgentType">
   <xs:sequence>
      <xs:element name="name" type="name" minOccurs="1"
        maxOccurs="1"/>
     <xs:element name="name" type="name" minOccurs="1"
        maxOccurs="1"/>
     <xs:element name="description" type="description"
        minOccurs="0" maxOccurs="1"/>
```

```
      <xs:element name="belief" type="BeliefType"
         minOccurs="1" maxOccurs="unbounded"/>
      <xs:element name="goal" type="GoalType"
         minOccurs="1" maxOccurs="unbounded"/>
      <xs:element name="plan" type="PlanType"
         minOccurs="1" maxOccurs="unbounded"/>
      <xs:element name="interaction"
         type="InteractionType" minOccurs="1"
         maxOccurs="1"/>
      <xs:element name="adaptation"
          type="AdaptationType" minOccurs="1"
          maxOccurs="1"/>
      <xs:element name="autonomy" type="AutonomyType"
          minOccurs="1" maxOccurs="1"/>
      <xs:element name="role" type="RoleType"
          minOccurs="0" maxOccurs="unbounded"/>
   </xs:sequence>
</xs:complexType>
…
```

For each MAS to be generated, an agent description XML document of the Agent-DSL must be offered. This document must conform to the XML Schema that defines the Agent-DSL.

3.2. Aspect-Oriented Agent Architecture

The aspect-oriented agent architecture was implemented using the Java [15] and AspectJ [22] programming languages. It is composed of: (i) a framework that contains class and aspect hot-spots [7]; (ii) a set of components implemented to support basic and specific functionalities of the agent; and (iii) a set of meta-components, implemented as EMF/JET [3] source templates, that are customized by the code generator using information of the Agent-DSL.

The framework defines some classes and aspects as the core of the aspect-oriented agent architecture. It also contains hot-spot classes that we use to define the agent knowledge, and hot-spot aspects used to concretize architectural components implementing the agenthood and additional agent features.

Many framework components were created to implement specific functionalities associated with the agenthood features, such as:

- *interaction features*: data structures to store received and sent messages; data structures to maintain sensors and effectors of the agent; and concrete sensors and effectors to specific agent platforms (such as JADE [2]);
- *adaptation features*: data structures to interrelate goals and plans; and data structures to call class methods to update beliefs when specific messages are received by the agent;
- *autonomy features*: data structures to instantiate goals from external messages (reactive goals) and from internal events (proactive goals); the basic concurrency strategy behavior; and concrete concurrency strategy, such as, thread pool and thread per request.

Finally, the agent architecture also contains some EMF/JET source templates. We use source templates to implement components that need to be customized based on information collected by the Agent-DSL. Java Emitter Templates (JET) a generic template engine of the Eclipse Modeling Framework (EMF) [3] has been used. JET enabled us to write source templates that express structure and behavior of classes and aspects that we want to generate using the agent description file. Examples of classes and aspects that we wrote as templates are: (i) concrete instances of hot-spots (classes or aspects), such as, specific Agent type classes, specific agenthood aspects; and (ii) specific agent plans and goal classes.

3.3. Agent Architecture Generator

The current version of the agent architecture generator was implemented as an Eclipse plugin [25]. We used a kit of technologies related to Eclipse project to implement the generator. JAXB plugin [18] was used to enable the reading of the agent description XML file of the Agent-DSL. This plugin permits to generate classes that represent the elements of a XML-Schema. So, you can use these generated classes to read XML files that conform to that XML-Schema.

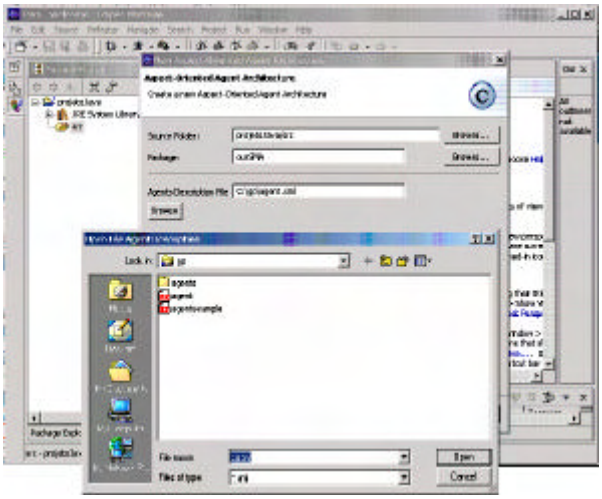


Figure 7. Agent AO-Architecture Eclipse Plug-in

The generator plugin offers a wizard in the Eclipse workbench to start the process of code generation (figure **7**). The wizard requests from the user: (i) a source folder to arrange the classes and aspects generated; (ii) a package name used as a root package to arrange classes and aspects generated; and (iii) the agent description XML file that contains descriptions of

agents to be generated. As soon as the user provides this information classes and aspects of agents are generated.

The internal implementation of the agent architecture generator plugin uses Java Development Tooling (JDT) API [25] to create Java packages and classes, and AspectJ aspects. Also, JET API is used to process templates and to pass information of agent description file to source templates.

4. Discussions and Conclusions

This paper presented a generative approach to assist the development of multi-agent systems. The aim of the approach is to explore the horizontal domain that MASs represent to enable the code generation of agent architectures. Aspectual components have been used to model crosscutting agent features from the architectural point of view. We have already used the generative approach to implement partially two case studies: (i) ExpertCommittee - a conference management system; and (ii) Portalware - a web-based environment for the construction and management of e-commerce portals [13, 14].

The definition of a generative approach, brings important benefits compared to other MAS development approaches [10, 17, 27]. The benefits include: (i) clear definition of the mapping between high-level features and implementation components (classes, aspects) of a agent architecture in the code generator; (ii) clear separation of orthogonal and crosscutting agent features; (iii) flexibility to evolve the problem space independent of solution space by creating new domain specific languages to address other MAS concerns; and (iv) flexibility to evolve or change the architectural model used in the solution space independent of problem space.

In our generative approach we have derived aspects from our domain knowledge based on previous studies [10, 11, 13, 17, 26, 27]. We identified that many important MAS features, such as, autonomy and adaptation, are commonly tangled in the specification artifacts and source code of agent implementations. Results gathered in previous empirical studies [11, 12, 13, 14] have emphasized that the separation of these MAS features improves the maintainability and reusability of MASs.

The aspect-oriented framework and code templates (Section 3.2) expose the identification and generalization of many MASs domain aspects (agenthood and additional agent features).

5. Future Work

This position paper presented a initial version of a generative approach for MASs. We are currently working in different ways to improve our ability to generate MASs source code and to handle its different artifacts (DSLs, components and aspects in the architecture, mapping between DSL elements and components/aspects).

The specification of an agent using Agent-DSL is supported currently by the edition of a XML file. We are defining a mixture of wizards and diagrams that help MAS developers to specify agents based on Agent-DSL. We have also focused on Agent-DSL refinement to support the modeling of other relevant MASs concerns, such as, learning, mobility, agent coordination, and agent organizations. We are working on a conceptual framework [26] that provides a more precise definition of these agent features in order to incorporate them into our Agent-DSL. Moreover we are investigating how to increment Agent-DSL to model the dynamic semantics of MASs, such as, plan execution, role creation and allocation.

Also, an aspect-oriented pattern language is being developed to refine our agent architecture as a set of design patterns and idioms that model progressively MASs concerns. The patterns and idioms provide detailed solutions for each architectural component in terms of classes and aspects.

The use of AspectJ in our case studies limited the capacity to reconfigure dynamically the MASs, not permiting attach and detach aspects from the agent and knowledge objects. We plan to implement a new case study based on dynamic weaving technologies, such as, PROSE [24], AspectWerkz [1] and JBoss [8].

Finally, an ongoing research work is a definition of an architectural definition language (ADL) that supports the definition of the aspectual components, normal and crosscutting interfaces (Section 2.2). We plan to use this ADL to formalize the definition of aspect-oriented agent architectures. The ADL can bring many benefits for our work, such as: (i) enable the specification of aspect-oriented architectures in a high-level independent of technologies; and (ii) facilitate architecture analysis and maintenance. When incorporating the ADL to our generative approach, we plan to offer flexible ways of specifying the transformations of elements in Agent-DSL to elements (components, aspect and interfaces) of the ADL. Also, transformation rules of the ADL to concrete technologies (for instance, Java and AspectJ) could be defined.

6. References

[1] AspectWerkz. Available at URL http://aspectwerkz.codehaus.org/

[2] F. Bellifemine, A. Poggi & G. Rimassi, "JADE: A FIPA-Compliant agent framework", Proc. Practical Applications of Intelligent Agents and Multi-Agents, pp. 97-108, April 1999.

[3] F. Budinsky, D. Steinberg, E. Merks, R. Ellersick, T. Grose. Eclipse Modeling Framework, Addison-Wesley, 2003.

[4] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. Pattern-Oriented Software Architecture: A System of Patterns, John Wiley and Sons, 1996.

[5] C. Chavez. "A Model-Driven Approach to Aspect-Oriented Design". PhD Thesis, PUC-Rio, 2004.

[6] K. Czarnecki, U. Eisenecker. Generative Programming: Methods, Tools, and Applications, Addison-Wesley, 2000.

[7] M. Fayad, D. Schmidt, R. Johnson. Building application frameworks: Object-oriented foundations of framework design, John Wiley & Sons, September 1999.

[8] M. Fleury, F. Reverbel. "The JBoss Extensible Server". Proceedings of the International Middleware Conference 2003, vol. 2672 of LNCS, pp. 344-373, Springer-Verlag, 2003.

[9] Foundation of Intelligent Physical Agent: FIPA Interaction Protocols Specification, (2002). Available at URL http://www.fipa.org/repository/ips.html

[10] A. Garcia, C. Lucena. Software Engineering for Large-Scale Multi-Agent Systems. ACM Software Engineering Notes, August 2002.

[11] A. Garcia, et al. "Engineering Multi-Agent Systems with Aspects and Patterns". Journal of the Brazilian Computer Society, September 2002.

[12] A. Garcia et al. Separation of Concerns in Multi-Agent Systems: An Empirical Study. In: C. Lucena et al (Eds). "Advances in Software Engineering for Multi-Agent Systems". Springer-Verlag, LNCS 2940.

[13] A. Garcia, C. Lucena, D. Cowan. "Agents in Object-Oriented Software Engineering". Software: Practice and Experience, May 2004, pp. 1-33. (to appear)

[14] A. Garcia, M. Cortés, C. Lucena. "A Web Environment for the Development and Maintenance of E-Commerce Portals based on a Groupware Approach". Proceedings of the Information Resources Management Association International Conference (IRMA'01), Toronto, May 2001,

[15] J. Gosling, B. Joy, G. Steele. The Java Language Specification. Addison-Wesley Longman, Inc., 1996.

[16] E. Harold, W. Means. XML in a Nutshell, 2nd Edition, O´Reilly, 2002.

[17] C. Iglesias, et al. "A Survey of Agent-Oriented Methodologies". Proceedings of the ATAL-98, Paris, France, July 1998, pp. 317-330.

[18] JAXB Eclipse Plugin. Available at URL http://sourceforge.net/projects/jaxb-builder/

[19] N. Jennings, M. Wooldridge. "Agent-Oriented Software Engineering". In: J. Bradshaw (ed.), Handbook of Agent Technology, AAAI/MIT Press, 2000.

[20] K. Kang, S. Cohen, J. Hess, W. Novak, A. Peterson. "Feature-oriented domain analysis (FODA) feasibility study". Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, 1990.

[21] G. Kiczales, et al. "Aspect-Oriented Programming". Proc. Of`ECOOP'97, LNCS 1241, Springer-Verlag, Finland, June 1997.

[22] G. Kiczales, et al. "Getting Started with AspectJ". Communication of the ACM. October 2001.

[23] Pace, A., Trilnik, F., Campo, M. "Assisting the Development of Aspect-based MAS using the SmartWeaver Approach". In: A. Garcia, C. Lucena, J. Castro, A. Omicini, F. Zambonelli (Eds). "Software Engineering for Large-Scale Multi-Agent Systems". Springer-Verlag, LNCS, March 2003.

[24] A. Popovici, T. Gross, G. Alonso. Dynamic Weaving for Aspect Oriented Programming. In 1st Intl. Conf. on Aspect-Oriented Software Development, Enschede, The Netherlands, Apr. 2002.

[25] S. Shavor, J. D'Anjou, S. Fairbrother, et all. The Java Developer's Guide to Eclipse. Addison-Wesley, 2003.

[26] V. Silva, et al. "Taming Agents and Objects in Software Engineering". In: "Software Engineering for Large-Scale MASs". Springer, LNCS 2603, March 2003.

[27] M. Wooldridge, P. Ciancarini (Eds.). "Agent-Oriented Software Engineering: The State of the Art". In: Agent-Oriented Software Engineering, Springer, LNAI, 2001.

[28] XML Schema. Available at URL http://www.w3.org/XML/Schema.

Identifying Aspects Using Architectural Reasoning

Len Bass
Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA, 15213 USA
+1 412 268-6763
ljb@sei.cmu.edu

Mark Klein
Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA, 15213 USA
+1 412 268-7615
mk@sei.cmu.edu

Linda Northrop
Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA, 15213 USA
+1 412 268-7638
lmn@sei.cmu.edu

1. Introduction

Software architecture, which encompasses the structures of software systems, has emerged as a crucial part of the design process. A software architecture is developed as the first step toward designing a system with certain desired properties. A growing body of experience and evidence suggest that the dominant design drivers for any software architecture are its quality attribute requirements [3]. By quality attributes we mean such properties as reliability, security, usability, modifiability, performance, portability, etc. Quality attribute requirements articulate the important quality attributes for a system in system-specific characterizations. We are developing a method to derive a software architecture from its quality attribute requirements via insights gained from quality attribute models called architectural tactics. We believe that some of the discoveries that occur during the derivation process can be viewed as candidate aspects. In particular, the derivation process illuminates what we call *architectural aspects*. Architectural aspects come with architectural analogous advice, pointcuts, and join points. The architectural aspects are candidate aspects to be carried through detailed design and implemented using AOP.

We describe this connection in a bottom-up fashion because we think it is helpful to see the concepts at work in a simple example before delving into the details of the method. So after we set the stage by introducing some new terminology, we begin with a small set of quality requirements for an example system, present a software architecture that satisfies those requirements, and highlight the architectural tactics at work in that architecture. We then identify architectural aspects and their constituent architectural advice, pointcuts, and join points. Having motivated the need to derive the relevant architectural tactics, we next summarize our method and how it can derive the tactics and architectural aspects already described. In practice, one would, of course, begin with the quality requirements, apply the method, derive the architecture, and, in the process, identify the candidate aspects. We conclude with a postulation of the benefits associated with aspect identification during architecture design, as well as a discussion of open issues and possible future work.

2. Terminology

Kiczales and colleagues use the following terms in their overview of AspectJ [6]:

- *Join points* are well-defined points in the execution of the program.
- *Pointcuts* are a means of referring to collections of join points and certain values at those join points.
- *Advice* consists of method-like constructs used to define additional behavior at join points.
- *Aspects* are units of modular crosscutting implementation composed of pointcuts, advice, and ordinary Java member declarations.

We use the same terminology extended with a prefix of *architectural* to emphasize that we are applying these aspect concepts to architectural reasoning rather than language reasoning.

- *Architectural join points* are well-defined points in the specification of the software architecture.
- *Architectural pointcuts* are means of referring to collections of architectural join points.
- *Architectural advice* is a specification (possibly informal) of transformations to perform at architectural join points.

- *Architectural aspects* are architectural views consisting of architectural pointcuts and architectural advice.

3. Example

The example we use is a variation of one that we have used before [1, 3], a product line of garage door opener systems. This system is a portion of a home integration system (HIS) that manages the opening and closing of the garage door.

The basic premise of our work in architecture design is that quality attribute requirements are the dominant design drivers for any system. We express quality attribute requirements as scenarios formulated in a structured fashion. The following are the quality attribute requirements for our example. In each case the relevant quality attribute is indicated in square brackets at the end of the requirement.

Reacting to obstacles: The garage door system must be able to detect obstacles and safely halt a closing garage door in a timely manner. Timely is defined to be within 0.1 seconds. Also, when an obstacle is detected, this event must be reported to the user interface (UI) display (if there is one) and to an HIS (if there is one). [performance]

Door commands: The garage door system can receive commands from a remote control in the car, the HIS, or buttons attached to the garage. Several commands must be handled: open, close, halt, and diagnosis. The garage door system must be able to detect a command and initiate its execution within 0.5 seconds. (Note that when "halt" is issued via obstacle detection sensors, it is subject to the .1 second requirement.) [performance]

UI: Various garage door openers have various controls. Some have displays and others do not; some may be integrated with an HIS while other cannot be; various types of remote controls should be supported. [modifiability]

Sensors and actuators: Sensors and actuators can change due to either obsolescence or changes in the marketplace. [modifiability]

4. Architecture of example

Figure 1 depicts the module decomposition view of an architecture for the garage door opener system. It has the following modules with associated responsibilities:

- *UI*. Manage interactions with UI sensors and displays. A change in UI devices will be handled here.
- *Virtual UI*. Translate interactions from the UI into a set of commands common for all UIs. This module hides all changes in UI devices so they are not visible elsewhere in the system. This module also checks security codes by sending a message to the House proxy (through the Pub/Sub module).
- *Pub/Sub*. Handle events received from modules on a subscription basis. Doing so keeps portions of the system that exist in some products and not in others (such as the HIS) from affecting the remainder of the system.
- *House proxy*. Manage interactions with the HIS (if it exists).
- *Door commands*. Manage all the door's actions.
- *Virtual sensor/actuator*. Hide details of actual sensors or actuators (except for obstacle detection). Doing so allows sensors or actuators to change without impacting the remainder of the system.
- *Device driver*. Handle interactions with particular sensors or actuators except for the obstacle detection sensor.
- *Obstacle detection algorithm*. Handle obstacle detection. It sends a halt instruction directly to the door motor.
- *Obstacle detection driver*. Handle interactions with the obstacle detection sensor.

Notice that a major influence on the module view is the requirement to accommodate change. The modules whose role is to hide the effects of changes include Virtual UI, Virtual sensors/actuators, House proxy, and Pub/Sub.

Figure 2 depicts the concurrency view of the same architecture. It shows some threads and associated communication paths among the components derived from the modules. The numbers in the circles reflect scheduling priority. The Obstacle Detection thread has highest priority, the Door Commands and the House Proxy threads have the next highest priorities, and the Status Reporting thread has the lowest priority. Notice that a major influence on the concurrency view is the need to satisfy stringent timing requirements. In particular, this architecture allows for high-performance obstacle detection.

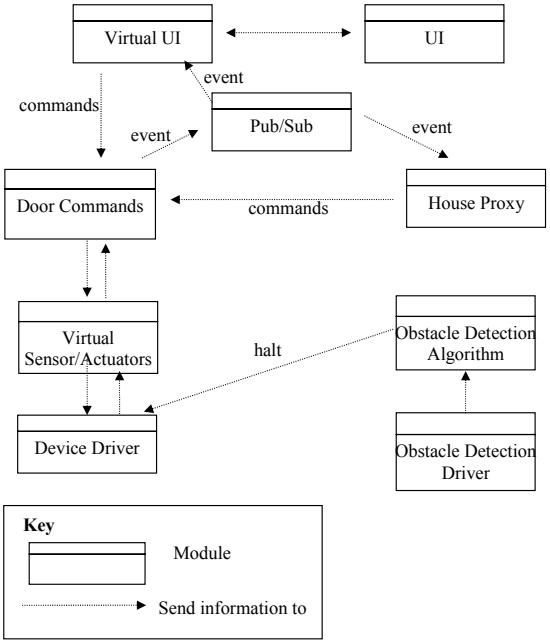


Figure 1. Module view

5. Architectural tactics and aspects

We define an architectural tactic as *"a means of satisfying a quality-attribute-response measure by manipulating some aspect of a quality attribute model through architectural design decisions"* [1]. We have enumerated architectural tactics for the achievement of modifiability and performance. A subset of the tactics we identified lead directly to architectural aspects.

Two key modifiability tactics are *Abstract common services* and *Break the dependency chain*. The *Abstract common services* tactic prescribes generating a new module comprising common or similar responsibilities that previously resided in multiple modules. This tactic localizes similar responsibilities. The *Break the dependency chain* tactic prescribes inserting an intermediary between the publisher and consumer of data or services. This tactic helps prevent the propagation of a change to the producer from propagating to the consumer (or vice versa). Looking again at Figure 1, we see that the *Abstract common services* tactic was applied to create the Virtual UI and the Virtual sensor/actuator modules, and the *Break the dependency chain* tactic was applied to generate the Pub/Sub module (between the Door commands module and several consumers of its events).

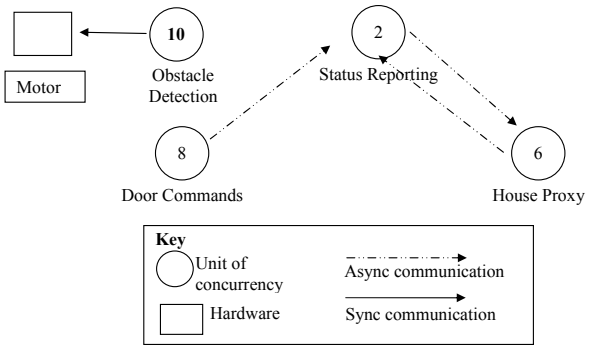


Figure 2. Concurrency view

The *Break the dependency chain* tactic leads directly to an architectural aspect. In this case the aspect implements the crosscutting concerns related to the publisher/subscriber:

- architectural aspect – the collection of pointcuts and advice related to the publisher/subscriber
- architectural pointcuts – specification for every place where an event is published or subscribed to, and every place where an event is defined
- architectural advice – establishes the right connections between producers and consumers of events, and provides the details of the types of generated events

This architectural aspect would allow the designer to defer the choice of the details of the publish/subscribe mechanism for communication, such as the choice of shared memory or message passing, and the subsequent decision would be woven in at either compile time or runtime. If there is no subscriber for certain events (e.g., no House proxy), the aspect can eliminate the generation of messages for those events.

Similarly the Virtual UI and Virtual sensor/actuator modules can be considered architectural aspects. Consider the Virtual sensor/actuator module:

- architectural aspect – the collection of pointcuts and advice related to sensor/actuator services
- architectural pointcuts – specification for every place where a sensor/actuator service is invoked
- architectural advice – specification of the device driver details associated with each generic sensor/actuator service—in effect the entire Device driver module.

One performance tactic evident in the concurrency view is *Use fixed-priority scheduling*. This tactic, which leads directly to an architectural aspect, assumes that each unit of concurrency (see Figure 2) is assigned a fixed priority.

- architectural aspect – the collection of pointcuts and advice related to the prioritization of units of concurrency
- architectural pointcuts – specification for every place where the priority of a unit of concurrency is specified
- architectural advice – specification of the rules inspecting the relative deadlines of units of concurrency and using those rules as the basis for assigning priorities

Making the priority assignment an architectural advice allows the priorities to be modified in a uniform fashion and modularizes this crosscutting concern.

Another performance tactic is *Increase logical concurrency*, which is likewise an architectural aspect.

- architectural aspect – the collection of pointcuts and advice related to communication between units of concurrency
- architectural pointcuts – specification for every place where communication services are used
- architectural advice – specification of the communication protocol—asynchronous or synchronous—based on the communication path

This architectural aspect localizes the communication strategy so that it can be changed uniformly.

The key point to observe from this discussion is that each use of an architectural tactic is manifested by particular points in the architecture, and each point is a candidate for an architectural join point. These join points would be controlled by the architectural advice. Whether an architectural aspect should be used to depict any given tactic is something for which we have, as yet, no general rules. At this point, what we can say is that each application of a tactic is a candidate for an architectural aspect. Moreover, treating the tactic as an architectural aspect seems to bring to the architecture the same advantages that aspects bring to programming.

The use of an architectural tactic is not a random event but rather the result of a deliberate design decision. In the next section, we describe how tactics are chosen.

6. Method for identifying tactics

The example in the previous section illustrates that the notion of aspects is applicable at the architecture level and that architectural aspects are discovered through the identification of the architectural tactics being applied. The obvious question that arises is: how does one go about identifying the appropriate architectural tactics to apply? We developed a method that shows how to derive architectural decisions from a set of quality attribute requirements through the application of architectural tactics. We can extend this method easily to identify candidate architectural aspects.

Figure 3 shows the key elements of our method. Quality requirements are expressed as quality attribute scenarios. In the figure we show only modifiability and performance requirements. For the purposes of our method, functional requirements are expressed in a variety of forms such as text, state charts, use cases, and then are translated into a responsibility graph.

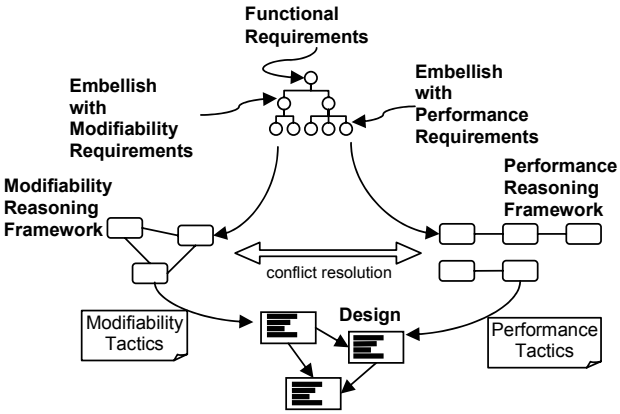


Figure 3. Elements used in our method

6.1 Formulate quality attribute requirements

We formulate quality attribute requirements according to a structured scenario format made up of these six parts: 1. stimulus to which the scenario reacts. This can be at runtime or prior to runtime, depending on the quality attribute; 2. the source of the stimulus; 3. the artifact affected by the stimulus; 4. the environment of the system when the stimulus arrives; 5. the desired response to the stimulus; 6. the measure for the response and the bound (constraint) on that measure that must be satisfied

Table 1 gives the parts and the values for the three scenarios we are considering.

Table 1. Scenarios being considered

	Obstacle detection	**Modify sensor/actuator except motor**	**Modify motor**
Source of stimulus	Obstacle detection sensor	New product request	New product request
Stimulus	Obstacle detected	Develop new product with different sensor/actuator	Develop new product with different motor
Environment	Garage door	Development	Development

	descending	time	time
Artifact	Motor	Source code for system	Source code for system
Response	Halt door	Produce new product	Produce new product
Response measure and constraint	Within .1 sec.	Within 1 staff-days	Within 1.5 staff-days

For each scenario the relevant reasoning framework is deducible from its response measure. The first scenario has a timing deadline, indicating that a reasoning framework that deduces latency would be appropriate; the Rate Monotonic Analysis [5] is therefore suitable. The other two scenarios have staff-days as the response measure, indicating that the Impact Analysis reasoning framework should be used to reason about the achievement of these scenarios.

6.2 Construct quality attribute model

We begin with the Rate Monotonic Analysis reasoning framework.

6.2.1 Rate Monotonic Analysis

The Rate Monotonic Analysis reasoning framework suggests an assignment of responsibilities to units of concurrencies and uses estimates of the execution time and the priorities of the units of concurrency to determine the latency in response to a particular event.

The architect must estimate the execution times for the responsibilities. During the action of the reasoning framework, the architect may be asked whether it is possible to reduce those estimates. After one successful satisfaction of the scenarios, the execution time estimates become budgets that must be adhered to by subsequent implementation.

Some of the architectural tactics comprising the Rate Monotonic Analysis reasoning framework for this example include

- *Reduce execution time*. This tactic causes the software architect to consider what the execution times are for each responsibility and whether they can be reduced.
- *Use fixed-priority scheduling*. This tactic determines the basis for assigning priorities and calculating the latency in response to the single event of obstacle detection.
- *Increase logical concurrency*. In our example, we used this tactic as a method for reducing the latency associated with responsibilities that have stringent timing requirements.

For example, the *Use fixed priority scheduling* tactic suggests assigning the responsibilities "detect obstacle" and "halt door" to the highest priority unit of concurrency, since those responsibilities have the most stringent timing requirement associated with them. Rate Monotonic Analysis gives us rules of thumb for assigning priorities (shortest deadline first) that will satisfy the constraints (if all the performance constraints can be satisfied). The Rate Monotonic Analysis reasoning framework, through the use of tactics, focuses the architect's attention on the design decisions that are the most influential in achieving our performance goals. As we saw in the previous section, architectural aspects provide us with a way to modularize these high-impact architectural decisions.

6.2.2 Impact analysis

The Impact Analysis [4] reasoning framework assigns responsibilities to modules in such a way as to satisfy the constraints from the scenarios. The reasoning framework may also refine the responsibilities according to various tactics.

The Impact Analysis reasoning framework constructs a dependency graph among modules. The nodes of the graph are modules, and the arc between nodes A and B represents the probability that a modification to node A will require a derivative modification to node B. For a given dependency graph, a cost function determines the cost of a modification to a particular responsibility within a given node. See [2] for details of the cost model; we will not go into its workings here. The cost model depends on the cost of modifying a responsibility, the packaging of the responsibilities into modules, and the probability of a modification of one responsibility propagating to another.

In the Rate Monotonic Analysis reasoning framework, the architect needed to specify the execution time of responsibilities, the assignment of responsibilities to units of concurrency, and the priority of those units of concurrency.

In the Impact Analysis reasoning framework, the architect must specify the cost of modifying each responsibility. Doing so is equivalent to specifying the execution time for the Rate Monotonic Analysis reasoning framework. The architect specifies the cost using experience and intuition—just as he or she specifies

the execution time. One tactic encourages the architect to think about the specified cost of modifying a responsibility and whether it can be reduced, again in parallel with Rate Monotonic Analysis.

There are two types of tactics within the Impact Analysis reasoning framework. One type is intended to reduce the cost of a modification to a responsibility by modifying the set of responsibilities, and one type is intended to reduce the likelihood that a modification is propagated to other responsibilities.

In our example, two scenarios are relevant to the Impact Analysis reasoning framework—replace sensors or actuators, and replace the motor.

The reasoning framework begins by considering the cost of modifying the software to manage a new sensor. If this cost is less than the constraint value, this scenario is satisfied. For our example, we assume that the constraint is not met and therefore that tactics need to be applied.

The first set of tactics to consider are those that attempt to reduce the cost of making a modification. The *Abstract common services* tactic suggests to the architect that the management of sensors can be considered a common service. This approach causes a potential re-arrangement of responsibilities into those that manage the sensors (the sensor device drivers) and those that use the sensor management responsibilities. Now the cost of changing the sensor is recalculated using the reasoning framework. Once again, we assume the cost is too high. The architect examines the cost model and sees that changes to the sensor cause changes to the sensor device driver that, in turn, are propagated through the remainder of the responsibilities. This observation causes the architect to apply the *Break the ripple effect* tactic. Again the responsibilities are rearranged so that a virtual sensor/actuator is interposed between the device drivers and the other responsibilities. This time, the cost of a change is examined and the constraint is now met.

Notice that the application of the tactics is a joint undertaking of the method and the architect. The method will highlight where problems occur and which tactics might be applicable. The architect then chooses appropriate tactics and how they are applied. Since nothing, thus far, has affected the scheduling of the obstacle detection, the performance constraint is still satisfied.

Now, the other modifiability scenario, one that calls for the modification of the motor, is examined. The same initial reasoning that led to sensor/actual device drivers leads to a motor device driver. This time, however, the constraint is 1.5 staff-days rather than 1 staff-day. This larger constraint is satisfied just by the device driver without interposing a virtual motor.

The Impact Analysis reasoning framework, through the use of tactics, focuses the architect's attention on the design decisions that are the most influential in achieving modifiability goals. As we saw for performance, aspects also provide us with a way of modularizing the high-impact architectural decisions that affect modifiability.

6.3 Convey information among reasoning frameworks

In addition to those responsibilities extracted from the scenarios, the other responsibilities for the garage door system must be enumerated. For our purposes, they are lumped into a single responsibility called "other."

The only vocabulary that the Rate Monotonic Analysis reasoning framework and the Impact Analysis reasoning framework have in common is the concept of responsibilities. A responsibility graph is used to capture the relationships among the responsibilities. Possible relationships are either is-a-part-of or precedes. Responsibilities can be decomposed to yield is-a-part-of relationships. This is what happens when the common services are embodied in the Virtual sensor/actual module. Responsibilities can also precede other responsibilities. For example, an obstacle must be detected prior to the door being halted in one of our scenarios.

Both reasoning frameworks, during their activities, drew on the responsibility graph to determine responsibilities for allocation to either units of concurrency or modules, and modified the responsibility graph through the addition of new responsibilities or the decomposition of responsibilities.

6.4 Satisfy conflicting constraints

In general, a reasoning framework considers a set of possible tactics (derived from looking at the analysis model that the reasoning framework uses) and attempts to find a combination of tactics (in conjunction with the architect) that will satisfy all the relevant constraints. Each application of a tactic may affect the responsibility graph that may, in turn, affect reasoning frameworks other than the one that is applying the tactic. Recalling our prior discussion, application of some tactics also identifies a candidate architectural aspect.

When the various data structures are stable, the implied design should be generated. If the data structures do not stabilize or if some constraints are unable to be satisfied, the system is overconstrained and cannot be constructed. In this case, either a constraint must be

relaxed or parameters (such as execution time) are adjusted.

6.5 Interaction between the design method and the architect

We did not show the derivation that led to two scenarios to reflect the cost of modification. Originally, there was only one scenario that dealt with the modification of sensors/actuators. That scenario led, using a chain of reasoning as above, to a Virtual sensor/actuator module. The performance constraint involving the motor, however, could not be met using a virtual motor. Thus, the architect was informed, through the method, that the quality requirements could not be met. This caused the architect to weaken the modifiability requirement for the motor.

Although the method provides guidance and information to the architect, he or she still makes the key decisions.

7. Conclusions and open issues

We have presented a method that begins with quality requirements cast as quality scenarios and moves from there (in a semi-automatic fashion) to a design. The method is based on using quality attribute reasoning frameworks to determine whether the requirements can be met, to identify places in an emerging design that cause them not to be met, and to provide the architect with guidance as to which architectural tactics to use to correct the problem.

We initially used an example to present our method from the bottom up for pedagogical purposes. From the top down, the method proceeds as follows: Represent quality attribute requirements as structured scenarios; Generate a design using reasoning frameworks and architectural tactics that depend on those reasoning frameworks to satisfy quality attribute requirements; Consider each use of an architectural tactic in generating the design as a candidate architectural aspect, where the join point is the place in the architecture where the tactic was applied and the advice is dependent on the quality attribute being achieved by the tactic.

This leaves two questions for future consideration:

- Which subset of architectural tactics leads to architectural aspects?
- Which subset of architectural aspects is realizable as language-level aspects?

We believe there is a systematic method for addressing both questions, but they remain open issues.

Some other ideas that we believe have promise at the architectural level and merit investigation include

- identifying an "aspect" view of the architecture that would display those places in the architecture where decisions have been deferred and where an "architectural aspect" weaver might be useful
- Many of the aspects we have identified require a weaver to operate at more than just a syntactic level. For example, a weaver may decide based on performance considerations identified by a method such as ours that for one join point, information can be passed as messages but for another, it should be passed through shared memory.
- some method for specifying functionality and architectural style independently and weaving them together

8. Acknowledgments

Our thanks to Gregor Kiczales, Gail Murphy, and Harold Osher for participating in a workshop at the Software Engineering Institute that led to this paper.

9. References

[1] Felix Bachmann, Len Bass, and Mark Klein. Deriving Architectural Tactics: A Step Toward Methodical Architectural Design (CMU/SEI-2003-TR-004). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2003.

[2] Len Bass, Felix Bachmann, and Mark Klein. Making Variability Decisions During Architecture Design, Proceedings Product Family Engineering – 5, Nov, 2003, Sienna, Italy, Springer-Verlag.

[3] Len Bass, Rick Kazman, and Paul Clements. Software Architecture in Practice, 2nd edition. Reading, MA: Addison-Wesley, 2003.

[4] Shawn A. Bohner and Robert S. Arnold. Software Change Impact Analysis. IEEE Computer Society Press, 1996.

[5] J.P. Lehoczky, "Fixed Priority Scheduling of Periodic Task Sets with Arbitrary Deadlines," Proc. of the 11th IEEE Real-Time Systems Symposium, pp. 201-209, December 1990.

[6] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm and William G. Griswold. An Overview of AspectJ In Proceedings of ECOOP, Springer-Verlag (2001).

Facets of Concerns

Crenguța Bogdan
"Ovidius" University, Constanta, Romania
cbogdan@univ-ovidius.ro

Abstract

The concern facet concept is introduced. We claim that in a world of "concerns" a cluster of facets is associated with each and every concern. The facets in the cluster may structure the concern specification. In order to demonstrate this statement, we define a group of concern facets for specifying the concerns of the software process and software product quality and use them in developing software systems.

1. Introduction

Yet there are debates on the exact definition (meaning, as much as possible close to the intuitive meaning) of the "concern" notion. Nevertheless, the scientific world more and more accepts the concern as "any matter of interest in a software system"[8, p.128].

A great deal of the set of concerns comes from the software process model as shown in [10, 9, 7, 2]. These concerns guide the development of software systems and structure the software product.

Our claim is that the concerns can be structured, too. The structuring element is the facet concept, as presented in [1]. In order to highlight the use of facets in structuring of the software process and product, we will consider RUP (Rational Unified Process) as a model [6] of the software process. Some of the RUP concerns are identified.

2. Facets of concerns

In our opinion facets are that kind of questions one might want to ask about a concern. A facet instance is a possible answer to such question. Usually, a stakeholder raises such questions. Answering to them, the stakeholder specifies new facet instances. If the identified facets completely describe the whole concern, the stakeholder will be able to better control the concern. There are also cases when even a concern may query other concerns for information.

In order to explain better the role of facets in concern specification, let us consider a concern C and some typical questions about it:

- Definition: Which is the definition of C?
- Example: What are the things that satisfy C's definition?
- Address: When can we say that C is addressed?
- Resolve: When can we say that C is resolved?
- Generalization: Which other concerns have less restrictive definitions than C?
- Specialization: Which concerns have C's definition plus some additional constraints?
- Life Cycle Phase: Which are those concerns that are addressed or resolved in the same life cycle phase with C?
- Workflow: Which are those concerns that are addressed or resolved in the same workflow of iteration with C?
- Artifact: Which are the other concerns that help together with C in realizing the same artifact?
- Role: If C is relevant to a role, which other concerns are relevant to the same role?
- Dependency: Which are concerns that C is depending on?

In addition, each facet F of a concern C has subfacets that answer at the following questions:

- Check: How can elements of C.F be checked?
- Evaluate: How can elements of C.F be evaluated?

where C.F represents the facet F of a concern C.

The facets of a concern break into three categories:

- basic facets which contain intensive information on concern C itself: Definition, Address, Resolve;
- context facets which are linking the concern to other concern(s): Example, Generalization, Specialization, Life Cycle Phase, Iteration, Artifact, Role, Dependency;
- other facets which can be tacked into any facets from above: Check, Evaluate.

The paper defines some basic and context facets of concerns and introduces several examples of facets.

3. Basic facets

In this section we present three basic facets of concerns, namely: Definition, Address and Resolve.

3.1. Definition

To work with a concern, first of all we have to define it. The layout of this facet is a list of elements, where each element, subfacet of the facet, consists of the following attributes:

- Identification:
  - type: global/local,
  - visibility: opaque/transparent,
  - life cycle: software process/life-cycle phase/iteration,
  - inclusion: aggregate/atomic,
  - behavior: initiator/active/passive,
  - locality: crosscutting/not-crosscutting.
- Responsible: who is responsible with the concern resolving (most likely, a stakeholder).
- Audience: stakeholders that are interested in the concern.
- Collaborators: Using definition of other concern X, Reducing to the definition of Y, Same definition as Y except…, Specialized version of Z, etc.
- Predicate: a test that verifies if a given thing is an example of the concern.

Let consider the attribute values of the Identification subfacet.

A *global concern* is a concern of the entire software project and will be checked during the major milestones. Instead, a *local concern* is a concern of one or few iterations and will be checked during the minor milestone of the iterations. By example, *system_objective* concern is checked at every major milestone; instead, during the minor milestones, the *evaluating-criterion* concern is checked.

We define a *transparent concern* as a concern that can be resolved and checked. Instead, an *opaque concern* can only be addressed, because we don't have all the necessary information to resolve it. All the quality concerns of the product are opaque because only when the software project is closed, we can say if the system is of quality or not. Instead, *architecture-baseline* is a transparent concern in the elaboration phase of the software process: it will be resolved in this phase and will be checked during the architecture life-cycle milestone.

Until a concern is resolved, it evolves during the entire software process or only a part of it. This part may be an entire life-cycle phase or only one or few iterations of it. The life-cycle parameter of the Identification subfacet indicates the area of software process where the concern is relevant. By example, the *architecture* concern evolves through the software process like that:

- in the inception phase, an architecture statement is synthesized by evaluating the design trade-offs, problem space ambiguities and available-space assets (technologies and existing components),
- the elaboration phase builds up an architecture baseline that resolves the critical use cases,
- in the construction phase, the architecture is enriched by integrating remaining components in order to fulfill all the requirements from the vision document,
- during the transition phase, the architecture is modified in order to resolve multiple-component design issues that affect the quality attributes of the system, like reliability, performance and maintainability.

A concern is *aggregate* if its resolution needs the resolution of other concerns. Otherwise, we will call it *atomic* concern. By example, *architecture* is an aggregate concern because for constructing an architecture, we must firstly construct the "4+1" views [4]: *design view*, *process view*, *component view*, *deployment view* and *use case view*, which are concerns themselves. Furthermore, to obtain a *stable architecture* we have to resolve the quality concerns such as *performance*, *reliability*, *adaptability*, *robustness*, *scalability*, *economic-trade-offs*, *technology-constraints*, *comprehensibility* and *ease-to-construct*. These concerns are aggregate concerns, too.

In general, concerns obtained during the software process are aggregate. Atomic concerns are relative to the software system and depend on this.

A concern is *active* if its addressing or resolving implies that the system executes one or more activities. Examples include some features such as *modify-component* or *execute-a-use-case* in order to check the *operational-concept* of the system. Such a concern is a system concern and it is used when the system is viewed as an "open box" by the developers.

By definition, a concern has *initiator behavior* if it comes from the software process of the system, where the system is seen like a "black box" and addressing or resolving of it implies, actually, addressing or resolving of active concerns. By example, *operational-concept* concern of the system is an initiator concern.

All the concerns in the software process are crosscutting concerns, if we consider the system to be a base.

The subfacet Collaborators of a concern indicates all concerns and their facets, which help us to define the concern.

As we mentioned above, the Predicate subfacet denotes a test that verifies if an entity is an example of the concern. The test has to be (*i*) faithfully described by Identification, and (*ii*) related to other concerns belonging to the Collaborators facet.

For example, let consider the Definition facet of the *architecture* concern:

Identification: *Global, Opaque, Software process, Aggregate, Initiator, Crosscutting*

Responsible: *Software Architecture Team*

Audience: *Software Management Team, Software Development Team, Software Assessment Team*

Collaborators:

> **Concerns***:* {*the-system's-structure, the-system's-behavior, heuristic-rules-for-the-architecture-design, heuristic-rules-for-the-architecture-evolution*}
>
> **Facets:** {*scope-of-the-system, objectives-of-the-system, requirements-set, development-risks, quality-attributes-of-the-architecture, estimate-scope, schedule*}

Predicate:
resolve *the-system's-structure* **and**
resolve *the-system's-behavior* **and**
resolve *heuristic-rules-for-the-architecture-design* **and**
resolve *heuristic-rules-for-the-architecture-evolution*
and
check *scope-of-system* **and**
check *objectives-of-system* **and**
check *requirements-set* **and**
check *development-risks* **and**
check *quality-attributes-of-architecture* **and**
check *estimate-scope* **and**
check *schedule*

If the Definition facet of a concern has a list of elements, that is the concern has more definitions, there should be no conflict among each other. Namely, they must have the same characteristics (indicated in Identification subfacet), the same responsible and audience, and have different collaborators and predicates.

For example, the *architecture* concern may have another predicate facet:

Predicate:
resolve *use-case-view*
for each Example.*use-case-view*
address *design-view*
if the system is concurrent **then**
resolve *process-view*
if the system is distributed **then**
resolve *process-view* and *deployment-view*
resolve *component-view*
check *scope-of-system*
check *objectives-of-system*
check *requirements-set*
check *development-risks*
check *quality-attributes-of-architecture*
check *estimate-cost*
check *schedule*

Two facets may overlap, that is their intersection is not empty or equivalent, if they contain one or more common concerns. In other words, the Predicate subfacets of the *architecture* concern are overlapping.

3.2. Address

A concern is *addressed* by a life-cycle phase or iteration of the software process if and only if it uses information from the phase or iteration, but this information is not sufficient for resolving the concern. Furthermore, this information should be checked during the major or minor milestones.

There are cases when addressing a concern needs to address or resolve other concerns. By example, the *architecture* concern is addressed in the inception phase of the process software when the *architecture-statement* is resolved. The same concern is addressed in the elaboration phase where the *architecture-baseline* concern is resolved.

Another example is given by *requirement* concerns that are addressed in the iteration of the elaboration phase, by example, where the set of *evaluating-criterion* concerns is resolved.

The Address facet contains the next subfacets:

- Locality: phase or iteration that addresses the concern,
- Collaborators: name of concerns that are addressing or resolving to address this concern,
- Predicate: a test that tells us if any given thing is an example of this concern.

By example, the Address facet of the architecture concern is the following:

Locality: *Elaboration phase*
Collaborators: *architecture-baseline*
Predicate: resolve *architecture-baseline*

3.3. Resolve

First of all, we are saying that a concern is *implemented* if there is a formalism or artifact that comprises descriptive material that include all information about the concern. A construction from some such formalism might be the unit concept from Hyperspace [5].

A concern is *resolved* if contains all needed information to implement the concern. This information must be evaluated with some metrics and obtain the consensus of all stakeholders who are interested in it. The *architecture-baseline* concern is resolved in the elaboration phase of the software process, when it is evaluated and approved by the manager and client. In this stage, we consider that the concern is closed, that is if we want to modify such concern, it enters in a SCO (Software Change Order) process.

The Resolve facet has the following subfacets:

- Locality: phase or iteration,
- Collaborators: names of the concerns that help to resolve this concern,
- Predicate: a test that tells us if some given entities are examples of this concern.

By example, the *architecture-baseline* concern has the following Resolve facet:

Locality: *Elaboration phase*

Collaborators: {*the-system's-scope*, *the-system's-objectives*, *the-project-vision*, *use-cases-set*, *quality-attributes-of-the-architecture*, *design-risks*, *planning-risks*, *work-and-progress-metric*, *change-traffic-and-stability-metric*, *breakage-and-modularity-metric*, *rework-and-adaptability-metric*}

Predicate:

check *scope-of-system* **and** *objectives-of-system*
check the *project-vision* **and** *use-cases-set*
check *quality-attributes-of-architecture*
resolve *design-risks* and *planning-risks*
check *work-and-progress-metric* **and**
change-traffic-and-stability-metric **and**
breakage-and-modularity-metric **and**
rework-and-adaptability-metric

Obviously, Definition, Address and Resolve facets intuitively represent the states of a concern life cycle: defined (created), intermediary (addressed) and closed (resolved).

4. Some Context facets

In this section we present four facets: Life Cycle Phase, Workflow, Artifact, and Role. These facets correspond to points of view, which link through affiliation a concern to others. These facets have the same subfacets:

- Name of the point of view,
- Structure: a set of concern names.

The set of concerns in the Structure subfacet contains the concerns derived from the objectives of the point of view we consider in applying the facet.

4.1. Life Cycle Phase

In case of the Life Cycle Phase facet, the point of view, as we mentioned before, is the phase of software process in which a concern is addressed or resolved. In such phase, a set of concerns are addressed or resolved, namely those concerns that fulfill the objectives of the phase.

For instance, the *architecture-baseline* concern has the next Life Cycle Phase facet:

Name of phase: *Elaboration*

Structure: *vision-baseline*, *plan-baseline-for-construction-phase*, *design-decision* and *demonstrating-architecture*.

4.2. Workflow

Workflow facet focuses on the workflow of iteration and derives the concerns that fulfill the evaluation criteria of iteration.

For instance, the *architecture-baseline* concern has the next Workflow facet:

Name of workflow: *Design*

Structure: *architecture-concept*, *design-components*, *source-component-inventory*, *bill-of-materials* and *executable-components*.

4.3. Artifact

This facet contains those concerns that contribute together with the given concern to the realization of the same artifact. For instance, the *architecture-baseline* concern has the next Artifact facet:

Name of artifact: *Architecture description*

Structure: *architecture-views*, *architectural-interactions*, *quality-attributes-of-architecture*, *architectural-rationale* and *architectural-trade-offs*.

4.3. Role

The Role facet contains the concerns that derive from responsibilities of the stakeholder that addresses or resolves the given concern. For instance, the *architecture-baseline* concern has the next Artifact facet:

Name of role: *Software Architecture Team*

Structure: *architecture-prototyping*, *make/buy-trade-offs*, *primary-scenario-definition*, *primary-sce-*

nario-demonstration, *make/buy-trade-offs-baseline*, *architecture-maintenance*, *multiple-component-issue-resolution* and *quality-attributes-of-architecture*.

5. Conclusions

The concern facet concept is introduced by focusing on the software process and the software system quality concerns.

In the paper, examples have been given "from around" the architecture concern. Besides scope, objectives and requirements, the architecture concern is one of the most important concerns.

5.1. Future work

We are working on the specification all other facets we enumerated at the beginning of the paper.

Other possible questions are: how can we relate the facets of concern with Cosmos model [8] and how can we apply them to the Pirol environment [3]?

Finally, our claim is that the concern facet concept could be useful for systems analysis, too. Our further research will also investigate in this direction.

5.2. Acknowledgments

The author thanks to Professor Luca Dan Şerbănaţi for the paper's review. Also, my thanks to Stephan Hermann for helpful comments about concerns.

6. References

[1] R. Davis and D.B. Lenat, *Knowledge-based systems in artificial intelligence*, McGraw-Hill, USA, 1982.

[2] W. Harrison, H. Ossher and P. Tarr, "Software engineering tools and environments: a roadmap", *A. Finkelstein ed., The future of software engineering, 22nd Int. conf. on soft. Eng.,* Limerick, Ireland, ACM, New York, June 2000, pp. 263-277.

[3] S. Hermann, "Views and concerns and interrelationships. Lessons learned from developing the multi-view software engineering environment PIROL", *Phd Thesis*, Berlin, 2002.

[4] P.B. Kruchten, "The 4+1 view model of architecture", *IEEE Software*, 12(6), Nov. 1995, pp. 42-50.

[5] H. Ossher and P.Tarr, "Multi-dimensional of concerns and the Hyperspace approach", M. Aksit editor, Software Architectures and Component Technology, Kluwer Academic Publishers, 2001.

[6] W. Royce, *Software project management: a unified framework*, Addison Wesley, USA, 1998.

[7] S.M. Sutton, Jr., "Multidimensional separation of concerns in testing, *First workshop on Multi-dimensional Separation of Concerns in Object-Oriented Systems, Conference on Object-Oriented Programming, Systems, Languages and Applications*, Denver, Colorado, Nov. 1999.

[8] S.M. Sutton, Jr. and I. Rouvellou, "Modeling of software concerns in Cosmos", *Proc. of the 1st International Conference on Aspect-Oriented Software Development*, Enschede, The Netherlands, Apr. 2002, pp. 127-133.

[9] P.Tarr, H. Ossher, W. Harrison. and S.M. Sutton Jr., "N degrees of separation: multi-dimensional separation of concerns", *Proc. of the 21st International Conference on Software* Engineering, Los Angeles, CA, USA, May 1999, pp. 107-119.

[10] E.S. Yu and J. Mylopoulos, "Understanding "why" in software process modeling, analysis and design", *16th International Conference on Software Engineering*, Sorrento, Italy, 1994, pp. 159-168.

Aspect-Orientation from Design to Code

Iris Groher
Siemens AG, CT SE 2
Otto-Hahn-Ring 6
81739 Munich, Germany
groher@informatik.tu-darmstadt.de

Thomas Baumgarth
Siemens AG, CT SE 2
Otto-Hahn-Ring 6
81739 Munich, Germany
thomas.baumgarth@siemens.com

ABSTRACT
The AO paradigm focuses mainly at the implementation phases of the software lifecycle and is missing standardized concepts for early stages of the development lifecycle. The term *Early Aspects* refers to crosscutting properties at the requirements and architecture level and this paper addresses the separation of crosscutting concerns at the architecture design phases by offering AML (*Aspect Modeling Language*), a notation for aspect-oriented architecture design modeling that is standard UML conform. Within the notation, crosscutting artifacts are clearly encapsulated and completely kept apart from the business logic to foster their reuse. A clear separation of the AO language dependent from AO independent parts simplifies the support of a number of different AO languages and concepts. To extend the support beyond the architecture phase a code generator is presented addressing low-level design support by offering an automated mapping from design models to programming models to prevent inconsistencies among design and implementation.

Categories and Subject Descriptors
D.2.2 [**Design Tools and Techniques**]: Computer-aided software engineering (CASE)

General Terms
Architecture, Design, Languages

Keywords
AOSD, aspect, crosscutting concerns, early aspects, architecture design modeling, UML

1. INTRODUCTION
Separation of concerns [1] is one of the fundamental principles in software engineering. It states that a given problem involves different kinds of concerns, which should be identified and separated in order to manage complexity and to achieve required engineering quality factors such as adaptability, maintainability, extensibility and reusability. OO software development proved its usefulness regarding the separation of functional concerns of a system. Concerns that crosscut these functional decompositions do not fit equally well into the OO model and have a potentially harmful impact on engineering quality factors mentioned above. Aspect-oriented programming [2] addresses these concepts at the implementation level and offers low-level support for separation of concerns. Aspects are implemented as first-class elements that are expressed in terms of their own modular structure, thus enabling the modularization of crosscutting concerns.

Early Aspects refer to crosscutting properties at the requirements and architecture level ([3]). The term denotes aspect-orientation within the early development stages of requirements engineering and architecture design. This paper focuses on the separation of crosscutting concerns at the high level architecture and the low level design while offering an approach for aspect-oriented modeling and automated code generation. Typically, design artifacts that crosscut an architecture cannot be encapsulated by single components or packages and are typically spread across several of them and therefore also make design hard to understand and maintain. This work addresses the specification of crosscutting concerns at the architecture level in order to maintain the separation of concerns at an early stage in the software development lifecycle. Crosscutting design artifacts can clearly be encapsulated avoiding tangling and scattering.

An extension to UML [4] [5] is presented, without changing its metamodel specification, to achieve standard UML conformity. This helps developers to become acquainted with AO modeling when they are already familiar with OO modeling and UML. A key intention was to offer standard development tool support and interchangeability between various tools. UML is customized by using standard extension mechanisms only. To gain the benefits of code and design reuse of AO software, the ability to reuse aspect and business logic separately is needed. A notation is presented where aspect and business logic are completely kept apart. Thus, both are reusable and at the same time independent of the implementation technology. Within this approach it is assumed that the requirements have already been defined and specified during previous development stages.

To ease the transition from design to implementation and to offer low-level architecture design support, a code generator was developed to support automatic generation of AO code skeletons from design models. This helps developers to focus on models having the code skeletons generated automatically to gain the benefits they are used to in OOSD. Code generation improves developer productivity, ensures syntactical correctness and reduces errors when mapping a model to code. The presented UML notation in combination with the code generator makes AOSD more usable and more efficient for software development by avoiding inconsistencies among design and implementation. Developers can then concentrate on AO design having the code skeletons generated automatically.

The remainder of this paper is organized as follows: Section 2 presents shortcomings of the current state of research on aspect-oriented modeling and describes the need for AO architecture design. Section 3 describes the syntax and semantics of the developed notation. Section 4 presents the automated transition

from design models to implementation models. We conclude with a note of related work and a summary in Section 5 and 6.

2. PROBLEM STATEMENT

The architecture design is an important step within the software development lifecycle. OO design has proved its strength when it comes to modeling common behavior. However, OO design does not adequately address design artifacts that crosscut an architecture. They cannot be encapsulated by single components or packages and are typically spread across several of them and therefore also make design hard to understand and maintain. Crosscutting concerns are present during all phases of a software development lifecycle, leading to code tangling or code scattering during the implementation phase and *graphical tangling* during the design phase. AOSD is still lacking standardized concepts at the design phase that would foster the specification of crosscutting concerns at the high level architecture and low level design. Development of large software systems follows processes that all include activities like requirements engineering, analysis, design and implementations. Following a design methodology like OOD, and focusing on AOP at coding level causes a shift of paradigms between OO design and AO code. This leads to inconsistencies between design and implementation as the AO paradigm is not seamlessly supported during the early stages of the development lifecycle. To avoid the divergence of design models and code, crosscutting concerns must be identified at the requirements and architecture level and carried forward in the implementation phase. Concepts are needed for a seamless integration of AO design and implementation and will be a first step towards an integrated AO development process. To make AOSD more widely accepted, the different phases of an AOSD lifecycle have to be integrated more smoothly by supporting the AO paradigm in every phase. This work includes both, a design notation as well as a code generator for automatic code generation and validation of AO models. Supporting design models and their transition to concrete implementations makes AOSD more usable, more efficient and more accepted among software engineers.

When analyzing OO design, one can see that OO modeling tries to adopt many of the OO programming features for design and analysis. Classes, their structures, and their relationships are identified and generalization and aggregation hierarchies are built. OO design techniques are not sufficient when focusing on the AO paradigm as crosscutting concerns also make design tangled and therefore hard to understand and maintain. When developing an AO modeling approach, the following requirements are obvious:

- A sufficient notation should be simple to understand and straightforward to use for developers who are familiar with common design notations (such as UML).
- Design modeling should be supported by powerful CASE tools to improve developer productivity and to ensure syntactical correctness of the AO model.
- Design notations should support modeling according to the paradigms behind the most common AO approaches and languages.
- Models should be easy to read and offer a clear separation of concerns to avoid crosscutting concerns spanning over many design elements.
- A direct mapping between the notation and supported implementation languages should allow automatic code generation based on the design model.
- The notation should be applicable in real-world development projects and should be part of an integrated AO development process.

This work can be seen as a step towards a standardized way to capture aspects at the design phase of an AO development process. Existing approaches and prototypes are well aware of the fact that aspect-oriented modeling is a critical part of AOSD. Obviously, to obtain an AO development lifecycle, the gap between AO requirements engineering and AOP has to be filled. This work makes a contribution to the problem of bridging this gap.

3. ASPECT MODELING LANGUAGE

This work specifies an approach for AO modeling to address the specification of crosscutting concerns at the architecture level in order to maintain the separation of concerns at an early stage in the software development lifecycle. A key intention is to offer standard development tool support and interchangeability among various CASE tools, thus an extension to UML was developed without changing its metamodel specification to achieve standard UML conformity. Using UML as a modeling language improves developer productivity and offers high acceptance, as it is the industry-standard modeling language for the software engineering community. When using standard UML for aspect-oriented modeling, developers do modeling by using familiar tools and environments to gain all the benefits they are used to in OO design. UML is an extensible modeling language that enables domain-specific modeling which raises its suitability as a modeling language for supporting aspect-oriented modeling. Another important goal was to gain the benefits both of code and design reuse of AO software, including the ability to reuse aspect and base elements separately. Thus, aspects and base elements should be completely kept apart and independent of the implementation technology in order to simplify the replacement of the AO language. A clear separation of the language dependent crosscutting parts eases the support of many different AO languages and concepts. This work focuses on adopting AspectJ [6] [21] concepts for the implementation language dependent parts of AML; the support of other AO concepts (such as Hyper/J [18] [19] [23]) is considered and part of some future work.

AML considers the fact that crosscutting concerns tend to affect multiple classes in a system. Sine a concern itself can consist of several classes and since all of these classes may be associated with the class the concern crosscuts, the module construct for a concern should be higher-level than a class. Otherwise associations modeled on class-level would supersede the logical grouping of the classes belonging to one concern. This would make the design models hard to read and lead to graphical tangling of crosscutting concerns instead of a clear separation.

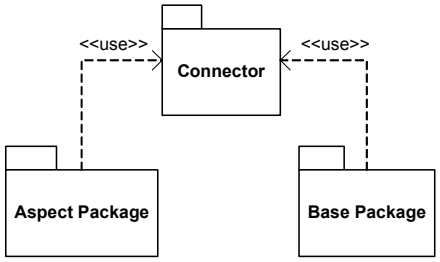


Figure 1: Package Level (De) Composition

Figure 1 provides an overview of the notation and its focus on package-level decomposition. AML includes a *base package* (containing the business logic), an *aspect package* (containing the crosscutting concern) and a *connector* to link aspects and base elements. This separation enables high reusability of the aspect and base elements since the connector is the only crosscutting element. Focusing on UML packages as a central decomposition unit leads to design models that are easy to read, as they avoid *graphical tangling*. Additionally, the connector encapsulates the underlying implementation technology (e.g. AspectJ). The aspect can be modeled independently of any design it may potentially affect. The connection between base design and aspect design is specified separately. Support of different AO technologies is therefore rather simple and straightforward, as it is only the connector's syntax that has to be changed.

The aspect package provides a graphical representation (class diagram) of the static view of a particular crosscutting concern and is, along with the base package, one of the OO parts of the AO model. The base package contains the business logic of the system and can be modeled without considering any crosscutting concern that may potentially affect the system. Similar to the aspect package, the base package can contain any valid UML model that describes the business logic of the desired system. There is no direct relationship among the aspect package and the base package; their relationship is only defined through the connector package containing the rules for the later recomposition of aspects and base elements. As AspectJ is currently the best known AO language, all connector semantics presented here have been developed according to AspectJ's connection model.

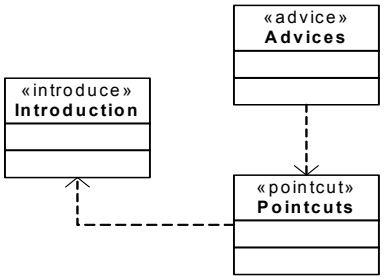


Figure 2: AspectJ specific Connector

As shown in Figure 2, the AspectJ-specific connector package can contain the following classes that conform to the concepts AspectJ offers for the specification of weaving rules:

1. The *Introduction* class, which defines the rules for AspectJ's introduction mechanism.
2. The *Pointcut* class, which defines execution points in the control flow of the program.
3. The *Advice* class, which defines the code to be executed at the pointcuts defined in the *Pointcut* class.

All classes contain operations with special semantics to specify how aspect and base elements have to be recomposed. The complete syntax of the AspectJ specific connector will not be presented here; the following example should provide a view of how the notation can be used and shows some of the most important constructs.

The example in Figure 3 shows how to model an aspect related to tracing to give some guidelines and indications on how to use our notation.

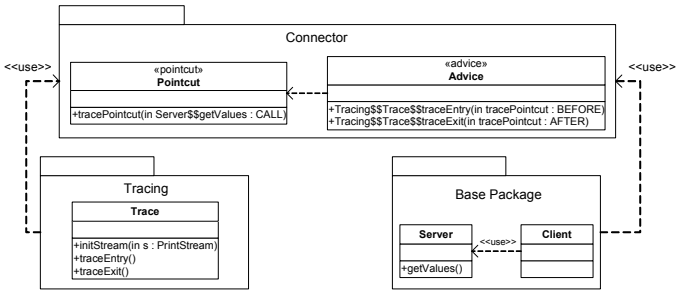


Figure 3: Tracing Design Example

Every time the user performs an invocation on the Server, the action should be traced. Both, tracing aspect and business logic, are independent from each other, no connection is modeled inside. The connector, specifying the weaving rules, includes program execution points (pointcuts) and actions performed at those points (advices). The pointcut (*tracePointcut*) is triggered every time the client invokes the *Server.getValues()* method. The action to be performed before the method call is tracing the entry of the method (*Trace.traceEntry()*) and after the method call is tracing the exit of the method (*Trace.traceExit()*). As within Java [7] dots are not allowed within operation names, it was soon discovered that dots could not be used to separate packages from classes and members. Therefore, we decided to separate them from each other using "$$". The "$" character can be found quite often within AML and it has been chosen as it is rarely used by developers within class or member names.

AML is a simple and powerful notation for aspect-oriented modeling. In order to reduce errors when mapping models to code and offer low-level architecture design support, a code generator is developed which is presented in the next chapter.

4. CODE GENERATION

To extend the support beyond the architecture phase a code generator is presented addressing low-level design support by offering an automated mapping from abstract design models to programming models. This low-level architecture design support prevents inconsistencies among design and implementation and helps developers concentrate on AO design having the code skeletons generated automatically. AspectJ has been chosen to be the target language, as it is the AO language that is mainly used at present. The semantics of the connector have been designed according to AspectJ concepts including concrete mapping rules between model and code. Before generating code skeletons, the model is validated for syntactical and semantical correctness. It is even possible for developers to have the model validated without generating code afterwards.

The development of the code generator is divided into two parts (see Figure 4):

1. The *model validation* part validates an AO design model for syntactical and semantical correctness (e.g., the existence of referenced pointcuts). It is possible for developers to have the design model validated without generating code afterwards.
2. The *code generation* part generates AspectJ source code for a validated AO model.

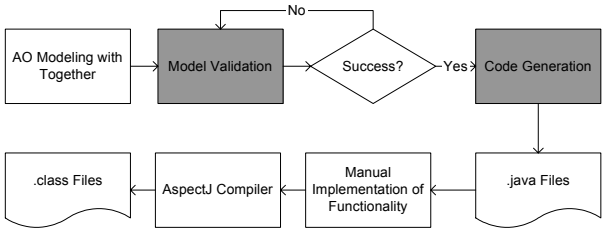


Figure 4: Flowchart of an AO Development Process

The CASE tool *Together* [22] from Borland is an enterprise development platform enabling application design, development, and deployment. It is extensible through an open Java API offering the possibility to develop custom software that plugs into the Together platform in the form of modules. The open API is composed of a three-tier interface that enables varying degrees of access to the infrastructure of Together. Altogether, Together's open API offers a lot of very powerful concepts for the manipulation of UML models and has therefore been chosen for the development of the code generator. The tool automatically validates and generates the OO parts of the model (aspect and base elements), the validation and code generation of AO parts (i.e. connector elements) is implemented as modules that plug into the Together platform.

Aspect elements and base elements map to Java source code. The aspect package and the base package are the OO parts of the notation. Connector elements map to AspectJ source code. The connector package consists of the AO part of the notation, linking aspect package and base package. To ensure syntactical and semantical correct AspectJ files that can then be compiled with an AspectJ compiler mapping rules have been defined between the notation and AspectJ concepts.

```
public aspect TracingAspect {
   pointcut tracePointcut () :
   call ( * BasePackage.Server.getValues(..));

   before () : tracePointcut () {
      System.out.println ("Entering method…");
   }
   after () : tracePointcut  () {
      System.out.println ("Leaving method…");
   }
}
```

Listing 1: Tracing Aspect with Copied Code

```
public aspect TracingAspect {
   pointcut tracePointcut () :
   call ( * BasePackage.Server.getValues(..));

   before () : tracePointcut () {
      Trace t = new Trace (/*parameters*/);
      t.traceEntry (/*parameters*/);
   }
   after () : tracePointcut  () {
      Trace t = new Trace (/*parameters*/);
      t.traceExit (/*parameters*/);
   }
}
```

Listing 2: Tracing Aspect with Instantiation

Listing 1 shows the aspect `TracingAspect` that is generated when code parts of the crosscutting concern are copied into the AspectJ file. The difference between Listing 1 and Listing 2 lies in the specification of actions being performed at pointcuts. In Listing 1, the code to be executed (declared inside the aspect package) is copied into the AspectJ file, whereby in Listing 2 the relevant classes are instantiated and the appropriate methods are called. When instantiating classes (as shown in Listing 2), the appropriate constructor and method parameters have to be inserted by the user which is not necessary when copying the code. The user can choose between the two options when generating the code (copied code and instantiation).

The generation of AspectJ code is a one-time/one-way generation, possible future extensions could support roundtrip engineering including reverse engineering for aspect mining.

5. RELATED WORK

Related aspect-oriented design approaches proposed to provide support for crosscutting concerns at the architecture design level are based on Composition Patterns [12] [13] [14], Aspectual UML [10] [11] and other UML based modeling approaches [15] [16] [17].

The *Composition Pattern* approach combines UML templates with a subject-oriented model. The notation focuses on package level decomposition and "binds" crosscutting concerns with business logic classes with the help of binding relationships between the decomposed packages. The modeling language is based on standard UML extension elements like stereotypes, constraints or templates, which are supported on all standard UML conform CASE tools. The Composition Pattern notation does not provide an explicit notation for advice specifications, instead advices are expressed through state diagrams. A designer is forced to provide an additional state diagram for each execution point. While modeling the notation requires switching between object and state diagrams. The notation might be sufficient for small designs, but gets complex and hard to read for larger systems.

Aspectual UML separates the design in aspectual collaboration modules and all linking rules in a separate "connector" package. Compared to Composition Patterns, the notation enhances the separation of base classes, crosscutting concerns and binding rules in independent modules. However the UML notation of this approach introduces two new relationships on package-level (package inheritance and package adaption), which are unknown to standard UML and will be problematic to realize in existing CASE tools. With binding by delegation and advice weaving, Aspectual UML provides two powerful binding concepts, but is lacking other AO concepts like introduction and full support for all AspectJ-like join point definitions.

Many of the other modeling approaches [15], [16], [17] are based on class level decomposition. This decomposition level does not seem ideal, since often several classes are involved in one crosscutting concern. There is a danger that class level decomposition may lead to redundant notations and graphical tangling in the design models. [17] complies to standard UML, however the tight coupling of specific notations to AspectJ concepts, will make it difficult to support other aspect-oriented languages (e.g. HyperJ). [15] remains unspecific, how advice or pointcuts can be modeled, It mainly provides concepts for static crosscutting of operations. [16] provides limited modeling capabilities for crosscutting concerns e.g. advices can only be expressed through state-chart diagrams.

6. SUMMARY AND FUTURE WORK

This work addresses the AO development process from the high level architecture to the low level design by presenting an approach for aspect-oriented modeling and automated code generation. When considering the requirements defined in chapter 2, the following goals have been reached:

- An approach for high level architecture design, called AML, has been developed to enable separation of concerns at the design level of an AO development process. Within this approach it is assumed that the requirements have already been defined and specified during previous development stages.
- Since AML is UML conform, any CASE tool that supports UML modeling can be used.
- Aspects and base elements are completely kept apart; they are connected via a special language-specific connector element that encapsulates the underlying implementation technology. Any desired AO technology can be supported; it is just the connector's syntax and semantics that have to be specified.
- Both, aspects and base elements, can be reused separately as the connector is the only crosscutting, language-dependent part. This sort of encapsulation offers a logical grouping of all classes belonging to one concern and eases the readability of design models as avoiding graphical tangling.
- To offer low-level architecture design support, a code generator has been developed to improve productivity and reduce errors when mapping model to code.

The work can be seen as a first step towards a simple and powerful modeling approach that fosters support from existing CASE tools since it is based on standard UML. AML in combination with the code generator should make AOSD more usable and more efficient for software development. The assumptions about the usefulness of the notation and the AO code generation have to be proven in the near future when using it in business development projects.

After evaluating the prototype's features in real world development projects, some concepts may have to be added (e.g. complex relationships between aspects). Another important feature will be a complete CASE tool support including roundtrip engineering for aspect mining. As Together plans to support the development of modules offering roundtrip engineering features in the next version, this should be included in the next version of the code generator.

The connector package encapsulates the underlying implementation technology. Currently, the syntax and semantics of an AspectJ specific connector type are defined. This sort of encapsulation eases the replacement of the AO language, the support of different technologies and language concepts (such as Hyper/J [18] [19] [23]) will be part of some future work. An automated code generation for different languages is rather straightforward, too. It is only the code generator's mapping rules that have to be changed.

There are still many issues to be solved until efficient AO development support comparable to current OO support is established. When offering an integrated development process, the gaps between the early phases and AO programming have to be filled as so far the paradigm focuses mainly at the implementation level. There is still a lot of challenging research to be done in the future until the paradigm is widely accepted and developers are aware of the benefits AOSD offers.

7. ACKNOWLEDGMENTS

We thank all researchers who explored the idea of crosscutting concerns to a state we could build on to gain a first experience in development projects, especially the group around AspectJ, Caesar [8] and Hyper/J. We also want to thank our colleagues at Siemens for their valuable feedback.

8. REFERENCES

[1] E.W. Dijkstra. A Discipline of Programming. Prentice Hall, Englewood Cliffs, NJ, 1976.

[2] G. Kiczales et al. *Aspect-Oriented Programming.* 2001

[3] Early Aspects Homepage, http://early-aspects.net

[4] J. Rumbaugh et al. *The Unified Modeling Language Reference Manual.* Addison-Wesley, Massachusetts, USA, 1999.

[5] The Unified Modeling Language, version 1.4, http://www.uml.org/

[6] AspectJ Homepage, http://www.eclipse.org/aspectj/

[7] Java Homepage, http://java.sun.com/

[8] CAESAR project, http://caesarj.org/

[9] E. Gamma et al. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, Massachusetts, USA, 2001.

[10] S.Herrmann. *Composable Designs with UFA*. Submission to AOSD 2002.

[11] S. Herrmann, M. Mezini. *Aspect-Oriented Software Development with Aspectual Collaborations*. Submission to ECOOP 2002.

[12] S. Clarke et al. *Composition Patterns: An Approach to Designing Reusable Aspects.* Workshop on AO Requirements Engineering and Architecture Design on AOSD 2002, Enschede, Netherlands, 2002.

[13] S. Clarke, R. J. Walker. Separating Crosscutting Concerns Across the Lifecycle: From Composition Patterns to Aspect J and Hyper/J

[14] S. Clarke, R. J. Walker. Towards A Standard Design Language for AOSD

[15] J. Suzuki, Y. Yamamoto, Extending UML with Aspects: Aspect Support in the Design Phase, Submission for Workshop at ECOOP 1999

[16] J.L. Herrero, F. Sanchez, F. Lucio, M. Torro, Introducing Separation of Aspects at Design Time, Submission for Workshop at ECOOP 2000

[17] R. Pawlak et al. *A UML Notation for Aspect-Oriented Software Design.* Workshop on AO Requirements Engineering and Architecture Design on AOSD 2002, Enchede, Netherlands, 2002.

[18] Hyper/J, http://www.alphaworks.ibm.com/tech/hyperj/

[19] Hyperspaces, http://www.research.ibm.com/hyperspace/

[20] I. Jacobson et al, *Unified Software Development Process*. Addison-Wesley Professional, February 1999.

[21] G. Kiczales. E.Hilsdale, J. Hugunin, M. Kersten, J. Palm and W. Griswold. *An overview of AspectJ*. In Proc. Of 15th. ECOOP, LNCS 2072, p. 327-353, Springer-Verlag, 2001

[22] Together Homepage, http://www.borland.com/together/

[23] H. Ossher and P. Tarr. *Multi-Dimensional Separation of Concerns and the Hyperspace Approach*. In Proceedings of the Symposium on Software Architectures and Component Technology: The State of the Art in Software Development. Kluwer, 2000.

Aspect-Oriented Context Modeling for Embedded Systems

Tomoji Kishi
Japan Advanced Institute of Science and Technology
1-1 Asahidai, Tatsunokuchi
Ishikawa 923-1292, Japan
tkishi@jaist.ac.jp

Natsuko Noda
NEC Corporation
2-11-5 Shibaura, Minato-ku
Tokyo 108-8557, Japan
n-noda@cw.jp.nec.com

Abstract

One of the characteristics of embedded systems is that they depend on their contexts, i.e. they react to the context changes, and their behaviors are constrained by the contexts. In modeling such context-dependent systems, we have the following difficulties: 1) As we have to treat various kinds of contexts such as logical contexts and physical contexts, it is difficult to model them from one point of view. 2) By its nature, the model for internal processing tends to depend on the model of external contexts, and this makes modifiability and extensibility of the model bad. 3) Though we have to make distinction between contexts and values shown by sensors, as sensors are the means of capture contexts, we sometimes confound contexts with sensors and make tightly coupled model. In this paper we propose an aspect-oriented context modeling technique for embedded systems, in which we show a strategy to model embedded systems that have context-dependent nature. The aspect-oriented context model is developed in the early stage of development and can be used as a reference model for software development in the following stages.

1. Introduction

With the development of information technology, embedded systems become more and more large and complicated. They have to control variety of hardware, they are required to show higher performance and reliability, they are connected to network to communicate with other equipments and information systems, and so forth.

These tendencies change the development style of embedded systems. Compared with the development style of business applications, that of embedded systems used to be ”implementation centric”. However, as embedded systems become larger and more complicated, we have to put much emphasis on early stages of development, and to that end, modeling becomes more and more important for the development of embedded systems.

One of the characteristics of embedded systems is context dependency. Here, context means any environment in which the system is operated. For example, it reacts to the context change (e.g. if it becomes dark, then turn on the light) and its behavior is constrained by the context (e.g. even if heating button is pressed, it does not start heating if the temperature is too high). These context-dependent systems have three important conceptual parts. One part consists of sensors that show some values that reflect phenomena in the external worlds, one part consists of logics to decide contexts based on these sensor values, and the other part consists of internal processings that are triggered or constrained by these determined contexts. For example, in vehicle, sensors for a hand-brake and a shift-gear detect their position, then the system determines whether the vehicle completely stops or not, and internal processing abort services that interfere driving.

As these three parts are essential for context dependent systems, requirements for such systems are defined in terms of these three parts, and the software architecture of the system reflects them. Furthermore, as each part relates to different properties of the system, each part requires different kinds of check and verification. Therefore it is desirable to develop model in the early stage that capture these three parts, and utilize the model as a reference model for architecture design, checking/verification, and so on. One of the problems in modeling such context dependent systems is that it is not straightforward to localize these three parts, as we have to handle various kinds of contexts such as logical contexts and physical contexts, and they have complicated relationships.

In this paper, we propose an aspect-oriented context modeling for embedded systems that have these context-dependent nature. In the technique we show a strategy to model embedded systems utilizing aspects. In our technique, we introduce a new modeling element, ”inter aspects relation”, that explicitly define the relationship among as-

pects.

2. Context Modeling

In this paper, we call any environment in which a system is operated as **context**. There are various kinds of contexts, and they may change time to time, and place to place. For example, considering the systems for automobile, the followings are some examples of possible contexts:

- the temperature, humidity, brightness around the vehicle.
- the position, velocity, acceleration of the vehicle.
- the engine status, door position, gear position.
- the intensity of radio field of cell phone and wireless LAN, the number of satellites they can capture for GPS measurement.
- the services available from outside information systems (for example, each gas station, parking lot, or toll booth may provides different services).

Systems cannot recognize contexts directly, and they utilize **sensors** as the means to capture these contexts, and judge and conjecture the current contexts. We may use multiple sensors to capture a context, same sensors may be used to capture different contexts and there may be multiple means to capture a certain context. Therefore, it is important to distinguish between contexts from values shown by sensors.

A **context-dependent system** is a system that has internal processing that depends on its contexts. The followings are the typical characteristics of the internal processing of a context-dependent system:

- **reaction:** the system reacts to their external events and/or context changes, i.e. contexts trigger the internal processing. e.g. if the temperature arises, then the system starts cooling.
- **constraint:** the system's behavior is constrained by the contexts, i.e. the system determines the type of the behavior, considering the current context. e.g. even if the operator sends cooling command, the system does not start cooling, if the temperature is too low.

In analyzing and designing context-dependent systems, we have to model contexts and relationships among contexts and internal processing. We call these modeling activities as **context modeling**.

3. Example: Onboard Software Management System

In this section, we introduce Onboard Software Management System (OSMS for short), and examine the problems in context modeling.

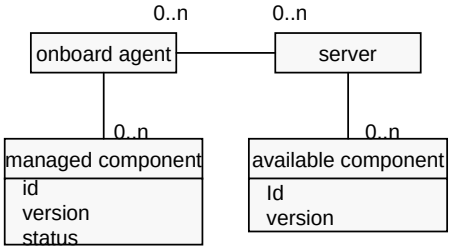
3.1. Basic Functionality

OSMS is the system to manage software components that are installed on the vehicle onboard system. Recently the size of software for onboard system becomes larger, it is required to manage these software components like those on personal computers and workstations. Namely, it becomes necessary to update and replace the software components when a problem arises and/or newer versions are released.

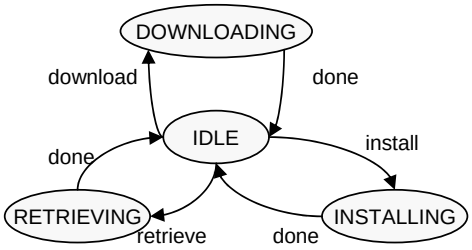
In this paper, we examine the OSMS that has the following functionalities.

- OSMS can communicate with central servers by cell phone. It can also communicate with local servers that are placed at local points such as gas stations and toll booths by wireless LAN. Even if OSMS communicates with the central server, when it comes into the area where wireless LAN is available, the system can switch the communication media.
- In the central servers and local servers, there are software components for downloading. Each server may have a different set of software components, as it is difficult to deliver the same version of software to all the servers simultaneously.
- OSMS can get the list of available software components from servers, via cell phone or wireless LAN. Based on the operation of the driver or passengers, OSMS can download the software components, and installs them on the vehicle onboard system.

Figure 1 shows the basic model of OSMS. Class "server" resides on the server side, and stores and manages the available software components. Class "available component" is the information about software component that is placed on the server. Class "onboard agent" resides on the vehicle side, and manages downloading and installing of the software components. Class "managed component" is the information about the available software component for vehicle sides, and its status indicates that it is downloaded or not, and it is installed or not. The "onboard agent" downloads the information about the available components (retrieve), downloads the components (download) and installs the downloaded components (install).



(a) class diagram



(b) state diagram for onboard agent

Figure 1: Basic Model of OSMS

3.2. Context Dependency

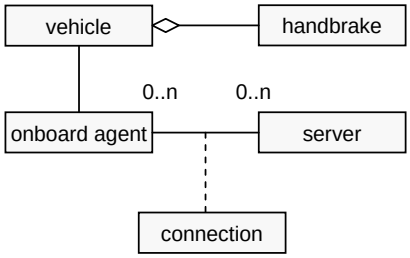
OSMS has the context-dependent features. The followings are examples of the context dependencies:

- Even if driver/passengers specify install command, ”onboard agent” does not start installing if the vehicle is not in the safe condition (typically vehicle is running). It is because installing new software components may causes dangerous situation during the driving.
- If the vehicle enter the unsafe condition (typically vehicle start running) during installation, ”onboard agent” aborts installation.
- If the communication path between ”onboard agent” and ”server” is disconnected during downloading, ”onboard agent” preserves the status, and resumes downloading if the connection is established again. As connected server may be changed (because vehicles are moving), ”onboard agent” has to check whether or not the same component is available from the new server.

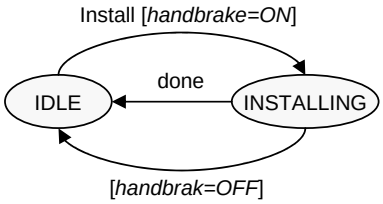
As described above, OSMS has to consider their contexts related to vehicle status (running or not), communication status (connected or disconnected) and servers status (current set of downloadable components).

3.3. Problem

Figure 2 shows an example of OSMS model (part) considering the context dependency. Here, in order to model contexts, class ”vehicle”, ”handbrake” and ”connection” are



(a) class diagram



(b) State diagram for onboard agent (installing part)

Figure 2: OSMS Model considering Context

introduced, and ”onboard agent” detects vehicle status by means of ”handbrake”.

Based on this model, we examine problems of context modeling.

various kinds of contexts

OSMS depends on various kinds of contexts: Server status is a logical context in which we are interested in available components provided by the connected server. On the other hand, communication status is a physical context that relates to connection/disconnection and bandwidth of the communication path.

Though these contexts are captured from different points of view, it is not straightforward to model them independently, because they relate each other. For example, the server status model includes logical connections to servers, and communication status model includes physical connections to servers. These two kinds of connections are different aspect of vehicle-center relationship, i.e. they are not exactly the same but they have relationship. It is common for context dependent systems to handle multiple contexts, and these contexts share the same objects, or they relate to different aspects of the same objects, and this makes the context modeling difficult.

strong coupling with context

As shown in Figure 2 (b), internal model directly depends on contexts. If we change the referred contexts, that causes direct effects on internal model. In the current version of the model, the state diagram refers the status of ”handbrake” in the guard condition. If we try to refer other

conditions (such as vehicle velocity and the type of operator), we have to change the model.

As stated in the previous item, a context dependent system handles different contexts that relate each other, we can easily introduce complicated relationship among contexts and internal processing. For example, internal processing may detect disconnection with the server in terms of server context (logical disconnection) or communication context (physical disconnection), but partial order of these two disconnections can change depend on how to model two contexts and relationship among them.

confound sensors with context

In Figure 2, we do not distinguish between contexts and sensors, and the state diagram of ”onboard agent” directly depends on sensors (”handbrake”). It is not good for the modifiability and extensibility of the model, because there are multiple ways to capture the same context (such as handbrake, brake, gear position, and velocity), and we may change the means, even if we are to capture the same context.

4. Aspect Oriented Context Modeling

In this section, we examine the application of aspect-oriented technology to context modeling.

4.1. Basic idea

Aspect-oriented technology is a technology, in which we use aspect as a new mechanism for modularity, that corresponds to a certain concern [1]. In this paper, we utilize the aspect as a mechanism to modularize models; Entire model consists of one or more aspects, and each aspect includes a class diagram and state diagrams relate to the concern.

The basic idea of applying aspects to context modeling is as follows:

- As we have to handle various kinds of contexts, we utilize aspects to model each context. Each aspect is defined from a certain point of view appropriate for each context. As a same modeling target (such as vehicle) may appear in multiple contexts, it is suitable use aspect rather than ordinary UML package.
- As mentioned in the previous section, it is not desirable to make models for ”internal processing”, models for ”context” and models for ”sensor” tightly coupled. We also utilize aspects to disjoint these three categories of model, and introduce three ”stereotypes” for aspect (<<process aspect>>, <<context aspect>> and <<sensor aspect>>) to explicitly express these three categories of model.

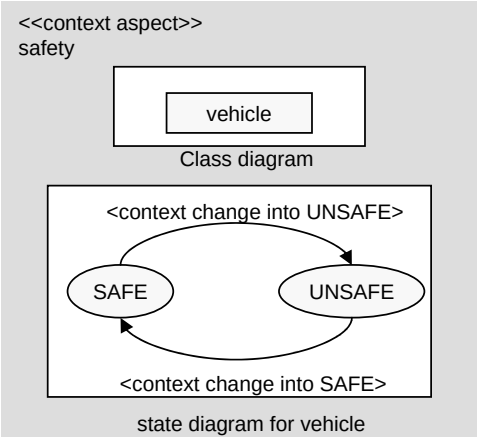


Figure 3: Context Aspect (safety)

- In aspect-oriented programming, mechanisms that relate different aspects are pre-defined as the language constructs. However, in aspect-oriented modeling, there are not common and widely accepted mechanisms yet. Therefore, we explicitly model the relationships among aspects using modeling element ”inter aspects relation”.

4.2. Application of Aspect-Oriented Technology

We will explain how we apply aspect-oriented technology to context modeling, using OSMS example.

various kinds of contexts

We define each context as an aspect, in which we view the system from a specific point of view. These aspects have stereotype <<context aspect>>.

- Vehicle status is about the safety of installation, and modeled in ”safety aspect”.
- Server status is about the logical connection among ”onboard agent” and ”server”, and modeled in ”server aspect”.
- Communication status (connection and bandwidth) is about the physical status of communication, and modeled in ”communication aspect”.

Figure 3 shows a context aspect (”safety” aspect). This figure denotes that ”safety” aspect of stereo type <<context aspect>> includes a class diagram (that defines class ”vehicle”), and a state diagram for ”vehicle” class.

strong coupling with context

In order to avoid strong coupling among contexts and internal processing, we define them as independent aspects. Aspects that represent context have stereotype <<context

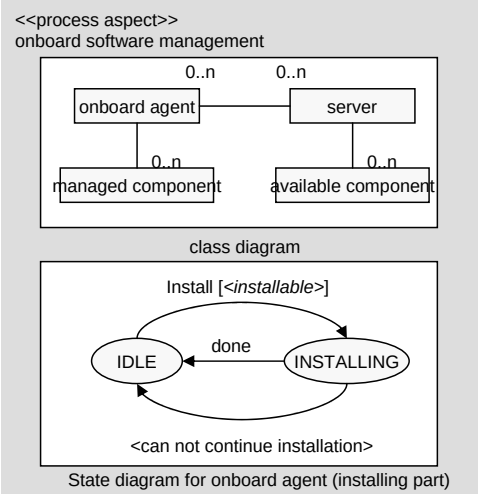


Figure 4: Process Aspect (onboard software management)

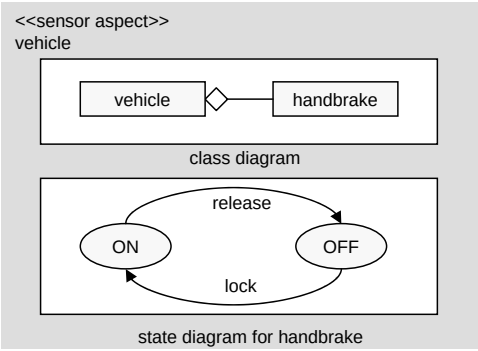


Figure 5: Sensor Aspect (vehicle)

aspect>>, and aspects that represent internal processing have stereotype <<process aspect>>. If we directly refer contexts in process aspect, internal processing and context become strongly coupled. Therefore, in process aspect, we use abstract context to avoid the direct coupling. In OSMS example, we use abstract names ”installable” and ”can not continue installation” to avoid direct coupling. These abstract names will be related to other aspects using inter aspects relation explained shortly.

Figure 4 shows a process aspect (”onboard software management” aspect).

confound sensors with context

In order to distinguish contexts from sensors, we also define them as independent aspects. Aspects that represent sensors have stereotype <<sensor aspect>>.

Figure 5 shows a sensor aspect (”vehicle” aspect). Note that ”vehicle” is appeared in ”safety” aspect and ”vehicle” aspect, but it is obvious that these two classes are modeled from different point of view.

4.3. Inter Aspects Relations

In order to model entire system, we have to relate independently defined aspects[10]. In aspect-oriented programming, we define each aspect using language constructs such as ”advice”, and the weaver merges aspects into a program.

In modeling field, on the other hand, there are no common and widely accepted mechanisms yet. Therefore, in this paper, we examine what kinds of relationships among aspects are required in context modeling, and introduce a modeling element ”inter aspects relation” to explicitly define relationships among aspects.

Based on OSMS example, we observe that relationships listed in Table 1 are necessary for context modeling.

Table 1: Inter Aspects Relation appeared in Context Modeling

response	Context changes in ”context aspect” trigger the activation of processes in ”process aspect”. E.g. if it becomes ”SAFE” in ”safety aspect”, then abort ”INSTALLING”.
condition	Processes in ”process aspect” refer contexts defined in ”context aspect”, and determine the behavior based on those. E.g. even if ”Install” event comes in ”onboard software management” aspect, it does not go into ”INSTALLING” if it is ”UNSAFE” in ”safety” aspect.
abstraction	Context aspect determines contexts by referring the contexts in other ”context aspects” and status in ”sensor aspects”. E.g. ”safety” aspects determines ”SAFE” and ”UNSAFE”, based on status in ”handbrake” aspect.
consistency	Keep consistencies among aspects. E.g. if physical connection in ”communication aspect” becomes disconnected, then delete the logical association between ”onboard agent” and ”server”.

In our context modeling, we introduce inter aspects relations to express these relationships. An **inter aspects relation** is a modeling element by which we define the dependency among modeling elements in aspects. Each inter aspects relation imports modeling elements defined in two or more aspects, and relates these elements using relationship shown in Table 2.

Figure 6 is an example of inter aspects relation, in which relations between ”onboard software management” aspect and ”safety” aspect are defined. Inter aspects relation is described as rectangle, and ”imports” the data elements from two or more aspects. (In the figure, we abbreviate class dia-

Table 2: Basic Relationship used in Inter Aspects Relation

refer	A model element in one aspect is defined in terms of other model elements in other aspects. E.g. a guard condition is defined in terms of attributes and states defined in other aspects.
trigger	A behavior in one aspect triggers the behaviors in other aspects. E.g. update of attribute value, invocation of method and firing of transition in one aspect trigger those in other aspects.

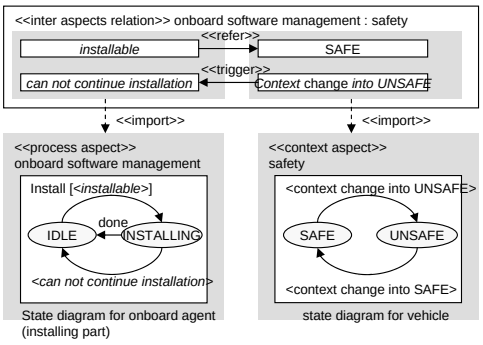


Figure 6: Inter Aspects Relation (response, condition)

gram in ”onboard software management” aspect and ”safety aspect”). In the rectangle, there defined dependencies (refer, trigger) among data elements imported from these aspects. Here, ”installable” in ”onboard software management” aspect is defined in terms of conditions defined in ”safety aspect (”SAFE” or not), and the transition from UNSAFE to SAFE in ”safety” aspect triggers the transition from INSTALLING to IDLE in ”onboard software management” aspect.

Figure 7 shows another example in which relations between ”safety” aspect and ”vehicle” aspect are defined. The states of ”vehicle” (SAFE, UNSAFE) are defined in terms of the status of ”handbrake” (ON, OFF).

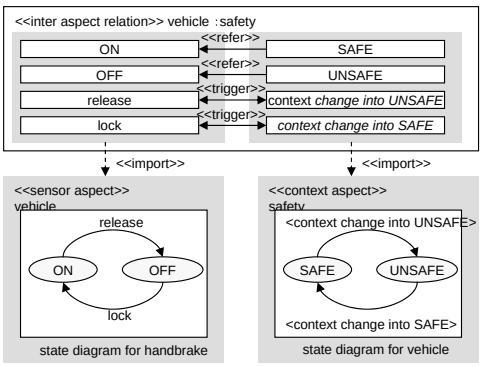


Figure 7: Inter Aspects Relation (abstraction)

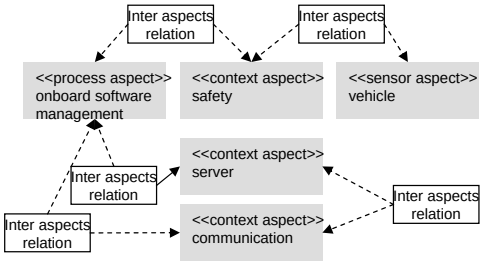


Figure 8: Overall Relations among Aspects for OSMS

The transitions in ”vehicle” aspect and transitions in ”safety” aspect triggers each other: As ”vehicle” aspect is concrete side, and ”safety” aspect is abstract side, the transition is firstly about to fire in ”vehicle” aspect. Before actually fire transition, it triggers the transition in ”safety” aspect, and if it actually fires, then it triggers back the transition in ”vehicle” aspect. This kind of interactions is necessary to keep the consistency between concrete side and abstract side, because in ”safety” aspect, there may be some guard conditions are given, and triggered transition may not fire. Figure 8 shows the overall structure of OSMS example.

Note that, the inter aspects relation is introduced in order to explicitly define relationships among aspects in order to overview the entire structure, as dependency design (how to reduce the strong coupling among models for internal processing, models for context and models for sensor) is one of the most important issues in context modeling. Therefore, concrete mechanisms to realize ”refer” and ”trigger” relations are not our main focus.

5. Discussion

In this section, we discuss a few technical issues.

5.1. Context Modeling

We examine advantages and disadvantages of our aspect-oriented context modeling.

- As we have to handle variety of contexts, it is good to model them from different points of view. On the other hand, if we develop aspects without careful consideration, the entire model becomes just a gathering of small pieces of aspect, and inter aspects relations become complicated. It is important to work out a strategy for deciding aspects and developing entire structure of the model.
- As each aspect basically does not depend on other aspects, we can avoid strong coupling among internal processing, contexts and sensors. This is a good characteristic, because in developing context dependent systems, we need many iterations before fixing

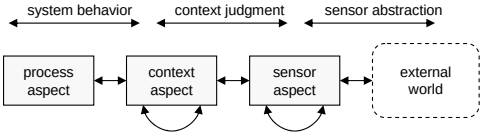


Figure 9: Reference Model for Checking Validity

what contexts should be captured and what sensors should be used to determine contexts.

For example, consider the situation in which we want to refine SAFE condition to be determined not only by handbrake but also by gear position. In our model, we just add ”gear” class to ”vehicle” aspect, and modify ”refer” relation for SAFE condition (Figure 7). We do not need to change the ”onboard software management” aspect.

- As pointed out previously, relationships among contexts and sensors are not one to one. Consider the situation in which the same sensor is used to capture different contexts, it is difficult to model sensor from one point of view. For example, if we use vehicle velocity to determine safety, it may be abstracted as safe speed and unsafe speed. However, if we use vehicle velocity to determine the time to distance, such abstraction does not work. If we use aspects, we can model them straightforwardly, i.e. define two aspects, one for safety, the other for time calculation.

5.2. Checking the Validity

It is important to check the validity of developed model. In order to effectively check the model, it is indispensable to have the good strategy of checking. There are some techniques to check the validity, such as review, test, simulation and model checking technologies[5], and we have to apply right technologies to right place. In context dependent systems, each conceptual part has different sets of properties to be checked, and we can use our context model as a reference model for checking the validity.

Figure 9 shows a reference model for checking the validity based on our context model. This reference model shows that there are three categories for checking the validity in context-dependent system. The followings are the rough sketch of the checking strategy, based on this reference model.

- **sensor abstraction**

  In the context model, ”sensor aspect” and relationship (via inter aspects relation) among them relate to abstraction of the external world.

  Here, we have to decide how to abstract sensor value (e.g. a sensor may be modeled to have two values ON and OFF) and relationship among them (e.g. sensorA and sensorB do not have value ON at the same time).

  To check if the abstraction is adequate or not is difficult, because it is a matter of recognition of the real world. For example, a sensor may be modeled to have three values ON, OFF and ERROR, instead of ON and OFF. To determine which abstraction is adequate must be checked by test (i.e. actually operate the system in the real world), or by expert’s review.

  On the other hand, it is relatively easier to check whether or not the model correctly reflects the determined abstraction. For example, using model checker, we can check important property such as ”sensorA never becomes ON whenever sensorB is ON, vice versa”.

- **context judgment**

  ”Context aspects”, relationship among them and relationship among ”context aspects” and ”sensor aspect” relate to judgment of context. Here we decide contexts to be captured, and determine these contexts by means of other contexts and sensors.

  It is difficult to check if contexts are correctly identified. For example, even though we identify the ”SAFE” and ”UNSAFE” situations, we cannot determine the adequateness of this abstraction, because it is again a matter of recognition of the real world. We use review or test to check the adequateness of them.

  On the other hand, it is relatively easier to check if the model correctly reflects the definition of each context. For example, using model checker, we can check important property such as ”whenever handbrake is released, it never judged as SAFE”.

- **system behavior**

  ”Process aspects” and relationship among ”process aspects” and ”context aspects” relate to the context dependencies of the main functionalities.

  We can check if the interaction among processes and contexts (i.e. reaction to the contexts and constraint by the contexts) are defined correctly or not by review, test, simulation and model checker. As ”process aspects” do not directly refer the ”sensor aspect”, but refer ”context aspects”, it is expected that we can reduce the number of states to be checked.

5.3. Related Work

We compare our work with other works.

- Aspect-oriented technologies are studied not only for programming phase[3, 7], but also wide range of software development. There pointed out that we have to

Understanding separation of concerns

Hafedh Mili , Amel Elkharraz & Hamid Mcheick
LAboratoire de recherche sur les TEchnologies du Commerce Électronique (LATECE)
UQAM, PO Box 8888, downtown station, Montréal (Qc) H3C 3P8, Canada
hafedh.mili@uqam.ca

Abstract

The separation of concerns, as a conceptual tool, enables us to manage the complexity of the software systems that we develop. There have been a number of approaches aimed at modularizing software around the natural boundaries of the various concerns, including subject-oriented programming [Harrison & Ossher, 93], composition filters [Aksit & Bergmans, 1992], aspect-oriented programming [Kiczales et al., 97], our own view-oriented programming [Mili et al., 99-02], and many others. The growing body of experiences in using these approaches have identified a number of fundamental issues such asm what is a concern, what is an aspect, which concerns are inherently separable, and which aspects are composable. To address these issues, we need to focus focus on the semantics of separation of concerns as opposed to the mechanics—and semantics—of aspect-oriented software development methods. We propose a conceptual framework based on a transformational view of software development. In particular, we distinguish between essential separability and inseparability, which characterize requirements, from accidental separability and inseparability, which characterize the realizations of those requirements.

1 Introduction

"Separation of concerns" is a general problem-solving idiom that enables us to break the complexity of a problem into loosely-coupled, easier to solve, subproblems. Underlying this idiom is the hope that the solutions to these subproblems can be composed relatively easily to yield a solution to the original problem. The history of programming languages may be seen as a perennial quest for modularisation boundaries that best map (back) to "natural modularisation boundaries" of requirements. Aspect-oriented software development methods are no different, and most of the research on AOSD has focused on the semantics of aspects and aspect composition, i.e. the solution domain, as opposed to the semantics of concerns and concern separation and composition, i.e. the problem domain. Yet, the early case studies have shown that these conceptually elegant techniques weren't intuitive to use (see [Kersten & Murphy, 99], [Kendall, 99], [Herrmann & Mezini, 00]). Further, a great number of users of these techniques were caught up in the "how-to" of language constructs, with no regard for the conceptual appropriateness of the AOSD technique for the problem at hand. Further, the various techniques seem to offer orthogonal, but equally useful constructs, with no clear guidelines as to which method is appropriate for which problem.

We believe that better understanding of the AOSD techniques will result from a characterization of, 1) the *input* of software development, and 2) the *process* of software development, to help characterize, if not identify, which concerns are separable, and which development steps are most likely to affect the separation (or separability) of the resulting artifacts. We propose a conceptual framework based on a transformational view of software development. In this context, all the requirements on a software product, be they functional (related to input/output relations) or otherwise (related to *how* the output is produced), are inputs in these transformations. These requirements fit into general areas, or *concerns*, which may end up embodied in separate or same artifacts. We distinguish *essential separability* and *inseparability*, which characterize *requirements*, from *accidental separability* and *inseparability*, which characterize the *realizations* of those requirements in development artifacts. Accidental inseparability can be remedied by better language design and user education. Accidental separability should even be discouraged as the conceptual complexity is often increased and not reduced, and maintenance of the resulting program is often made harder.

2 Understanding the separation of concerns problem

Design, in and of itself, is a very complicated cognitive task bringing to bear a host of knowledge types and sources and a myriad of problem solving skills [Dasgupta, 1991]. When the artifacts, themselves, are

complex, a number of the conceptual and methodological tools fall apart because of scalability problems. A number of researchers have shown that complexity is an *essential* property of design activities *in general*, due in part to the *inevitably incomplete formulation* of the problem, and in part to our inability to cope *simultaneously* with all of the constraints of a given problem (our *bounded rationality* [Simon, 1982]).

The *separation of concerns* technique is a general problem solving heuristic that consists of solving a problem by addressing its constraints, first separately, and then combining the partial solutions with the expectation that, 1) they be composable, and 2) the resulting solution is nearly optimal. For this heuristic to yield satisfactory results, the concerns that we are trying to treat separately must be fairly independent, to start with, so that they don't interfere with each other. Further, the problem solving activity itself needs to yield solutions that are composable.

In this section, we try to define the separation of concerns problem for the case of software. In this case, the "problem" is a set of requirements, and the "problem solving" process is the software development process. We first start by characterizing the software development process. In section 2.2, we try frame the separation of concerns problem.

2.1 A transformational view of software development

Simply put, software development may be seen as the process of going from precise specifications of what is to be done (requirements), to precise specifications of how it is to be done. Dasgupta identified two kinds of requirements in any design problem, *empirical requirements*, which specify externally observable or empirically determinable qualities that are desired of the artifacts, and *conceptual requirements*, which specify adherence to a particular style [Dasgupta, 1991]. For the case of software, there are two kinds of externally observable qualities, *functionality*—the *what*—on one hand, and run-time behavior—the *how*, including performance, and the like. Accordingly, we see three major categories of requirements for software development:

1) *Requirements of functionality*. These requirements specify an input/output relationship. To satisfy these requirements, we need a function that takes an input/output relationship and returns a function that returns the output for a given input
2) *Run-time requirements*. These are requirements on run-time behavior such as performance, distribution, the underlying machine (virtual or otherwise), etc.
3) *Requirements on the software artifacts*. These are requirements dealing with things such as modularity, reusability, choice of programming language, adherence to specific programming style, etc.

These correspond closely to the categories of architectural qualities identified by [Bass et al., 1998]. Describing a program using an executable specification languages may be seen as performing a first step of the design process, i.e. ensuring functionality. Later steps can worry about run-time behavior and artifact quality.

In practice, these three sets of requirements are addressed simultaneously. Further, except in new projects where a complete system is built from the ground up, new functionality often has to integrate into an existing architecture, which embodies a specific point in the design space that addresses a set of run-time and artifact requirements. However, for the purposes of our presentation, we will assume that the three major design dimensions are commutative; two design transformations T_1 and T_2 are said to be commutative if given D_i, the description of the software at step i, we have $T_2 \circ T_1 (D_i) = T_1 \circ T_2 (D_i)$ (see e.g. [Baxter, 1992]). With this mind, let us propose a first-cut description of software development.

Handling functional requirements: operationalizing requirements: This first transformation handles functional requirements. Given a relation R: A×B, we need to obtain a function f: $A \rightarrow B$, such that for all $a \in A$, $f(a) \in Image_R(a)$. We say that f(.) is an implementation of R. R describes the relationship that must exist between the input and the output; f(.) provides an effective procedure for computing the output, given the input. If R(.,.) is not a *function* (i.e. some elements of A have more than one image), then f(.) picks one element. Automatic programming consists, to a great extent, of automating the "operationalization of requirements". This transformation may be described by a relation *OR*: $\{R\} \times \{f(.)\}$ from the set of relations to the set of functions. Let $\underline{R}$ be the set of relations and $\underline{F}$ the set of functions. OR is thus a subset of $\underline{R} \times \underline{F}$. This relation may be known intensionally (as we just described), or extensionally (through exemplar pairs). Automating this step consists of finding a function g: $\underline{R} \rightarrow \underline{F}$ such that given a relation $R \in \underline{R}$, $g(R(.,.)) = f(.)$ where $(R, f(.)) \in OR$. We say that g is an implementation of *OR*.

Handling run-time requirements. These include *performance requirements* and *execution model*. These

requirements are handled differently from functional ones. Whereas the operationalization of requirements associates a requirement with *any* function that implements the requirement, here we are picky about the properties of such functions. For example, such functions have to be efficiently computed. Instead of the relation *OR* shown below, we now have a subrelation *EOR* (Efficient Operationalization of Requirements) such that *EOR* $\subseteq$OR, where Domain (*EOR*) = Domain (*OR*), but $\text{Image}_{\text{EOR}}(\text{R}) \subseteq \text{Image}_{\text{OR}}(\text{R})$. In other words, out of all the functions that implement R, we pick the ones that are efficient.

Issues related to the execution model include things such as distribution, synchronization, and security. This does not change the function that is computed but changes things about where the different pieces are executed and how[1]. We can represent the execution of function f() as follows: EX: $\underline{\text{F}} \times \text{I} \times \text{M} \rightarrow \text{O} \times \text{M}$. EX takes as argument the function to be computed and its input, and produces its outputs, while modifying the state of the machine that executed it (M). Those are the side effects on the machine, and may involve things such as establishing or terminating connections, modifying the state of data on permanent storage, logging, collecting statistics, etc. If f(*i*) = *o*, then EX (f(.), *i*,*s*) = <*o*,*s'*> where *s'* is the state of the machine after it has finished execution of function f(.). If f(.) is written in machine language, then EX() is the function implemented by the CPU (the hardware's fetch, load, execute cycle).

Generally speaking, EX is a composition of several functions. For example, with a virtual machine architecture, EX(f(.),*i*) = H (VM(f(.),*i*,*s*)) = <*o*,*s'*>, where H is the hardware execution, and VM is the virtual machine. The virtual machine itself could consist of a set of layered (composed) services or parallel services. An example of layered services is VM(f(.), *i*) = VM_1o VM_2(f(.),*i*,*s*). An example of parallel services is represented as < VM_1;VM_2>(f(.),*i*,*s*) where VM_1 and VM_2 are two services that are performed in parallel but such that the end result is the pair <*o*,*s'*>. It may be that VM_1(f(.),*i*,*s*) = *o*, while VM_2(f(.),*i*,*s*) = *s'*, or, provided that *s*= <s_1,s_2>, VM_1(f(.),*i*,<s_1,s_2>) = <*o*, <s'_1,s_2>> and VM_2(f(.),*i*, <s_1,s_2>) = <s_1,s'_2>. In other words, VM_1 and VM_2 modify different parts of the state of the executing machine. The output itself may be computed by one or two of the virtual machines.

Handling requirements on the artifacts. This involves taking into account the packaging of the function *f*(.) based on a number of criteria, including a reasonable division of labor, reusability, cohesion and coupling of the resulting modules, etc. It also includes things such as the choice of a programming language, programming style, etc. With object orientation, we end up implementing more than we need to for any particular use: classes are supposed to accommodate the needs of an application domain, but any application may use only a subset of that. Reuse (code sharing) happens by breaking down functions in such a way that the same sub-functions appear in two different functions (or systems). Let us take a problem R(.), and its realization, some function *f*(.). Idem for a problem R'() with realization *f*'(). If we can write $f = f_{post} \circ g \circ f_{pre}$ and $f' = f'_{post} \circ g \circ f'_{pre}$, then we reduce the amount of new code to be developed[2].

2.2 Framing the separation of concerns problem

For the purposes of our discussion, we define a *concern* as a set of related requirements. *The basic premise of separation of concerns approaches is that requirements have nice properties, and to the extent that we can associate artifacts with concerns, we would like the artifacts to have similar properties*! Precisely, the "separation of concerns" methods rely on the existence of a development homomorphism such as the one illustrated in Figure 1. Assume that requirements are represented by predicates, and let $A_P = OR(P(.))$ be the artifact that corresponds to predicate *P*(.). Development (represented by the thick arrow) is a homomorphism if there exists an operator $\oplus$ defined on artifacts such that $OR(P(.) \wedge Q(.)) \equiv OR(P(.)) \oplus OR(Q(.))$.

We have some intuitions about cases where this homomorphism between requirements and artifacts holds. For example, given two requirements defined by relations R_1: $A \rightarrow B$, and R_2: $B \rightarrow C$, we know of several operators $\oplus$ such that $OR(R_2 \circ R_1) \equiv OR(R_1) \oplus OR(R_2)$. For example, if the implementation adopts the call-and-return style, the operator $\oplus$ consists of the call relationship between procedures. If the publish-and-subscribe style is used, the operator $\oplus$ consists of registering $OR(R_1)$ as a publisher of some message, and $OR(R_2)$ as a subscriber to that message.

[1] Some aspects of security may be represented as functional requirements: adding the requester (another program or a user-id) as an input parameter..

[2] Developing with reuse may be framed as follows: given a desired function *f*(.), and an available library function *g*(.), find two functions *h*(.) and *i*(.) such that *f*(.) = *h*(.)o *g*(.) o *i*(.), and such that dev_cost(*h*(.)) + dev_cost(*i*(.)) $\leq$ dev_cost(*f*(.)).

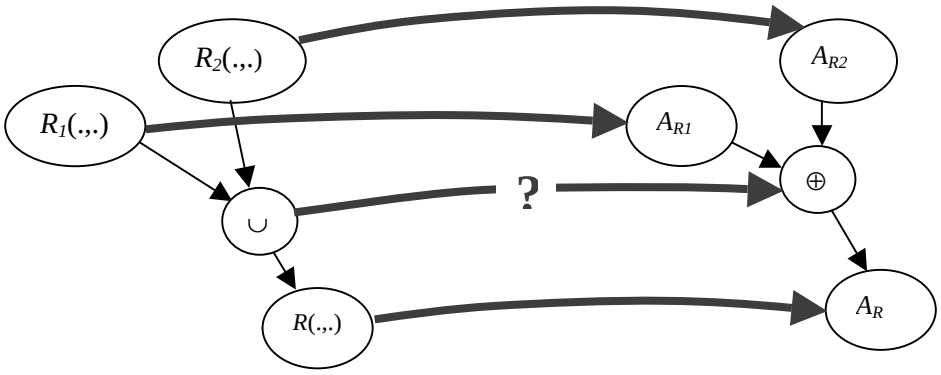


Figure 1. Development is a homomorphism from requirements to artifacts.

The advantages of this homomorphism include reusability, configurability, and separate maintenance. A number of object-oriented programming constructs and design idioms may be seen in this light. The new generation of separation of concerns techniques may be seen as defining new modularization boundaries for requirements, that are different from the ones afforded by regular object-oriented programming, and that are *realizable* in artifacts that are *composable* according to some composition operator. For example, OORAM uses *role models* [Reenskaugh, 1995] as new behavioral modules, and *role synthesis* to compose role models. Subject-oriented programming defined *subjects* [Harrison & Ossher, 93] as new modular structures, and *subject composition*, as a composition mechanism [Ossher et al., 96]. Aspect oriented programming defines *aspects* as new module boundaries, and *aspect-weaving* as a way of composing aspects with regular classes [Kiczales et al., 97]. Our own view-oriented programming uses *viewpoints* as a way of representing domain-independent business processes, and *view instantiation* and *attachment* as a way of adding that behavior to objects [Mili et al., 99],[Mili et al., 02].

Notwithstanding the case of OORAM, where the emphasis is on requirements level separation (*role models*) and composition, the other approaches have focused on the mechanics of artifact composition, sometimes losing sight of, 1) the requirements that these artifacts are supposed to embody, and 2) whether that composition (or separation) makes sense, from a requirements point of view. Further, even in those cases where AO techniques seemed appropriate, there were sometimes better non-aspect oriented solutions (see e.g. [Robillard & Murphy,2001]).

If we view requirements as predicates on the solution, then requirements are clearly composable using logical composition ($\wedge$)—whether the resulting conjunction has solutions or not. However, for the homomorphism of Figure 1 to hold:

1) the composed requirements have to be independent, and
2) the development transformations have to preserve such independence so that the resulting artefacts may be combined.

In the next section, we try to characterize both conditions.

3 Characterizing the separability of requirements

In this section, we attempt the overly ambitious goal of answering two dual questions:

1) Given two requirements, under what conditions can they be "developed" separately, and can their realizations (aspects) be composed at will. The answer to this question will help determine the *domain* or *operating range* of the development homomorphism we illustrated in Figure 1. We refer to this problem as the *composability of requirement realizations*.
2) Given a realization that addresses several concerns, under what conditions can that realization be untangled into separable aspects, each of which addressing a subset of concerns. The answer to this question may help us assess which systems may be re-engineered in such a way that different concerns are addressed in separate—and readily reusable—aspects. We refer to this problem as the *separability of requirement realizations*.

In addition to its practical importance, an answer to the second question will also help us understand why case studies have not been as convincing as the textbook

cases that the original method authors have presented in support of their techniques.

We identified in section 2.1 three distinct kinds of requirements, requirements of functionality, run-time requirements, and requirements on the software artifacts themselves. Does it even make sense to try to address the composability of requirement realizations when talking about one functional requirement, and one requirement on the software artefacts. We should address the issue of whether run-time requirements even have realizations, before we concern ourselves with composing them. Accordingly, we start our discussion by first characterizing the ways in which requirements in each category are handled (individually). We will argue that run-time requirements can be represented as functional requirements on the virtual machine; requirements on artefacts are more difficult to formalize.

In section 3.2 looks at the *composability of requirement realizations* problem for the case of functional requirements. We examine the problem from a purely mathematical point of view, reducing the separability of two requirements, seen as (input,output) relations, to conditions on their domains and ranges. This will enable us to address composability issues between runtime requirements or between functional requirements, but not between a functional requirement and a run-time requirement, say[3]; this kind of (cross-type) composition will be discussed in section 4.

Section 3.3 tries to answer the *separability of requirement realizations* for functional requirements by looking at the problem of decomposing a function into separate sub-functions. We look at a range of decomposition/recomposition operators with different semantics preserving properties.

3.1 Handling the different types of requirements

3.1.1 Handling run-time requirements

We consider run-time requirements to be functional requirements on an imaginary virtual machine that will execute the program in the context of the real machine. The virtual machine will add a number of services including distribution, persistence, security, and others.

Persistence services may be seen as providing the program with an execution environment (a virtual machine) that persists automatically the objects that the program manipulates. Most object-oriented databases operate this way (Versant, ObjectStore): developers write programs that manipulate persistent objects in a seamless fashion. It is as if databases come with their own run-time object model, built on top of the host language object model. We later see how this is actually implemented—interestingly, a limited form of aspect-oriented programming.

Distribution is similar to persistence in principle. Lest we oversimplify, distribution may be seen as providing a virtual machine whose run-time representation of objects accommodates remote objects, with what that implies in terms of referencing and in terms of method invocation. Consider the following CORBA or RMI-like code sequence:

```
Bank bank =
naming.bind("//www.mycompagny.c
om/mybusinessdomain/bank23");
Client cl =
bank.getCustomer("JohnDoe234");
String address =
cl.getAddress();
```

Notwithstanding the first line, which suggests the use of a naming service, the subsequent lines are indifferent from the location of the objects. We could imagine the same program being run in local mode, where the default Java virtual machine run-time representation of objects is used, and "a distributed Java virtual machine" that uses a level of indirection for run-time object representation to access remote objects, and that invokes an ORB to execute methods. Existing implementations of distribution use a slightly different implementation but the idea is the same.

The way distribution and persistence have been commonly implemented present some commonality. Transparency to the developer dictates a virtual machine metaphor. However, both techniques instrument user code with service-specific code that invokes those services (persistence or remote access). With Java-style persistence (e.g. ObjectStore), the code that is injected is added directly to the compiled Java bytecodes. With distribution, the IDL compiler injects, along with user code, code that is meant to be executed by the distribution virtual machine.

The same can be said about some aspects of security. Both authentication and encryption can be easily (and naturally) implemented at the virtual machine level: one involves encrypting exchanged data (through method calls), and the other authenticates the caller. In fact, Java's own security model is supported and enforced by

[3] This is something that AspectJ is presumably well-suited for, but that also explains some of its weaknesses.

the virtual machine. J2EE's security model is enforced by the containers—a higher level yet virtual machine.

One reason why virtual machine-like implementations of these services are not common—with the exception of security, for which we want no loopholes—is performance. The other is selectivity: because these services involve an overhead, if we embed it in the virtual machine, then all objects will use it, whether they need it or not. With this code injection mechanism, the code will only be injected in those objects/classes that need it.

As mentioned above, common implementations of persistence use a variant of aspect oriented programming: persistence code is added into designated class files (typically specified in configuration files) so that object creation, accessing, and modification accesses the database client. The same is true for distribution, where client-side stubs (proxies) go through the ORB to get the data they need.

Viewing run-time requirements as functional requirements on the virtual machine helps us understand which services are separable and/or composable, *in principle*, and also helps us understand which solutions are feasible under which situations, and understand some of the anomalies that arise from composing virtual machine-level services.

3.1.2 Handling requirements on the artifacts

Requirements on the artifacts deal with development-time "abilities", with no regard for functionality or performance. Such requirements include understandability, reusability, maintainability, etc. Let $R(.)$ be a functional requirement, and $f(.)$ be an operationalization of $R(.)$, i.e. $f(.) \in OR(R(.))$. The various "abilities" on the artifacts can typically be written as constraints on various metrics on the artifacts, such as:

- $M_i(f(.)) = \text{MIN}_{g \in OR(R(.))} (M_i(g(.))$ (relative constraint) or
- $M_i(f(.)) \leq \alpha$, for some constant α (absolute constraint)

These *meta-level* constraints determine the packaging of the functionality[4].

[4] Normally, we would distinguish between algorithms/procedures and packaging. Algorithms determine how outputs are produced from inputs, and determine performance. Packaging puts boundaries around parts of the algorithm to promote various qualities.

Separation of concerns *is* a requirement on software artifacts that is being addressed with AOSD techniques. Thus, our discussion of how development affects separation of concerns will be limited to the development activities related to accommodating functional requirements and those related to handling run-time requirements.

3.2 Composable requirements

Given a development transformation T, we consider two requirements R_1 and R_2 to be T-composable if:

1) we can associate separate realizations to them ($T(R_1)$ and $T(R_2)$), and
2) there exists a composition operator $\otimes$ on their realizations that satisfies them both, i.e. $T(R_1 \Lambda R_2) = T(R_1) \otimes T(R_2)$

We showed in section 2.1 that functional requirements are transformed using an *operationalization operator—OR*, turning an input-output relation into a *function* that produces the output given the input. Having argued in section 3.1.1 that run-time requirements are nothing but functional requirements on the virtual machine, we look at the problem of composing two functional requirements through the operationalization operator.

We would like the operationalization of functional requirements to be additive at least in those cases where the two requirements have disjoint domains. Consider two relations R and R' such that $\text{Domain}(R) \cap \text{Domain}(R') = \Phi$.The simplest way of implementing $R \cup R'$ is by taking $f(.) \oplus f'(.)$, where $f(.) \oplus f'(.) = g(x)$ such that:

$g(x) = f(x)$, if $x \in \text{Domain}(R)$
$= f'(x)$, if $x \in \text{Domain}(R')$

In other words, the simplest $OR(.)$ would behave as follows:

$OR(R \cup R') = f(.) \oplus f'(.)$

Note that if we take into account reuse, then we may be able to write $f = f_{post} \circ g \circ f_{pre}$ and $f' = f'_{post} \circ g \circ f'_{pre}$. We do have $\text{Domain}(f'_{pre}) = \text{Domain}(f')$ and $\text{Domain}(f_{pre}) = \text{Domain}(f)$, and thus $\text{Domain}(f_{pre}) \cap \text{Domain}(f'_{pre}) = \Phi$, but we don't know whether $\text{Domain}(f_{post})$ and $\text{Domain}(f'_{post})$ are disjoint, and we can't write $OR(R \cup R')$ (or $f(.) \oplus f'(.)$) as $[f_{post}(.) \oplus f'_{post}(.)] \circ g \circ [f_{pre}(.) \oplus f'_{pre}(.)]$.

If the relations have intersecting domains, we can define them as follows: $R = R_1 \cup R_2$ and $R' = R'_1 \cup R'_2$ such that: $\text{Domain}(R_1) = \text{Domain}(R) - \text{Domain}(R')$, $\text{Domain}(R'_1) = \text{Domain}(R') - \text{Domain}(R)$, and $\text{Domain}(R_2) = \text{Domain}(R'_2) = \text{Domain}(R) \cap$

Domain(R'). In this case, the relation to implement is $R_1 \cup R'_1 \cup (R_2 \cup R'_2)$, where R_1, R'_1, and $R_2 \cup R'_2$ have mutually disjoint domains. Thus, we have $OR(R_1 \cup R'_1 \cup (R_2 \cup R'_2)) = OR(R_1) \oplus OR(R_2) \oplus OR(R_2 \cup R'_2)$.

This relationship is trivially satisfied in case $R_2 = R'_2$. This is the ideal case in the sense that both requirements agree on what the output should be for the same inputs. In that case, the two requirements (R_1 and R_2) may be seen as two restrictions of the same relationship defined on the domain $Domain(R_1) \cup Domain(R'_1)$. If the two relationships disagree on the output, then we have a problem. We see two levels of disagreement. The first level of disagreement is illustrated in the following example. Consider the two relations $R_1 = \{ (x,y) \mid 0 < x < 100$, and $x^2 = y\}$ and $R_2 = \{ (x,y) \mid 50 < x < 150$, and $x^2 = y\}$. The intersection of the two domains consists of the interval [50..100]. If both the realizations of *R1* and *R2* use the positive square root of *x*—or both use the negative square root—then we are fine. If they use different square roots, then we have a problem. This incompatibility is due to an inconsistent choice of

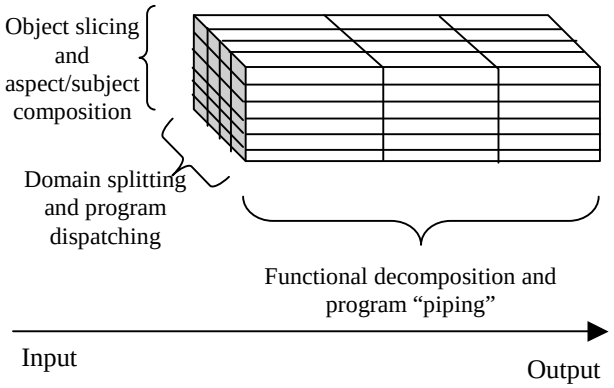


Figure 2. Comparing three decomposition paradigms

realizations, and is a common and acceptable course of action. Intuitively, what we need in this case to make sure that we use consistent realizations. This is not unlike the problem of choosing consistent specializations when we instantiate a framework, i.e. the kind of situations for which things such as the *factory pattern* is applicable.

The second level of disagreement is the case where the requirements themselves disagree, i.e.:

$\exists x \in Domain(R_1) \cap Domain(R_2)$ s.t. $R_1(x) \neq R_2(x)$

In our view, this is not a case for separation of concerns methods to handle: the requirements disagree, so there is no point in trying to compose the artifacts.

3.3 Separable requirements

Given a development transformation *T*, we consider a requirement *R* (an element of the domain of *T*) to be *T-separable* if there exist, 1) two requirements R_1 and R_2, 2) a composition operator • defined on the domain of *T*—the requirements—and, 3) a composition operator ⊗ on the image of *T*—the *artifacts*—such that:

1) $R = R_1 \bullet R_2$
2) $T(R) = T(R_1) \otimes T(R_2)$

This is nothing but the good old divide-and-conquer analytical development paradigm. With structured analysis and design (and programming), the operator is functional composition, in the mathematical sense, and ⊗ is "piping", in the programming sense (the output of a program or procedure is used as an input to the other). Functional decomposition is not only useful for reducing complexity, it is also useful for reuse.

Another valuable pair of operators corresponds to the combination of domain splitting and dispatching. Consider the requirement *R* where $domain(R) = D = D_1 \oplus D_2$.—the symbol ⊕ referring to disjoint union (partition). Let *T* be the operationalization of requirements ($OR(.)$), and $R_1 = R|D_1$, and $R_2 = R|D_2$. Then:

$$OR(R(.)) = \begin{cases} \textbf{if } x \in D_1 \textbf{ call } OR(R_1) \\ \textbf{if } x \in D_2 \textbf{ call } OR(R_2) \end{cases}$$

We are all familiar with these two techniques, and have used them—and should continue to do so—to good measure. Aspect-oriented development techniques advocate *other* pairs of decompose/recompose or split/join operators which are specific to the object-oriented context. These new pairs of operators operate simultaneously on functions and data, along the lines of object or class hierarchy slicing (see e.g. [Tip et al.,1996]). In this case, instead of considering the input domain (*D*) as consisting of simple value, we consider it as a tuple (of state variables), and functions (object methods) may operate on various "sub-tuples".

Figure 2 illustrates the three decomposition paradigms. For each paradigm, we mention the decomposition technique used on requirements, and the corresponding composition technique used on the corresponding artifacts. In the next section, we look more closely at the problem of sliceability of requirements. We start with a strict definition of sliceability which supports unrestricted (commutative) recomposition of the artifacts. We then propose a weaker form of sliceability which requires an ordered (non-commutative) recomposition.

3.3.1 Sliceability

Let R $\subseteq$ A×B, let *f*(.) = *OR*(R), and assume that A = S_1 ×S_2 ×… ×S_i ×S_{i+1} ×… ×S_n×I and B = S_1 ×S_2 ×… ×S_i ×S_{i+1} ×… ×S_n×O. We say that R (or *f*()) is *sliceable* if there exist two functions $f_1(x_1,\ldots,x_i,i)$ et $f_2(x_{i+1},\ldots,x_n,i)$ such that $f(x_1,\ldots,x_i,x_{i+1},\ldots,x_n,i) = f_1(x_1,\ldots,x_i,i) \bullet f_2(x_{i+1},\ldots,x_n,i)$. In other words, the function *f*() can be computed as the concatenation of two functions.

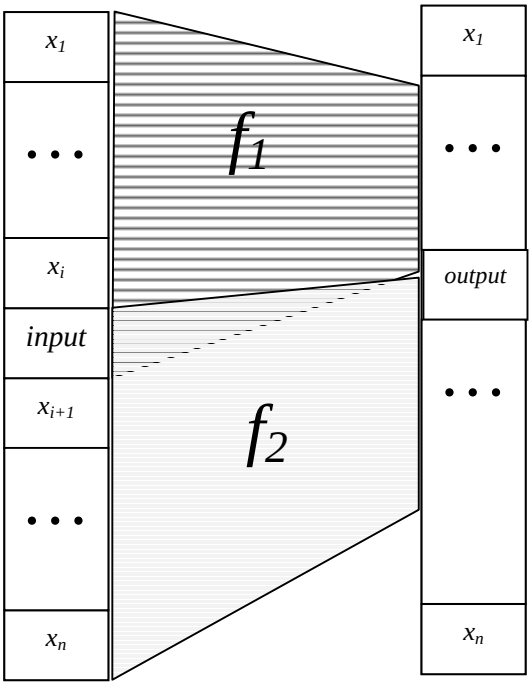


Figure 3. A function is *sliceable* if it can be written as the concatenation of two functions that access and modify disjoint parts of an object

The idea of sliceability is related to the idea that a relation may be written as a subset of the product of two relations. For example, let R_1 and R_2 be two binary relations. We can define the relation $R_1 \times R_2$ as follows: $\langle x_1,x_2,y_1,y_2\rangle \in R_1 \times R_2$ if and only if $\langle x_1,y_1\rangle \in R_1$ and $\langle ,x_2,y_2\rangle \in R_2$.
Intuitively, the sliceability of a corresponds to the case where we have two functions that take the same input and that use and modify different parts of an object, i.e. they correspond to two disjoint slices of the same data (or object). Sliceable functions can be put together, with no problem. Notice that we require that both functions take the input (which may be either a real input or a method selector), and that the output is produced between them. In the context of an object-oriented program, if we have a method that returns void but modifies the state of the object, then each subfunction will have modified its slice. If the function returns a value, then we might be able to find a subset of state variables based on which the output is computed, and the slice may be made along that. Note, however, that not all relations/functions are sliceable. A function that averages the state variables will not be sliceable[5].

Subject-oriented programming (and hyperspaces) works best with this ideal case in mind. Problematic cases occur when the sliceability hypothesis fails. Interestingly, the broken delegation problem can be understood in terms of sliceability of functions. Broken delegation happens when a function that occurs on one side (i.e. in a single object fragment) calls a separable function that occurs on several object fragments (see e.g. [Bardou & Dony, 1996]): the result is no longer separable.

3.3.2 Effective sliceability

Let R $\subseteq$ A×B, let *f*(.) = *OR*(R), and assume that A = S_1× … × S_i ×… × S_j ×… ×S_n×I and B = S_1× … × S_i ×… × S_j ×… ×S_n×O. Let *f*(…) be a function that implements R. Let f_1(…) and f_2(…) be two functions with domains S_1× … × S_i ×… × S_j ×… ×S_n×I. If $f(x_1,\ldots,x_i,x_{i+1},\ldots,x_n,i) = \langle x'_1,\ldots,x'_i,x'_{i+1},\ldots,x'_n,o\rangle$, we use the notation $f_{|i}$ to refer to the projection of *f* over the set S_i , i.e., $f_{|i}(x_1,\ldots,x_i,x_{i+1},\ldots,x_n,i) = x'_i$. Similarly, we define $f_{|S}$ as the projection of *f* over the set S = S_i ×… × S_j for some *i* and *j*. Let Ref(*f*) be the set of variables used in the computation of *f*(…) and Mod(*f*) be the set of variables modified by *f*(…) be the set of state variables that are modified by *f*, i.e. the set of variables $\{x_i\}_i$ such that $f_{|i}(x_1,\ldots,x_i,x_{i+1},\ldots,x_n,i) = x'_i \neq x_i$. A function *f*(…) is said to be *effectively sliceable* if and only if there exist two functions $f_1(x_1,\ldots x_n,i)$ and $f_2(x_1,\ldots x_n,i)$ such that:

$$\mathrm{Mod}(f_1) \cap \mathrm{Ref}(f_2) = \Phi$$
$$\mathrm{Mod}(f_2) \cap \mathrm{Ref}(f_1) = \Phi$$
$$\mathrm{Mod}(f_1) \cap \mathrm{Mod}(f_2) = \Phi$$
$$\mathrm{Mod}(f_1) \cup \mathrm{Mod}(f_2) = \mathrm{Mod}(f)$$
$$f_{|Mod(f)}(x_1,\ldots,x_n,i) = f_{1|Mod(f1)}(x_1,\ldots,x_n,i) \bullet f_{2|Mod(f2)}(x_1,\ldots,x_n,i), \text{ and}$$
$$f_{|o}(x_1,\ldots,x_n,i) = f_{1|o}(x_1,\ldots,x_n,i) \bullet f_{2|o}(x_1,\ldots,x_n,i)$$

for some ordering of the state variables $x_1,\ldots,x_n$. Figure 4 illustrates the first three equalities in a Venn Diagram.

Note that a *sliceable* function is also *effectively sliceable*. An interesting property of *effectively sliceable* functions is that the component functions may be executed in any sequence.

[5] Intuitively, intensive functions (of the state variables) are not separable, whereas extensive functions are.

There are other cases of sliceability, but in this case, the subfunctions have to be executed in a particular order. We call this *temporal sliceability*. Temporal sliceability is a weaker condition than effective sliceability, and is described as follows. Let $R \subseteq A \times B$, let $f(.) = OR(R)$, and assume that $A = S_1 \times \ldots \times S_i \times \ldots \times S_j \times \ldots \times S_n \times I$ and $B = S_1 \times \ldots \times S_i \times \ldots \times S_j \times \ldots \times S_n \times O$. Let $f(\ldots)$ be a function that implements R. Let $f_1(\ldots)$ and $f_2(\ldots)$ be two functions with domains $S_1 \times \ldots \times S_i \times \ldots \times S_j \times \ldots \times S_n \times I$. Using the same notation as above, we say that function $f(\ldots)$ is said to be *temporally sliceable* if and only if there exist two functions $f_1(x_1,\ldots x_n,i)$ and $f_2(x_1,\ldots x_n,i)$ such that:

$$f_{|Mod(f)}(x_1,\ldots,x_n,i) = f_{1|Mod(f1)-Mod(f2)}(x_1,\ldots,x_n,i) \bullet f_{2|A-(Mod(f1)-Mod(f2))}(f_1(x_1,\ldots,x_n,i)).$$

$Mod(f_1) - Mod(f_2)$ represents the set of variables that are modified by f_1 but not by f_2. Some of these variables may, however, be referenced by f2 and we don't care about that. Obviously, the relationship between f_1 and f_2 is not a symmetrical one, and the functions have to be executed in a particular order.

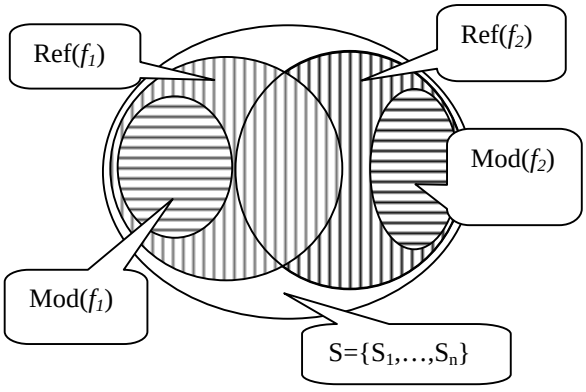


Figure 4. A function is *effectively sliceable* if it can be written as the concatenation of two functions that, 1) modify disjoint parts of an object, and 2) don't refer to the parts that the others modify

In [Mili, 1996], we showed that provided that methods of objects do not modify objects other than the executing objects or their components, any method that computes a function and modifies the receiver object can be decomposed into a sequence of pure functional and purely side-effectal functions. To compose two hybrid functions, we can decompose them along the purely functional versus purely side-effectal dimensions, find the smallest granularity decomposition between the two, and then compose them slice-by-slice.

The major problem, of course, is our tendency to code "service-oriented functions", i.e. functions that are application level but that are coded at the domain class level. These functions are not composable because they address an application specific need, each. You would want to compose them because they embody a general behavior that is not encapsulated elsewhere. Obviously, not choosing the right granularity is a problem, and leads to methods that are not composable.

4 Discussion

This is a very preliminary investigation into the principles of separation of concerns and the foundations of the techniques that promote separation of concerns. The yardstick by which innovations in software engineering are to be assessed has always been—and rightly so—to determine the problem that a given method, technique, or tool, solves. Separation of concerns is only useful to the extent that once the concerns have been addressed separately, we are able to re-combine the individual and partial solutions into one that addresses all of them.

Some of the case studies that are available in the literature show cases where concern separation is difficult in practice [Kersten & Murphy, 99], [Kendall, 99], [Herrmann & Mezini, 00]. Others showed that *aspect/subject* composition is difficult, even in cases where the aspects or subjects embody distinctly different concerns [Mili et al.,96], [Kersten & Murphy, 99], [Muprhy et al., 01]. Others yet have reminded us of simple solutions to separation of concern problems (see e.g. [Robillard et al.,'99]).

We attempted to frame the separation of concerns in software development in terms of homomorphisms of development transformations, and then we tried to determine the "operating range" of these homomorphisms. This preliminary work raised more questions than it answered, and some of the answers are reassuringly common-sensical, but are worth stating:

- Not all requirements (concerns) are composable in the sense that they lead to composable artifacts. Viewing requirements as input-output relations, we identified simple conditions on the domains and images of these requirements, which essentially say that the requirements should not be conflicting. In particular, method cancellation through subject composition or aspect weaving is no less dangerous than with inheritance: they are both a sign of either a violation of intent, or of sloppy realization (implementation).
- We should treat aspects that embody run-time requirements differently—and separately—from aspects that embody functional (domain) requirements. We framed run-time

requirements (persistence, fault-tolerance, etc.) in terms of *functional requirements of the virtual machine*. In an ideal world, such concerns should also be handled by virtual machine—or more generally, meta-level—aspects. However, performance considerations may suggest otherwise at the risk of inducing composability problems[6].

- Not all programs that implement several concerns can or should be re-engineered into separate aspects. The underlying concerns/requirements may not be separable (essetial inseparability), or the current implementation may not lend itself to such a separation (accidental inseparability). Object slicing can help with accidental inseparability.

This work is in its infancy. We have started to take a closer look at the existing AOSD methods and the case studies to judge the usefulness of the above framework. We were able to explain known difficulties with subject-oriented composition (see e.g. [Ossher et al.,95-97], [Mili et al.,96]) and view attachment [Mili et al., 99-02] in terms of violations of some of the principles outlined above. More work is needed with the other methods and the case studies to ascertain the usefulness of our framework.

5 Bibliography

[Aksit et al., 1992] M. Aksit, L. Bergmans, and L. Vural, "An object-oriented language-database integration model: the composition filters approach", *in Proc. of ECOOP 92*, Springer Verlag 1992.

[Bardou & Dony, 1996] D. Bardou & C. Dony, "Split objects: a disciplined use of delegation within objects," in proc. OOPSLA'96, pp. 122-137, 1996.

[Bass et al.,1998] L. Bass, P. Clements & R. Kazman, *Software Architecture in Practice*, 1998.

[Baxter, 1992] I. Baxter, "Design Maintenance Systems," *CACM*, vol. 35, no. 4, April 1992, pp. 73-89

[Dasgupta, 1991] S. Dasgupta, "The Nature of Design Problems," in *Design Theory and Computer Science*, Cambridge University Press, 1991, pp. 13-35.

[Harrison & Ossher, 93] W. Harrison & H.Ossher, "Subject-oriented programming: a critique of pure objects," *in Proc. OOPSLA'93*, pp. 411-428.

[Herrman & Mezini, 2000] S. Herrmann & M. Mezini, "On the Need for a Unified MDSOC Model: Experiences from Constructing a Modular Software Engineering Environment", MSDOC workshop, OOPSLA'2000

[Kendall, 1999] E. A. Kendall. *Role Model Designs and Implementations with Aspect Oriented Programming*. In Proc. OOPSLA'99, pages 353-369, Denver, Co., 1999

[Kersten & Murphy, 1999] M. Kersten & G. Murphy, "Atlas: a case study in building a Web-based learning environment using aspect-oriented programming", in *proc. OOPSLA'99*, pp. 340-352, 1999.

[Kiczales et al., 1997] G. Kiczales, J. Lamping, C. Lopez, "Aspect-Oriented Programming," in *Proc. ECOOP'*97.

[Mili, 1996] H. Mili, "On behavioral descriptions in oject-oriented modeling", *Journal of Systems and Software*, summer 1996.

[Mili et al.,99] H. Mili, A. Mili, J. Dargham, O. Cherkaoui & R. Godin, "View Programming: Towards a Framework for Decentralized Development and Execution of OO Programs," *Proceedings of TOOLS USA '99*, Aug. 1-5, 1999, Prentice-Hall, pp. 211-221

[Mili et al., 2002] H. Mili, H. Mcheick & J. Dargham, ''CorbaViews: Distribting objects with several functional aspects,'' *Journal of Object Technology*, August 2002,.

[Ossher 96] H. Ossher, M. Kaplan, W. Harrison, A. Katz, and V. Kruskal, "Specifying subject-oriented composition," in *TAPOS*, 2(3), 1996.

[Robillard & Murphy, 2001] M. Robillard & G. Murphy, "Analyzing Concerns Using Class Member Dependencies," Position paper for the ICSE 2001 Workshop on Advanced Separation of Concerns in Software Engineering, 2001.

[Simon, 1982] H. A. Simon, *Models of bounded rationality*, vol. 2, Cambridge, MA (MIT Press, 1982).

[Reenskaugh, 1995] Trygve Reenskaugh, in *Working with Objects*, Prentice-Hall, 1995

[Sullivan, 2001] G. T. Sullivan, "Aspect-Oriented Programming Using Reflection and Metaobject Protocols," *CACM*, Oct. 2001, 44 (10), pp. 95-97

[Tip et al., 1996] F. Tip, J-D Choi, J. Field, and G. Ramalingam, "Slicing class hierarchies in C++", In *Proc. OOPSLA'96*, San Jose, CA, pp. 179-197.

[6] With asectJ, if we have two aspects, one that adds a function, and one that traces all functions, the order of aspect weaving is important….the thing is, the trace aspect is meta-level: it is a functional aspect of the virtual machine (to execute methods and produce a log) defined *intensionally* for all methods. But by making it a user-level aspect, we define it *extensionally*, i.e. on all the *currently defined/available* user functions, and hence, the order dependency!

Refining Feature Driven Development - A methodology for early aspects

Jianxiong Pang Lynne Blair

Computing Department
Lancaster University, Bailrigg
Lancaster, LA1 4YR, U.K.

j.pang@lancaster.ac.uk *lb@comp.lancs.ac.uk*

Abstract

This position paper focuses on refining an agile processes approach named FDD to make it more aspect-oriented, hence a natural candidate for early aspects.

We show that only a slight refinement is needed to adapt FDD to aspect-oriented development. Within the refinement, all requirements, be they concerns (architectural, non-functional and functional) or properties or rules, are described by using the feature template in FDD. Features as development units are first class entities throughout the whole development period. Since the later stages of FDD are also enhanced with aspect-oriented technique, this makes the transition from requirements to design and implementation much easier and smoother.

Keywords: early aspects, feature driven development, agile process, feature template, aspect-oriented, requirement engineering.

1. Introduction

A *feature*, as a term, is used for describing a small piece of particular valuable (sometimes also attractive) capability/functionality. Within telecommunication systems, it is defined as "a unit of functionality existing in a system and usually perceived as having a self-contained functional role" [2].

A feature is a concept mostly belonging to the problem domain rather than the solution domain [3], hence is highly relevant to requirement analysis. This has been reflected by the fact that the "feature engineering" is categorised as a branch of requirement engineering [4]. A feature is suitable for behaviour modularity, which in turn, supports change of system's behaviour.

Feature Driven Development (FDD) is a lightweight and model-driven software development process tailored to the delivery of frequent, tangible, working results. The lightweight characteristics make these processes easy-to-follow and agile. Another remarkable characteristic of FDD is that it gives a slight variation to the definition of "feature", with a feature being referred to as "a client-valued functionality that can be implemented in two weeks or less". By that, it includes a "timing factor", which is believed to have subtly combined the technical factors (e.g. agility) and social psycho factors (e.g. encouraging value) in the software development activity. Under FDD, features as incremental units are planned and developed one by one, making tangible and solid results. This makes quality control more manageable. Furthermore, a feature is carefully tailored, so as to be implementable in a relatively comfortable time. This gives confidence, encouragement, and incentive to developers.

General feature driven methodology has been proven effective in modern software systems. In *product line development*, a company produces a range of similar software products; the product can be structured in such a way that common units of development (e.g. feature or feature sets) are shared. Such development has been shown to significantly reduce development costs, and benefits end-users in terms of flexibility, in being able to choose a customized combination of features for their product [5]. *The telecommunication industry* has particularly benefited from a development centred on features. The introduction of Intelligent Network (IN) brought in a generic model where a basic call could be updated by

adding features implemented as discrete components (Service Logic Programs). As a result, the telecommunication industry has a tradition of organizing development projects, people, and even marketing by features [7]. Microsoft has also apparently followed some feature-centric processes in their software product line for a number of years [8].

Building on the object-oriented paradigm and reflective programming, aspect-oriented programming is emerging as a technique that supports more advanced separation of concerns. Recently, as this technique has become more broadly recognized, more techniques have been merged under the umbrella of aspect-oriented software development (AOSD). The worthiness of aspect-oriented techniques being combined with FDD lies in that aspect-oriented technology is capable of flexible behaviour modification being carried out on an existing, or even, running system. Although FDD has existed for quite a long time now, it is not easy for a traditional OO technique to implement features in an entirely modular way. It is with the emergence of aspect-oriented techniques that the development of features in a neat and clean way becomes a reality. As feature modularisation and localization is dramatically improved, a chance certainly exists to refine FDD into an aspect-oriented process.

2. Related work

As has happened in object-oriented programming, researchers have applied aspect-oriented ideas to higher levels of the software lifecycle, e.g. requirement analysis and design.

Works that are most relevant are those about *aspect-oriented component based software development* [9] and *aspect-oriented requirement engineering* [10].

In [9], an approach called "aspect-oriented component requirement engineering process" is proposed "to address some difficult issues of component requirement engineering by analysing and characterising components based on different aspects of the overall application a component addresses". This approach applies aspects to categorized components with properties, and provides facilities for the requirement changes (e.g. the change of stakeholders or running context). Our analysis reveals that it is not clear, in this approach, the relationship of aspects with the component architecture, and other aspects. Aspects in this approach are concepts rather than first class entities that later are mapped to some design and implementation artefacts (i.e. aspects are still identifiable). The lack of simple and unified concept of aspects and their associated base systems contributes to making this software process relatively 'heavy'. We believe that keeping it lightweight and agile is vital for today's software, especially for service-oriented systems, which rapidly become pervasive.

In [10], a model for "aspect-oriented requirement engineering" is proposed that supports "the reconciliation of *separation of concern* with the need to satisfy broadly scoped requirements and constraints". This model is built on an existing approach called PREView [11] that already supports separation of crosscutting properties but lacks guidelines on avoiding inconsistencies between concerns, and also lacks the mapping or influence of crosscutting properties on artefacts at later development stages. Therefore, the model can be viewed as a refinement of PREView, aiming to overcome the shortcomings mentioned above. The model uses "concerns" to represent aspects at the requirements level. We believe that these "concerns" are close to "features" in meaning, except that there is no concern template equivalent to the feature template, therefore the description of a concern seems ad-hoc. Furthermore, the model expressed in [11] does not require building an overall model (the backbone) as in FDD. Therefore, it is not clear how to make a smooth transition to the design and implementation stages given there is no placeholder for concerns in the requirement stage, or advice on how to deal with multi views if there are multiple models.

3. Comparing FDD and aspect-orientation

There are 5 processes within FDD (taken from [12]):

1. ***Develop an overall model (focussing more on shape than on content)***
2. ***Build a detailed, prioritised feature list***
3. ***Plan by feature***
4. ***Design by feature (focussing more on content than on shape)***
5. ***Build by feature***

Note: process 4-and 5 are iterated

Further information on FDD can be found in [12],[13] and [14].

It can be seen that, among the 5 process steps, the first three steps belong to the *requirement engineering*. The *feature*, as a development unit, is emphasised throughout these processes. Note that before any feature is developed, there has already been an *overall model* produced, typically represented as a class diagram. Thus, all features have to be composed or integrated into the overall model later. This arrangement makes it comparable to some important AOP techniques (e.g. AspectJ [15]), i.e. a system can be developed by separating a base system and a number of advice modules that later are woven to the base.

The determination of features is largely guided by user's value rather than by the class structure of the overall model[1], such that if, during the integration/composition of features into the overall model, a feature looks unsuitable to fit in a class, then it should be spread into several classes.

At a more detailed level, every feature in the feature list is described by using a feature template. Such a template is as follows:

<action> the <result><by | for | of | to>a(n)<object>

e.g. *calculate* the *total* of a *sale*, *invoke* the *spell-checker* for a *document*, *upgrade* the *quality* of a *service*, etc. Among these features, some can be naturally included into one class, such as the first and the third one; some cannot, like the second one (involving at least two classes the *spell-checker* and the *document*).

This has led us to conclude that FDD closely resembles aspect-oriented development in nature if purely examined from a perspective of the high levels (e.g. requirement and design levels), because it has satisfied two key aspect-oriented conditions:

1. *Crosscutting*: features do crosscut the overall model. (See the "spell-checker" example)
2. *Modular*: features are semantically complete and self-contained entities.

On the other hand, FDD is not completely an aspect-oriented technique because it lacks explicit guidelines on how to:

1. incorporate crosscutting concerns into features.
2. seemingly map features to design and implementation artefacts in a aspect-oriented programming environments at later stage.

Therefore, it is necessary to refine the FDD processes so as to complement it with respect to aspect-oriented development.

4. Refining the FDD processes

The proposed refinement is made with the following goals in mind:

- Facilitate the separation of concerns.
- Assist the detection and avoidance of inconsistency between features
- Maintain a smooth transition to the next stage of the development process

[1] It is also guided by development time, as can be seen from the FDD description. The time factor has been omitted here for simplicity.

To facilitate the separation of concerns, the basic arrangement of the system structure is preserved, i.e. an overall model plus a set of features. Features can be grouped into a feature set, according to their characteristics of functionality, which might result in a hierarchical structure of features.

To assist the detection and avoidance of inconsistency between features, we have proposed a method called *boundary condition exploration* [16] based on our study of a large number of feature interaction cases within and beyond telecommunication systems [2][17]. This method is built on the fact that most of the subversions or conflicts between features happened across the boundary condition of features.

To maintain smooth transitions between process stages, we keep a sharp decomposition based on features throughout the whole development period. The requirement is represented by a *feature list*, a development plan is made for each *feature*, the software is designed by *features*, and finally the software is built by *features*. By using aspect-oriented programming technology, a feature can even be localized within its own module, becoming a truly separate entity [16][18].

The refinement of FDD is briefly showed in Table 1.

Table 1: a refinement to FDD

Num	*Activity*	*Refinements*
# 1	Build an overall model	Preserve without refinement
# 2	Work out a feature list	Preserve all the principles of identification of features. All the "rules", "policy", "non-functional and functional concerns", "architectural concerns", "properties" are uniformly represented as "features". Use the "boundary condition exploration" technique to detect inconsistency/conflicts/subversion between features
# 3	Plan by features	Preserve all the principles of planning. And the plan should consider the influence of adopting aspect-oriented development.
# 4	Design by features	Adopting an aspect-oriented design principle and notation. Features and conflict resolutions are designed as "aspects".
# 5	Build by features	Adopting an aspect-oriented programming language. Features are implemented as "aspects".

Here we use some examples to illustrate how different forms of requirement information can be represented as "features". We use the feature template of section 3 to describe all the features. For example, in an E-learning system we are currently developing, there is a pedagogical rule described as follows:

If a learner answers a question wrong in a Quiz *then* guide him/her to a corresponding study unit.

This rule can be transformed to a feature style in FDD as follows:

Direct *learner* to *a particular study unit* for a wrong answer to *a question* in a *Quiz*.

- where words in italics are potential classes *of object.*

Regarding the early aspects work of [19] and [10], all aspectual requirements are described as "concerns" with each concern being defined in a XML document. In fact, each concern corresponds to a feature set in FDD. For example, one such concern is described as follows:

```
<?xml version = "1.0" ?>
<Concern name="Compatibility">
 <Requirement id="1">
   The system must be compatible with systems used to
    <Requirement id="1.1">
         read the gizmo identifier;
    </Requirement>
    <Requirement id="1.2">
         deal with infraction incidents;
    </Requirement>
    <Requirement id="1.3">
         charge for usage;
    </Requirement>
 </Requirement>
</Concern>
```

Note that each sub concern looks very similar to the FDD feature template.

Furthermore, most requirement examples in [9] are typically architectural features by nature. For example:

Persistency Aspects

+save and load function

- file manager

- remote data manager

Where: "+" denote functionality is *provided* by components, while "-" means the functionality is *required* from some other components

Figure 1. An aspectual requirement in [9]

While this is quite different to the FDD way of describing features, it can still be represented naturally by feature templates, in which, a feature set named "Persistency" is created, with the following feature included in the feature set:

-Save&load *object/components* from/to *the storage*.

-Request to *file manager* for *storage services*

-Request to *remote data manager* for *data*.

A more complete example is from a *building control system*. A feature list of the subsystem "*Room light control* " derived from [1] is given as follow.

Table 2: A feature list in a building control system

ID	Name	Description
f01	Illuminance	Turn off the lamplight of a room for saving energy
f02	EnergySaving_Blind	Control a list of blinds for energy saving.
f03	EnergySaving_Lamp	Control a list of lamplight for energy saving
f04	GlarePrevention_Blind	Control a list of blinds for glare prevention
f05	LightTurnOn_Request	Turn light on or off on request
f06	BlindOnOff_Request	Open or close blind on request
f07	RoomStatus	Determine and report if a room is being occupied
f08	RadiatorControl	Control room temperature by using the radiator
f09	RadiatorValve_Request	Open or close radiator valve on request
f10	TemperatureStatus	Determine and report current temperature

The features and feature set are organised into hierarchical structure and prioritised, weighted in the process "working out a feature list".

With the completion of a feature list, a "plan by feature" process can be carried out with the purpose of "making high-payoff results" in mind.

In the process of "design by feature", a feature's functionality specification is viewed as "the core business logic" or "hard logic", which is designed into one or more classes in an OO paradigm. For example, the core business logic of the feature "Illuminance" can be designed as in Figure 2 (in pseudo java code).

For the interaction, composition and connection between features, modules called "resolutions" or "soft logic" are designed to guarantee the inter-working of features. A resolution module can be designed in a pseudo AspectJ [15] as showed in Figure 3.

```
public class Illuminance {
 ArrayList rooms;
 public void peopleArrive(in roomNumber){
  turnOn(roomNumber);
 }
 public void turnOn(int roomNumber){
//code used for turning on the light in a
//room with a known number
 }
 public void peopleDepart(int roomNumber){
  turnOff(roomNumber);
 }
 public void turnOff(int roomNumber) {
//code used for turning off the light in a
//room with a known number
 }
}
```

Figure 2. Illuminance's hard logic that implements exactly the specification of the feature

```
aspect  ResolutionForIllmnce_EnergySaving
{
 void around(int roomNumber):
       call(void Illuminance.turnOn(int)) &&
                          args(roomNumber)
  {
   if (isDayLightSuitable(roomNumber))
      new EnergySaving().open(roomNumber);
   else proceed(roomNumber);
  }
 boolean isDayLightSuitable(int roomNumber){
   .......;
  }
 void around(int roomNumber):
    call(void Illuminance.turnoff(int)) &&
                            args(roomNumber)
  {
    if (isDayLightSuitable(roomNumber))
       new EnergySaving().close(roomNumber);
     else proceed(roomNumber);
  }
}
```

Figure 3 Resolving Illuminance vs. EnergySavingFeature Interactions

The resolution module ensures that "whenever there is suitable daylight, use the energy saving feature instead the normal illuminance feature".

In the process of "build by feature", all features are carefully examined, unit tested and deployed by using an AOP implementation language.

5. Brief evaluation of FDD refinement

The above examples have highlighted interesting overlaps between FDD and the early stage of the software lifecycle. The proposed refinement to FDD (table 1) attempts to integrate AOSD techniques into FDD. More specifically, the refinements preserve all the good properties of FDD, and at the same time, introduce the benefit of aspect-oriented techniques. The improved separation of concerns in all the processes helps control the complexity of software development, which in turn, helps to improve the maintenance and the capability of evolution (feature modification, replacement, addition and deletion, etc.). The assistance of inconsistency detection and avoidance facilitates secure and high quality software development, and supports the integration of new features in the future. Furthermore, the smooth transitions from requirement analysis to design and then to the implementation stage make the implementation more aligned to the requirements, which is highly helpful for the maintainability and the application of generative programming.

The refinement of FDD by adopting aspect-oriented techniques has so far had a positive impact on the later stages, and has allowed us to elegantly design and implement features, that are also faithful to the feature specification. For further information on the design and implementation of features in an aspect-oriented technique, we refer the interested reader to [16] and [18].

6. Conclusions and Future Work

In summary, FDD is an excellent agile process, and its development strategy bears similarities to that of aspect-oriented techniques. However, because it is not originally designed for aspect-orientation, it still lacks some properties to be a complete aspect-oriented solution to software development. A refinement based on an aspect-oriented technique can make the first three FDD processes good candidates for early aspects, namely aspect-oriented requirement analysis.

With the three feature representation examples, we conclude that the idea "early aspects can all be uniformly described with a feature template, then planned, designed and implemented later with an aspect-oriented programming technique" is viable.

While this theory came from our software development practices, it is vital to use this theory back to the real world development. The future work is to study the usefulness of this methodology in a variety of domains, such as server-side development, Grid/web service, P2P, Internet telephony and reactive control systems etc. It is also important to collect cases of feature (or aspect) conflict/subversion, and abstract the resolution pattern for the interworking of features.

References

[1] Andreas Metzger, Christian Webel, “Feature Interaction Detection in Building Control Systems”, in [20], pp60-66, June 2003.

[2] L.Blair. & J.Pang., “Feature Interactions - Life Beyond Traditional Telephony”, In [6], pp 83-93, 2000.

[3] C. R. Turner, A. Fuggetta, L. Lavazza, and A. L. Wolf. A conceptual basis for feature engineering. Journal of Systems and Software, 49(1):3--15, Dec. 1999

[4] P.Zave, "Requirements for evolving systems: A telecommunications perspective", in Proceedings of the Fifth IEEE International Symposium on Requirements Engineering, pages 2-9, IEEE Computer Society, 2001.

[5] Clements P., Northrop L., Software Product Line Practice Patterns. Addison-Wesley, 2001.

[6] M.Calder, E.Magill , editors, “Feature Interactions in Telecommunications and Software Systems VI”, Glasgow, Scotland, IOS Press(Amsterdam), 2000.

[7] P.Zave, “ FAQ Sheet on Feature Interactions ”, AT&T, 2001, Available via: http://www.research.att.com-/~pamela/faq.html.

[8] M.A.Cusumano and R.W.Selby, “Microsoft Secrets: How the World’s Most Powerful Software Company Creates Technology, Shapes Markets, and Manages People”, Simon & Schuster, 1998. ISBN: 0684855313.

[9] J. Gundy, Aspect-Oriented Requirements Engineering for Component-based Software Systems”, 4th IEEE International Symposium on RE, 1999, IEEE Computer Society Press, pp. 84-91.

[10] Rashid, A., P. Sawyer, A. Moreira and J. Araujo."Early Aspects: A Model for Aspect-Oriented Requirements Engineering". IEEE Joint International Conference on Requirements Engineering. IEEE Computer Society Press. Pages 199-202.2002

[11] I.Sommerville and P.Sawyer, Requirements Engineering – A Good Practice Guide: John Wiley and Sons, 1997.

[12] P.Coad, J.de Luca, E.Lefebrve, *Java Monitoring in Color with UML*. Chapter 6:Feature Driven Development, Prentice Hall, 1999.

[13] P.Coad and S.Palmer, Feature-Driven Development, TogetherSoft Corporation 2002, Available via http://www.nebulon.com/articles/fdd/latestprocesses.html.

[14] S.Palmer, Feature-Driven Development and Extreme Programming, informIT, 2002, Available via http://www.informit.com.

[15] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm and W. Griswold, Getting started with ASPECTJ , Communication of the ACM, Vol. 44, Issue 10, ACM Press (New York), pp59-65, October 2001.

[16] J. Pang, L. Blair, "Separating Interaction Concerns from Distributed Feature Components", Proceedings Software Composition workshop (SC 2003), held in conjunction with ETAPS 2003, Warsaw, Poland, Electronic Notes in Theoretical Science, Vol. 82, Issue 5, Elsevier, April 2003.

[17] E.Jane Cameron, Nancy D.Griffeth, Y. Lin, M.Nilson, W.Schnure, and H.Velthuijsen. A feature-interaction benchmark for IN and beyond. IEEE Communications XXXI(3):64-69, March 1993.

[18] Blair L., Pang J., "Aspect-Oriented Solutions to Feature Interaction Concerns using AspectJ", in [20], pp87-104, June 2003.

[19] Rashid, A. Moreira, and J. Araujo."Modularisation and Composition of Aspectual Requirements". 2nd International Conference on Aspect-Oriented Software Development. ACM. Pages 11-20.2003

[20] D. Amyot, L. Logrippo (eds), “Proceedings 7th International Workshop on Feature Interactions in Telecommunications and Software Systems (FIW'03), Ottawa, Canada, Amsterdam: IOS Press, June 2003.

On Imperfection in Information as an "Early" Crosscutting Concern and its Mapping to Aspect–Oriented Design

Miguel–Ángel Sicilia, Elena García
Computer Science Department, University of Alcalá
Polytechnic School, Ctra. Barcelona km 33.6
28871 – Alcalá de Henares (Madrid), Spain.
{*msicilia,elena.garciab*}*@uah.es*
http://www.cc.uah.es/msicilia

Abstract

Imperfection in information can be considered a cross–cutting concern that encompasses diverse kinds of imprecision, uncertainty or inconsistency management inside a software system. An early consideration of the type(s) and required level of imperfection handling for the system is necessary to properly inform design, and to serve as criteria for choosing the appropriate mathematical model(s) that will be implemented in the final software. In this paper, imperfection is discussed as an integral part of a concern–oriented approach to requirements and domain engineering, and some illustrative examples of the mapping of these concerns in aspect–oriented design are also provided.

1. Introduction

The *separation of concerns* principle has recently been applied to early stages of development like requirements engineering [15, 7, 3] and software architecture [16]. The early separation of cross–cutting concerns results in improved localization, and eventually in improved development and maintenance activities. In the research literature, a number of examples of cross–cutting concerns are often used for illustration purposes, or they are described when reporting case studies. Recurring examples include, for example, security, usability, persistence or performance.

The focus of this paper is that of *imperfection* in information as a cross–cutting system concern that is currently overlooked in many application models. Imperfection is a multifaceted concept including imprecision, uncertainty and inconsistency, being classical probability a model for a specific type of imperfection among many others. Currently, general and mature mathematical frameworks for the management of imperfection are available [5], although their widespread use in mainstream development technologies and industrial systems is still to come. To adhere to an unambiguous interpretation of the terms used for the various sub–aspects of information imperfection, we'll use them in the sense given in Smet's taxonomy described in [14]. It should be noted that this taxonomy reflects the diverse concepts of imperfection in information, and not their mathematical handling, so that it is related to *early* domain modelling, that will be *later* mapped to a concrete representation.

Imperfection in information should be addressed early in the lifecycle due to the specifics of uncertainty and imperfection in conceptual modeling [2], and its impact on architectural and implementation decisions, most notably in persistence and querying [10]. In any system dealing with imperfection, a mathematical model (or several of them) for its representation must be selected, according to the concrete concerns stated in the requirements and domain models. To do so, enough detail must be provided so that system tests could eventually be derived from them, in order to assess the validity of the representation chosen with regards to the required capabilities of the system.

In this paper we approach the problem of specifying information imperfection in software requirements and domain models as cross–cutting concerns, and some examples of mapping such concerns into aspects at the design and implementation stages. The motivation of our present work is that of initiating research in Software Engineering models that are conceptually connected with the diverse mathematical representations of imprecision and uncertainty, helping practitioners to make decisions regarding the use of the growing research results in the area. It should be noted that our focus is modeling imperfection that will be managed in the final system. Previous related work, e.g. [6] has focused on imperfection in the modeling process itself.

The rest of this paper is structured as follows. Section

2 examines how imperfection can be specified as an integral part of concern–oriented requirements and domain engineering activities, to a level of detail sufficient to inform subsequent architectural and modularization decisions. In Section 3, illustrative examples are given about the mapping of domain–level concerns to implementation structures. Finally, conclusions and a future outlook are provided in Section 4.

2. Imperfection as a Concern in Requirements and Domain Engineering

Information imperfection usually arises at the early stages of the development, since it pervades the description of the domain or real world situation. In fact, classical error theory that manifests in required degrees of precision in numerical computations is just a model to deal with a facet of imperfection related to numerical representation and error in measurement instruments. Probabilistic or statistical considerations arising, for example, in regression models or questionnaire–based tests are just a manifestation of uncertainty. In addition, some classes of systems work in a context that inherently entails imprecision and uncertainty, like personalized Web systems that use user's navigation to tailor their structure [12].

Information imperfection is a logical "matter of interest", according to COSMOS [15] terminology [1]. It can be organized in several *classifications*, one for each of the principal aspects of imperfection: `Imprecision-Related`, `Uncertainty-Related`, `Inconsistency-Related` or `Hybrid`. Classes inside those categories may refer to more specific types of imperfection according to Smet's taxonomy, e.g. `FuzzyElement` refers to "imprecision without error" without decidability as in "age is close to 30", while `PossibleElement` refers to "happen–ability" as a kind of uncertainty. In addition, imperfection manifestations can be classified in `Domain-Imperfection`, `UserImperfection` and `System-Imperfection` according to the source from which the imperfection originates. For example, inferences internal to the system may generate imperfect information from perfect inputs. According to the kind of element in the conceptual model that is subject to imperfection, we can have additional classifications, namely `ImperfectElement` and `ImperfectRelationship`, the former containing classes `ImperfectClass`, `ImperfectAttribute`, `ImperfectFunction` and `ImperfectResult` which roughly correspond to classes, attributes, method (or more generally, functionality), and method results in conceptual models, and the latter containing classes for each type of relationship (association, generalization, etc.). Imperfect conceptual model elements can be expressed in the domain model through extensions to the `UML` like the one sketched in [9]. Figure 1 depicts some of the just described elements and some of example relationships between them, representing classifications as `UML` packages.

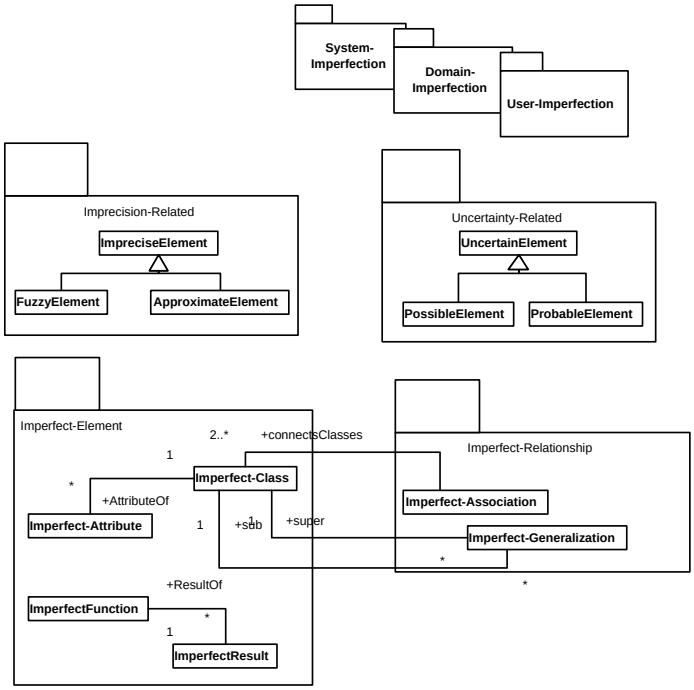


Figure 1. General concern dimensions related to information imperfection

Properties as defined in COSMOS are "concerns that characterize other logical concerns". Concerns that arise in the context of imperfection include `Granulation-level` and `Interpretability`. The former refers to the degree of "summarization" of an element, e.g. "large pages" may be interpreted to subsume "very–large" and "moderately large", thus summarizing information. The latter refers to the ease of interpretation of the information by humans, and is often considered as a quality criteria in rule bases. They can be used as requirements constraining the design of other concerns, e.g. `High-Interpretability` may involve selecting a fuzzy rule simplification algorithm at later stages for the computation of a given conceptual model element.

It should be noted that the imperfection–related concepts introduced so far do not require understanding neither about the mathematical frameworks for uncertainty handling not about their software representation, so that they can be con-

[1] In what follows, concern–modeling concepts are taken from COSMOS, in spite of the fact that the matter dealt with here is related to a greater extent to domain modeling, while COSMOS is intended for earlier stages of development.

sidered a concern in domain and requirements engineering.

In what follows, we briefly sketch concern space analysis with regards to imperfection for two case studies.

2.1. Case Study: Market Segmentation

Market segmentation systems provide support for the classification of customers with the purpose of targeting marketing strategies [17]. Market segmentation criteria are in many cases uncertain, and the resulting subsets can be considered to have no sharp boundaries, specially in the new relationship marketing paradigm [13].

A basic model for market segmentation using the relationship value model in [13] will result in the classes and instances showed in Table 1.

Table 1. Example imperfection–related concerns in a basic market segmentation setting.

Instances	Classes/classifications
Expected-Incremental-Purchases	SubjectiveUncertainElement, User-Imperfection, ImperfectAttribute
Estimated-Relationship-Duration	SubjectiveUncertainElement, User-Imperfection, ImperfectAttribute
Relationship-Value	ImpreciseElement, System-Imperfection, ImperfectResult
Customer-Segment	FuzzyElement, System-Imperfection, ImperfectClass
Customer-Similarity	FuzzyElement, System-Imperfection, ImperfectAssociation
Net-Relationship-Value-Model	FuzzyElement, System-Imperfection, ImperfectFunction

Segments can be considered imperfect classes of users inferred by the system from a net value relationship model that estimates the value of each customer relationship from rough estimates of increments in purchases and expected relationship duration. Those estimates are rough pessimistic and optimistic values obtained from experts. Customer similarity is derived from the segments and perhaps from the analogies in purchasing behavior of a pair of customers, as often interpreted in recommender systems [8].

2.2. Case Study: Adaptive Learning System

Adaptive Learning Systems are characterized by tailoring the hypermedia structure of the learning contents to the knowledge and/or characteristics of the learners. The case study described here is based in the adaptive technology described in [11] where a rule–based system was used to adapt the presentation of links ("fuzzy links") in the courseware. The main imperfection–related concerns are showed in Table 2.

Table 2. Example imperfection–related concerns in a rule–based adaptive Web system.

Instances	Classes/classifications
Knowledge-about-Lesson$_i$	SubjectivePossibleElement, System-imperfection, Imperfect-Association-Attribute
Fuzzy link	FuzzyElement, Domain-Imperfection, Imperfect-Class
Rule$_1$, ... Rule$_n$	FuzzyElement, User-Imperfection, Imperfect-Function
User-Categorization-Inference	UncertainElement, FuzzyElement, System-Imperfection, Imperfect–Class
Content-Tailoring-Inference	FuzzyElement, System-Imperfection, Imperfect–Attribute
Content-Tailoring-Algorithm	FuzzyElement, System-Imperfection, Imperfect–Function
Learner-Style	ObjectivePossibleElement, FuzzyElement, Imperfect-Class, Uncertain-Class

The knowledge a given learner is supposed to have about a given lesson can be modeled as an association attribute, a specialization of `Imperfect-Association`. The kind of uncertainty for such knowledge levels is considered in terms of epistemic possibility. Fuzzy links represent typed semantic relationships about nodes or contents, so that the vagueness expressed in them come from the domain being teach. Rules use fuzzy links and other model characteristics both to infer imprecise categories of users and also to establish personalized content attributes for each user or group. Rules are elicited from experts, so that it can be considered that the source of imperfection are a special kind of users. Content tailoring algorithms are an alternative to rules in the form of predefined function that obtains imprecise personalization of content attributes. Learner styles are both uncertain and imprecise, since the techniques used to obtain them, although based on recurring patterns of interaction, do not provide reliable answers.

3. Mapping Concerns of Fuzziness to Aspect–Oriented Design

In this section, a number of concrete mappings of information imperfection concerns are described for the purpose of illustration.

3.1. Fuzzy arithmetics

Numerical `ImperfectAttribute`s can be mapped to fuzzy numbers and fuzzy arithmetics [4], irrespective of the kind of imperfection (imprecision, uncertainty or both) they represent. This is the case of the uncertain `Expected-Incremental-Purchases` and `Estimated-Relationship-Duration` values that are used to compute the `Net-Relationship-Value-Model` in the market segmentation case study. The design of systems dealing with fuzzy numbers requires a library supporting them like the `Fuzzy Java Toolkit`. Nonetheless, it would be desirable that fuzzy arithmetics become an standard part of programming language libraries, so that developers could select using crisp or fuzzy arithmetics. The arithmetics of fuzziness can be modularized in aspects that modify the behaviors of classes representing numbers. For example, let's consider an hypothetical extension to Java `Double` class providing arithmetic functions [2] with an interface like the following:

```
public class ArithmeticDouble extends Double{
   public add(Double x){ ... }
   public times(Double x){ ... }
   //...
}
```

Fuzzy arithmetic can be modularized in aspects by *introducing* attributes to describe the shape of the fuzzy numbers and intercepting calls to *getter* methods like `doubleValue()` to compute the appropriate defuzzification of the triangular number, as sketched in what follows:

```
public aspect FuzzyArithmetics{

  // Introduces upper and lower bounds
  // of triangular fuzzy numbers:
  private double ArithmeticDouble.upper;
  private double ArithmeticDouble.lower;


  double around  (ArithmeticDouble d):
        target(d) &&
        call(public double doubleValue()){
        double crisp = proceed(d);
        // defuzzify triangular number
        // if required...
        return crisp;
  }


}
```

This enables the co–existence of normal and fuzzy numbers in the same framework, as an option to changing programming language support by special–purpose interfaces.

3.2. Fuzziness in databases and queries

In almost every application, the use of specialized representations for imperfect information result in special persistence requirements. In consequence, database programming interfaces require extensions for such purpose. Concretely, here we focus in orthogonal persistence interfaces similar to those exposed by `Jdo`[3]. The extension for fuzziness of such kind of interfaces can be accomplished by adding elements to the query syntax — as is done in [1]— and also by augmenting programming interfaces to deal with the desired fuzzy modeling capabilities — e.g. as in [10]. Aspect–oriented design can be used to extend existing programming libraries for fuzziness without obscuring their original design. In what follows, we briefly sketch some design points of the extension of the `OJB`[4] interfaces for illustration purposes.

The core of such extensions is adding new schemata that van be made processed by the libraries by an aspect like the following:

```
public aspect FuzzyMetadataManagement
{
    private DescriptorRepository globalRep;
    void around (MetadataManager m):
         target(m) && call(
           * MetadataManager.init(..)){
        try{
                proceed(m);
        }catch(MetadataException e){throw e;}
        globalRep = loadFuzzyDesRep();
        m.mergeDescriptorRepository(globalRep);
    }
private DescriptorRepository loadFuzzyDesRep()
 {  //  load descriptor repository..  }
    //...
}
```

Then, the extended meta–schema describing the storage details of each kind of imperfect modeling concern becomes available. For example, explicit storage of membership values of objects belonging to fuzzy classes can be achieved through the following design, which introduces a new method in the `PersistenceBrokerImpl` class.

[2] this is actually not possible, since the class is declared `final`.

[3] http://access1.sun.com/jdo/

[4] http://db.apache.org/ojb/

```
public aspect FuzzyStorageHandling{
    public void PersistenceBrokerImpl.store(
     Object obj, Double m,  String fuzzyClass)
     throws PersistenceBrokerException {
        store(obj);
        storeMembership(obj, m, fuzzyClass);
    }
 //...
}
```

Such aspect code could be automatically generated from models with elements that have been specified with the `FuzzyElement` and `ImperfectClass` model concerns. In addition, the retrieval of fuzzy grades from the database can be accomplished by wrapping result collections with the following design:

```
public aspect FuzzyObjectWrapping{
    Collection around (PersistenceBroker p):
    target(p) && args(query)
    && call(Collection PersistenceBroker.
    getCollectionByQuery(Query query)){
           Collection aux=null;
             try{
                  aux =  proceed(p, query);
            }catch(PersistenceBrokerException e)
                {throw e;}
            return this.wrapResultCollection(aux);
    }
    // ...
}
```

Such kind of aspect–oriented design elements point out the possibility of a comprehensive model–based database implementation of the concerns of fuzziness through standard mappings.

4. Conclusions and Future Research Directions

Information imperfection can be considered a cross–cutting concern that arises at early stages of the development lifecycle. A concern–oriented requirement and domain analysis process can be used for the early specification of requirements for imperfection handling and their associated domain model elements, so that design and implementation decisions can be based on them. A tentative concern space analysis for information imperfection has been described, along with some examples of the mapping of concerns to specific aspect–oriented design options.

Further work is needed in the analysis of concerns regarding imperfection and its mapping to the range of available mathematical models that can be used to map them into design. Such analysis would eventually come up with the seamless integration of Fuzzy Set Theory and other related frameworks [5] into the software engineering process, facilitating the adoption and development of fuzzy techniques in all industrial areas. In addition, the UML language and its associated tools should be extended to explicitly address the requirements of requirements and domain engineering regarding imperfection (preliminary work is described in [9]).

References

[1] Callens, B. de Tré, G., Verstraete, J., Hallez, A.: A Flexible Querying Framework (FQF): Some Implementation Issues. Lecture Notes on Computer Science 2869: Proceedings of International Symposium of Computer and Information Sciences (ISCIS) 2003, 260–267.

[2] Chen, G. Fuzzy logic in data modeling : semantics, constraints, and database design, Kluwer Academic Publishers, 1998.

[3] Grundy, J. Aspect-Oriented Requirements Engineering for Component-based Software Systems. In Proceedings of the 4th IEEE International Symposium on Requirements Engineering. IEEE Computer Society Press (1999): 84–91.

[4] Kaufmann, A., and M. M. Gupta. Introduction to fuzzy arithmetic: theory and applications. New York: Van Nostrand Reinhold, 1985.

[5] Klir, G., Wierman, M.: Uncertainty-Based Information: Elements of Generalized Information Theory. Springer-Verlag (1998).

[6] Marcelloni, F. and Aksit, M. Fuzzy logic based object-oriented methods to reduce quantization error and contextual bias problems in software development. Fuzzy Sets and Systems (2004) (to appear).

[7] Rashid, A., Sawyer, P., Moreira, A. and Araujo, J. Early Aspects: A Model for Aspect-Oriented Requirements Engineering. In Proceedings of the IEEE Joint International Conference on Requirements Engineering. IEEE Computer Society Press. (2002): 199–202.

[8] Sarwar, B. M., Karypis, G., Konstan, J. A., and Riedl, J. "Analysis of Recommender Algorithms for E-Commerce". In proceedings of the ACM E-Commerce 2000 Conference. Oct. 17-20, 2000, pp. 158-167.

[9] Sicilia, M. A., García, E., Gutiérrez, J. A. (2002). Integrating fuzziness in object oriented modelling languages: towards a fuzzy-UML. In: Proceedings of the International Conference on Fuzzy Sets Theory and its Applications (FSTA 2002), 66-67.

[10] Sicilia, M.A., García, E., Díaz, P. and Aedo, I.: Extending Relational Data Access Programming Libraries

for Fuzziness: The fJDBC Framework. In: Proceedings of the 5th International Conference on Flexible Query Answering Systems. Lecture Notes in Computer Science 2522, Springer (2002):314–328

[11] Sicilia, M.A., García, E., Díaz, P., Aedo, I. (2002): LEARNING LINKS: Reusable Assets with Supports for Vagueness and Ontology-Based Typing. In *Proceedings of the International Conference on Computers in Education*, 1567-1568

[12] Sicilia, M.A. (2003). Observing web users: conjecturing and refutation on partial evidence. In *Proceedings of the 22nd North American Fuzzy Information Processing Society*, NAFIPS 2003, 530 -535.

[13] Sicilia, M.A., García, E. On Fuzziness in Relationship Value Segmentation: Applications to Personalized e-Commerce. ACM SIGECOM Newsletter, 4(2):1–10.

[14] Smets, P.: Imperfect information: Imprecision-Uncertainty. Uncertainty Management in Information Systems: From Needs to Solutions. Kluwer Academic Publishers (1997), 225-254.

[15] Sutton Jr., S.M. and Rouvellou, I. Modeling Software Concerns in Cosmos. In *Proceedings of the First International Conference on Aspect–Oriented Software Development* (AOSD 2002), Enschede, The Netherlands, Apr. 2002, ACM Press, 127-133.

[16] Tekinerdogan, B. ASAAM: Aspectual Software Architecture Analysis Method. In Proceedings of the Aspect–Oriented Requirements Engineering and Architecture Design Workshop, Boston, US, 2002.

[17] Wedel, M., Kamakura, W.A. 1999. Market Segmentation: Conceptual and Methodological Foundations, Kluwer Academic Publishers, 2nd edition.

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.

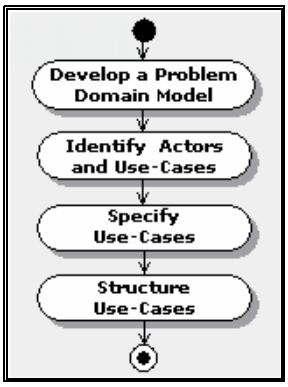


Figure 1 – Requirements activities

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.

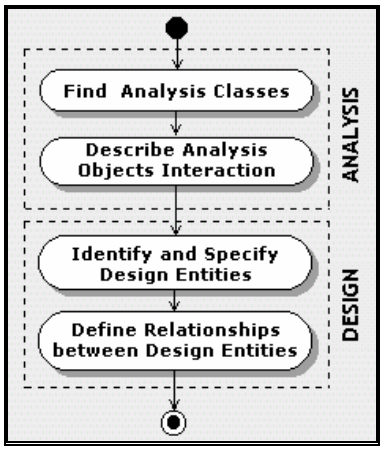


Figure 2 – Analysis and Design activities

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 "!!").

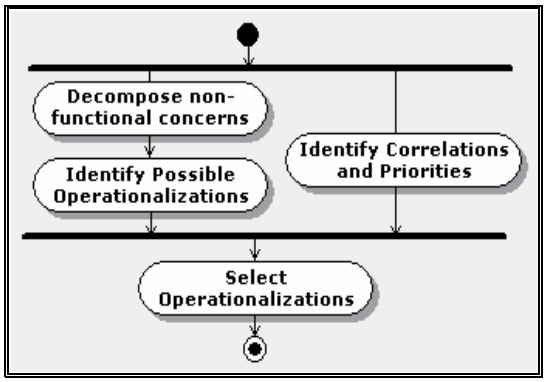


Figure 3 - NFR Framework Activities

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.

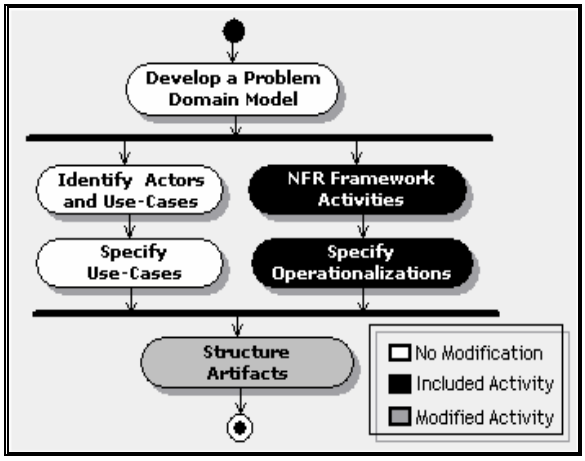


Figure 4 - Requirements Activities of Our Proposal

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.

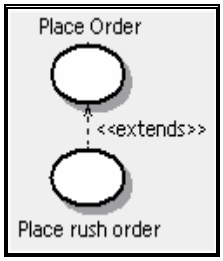


Figure 5 – Extension example (reproduced from [7])

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 includes 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 extends 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 crosscuts 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.*

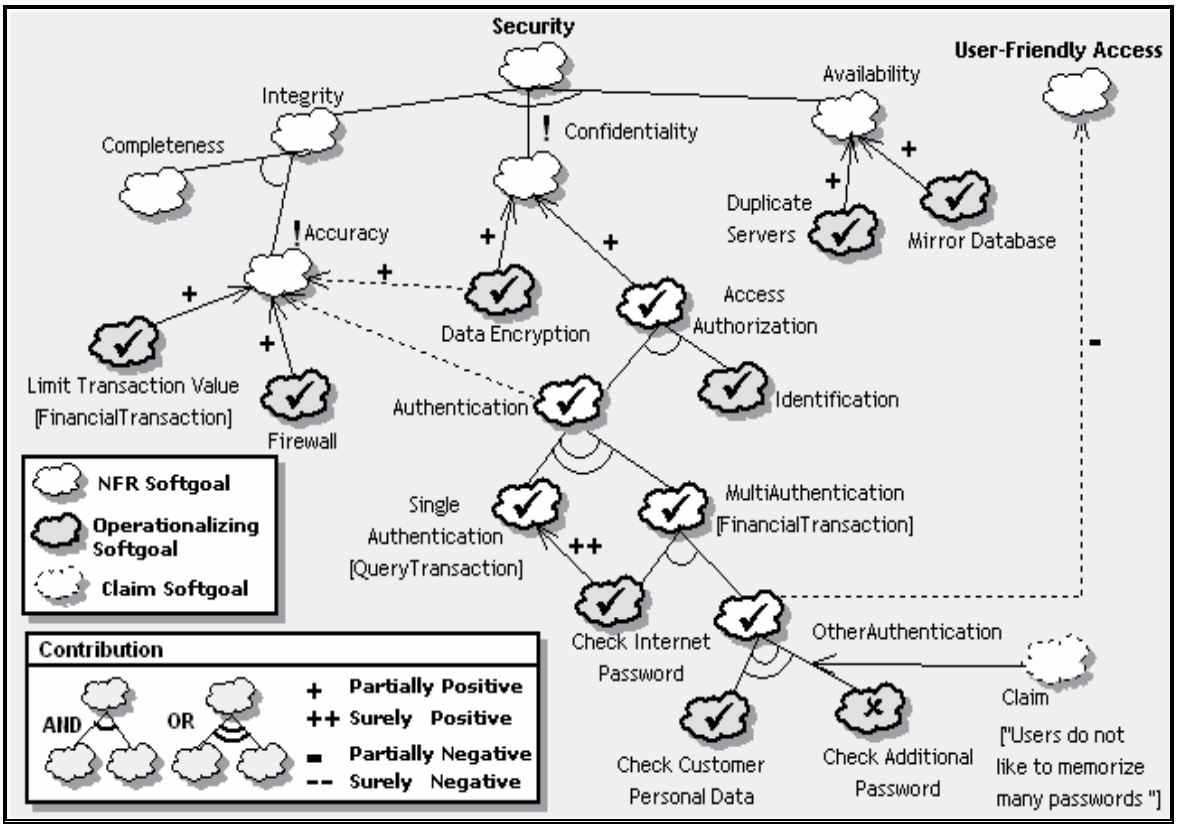


Figure 6- Softgoal Interdependency Graph for Security non-functional concern

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.

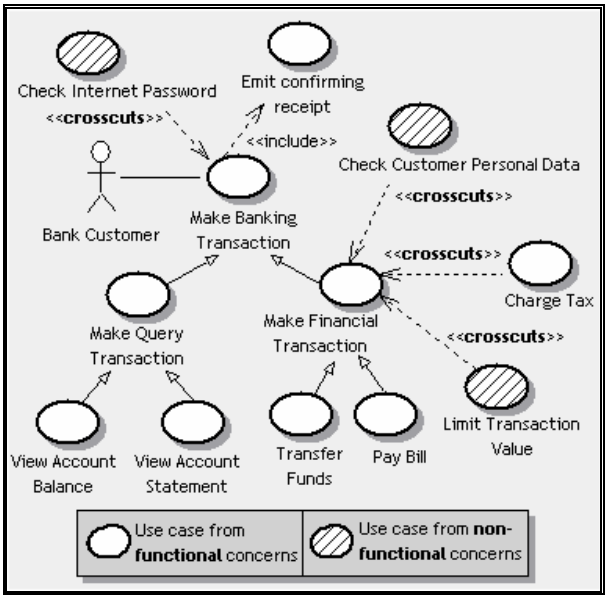


Figure 7 – Structured Use Case Diagram

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.

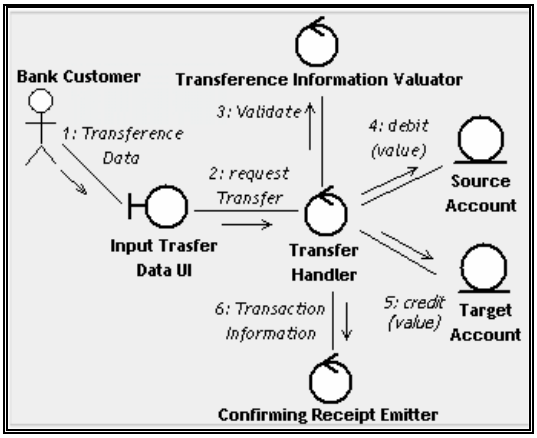


Figure 8 – A Collaboration Diagram for a realization of the *Transfer Funds* use case

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.

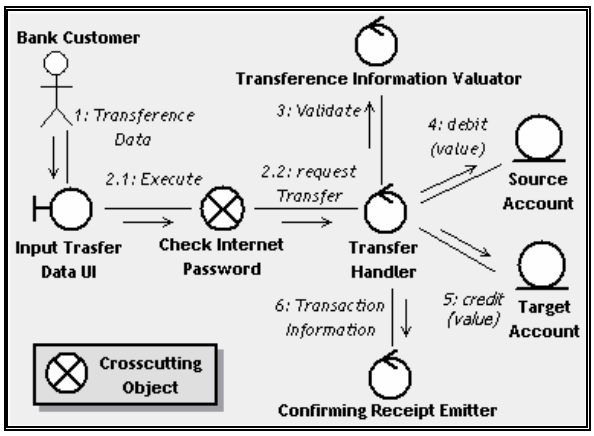


Figure 9 – Crosscutting View of the *Transfer Funds* Collaboration Diagram

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.

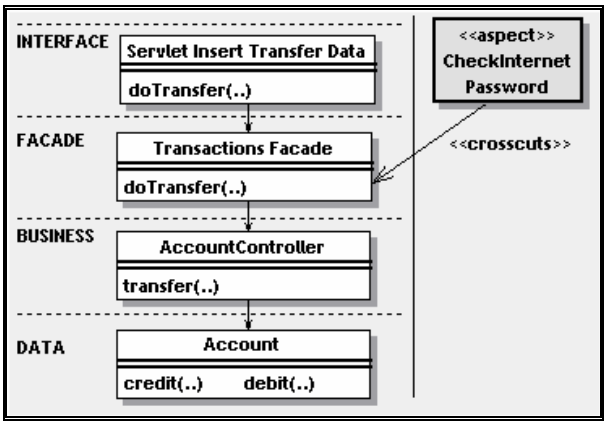


Figure 10 - Relationships between Design Entities

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.

Modeling Pointcuts

Dominik Stein, Stefan Hanenberg, and Rainer Unland
Institute for Computer Science and Business Information Systems
University of Duisburg-Essen, Germany
{dstein | shanenbe | unlandR}@cs.uni-essen.de

Abstract

Modeling pointcuts, i.e., modeling the places where to crosscut and/or the conditions under which to crosscut, is a principal task in aspect-oriented modeling. It is a fairly independent design issue and can be accomplished separate from other modeling tasks such as modeling the crosscutting effects. Modeling pointcuts is basically about modeling selection queries. It requires novel modeling means. This paper gives a short overview on a new graphical approach to model pointcuts. It presents its semantics using OCL code. It presents its use in the early aspect phase, and it demonstrates its capabilities to capture the pointcut semantics of prevailing aspect-oriented programming techniques with help of examples.

1. Introduction

Aspect-Oriented Software Development (AOSD) is about encapsulating crosscutting concerns in aspects, and weaving these aspects together with other aspects and basic functionalities to a final aspect-enhanced program. When designing aspects or aspect-oriented software architectures, a principal task of aspect-oriented software developers is to investigate the relationships between aspects and their target artifacts. The software developer needs to contemplate about the aspect's assertions made on the final program as well as on the places and conditions at/under which the assertions apply to the program. Ultimately, the aspect-oriented weaver will take this information and enhance the final program accordingly. The key goal of aspect-oriented software development is to make reuse of aspects, i.e., crosscutting concerns, as easy as adjusting places and conditions of crosscutting to new programs and/or new requirements. To achieve this goal, aspect-oriented designers require means to specify such places and conditions of crosscutting from the early stages of architecture design.

Currently, aspect-oriented software development is greatly advancing on the implementation level. However, comprehensive design support is still in its infancy. Promising methodologies for requirements analysis, architecture design, and graphical visualizations are around (e.g., [13], [32], [35], [20], [30], [19], etc.); yet improvements are necessary to span the entire software life cycle and to cope with multiple aspect-oriented implementation techniques.

This paper deals with modeling and graphical visualization of places and conditions of crosscutting. Defining places and conditions at/under which an aspect affects the final program turns out to be a critical design activity in AOSD. Pointcut design in aspect-oriented architecture design is likewise crucial as interface design in conventional architecture design since both specify the connection points between different software artifacts of the architecture. Inaccurate pointcut definition may thus easily require fundamental redesign of the aspect and/or the architecture at a later stage in software development. Furthermore, lax definition of places and conditions of crosscutting quickly designates much more parts in the final program than was intended. Such will severely obstruct reasoning on the crosscutting effects of aspects and, in the ultimate, lead to unpredictable software comportment. Hence, it is essential that software designers are supported from the early stages of system design in identifying and documenting (!) the places and conditions at/under which an aspect crosscuts the final program.

Our work is focused on the Unified Modeling Language (UML) [28]. The UML is a powerful and broadly used modeling language for use case driven, architecture oriented, iterative, and incremental software development [12]. It may be used throughout the entire software development process [18] and may be used with different object-oriented implementation languages. Hence, the UML already provides much of what we are trying to achieve for aspect-oriented software development. It appears very appealing to exploit its capabilities for aspect-oriented software modeling.

Aspect-oriented modeling using the UML has been one of the core subjects of the ongoing series of workshops on Aspect-Oriented Modeling [1] [2] [3] [4]. The fruitful discussions and important insights given at these workshops have strongly influenced this work.

In this paper, we borrow the terms “join point” and “pointcut” from AspectJ [9] terminology. In difference to AspectJ, however, we contemplate on “join points” as “hooks where enhancements may be added” (cf. [14]) rather than as “principal points in the execution of a program” (cf. [10]). That means, we consider “join point” to refer to both points of crosscutting in the control flow as well as points of crosscutting in the class structure. We shall use the term “pointcut” to refer to a set of join points that possibly is attributed with conditions of crosscutting.

The remainder of this work is structured as follows: Section 2 discusses why pointcut modeling is a distinct design issue and why it must be given special care. We do this while first looking at the general case in aspect-oriented requirement engineering [30]. Afterwards, we reflect on the current aspect-oriented programming techniques. Section 3 presents our approach to model pointcuts. It describes the graphical means as well as their semantics in terms of Object Constraint Language (OCL) [37] expressions. Section 4 presents some related work. Section 5 concludes the paper.

2. Pointcut Specification as a Distinct Design Issue

When looking at the specification of crosscutting features in aspect-oriented software development, we identify the specification of the elements (Figure 1, A) that crosscut a given decomposition and the specification of the set of join points (Figure 1, B) where that crosscutting takes place to be two fairly independent issues that can be seen as distinct design problems. We do this because we contemplate that the reasoning on the crosscutting assertions can be accomplished not knowing where exactly that crosscutting assertions are applied to; and vice versa, we suppose we can reflect on the join points at which crosscutting should occur while neglecting what exactly is to be inserted at these points.

Figure 1. Aspect-oriented design issues (non-UML diagram) (cf. [33])

In an aspect-oriented software framework, for example, when we specify a particular synchronization strategy we do not want to determine yet which objects it should be applied to. Oppositely, we could identify the actions that need to be synchronized first, and leave it to a third party to design the synchronization strategy. In a third case, we have both a set of different synchronization strategies and a set of join point collections that designate actions to be synchronized. Now, we are able to combine crosscutting assertions and hooks of crosscutting in any manner, and thus we are set to realize any synchronization requirement by simple hooking the right strategy onto the right set of join points. In conclusion, the separate treatment of crosscutting details and points of crosscutting is a very important issue to achieve incremental programming (cf. [38]). And after all that's what we're heading for in aspect-oriented programming, too.

Looking at current aspect-oriented programming techniques we recognize that most of them commit to this separation of concern. AspectJ, for example, provides advice to specify crosscutting code, and pointcuts to specify where that code is to be introduced into the base program. We may combine advice and pointcuts in arbitrary manners: For example, we can specify an advice that hooks onto an "abstract pointcuts", i.e., empty sets of join points. That set can be filled by diverse subentities later-on for the crosscutting to take effect in multiple concerns. On the other hand, we can specify pointcuts first and let subentities implement the crosscutting behavior that is to be executed at those pointcuts. In Hyper/J [17], hyperslices designate all model elements in a given decomposition that belong to a particular concern (this process is call "concern mapping"). Two hyperslices may be composed by a hypermodule, which contains correspondence rules that determine at what points the hyperslices should be joined. Usually, we specify the hyperslices first and use a hypermodule to join one to the other afterwards. We could, however, also assign the composition of two concerns in advance and then define (or change) the hyperslices (or concern mappings) that are to be involved. That way we can substitute the crosscutting details to be introduced to a particular join point according to our need and desire. In Adaptive Programming [5], the affiliation between the specification of crosscutting details and the specification of crosscutting hooks is much stronger. However, even though we cannot change the one thing without adapting the other, Adaptive Programming distinguishes between traversal strategies that specify the locations at which crosscutting is to take place and visitor methods that specify how these locations are to be augmented.

Of course, even though we may reason on the specification of crosscutting details (Figure 1, A) and the specifications of the hooks (Figure 1, B) separately there certainly exist strong correlations between these two issues. In particular, dependencies arise from the charge of the latter to designate elements in the environment of

the crosscut decomposition (Figure 1, C) that are used by the former. Nevertheless, we will neglect these kinds of dependencies for now and concentrate on the specifications of the hooks, i.e., on the designation of join points and of join conditions. For a closer elucidation on the dependencies between the specification of hooks and the specification of crosscutting details, please refer to [33].

3. Modeling Means for Pointcut Specification

On implementation level, join points represent "hooks where enhancements may be added"[1] (cf. [14]), e.g., classes or method calls. On modeling level, these join points are rendered by model elements in models – may it be a structural model describing a hierarchy of classifiers or a behavioral model describing control flow. On implementation level, pointcuts characterize the (sets of) hooks and/or (optional) conditions at/under which crosscutting takes place. For modeling pointcuts on modeling level, we thus need a means to render a set of model elements together with a set of conditions that must evaluate true for those model elements.

We choose UML Classifiers to represent join points in structural models, and UML Messages to represent join points in behavioral models. For the designation of join points – i.e., UML Classifiers and UML Messages, respectively – we introduce a new graphical means called "Join Point Designation Diagram" (JPDD). JPDDs resemble UML collaboration templates, however they lack the generative semantics of templates. That is, JPDDs describe "selection patterns" rather than "generation patterns". They specify all properties a model element (i.e., UML Classifier or UML Message) must provide in order to represent a join point (rather than the properties that will be added to or modified at those join points). These properties may be of structural or behavioral kind.

Structural properties are defined by means of class diagrams. Class diagrams may be used to model structural conditions of crosscutting, e.g., a particular feature or relationship that must be present for a classifier to represent a join point. Behavioral properties are defined by means of interaction diagrams (i.e., sequence diagrams or collaboration diagrams). Interaction diagrams are used, for example, to model behavioral conditions of crosscutting, e.g., that a particular message must be called within the control flow of some other message in order to represent a join point. JPDDs may contain both class diagrams and interaction diagrams in order to describe structural and behavioral properties at the same time (just like ordinary UML collaborations). Recall, though, that the semantics of class diagrams and interaction diagrams contained in JPDDs is different from their conventional variants as they specify a query on model elements rather than the model elements themselves.

The actual join points in a JPDD are modeled as JPDD's template parameters. JPDDs may designate both kinds of join points – i.e., UML Classifiers and UML Messages – at the same time.

The semantic of JPDDs is specified by means of OCL expressions: Each JPDD can be transformed into a OCL selection query picking out all model elements from a given UML model that represent join points. Those OCL statements make use of various meta-operations that we have appended to the UML meta-classes. Note that not all OCL operations are shown here due to limitations in space. Have a look at [6] to obtain the full OCL code.

In the following, we demonstrate with help of various examples what JPDDs look like and how they can be put to use to model various kinds of pointcuts. For each example, we present the relevant OCL expressions that are involved in matching UML Classifiers and UML Messages with the selection pattern described by a JPDD.

3.1. Pointcuts in the Early Aspects Stage

In the early aspect stage, JPDDs come in when we map aspect-oriented requirements to an aspect-oriented architecture. For example, Figure 2 shows two use cases that model two requirements, one (aspectual) synchronization requirement and some (core) functionality requirement which needs to be synchronized. The crosscutting relationship between one and the other has already been identified[2].

When mapping the requirements to an architecture, we must determine how the software artifact realizing the

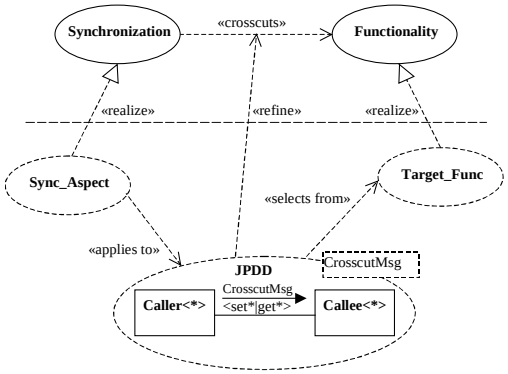


Figure 2. JPDDs in the early aspects stage

[1] Remember the difference we make here to join points in AspectJ terminology, where join points represent "principal points in the execution of a program" (cf. [10]).

[2] Note the subtle yet essential difference between «crosscut» relationships in aspect-oriented software development and «extend» relationships in use case driven software development (cf. [32]).

aspectual requirement connects to the software artifact realizing the functional requirement. This is the task of JPDDs. In Figure 2, for example, the two software artifacts are represented by collaborations. A JPDD is used to characterize the connection points at which the aspectual collaboration crosscuts the functional collaboration. In doing so, the JPDD gives some more details on the «crosscut» relationship between the use cases. The JPDD describes what is expected by the aspect, or what is exposed to the aspect, from the target environment so that it can accomplish its task. In a sense, the JPDD specifies an aspect-specific view on the target artifact.

We can learn from the example in Figure 2 that the synchronization aspect is concerned about synchronizing method calls ("CrosscutMsg") from one entity ("Caller") to another entity ("Callee"), which are expected to be some kinds of classifiers in the target artifact. Further, we can see that the aspect expects the methods' names to begin with "set" or "get". This name restriction may originate in a design decision on the aspectual or the functional requirement. For some (incomprehensible) reason, for example, we could be required to distinguish between synchronized "setter" and unsynchronized "putter" methods.

In the following we explain further capabilities of JPDDs to express views on the deployment environment of aspects and briefly sketch how they map to OCL expressions. For doing so, we chose to use examples from common aspect-oriented programming techniques to demonstrate the practical relevance of the designation means in JPDDs.

3.2. Pointcuts in AspectJ

Figure 3 models the following AspectJ pointcut (adopted from [21]):

```
pointcut aspectj_pc():
 cflowbelow(call(* ColoringClient.*(..))
    && this(SomeCaller))
 && call(FigureElement Figure.make*(..))
```

It designates all messages that invoke a method beginning with "make" on class "Figure" (returning an instance of class "FigureElement") from within the control flow of any method called on class "ColoringClient" from class "SomeCaller" (returning any or none return value). The message being crosscut is rendered as template parameter "CrosscutMsg".

Table 1 gives a general description on how message matching is accomplished – the depicted (meta-)operation "matchesMessage" is invoked on each message in the UML model. At first, the messages' names are matched. If the message in the JPDD is tagged with a

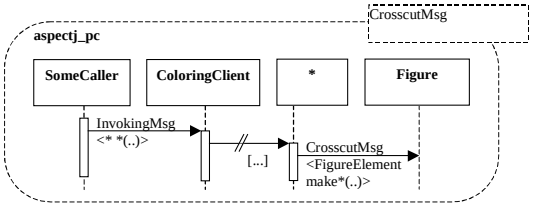


Figure 3. An AspectJ pointcut as JPDD

"joinPointPattern" (which are enclosed by sharp brackets "<...>"; see Figure 3), the "joinPointPattern"'s value (e.g., "FigureElement Figure.make*(..)") is passed for matching rather than the message's name (e.g., "CrosscutMsg"). Then, the message's sender and receiver are matched. This includes matching of their relationships (association roles, generalizations, etc.). After that, the associations used for transmitting the messages are compared.

Note that sender and receiver comparison is accomplished by matching the sender's and receiver's *role* in the JPDD to the sender's and receiver's *base* classifiers in the target model. This is because behavioral crosscutting takes place in every target model whose participants provide the set of features specified in the JPDD – may they be explicitly required by means of the role specification in the collaboration, or implicitly present by means of the base classifier specification in the class hierarchy. The same counts for the associations used for transmitting the messages.

In case the JPDD defines predecessors and/or an activator to the crosscut message (like "InvokingMessage" in Figure 3), the message in the target model must provide corresponding messages among its predecessors. The

Table 1. Matching messages in UML models

```
Context Message::
matchesMessage(m : Message) : Boolean
post: result =                          -- evaluate name pattern ('<...>')
if m.taggedValue->includes(tv | tv.type.name = 'joinPointPattern')
then
    self.matchesNamePattern(m.taggedValue->select(tv |
        tv.type.name = 'joinPointPattern').dataValue->at(1))
else
    self.matchesNamePattern(m.name)
endif
                                        -- evaluate sender/receiver/...
and self.sender.base->includes(C |
    C.matchesRelationships(m.sender) and
    C.matchesClassifier(m.sender))
and self.receiver.base->includes(C |
    C.matchesRelationships(m.receiver) and
    C.matchesClassifier(m.receiver))
and self.communicationConnection.base
    .matchesAssociation(m.communicationConnection)
                                -- evaluate predecessors/activator
and m.allPredecessors->union(m.activator)->reject(m2 |
    m2.stereotype->includes(st | st.name='indirect'))->forAll(m2 |
    self.allPredecessors->includes(M | M.matchesMessage(m2)))
                                                -- evaluate action
and self.action.matchesAction(m.action)
```

precise position is not important. For message matching, messages of stereotype "indirect" (denoted by double-striked-through lines; see Figure 3) are neglected. Their only purpose in JPDDs is to indicate auxiliary control flow the predecessors may provoke.

Finally, the message's actions are matched.

3.3. Traversal Strategies in Adaptive Programming

Figure 4 models the following traversal strategy in Adaptive Programming (adopted from [22], [23]):

```
*from* Conglomerat
 *bypassing* -> *,subsidiaries,*
 *via* Officer *to* Salary
```

The traversal strategy starts at object Conglomerat and traverses a characterized path to object Salary. It states that on its way through the class hierarchy the traversal must pass object Officer. At the same, it requires that traversal must not pass an association end named "subsidiaries".

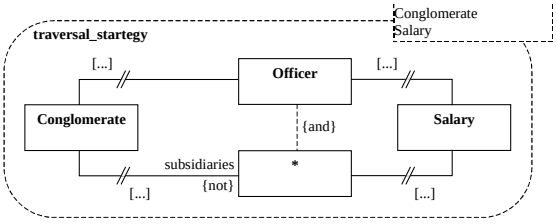


Figure 4. Traversal strategies as JPDD

In a UML model, the classifiers being traversed are identified with help of the (meta-)operation shown in Table 2. The operation analyzes if a classifiers possesses (a set of) associations matching to the ones specified in the JPDD. The operation distinguishes between standard associations and associations of stereotype "indirect" (denoted by double-striked-through lines; see Figure 4). In the former case, comparison is successful if the classifier provides a matching association with matching association ends. In the latter case, there must exist a navigable path from the current classifier to a classifier matching the associate in the JPDD. The association ends at which navigation starts and ends must match the association ends of the association specified in the JPDD.

For example, the bottom left association in Figure 4 denotes a navigation path starting with an association end whose participant is of type Conglomerate. And it ends with an association end whose name must not be "subsidiaries" – no matter of the type of its participant. From the participant, however, there must be a navigable path that ends with an association end whose participant is of type Salary (bottom right association in Figure 4).

Table 2. Matching associations in UML models

```
Context Classifier::
possessesMatchingAssociation(a : Association, c : Classifier) :
Boolean
post: result =                          -- evaluate indirect neighbours
if a.stereotype->includes(st | st.name='indirect') then
    self.associations->includes(A | A.matchesAssociation(a) and
    a.allConnections->select(ae | ae.participant = c)->forAll(ae |
      A.allConnections->select(AE | AE.participant = self)
        ->includes(AE | AE.matchesAssociationEnd(ae) and
        a.allConnections->select(ae | ae.participant <> c)
        ->forAll(ae2 | self.allIndirectNeighbors(A)
        ->includes(AE2 | AE2.matchesAssociationEnd(ae2))))))
else                                    -- evaluate direct neighbours
    self.associations->includes(A | A.matchesAssociation(a) and
    a.allConnections->forAll(ae | A.allConnections
        ->includes(AE | AE.matchesAssociationEnd(ae))))
endif
```

3.4. Composition Rules on Declaratively Complete Hyperslices in Hyper/J

Composition rules in Hyper/J specify how the elements of one hyperslice are to be composed with the elements of another hyperslice. For that purpose, composition rules designate the join points in each hyperslice. Likewise, JPDDs are capable to designate model elements from UML models. While doing so, JPDDs may also reflect on the "declarative completeness" constraint in Hyper/J: In Hyper/J, each hyperslice needs to be "declaratively complete" (cf. [36]). That means that each hyperslice declares the structural properties it expects to be provided by another hyperslice. We can use JPDDs to model such structural requirements.

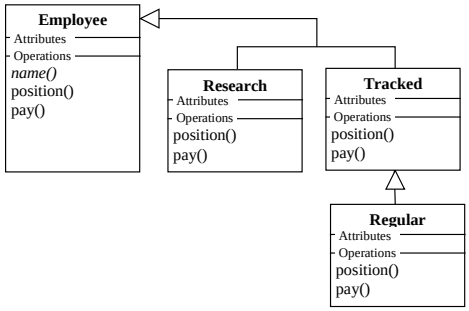


Figure 5. A payroll hyperslice (cf. [29])

For example, let's imagine a payroll hyperslice that implements "position()" and "pay()" operations on four classes "Employees", "Research", "Tracked", and "Regular" (see Figure 5). For their execution, the hyperslice requires the presence of a "name()" (declared as abstract in Figure 5), whose implementation must be provided by some other hyperslice – e.g., a personnel hyperslice (the example is adopted from [29]).

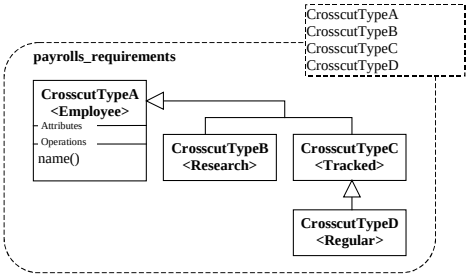


Figure 6. Specifying structural requirements in JPDDs

Figure 6 depicts a sample JPDD which designates the join points in the personnel hyperslice and specifies structural requirements being imposed on those join points. Table 3 and Table 4 describe the (meta-)operations for locating join points in UML models according to the specifications made in the JPDD.

Figure 6 depicts a JPDD that selects four classes from the personnel hyperslice as join points ("Employee", "Research", "Tracked", and "Regular"). These classes are meant to be augmented by the payroll hyperslice during composition. Besides designating the hyperslices' join points, though, the JPDD in Figure 6 specifies a couple of structural requirements that those join points must fulfill. At first, it requires class "Employee" to provide an operation "name()". Further, it requires class "Research" and "Tracked" to be subclasses of class "Employee", while class "Regular" in turn must be a subclass of class "Tracked". Composition may only take place if these constraints are satisfied.

Table 3. Matching classifiers in UML models

```
context Classifier::
matchesClassifier(C : Classifier) : Boolean
post: result =                                  -- evaluate name pattern
if C.taggedValue->includes(tv | tv.type.name = 'joinPointPattern')
then
    self.matchesNamePattern(C.taggedValue->select(tv |
        tv.type.name = 'joinPointPattern').dataValue->at(1))
else
    self.matchesNamePattern(C.name)
endif
                                -- evaluate defined meta-properties
and (self.isRoot = C.isRoot or C.isRoot = '')
and (self.isLeaf = C.isLeaf or C.isLeaf = '')
and (self.isAbstract = C.isAbstract or C.isAbstract = '')
                                -- evaluate attributes and operations
and (C.feature->select(f | f.oclIsKindOf(Attribute))->forAll(ATT |
    self.possessesMatchingAttribute(ATT))
    or C.feature->select(f | f.oclIsKindOf(Attribute))->size = 0)
and (C.feature->select(f | f.oclIsKindOf(Operation))->forAll(OP |
    self.possessesMatchingOperation(OP))
    or C.feature->select(f | f.oclIsKindOf(Operation))->size = 0)
```

Table 3 describes how join points are selected from UML models. Join point selection is accomplished by name matching. If the classifier specifications in the JPDD are tagged with a "joinPointPattern" (which are enclosed by sharp brackets "<...>"; see Figure 6), the "joinPointPattern"'s value (i.e., "Employee", "Research", "Tracked", or "Regular") is passed for matching rather than the classifiers' names (e.g., "CrosscutTypeA", etc.). Apart from their names, the classifier's meta-properties must match ("isRoot", "isLeaf", "isAbstract"). At last, the classifiers' features, i.e., attributes and methods, are compared. A classifier must possess all attributes and all methods being defined as structural requirement in the JPDD (like operation "name()" in Figure 6, for example) in order to be selected as join point.

Further, classifiers must possess all relationships, i.e., associations, generalizations, and specializations, that are defined in the JPDD (like the inheritance relationships in Figure 6, for example) in order to be selected as join point. Matching of relationships is accomplished by a second (meta-)operation for associations, generalizations, and specializations separately (see Table 4).

Table 4. Matching relationships in UML models

```
context Classifier::
matchesRelationships(B : Classifier) : Boolean
post: result =                                  -- evaluate relationships
 (B.parent->forAll(P |
     self.possessesMatchingParent(P))       or B.parent->size = 0)
and (B.child->forAll(CH |
     self.possessesMatchingChild(CH))       or B.child->size = 0)
and (B.associations->forAll(A |
     self.possessesMatchingAssociation(A, self))
                                            or B.associations->size = 0)
```

4. Related and Future Work

A couple of other approaches deal with modeling pointcuts using OCL, UML, and even MDA:

[31] makes use of OCL code [27] to bind elements form an application models to "hot spots" in aspect-oriented frameworks. In doing so, they select model elements that are to be enhanced like we do. Unlike us, however, they define the enhancements in the same OCL statement which hinders reuse of the query specification. Specifying enhancements is not duty of JPDDs.

A more sophisticated approach is described in [15] which presents a domain-specific extension to the OCL for the specification of crosscutting constraints. In particular, it introduces reflective operators to advance selection of model elements. Again, though, selection queries and modification assignments are instantly coupled together, and so, reuse of queries is not possible.

[34] [16] chooses to use UML Action Semantics [28] to define model transformations and OCL [37] to express selection criterions for those transformations. As queries are hard-coded into transformations, they cannot be reused in a different context.

[24] discusses how Model-Driven Architecture (MDA) [25] may support aspect-oriented modeling. It points out that a pointcut can be expressed as a query on one model. We share that conception and have defined a graphical notation to define such queries. We see another application area of our approach in connection with the Query View Transformation Language (QVT) [26] which is currently under review by the OMG: JPDDs can be used as a graphical query language to select model elements from UML models that are subject to transformations.

Besides that we see complementary contribution of our work to existing aspect-oriented modeling and design approaches, for example [13], [32], and [19], which lack graphical means to specify selection queries. Moreover, as JPDDs map onto OCL expressions, our approach can be seamlessly integrated into [16] and [15]. From using parameterized OCL (meta-)operations, we even gain greater flexibility because we may feed the operations with different JPDDs at a time.

Future work will involve investigations on how to specify selection queries in the context of aspect-oriented modeling with state charts [8] or activity diagrams [11]. Further, JPDDs are to be integrated into a UML profile for aspect-oriented modeling (cf. [7]) in order to advance its application in the aspect-oriented software development process.

5. Conclusion – Going Beyond

In this paper we have exemplified the need for distinct modeling means for the specifications of pointcuts, i.e., the specification of places and conditions at/under which crosscutting takes place. We presented a graphical notation that suits this purpose, and we have exemplified its use and semantics when designing a synchronization requirement in an aspect-oriented manner as well as with help of examples from different aspect-oriented implementation techniques.

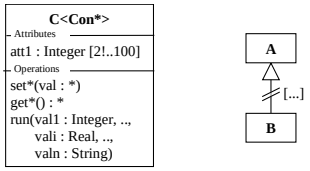


Figure 7. Going beyond in aspect-oriented modeling

Yet, note that the capabilities of our modeling notation go beyond the designation means of current aspect-oriented implementation techniques and allow advanced aspect-oriented modeling. The stereotype “indirect”, for example, is not limited to association relationships (denoted by double-striked-through lines; see Figure 4) but to generalization and specialization relationships as well (denoted by double-striked-through lines with hollow arrow heads; see Figure 7 right side). Using this symbol in JPDDs signifies that a given classifier must provide an ancestor or a descendant, respectively, that matches the specification of the JPDD.

Besides that the notation provides for the specification of operations using wildcards “*” and “..” in their parameter list (see Figure 7 left side for an example). Further, we allow the specification of multiplicity ranges for attributes (see Figure 7 left side). Classifiers representing join points must provide a matching attribute whose multiplicity resides in the range specified by the JPDD (e.g., “[2..100]”). An exclamation mark denotes a fixed lower or upper bound (e.g., “2!”). Please refer to [6] for the corresponding OCL code.

Provided with these novel modeling means, software designers are capable to design pointcuts in wholly new ways. Being implementation language independent, the modeling notation allows design of pointcuts in the very early stages of software development, e.g., when designing connection points in aspect-oriented software architectures. Further, aspect-oriented software developers may fully concentrate on design first, and finally can map their design to whatever aspect-oriented programming language seems best suited.

6. References

[1] 1st Workshop on Aspect-Oriented Modeling, at AOSD'02 (Enschede, The Netherlands, Apr. 2002), http://lgl.epfl.ch/workshops/aosd-uml/index.html

[2] 2nd Workshop on Aspect-Oriented Modeling, at UML’02 (Dresden, Germany, Sep. 2002), http://lglwww.epfl.ch/workshops/uml2002/

[3] 3rd Workshop on Aspect-Oriented Modeling, at AOSD'03 (Boston, MA, Mar. 2003), http://lglwww.epfl.ch/workshops/aosd2003/

[4] 4th Workshop on Aspect-Oriented Modeling, at UML’03 (San Francisco, CA, Oct. 2003), http://www.csam.iit.edu/~oaldawud/AOM/

[5] Adaptive Programming, http://www.ccs.neu.edu/research/demeter/

[6] Addendum to Stein, D., Hanenberg, S., Unland, R., Aspect-Oriented Modeling in the Light of MDA, submitted to the Special Issue of Science of Computer Programming (Elsevier) on Model Driven Architecture: Foundations and Applications, http://dawis.informatik.uni-essen.de/site/staff/stein/

[7] Aldawud, O., Bader, A., Elrad, T., *UML Profile for Aspect-Oriented Software Development*, 3rd AOM Workshop at AOSD'03 (Boston, MA, Mar. 2003)

[8] Aldawud, O., Bader, A., Elrad, T., *Weaving with Statecharts*, 1st AOM Workshop at AOSD'02 (Enschede, The Netherlands, Apr. 2002)

[9] AspectJ, http://www.aspectj.org

[10] AspectJ Team, *The AspectJ Programming Guide*, http://dev.eclipse.org/viewcvs/indextech.cgi/~checkout~/aspectj-home/doc/progguide/index.html, Jan. 2004
[11] Barros, J.P., Gomes, L., *Towards the Support for Crosscutting Concerns in Activity Diagrams: A Graphical Approach*, 4th AOM Workshop at UML'03, (San Francisco, CA, Oct. 2003)
[12] Booch, G., Jacobson, I., Rumbaugh, J., *The Unified Modeling Language User Guide*, Addison Wesley, Reading, MA, 1999
[13] Clarke, S., Walker, R.J. *Composition Patterns: An Approach to Designing Reusable Aspects*. in Proc. of ICSE '01 (Toronto, Canada, May 2001), ACM, 5-14
[14] Elrad, T., Aksit, M., Kiczales, G., Lieberherr, K., Ossher, H., *Discussing Aspects of Aspect-Oriented Programming*, in: ACM Communications, Vol. 44(10), Oct. 2001, pp. 33-38
[15] Gray, J., Bapty, T., Neema, S., Schmidt, D.C., Gokhale, A., Natarajan, B., *An Approach for Supporting Aspect-Oriented Domain Modeling*, in: Proc. of GPCE '03 (Erfurt, Germany, Sep. 2003), Springer, pp. 151-170
[16] Ho, W.M., Jézéquel, J.-M., Pennaneac'h, F., Plouzeau, N., *A Toolkit for Weaving Aspect Oriented UML Designs*, in: Proc. of AOSD '02 (Enschede, The Netherlands, Apr. 2002), ACM, pp. 99-105
[17] Hyper/J, http://www.alphaworks.ibm.com/tech/hyperj
[18] Jacobson, I., Booch, G., Rumbaugh, J., *The Unified Software Development Process*, Addison Wesley, Reading, MA, 1999
[19] Kandé, M.M., PhD Thesis, EPFL, Lausanne, Swiss, 2003
[20] Katara, M., Katz, Sh., *Architectural Views of Aspects*, in: Proc. of AOSD'03 (Boston, MA, Mar. 2003), ACM, pp. 1-10
[21] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G., *Getting Started with AspectJ*, ACM Communications, Vol. 44(10), Oct. 2001, pp. 59-65
[22] Lieberherr, K., *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns*, PWS Publishing Company, Boston, 1996
[23] Lieberherr, K., Orleans, D., Ovlinger, J., *Aspect-Oriented Programming with Adaptive Methods*, ACM Communications, Vol. 44(10), Oct. 2001, pp. 39-41
[24] Mellor, St., *On A Framework for Aspect-Oriented Modeling*, 4th AOM Workshop at UML'03, (San Francisco, CA, Oct. 2003)
[25] Object Management Group (OMG), *MDA Guide Version 1.0*, May 2003
[26] Object Management Group (OMG), *Request for Proposal: MOF 2.0 Query / Views / Transformations RFP*, Apr. 2002
[27] OMG, *Response to the UML 2.0 OCL RFP*, Revised Submission, Version 1.6, January 2003
[28] OMG, *Unified Modeling Language Specification*. Version 1.5, Mar. 2003
[29] Ossher, H., Tarr, P., *Using Multi-Dimensional Separation of Concerns to (Re)Shape evolving Software*, in: ACM Communications, Vol. 44(10), Oct. 2001, pp. 43-50
[30] Rashid, A., Moreira, A., Araújo, J., *Modularisation and Composition of Aspectual Requirements*, in: Proc. of AOSD'03 (Boston, MA, Mar. 2003), ACM, pp. 11-20
[31] Rausch, A., Rumpe, B., Hoogendoorn, L., *Aspect-Oriented Framework Modeling*, 4th AOM Workshop at UML'03, (San Francisco, CA, Oct. 2003)
[32] Stein, D., Hanenberg, St., Unland, R., *A UML-based Aspect-Oriented Design Notation For AspectJ*, in: Proc. of AOSD '02 (Enschede, The Netherlands, Apr. 2002), ACM, pp. 106-112
[33] Stein, D., Hanenberg, St., Unland, R., *Issues on Representing Crosscutting Features*, 3rd AOM Workshop at AOSD'03 (Boston, MA, Mar. 2003)
[34] Sunyé, G., Pennaneac'h, F., Ho, W.-M., Le Guennec, A., Jézéquel, J.-M., *Using UML Action Semantics for Executable Modeling and Beyond*, in: Proc. of CAiSE'01 (Interlaken, Switzerland, Jun. 2001), Springer, pp.433-447
[35] Sutton, St., Rouvellou, I., *Modeling of Software Concerns in Cosmos*, in: Proc. of AOSD '02 (Enschede, The Netherlands, Apr. 2002), ACM, pp. 127-133
[36] Tarr, P., Ossher, H., *Hyper/J User and Installation Manual*, IBM Corp., 2000
[37] Warmer, J., Kleppe, A., *The Object Constraint Language: Precise Modelling with UML*, Addison-Wesley, 1998
[38] Wegner, P., Zdonik, S., *Inheritance as Incremental Modification Mechanism or What Like is and Isn't Like*, in: Proc. of ECOOP'88 (Oslo, Norway, Aug. 1988), pp. 55-77