A Taxonomy of Aspects in Terms of Crosscutting Concerns

Jorge Fox

Institut für Informatik, Technische Universität München
Boltzmannstr. 3, D-85748 Garching b. München, Germany
fox@in.tum.de

Abstract. Aspect-orientation provides support for "Separation of Concerns" by means of techniques that first isolate and then weave concerns. Most work in aspect-orientation has achieved such goals at the programming level, even also at the modeling level. Though, in some cases the application of these techniques is independent of the problem itself. In other words, the techniques for weaving either code or models are in principle applicable to a number of problems without a clear criterion to answer questions like: in what software processes we may actually discuss aspect-orientation? This also brings other questions: what do we consider an aspect?, how do we deal with it?, are aspects crosscutting concerns? The first notions of aspect-orientation relate to crosscutting in code. We consider this a *bottom-up* approach. We believe though, that aspect-orientation can be better understood from an architectural perspective. We call this a *top-down* approach. We explore the question of "what makes an aspect an aspect" and "when do aspects arise" from a top-down perspective. Our work relates to a definition of aspects in terms of requirements traceability, proposes a classification, and altogether a taxonomy.

Keywords. Aspect-orientation, requirements, taxonomy

1 Introduction

As aspect-orientation turns into a more established field of research, some of its concepts need to be made more precise. Concepts like *crosscutting*, *concern*, and *aspect* itself, do convey an approximate notion of what we mean by using them. Historically seen, aspect-orientation seems to have emerged in a code-oriented community of researchers that developed some of the first *aspectual* languages (AspectJ, Hyper/J, Sina) mostly motivated by the need to improve object-oriented systems. So far back as 1979 (See [1]) some of these ideas were already being used. At a fast pace in recent years the term *aspect* is being used more and more, nevertheless, it may sometimes give the impression of being an umbrella idea behind which aspectual as well as non-aspectual concepts are explored. We would like to propose an approach at the hand of which we may further clarify aspect-orientation and detach it as far as possible from object-orientation to present it in a more general way.

Dagstuhl Seminar Proceedings 06351
Methods for Modelling Software Systems (MMOSS)
http://drops.dagstuhl.de/opus/volltexte/2007/860

In this paper we propose a classification of crosscutting concerns from the point of view of requirements traceability. We explore a set of definitions that support the classification, and in this way we suggest a taxonomy of aspects. We introduce the current concepts in aspect-orientation in order to determine its central notion. After this we introduce the related definitions, and then a classification (this is exemplified on an E-Learning system design).

We depart from the idea that aspects are crosscutting concerns. These acquire meaning with respect to requirements specifications. Our method is therefore to consider the general notion of software development process in which requirements are translated into design by means of software modeling concepts, and the design into code (there may be several steps, cycles , and iterations, however the goal is to produce code).

We consider now some possible ways to classify aspects[1], for instance, with respect to code, what we mentioned as a bottom-up perspective which we explore in Sect. 2 more in depth. At the programming language level we may classify them based on pointcut granularity as in [2]. We may also classify them according to the present main available technologies represented in programming languages as follows:

Table 1. Join point model and corresponding programming language

Join point model	Aspect-oriented language
Pointcut and Advice	AspectJ
Message interception via filters	Compose*
Multidimensional SoC	Hyper/J
Traversals	DemeterJ

Another classification may also separate them in *Dynamic* and *Static* aspects from a systems analysis and design point of view as well as from a programming perspective, considering the time at which the *weaving*[2] takes place. Another interesting classification may relate to the nature of the crosscutting expressed in the aspect language. Kiczales in [3] proposes a definition of "Crosscutting" independent of a programming perspective, though it is actually an explanation of when a programming language is considered *aspect-oriented* related or not.

However, in the search for a thorough understanding of aspect-orientation such approaches may not suffice. Consider that one of the problems aspect-orientation aims at solving is improving modularization. For modularization we understand the process of obtaining independent pieces of software or modules

[1] understood by now as a code construct that allows to modify an existing object-oriented program

[2] The programming language level concept is found in Sect. 2, we define it more formally in Sect. 3

that put together constitute a (whole) system. We refer to modules independently of the step in the development cycle, for instance, we consider a clearly identifiable and independent model element as a module. Where a model is "an approximation, representation, or idealization of selected aspects of the structure, behavior, operation, or other characteristics of a real-world process, concept, or system" [4].

Moreover, we refer to modularization of software requirements that are finally translated to code through the software development cycle. We suggest that modularization relies on the concepts (in the sense of *mental frameworks*) available for software modeling, for instance: class, method, inheritance as in object-oriented programming languages. The design decisions taken in the whole process of translating requirements into code might also be considered as a collection of composition and decomposition steps. At the general level, composition of the concerns in a system with composition mechanisms that support the interaction of the entities pertaining to a system. At a more particular level, the decomposition of functionality[3]. In this sense, crosscutting concerns exist from the perspective of decomposition and modularization in a tracing relation with respect to requirements i. e. software specifications. Both decomposition and modularization imply a subsequent composition in order to produce an operational software system. In this regard *compositionality*[4] is a central notion in aspect-orientation. This can be proved by the fact that the notion of *weaving* is present in every aspect-oriented technique, paper, or related proposal.

It here is suggested that the production of a brand new software, as well as the modification or maintenance of an existing one, relate to requirements. Also in the case of non-preventive maintenance since it deals with fixing of errors or failures stemming from changes in the environment or a wrongful implementation with respect to some desired behavior (implicitly or explicitly) represented by the requirements. In such case the system is corrected in order to fulfill the desired functionality as stated by the stakeholders in the specification.

Summarizing, crosscutting concerns represent interests of the stakeholders and are stated in the requirements artifacts. On this basis, we classify them in view of its tracing from requirements instead of other possible classification schemes.

In the coming section we look at the problem from a bottom-up approach (see also Appendix C). We then introduce our own concepts at a more abstract level in Sect. 3. Based on our top-down approach we identify two sources of crosscutting and outline a thesis with the conclusion that crosscutting should be managed in design and code i. e. later phases of requirements engineering and on. We later introduce our classification in Sect. 4. Related work is introduced

[3] as defined by Harel and Pnuelli in [5]

[4] "...the meaning of a complex expression is fully determined by its structure and the meanings of its constituents - once we fix what the parts mean and how they are put together we have no more leeway regarding the meaning of the whole. This is the principle of compositionality, a fundamental presupposition of most contemporary work in semantics."[6]

in Sect. 5. We close with some conclusions in Sect. 6. Appendix A provides the sample case study. Appendix B completes the definition of aspect from Sect. 3.

2 Aspect-oriented Programming (AOP)

In this section, we explore aspect-orientation from a brief bottom-up and later a more top-down perspective in order to provide some concepts that support our theory in Sect. 3. Most common definitions of aspect fall in two groups. The first group consisting of definitions stating that aspects are concerns that cut across other concerns (whereas two concerns crosscut if the methods related to those concerns intersect [7]). The second group assuming a very pragmatical approach and defining aspects in terms of the constructs from aspect-oriented programming languages, where an aspect is "a join point and advice." A *join point* is defined as "a well defined point in the execution flow of the program", and an *advice* as "the actual implementation of the aspect." Weaving is the mechanism that actually inserts the advice at the points in the execution flow indicated by the join points.

The first group argues further that aspects produce tangled representations that are difficult to understand and maintain [8]. Moreover, crosscutting is relative to a particular decomposition [9]. See for instance Fig. 1 which corresponds to a graphical representation of an aspect as compared to modular units. This illustration is based on the example introduced in the previous section.

In any case, aspects can be system properties involving more than one functional component, crossing the static and dynamic structure of a program. Aspects affect and modify several classes. This means that aspects might be static or dynamic properties that affect not only the behavioral definition of classes, but also its data structure. Please note that most work on the field argues that aspect-orientation is an enhancement of object-oriented programming.

Some authors define that issues that are not well localized in functional designs, such as: synchronization, component interaction, persistency, security control, fault tolerance mechanisms, quality of service, and the like; are concerns that are typical candidate aspects. Another guideline to identify aspects is that they "are usually expressed by small code fragments scattered throughout several functional components" [10]. Compatibility, availability and security requirements are crosscutting concerns [11]. Also exception handling, multi-object protocols, synchronization, and resource sharing would be extended across the source code if only using traditional implementation techniques [12].

Although the above ideas provide material for a first definition of aspect, we still have to consider means to identify aspects at different abstraction levels and relate them to the different software process activities.

As Ossher in [7] mentions, "One of the hard things about crosscutting concerns is understanding just what cuts across what". There is still a need for research on aspect identification at the early stage of software requirements, because it is at that stage that many of the later difficulties in software development can be generated. It is left to the criterion of the analyst to determine

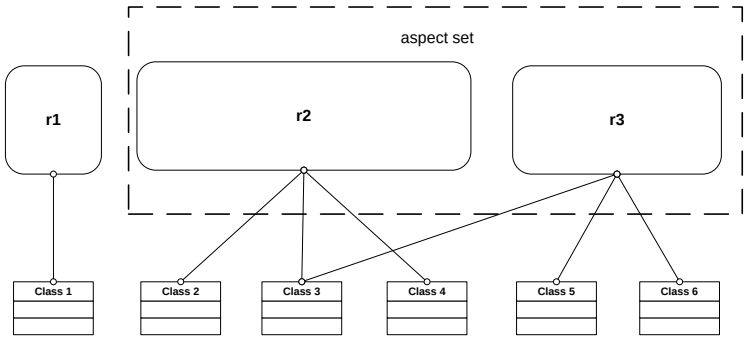


Fig. 1. Requirements and aspects in relation to classes. First published in [13]

requirements, and then concerns, and out of these, select candidate aspects and test them. As [14] states: "Designers must rely on their discretion to decompose the problem effectively". Kiczales in [14] points out a simplified thumb rule for aspect identification: " A property that must be implemented is

- A component, if it can be cleanly encapsulated in a generalized procedure (that is object, method, procedure, API). By cleanly, we mean well localized, and easily accessed and composed as necessary. Components tend to be units of the system's functional decomposition, such as image filters, bank accounts, and GUI widgets,
- An aspect, if it can not be cleanly encapsulated in a generalized procedure. Aspects tend not to be units of the system's functional decomposition, but rather to be properties that affect the performance or semantics of the components in systemic ways. Examples of aspects include memory access patterns and synchronization of concurrent objects."

This simple "thumb rule" points in the right direction, because it is consistent with the aspect concept. Our critique to this simple rule is that it does not help discriminating what is an aspect and what is not. For instance, would a synchronization protocol be an aspect or would only a part of the protocol be considered an aspect? Or would every call to a database in a front-end be considered an aspect just because it might be carried out in different parts of the front-end? Furthermore, how useful would it be to consider systemic concerns such as quality of service or performance as aspects?

In other words, aspects do not exist as such. These acquire meaning through the semantic dependencies in a software system, whether they are constraints affecting more than one functional unit or user concerns that are implemented by more than one functional unit.

It is commonly accepted that *aspects* are "concerns that cut across other concerns". According to the early notions of aspects, two concerns crosscut if the methods related to them intersect [15]. Therefore, they are responsible for producing tangled representations that are difficult to understand and maintain [16]. This crosscutting is relative to a particular decomposition [9]. A concern is considered as any area of interest in a software system. These concepts are too general. That is why we formulate our concepts in Sect. 3. In order to achieve it, we depart from the meaning of the word aspect. This helps shaping the concept correctly while building on the existing concepts in the aspect-oriented community.

The word aspect has the following meaning:

> *a particular status or phase in which something appears or may be regarded* [17]

Besides, the word aspect comes from the latin *aspectus*, from *aspicere*, that is, to look at. All in all the selected definition and the etymological meaning of the word reflect the idea of a perspective toward an artifact, in this case toward a system. The word aspect is then consistent with conceiving a system as composed of different parts (subsystems) or angles, and allows us to separate such subsystems as areas of interest. In short, this way we are able to arrange a system in concerns.

A number of difficulties for aspect identification arise from a notion of aspect that leaves room for interpretation. Let us for instance consider the proposal on early aspect identification as in [18]. Their approach toward aspect identification relies on use cases, namely when a use case extends more than one use case or when a use case is included by one or more viewpoints, then that use case is considered an *aspectual* use case. There are a number of difficulties associated with aspect identification by doing so, for instance, prioritization of conflicts that stem from different viewpoints is done by hand, and by hand is made also the decision of what is an aspect and what is not an aspect once the requirements people identify candidate aspects. It is nevertheless a valuable approach that gives an important insight toward aspect identification.

Further, aspect identification relates to the need of an integral comprehension of concern crosscutting altogether with its context. As authors like [19] have already outlined, the problem AOSD solves is one of complexity in today's software applications.

Summarizing, aspect-oriented programming consists of a traditional base language, usually object-oriented such as Java or C++ (but not necessarily object-oriented) together with additional language constructs that transform code, namely:

1. "pointcuts" (set of join points),
2. "advice", and a
3. weaving mechanism

Similar concepts are used in most work related to aspect-oriented modeling.

The prevailing aspect-oriented language is AspectJ. For a reader interested in having an overview of it, we explain its constructs at the hand of a case study in Appendix C.

3 Our Theory

In order to explain *aspect-orientation* we provide here some concepts that build our taxonomy. It is based on software requirements considered as some kind of "focal reference" to discuss aspects.

We consider that requirements relate to a *problem space*, in the early requirements stage, and are mapped to a *solution space* in the late requirements and design stages. These concepts are introduced in the following.

> ...the expression "requirements specification" by itself is virtually meaningless. Whenever we use the term, it refers to a deliverable of development consisting of product objective and of required product behavior [20].

Definition 1. ***(Requirements)*** *The definition of requirements we refer to is: "The first stage of software development which defines what the potential users want the system to do."*

Definition 2. ***(Problem space)*** *The problem space is defined as the set of requirements together with the explicit or implicit definition of the environment in which the system-to-be should operate.*

We exemplify the requirements expressed as part of the problem space in the form of interviews (Appendix A.3) and statements extracted from these (Appendix A.5). All of these is what we consider the problem space in the particular case of the ELSS, plus the assumptions the requirements engineer and design expert make with respect to domain knowledge.

Definition 3. ***(Solution space)*** *The solution space is the set of artifacts that belong to a running implementation of a software system and without which the system would not be able to operate or would not have been produced (requirements are explicitly not in this set).*

Examples of artifacts in the solution space are the class diagram in Fig. 3, and methods specific to the classes such as the ones in Table 2. The code implementing the system belongs also to the solution space.

Definition 4. ***(Mapping from problem space to solution space)*** *Given the notion of refinement as the transformation of an abstract i. e. high-level specification into one or more concrete i. e. low-level executable software artifacts, we define the mapping from problem space to solution space as the process of transforming a given requirement into one or more executable artifacts by means of refinement. This mapping is performed explicitly or implicitly with the help of a given conceptual model that translates requirements to software models and finally to code.*

Solution and problem spaces are represented in Fig. 2. The mapping is represented by the arrow from one to the other, we may formally relate the mapping either to a transformation function from problem to solution space in the form of refinement steps.

Requirements give shape to an architectural style, platform, interfaces, etc. in a certain domain and following the conceptual model in use, we illustrate this in Fig. 2. In short, the translation process from the problem space to the solution space is performed via conceptual models e.g. object orientation, components, service-oriented approaches, etc.

Some requirements are implemented in several modularization units. This suggests a way to define crosscutting as behavior relating to more than one (logical at design, physical at code) module from one abstraction level to the next. This considering an abstraction level as a refinement step in a series of steps from requirements to code, beign the previous step the more abstract in relation to the next one, such relation is clearly transitive. As a matter of fact, we may build a more abstract definition of aspect than the ones explored so far in this section, by considering behavior as a property[5] implemented in several modules i. e. *more than one* modularization entity. The modularization entity depends on the chosen architectural paradigm e. g. object-orientation, components, agents, etc.

The above reasoning leads to the following definition of crosscutting concern.

Definition 5. ***(Crosscutting concern)*** *Given a problem space, understood as set of requirements, a crosscutting concern is a requirement that under every possible translation from the problem space to the solution space is expressed in more than one modularization unit in a lower level of abstraction.*

Up to now we have considered *aspect* and *crosscutting concern* as the same entity. The difference is that aspects arise due to the translation from the problem space to the solution space (Fig. 2) due to the fact that no modularization abstraction is perfect. Emphasizing, aspects exist at the software architecture, design, and implementation stages while crosscutting concerns can be seen as a superordinate concept i. e. more generic concept. This makes aspects a subset of crosscutting concerns. We have a precise definition of aspect which we quote in the following lines (unpublished joint work with Professors Dominikus Herzberg and Manfred Broy).

Definition 6. ***(Aspect-Orientation, Aspect Weaving, Aspect)***

> *"...aspect-orientation can be completely subsumed by the notion of communication refinement (*defined in Appendix B*). In this regard, concerns are regarded as sets of components interacting via some means of*

[5] A property is formally defined as a set of behaviors, so that an execution of a system Π satisfies a property P if and only if the behavior (a sequence of states and agents) that represents the execution is an element of P. [21]

> *communication, e.g. connection-oriented or connectionless communication. Communication refinement is viewed as a kind of behavioral refinement that helps adding behavior on a given set of components at the communication channels between the components i.e. adding behavior to the concern. Communication refinement is applied to the communication service. This way of adding behavior opens up a way to stepwise add layers of concerns (i.e. sets of new components) to the previously existing ones. This is* aspect weaving*. In other words, aspect orientation can be explained as a process resulting from the viewpoint taken, or – to be more precise – of the way, how components are grouped. From a "horizontal" viewpoint (i.e. grouping), aspects turn out to be components added in the layers of the software structure. Here, a layer is an isolated concern, which builds up an abstraction hierarchy with other layers. From a "vertical" viewpoint, the layer is sliced into pieces fragmenting the individual concerns."*

Back to our taxonomy of aspects. We have discussed the subject, identified its main concepts, introduced our definitions and now we introduce our classification. An accepted classification of requirements divides them into *Functional (FR), Non-Functional (NFR), Design, and Implementation.*

We understand FR as defining characteristics of the problem space[6], and NFR as constraints in the solution space[7]. The behavior of the SuD is expressed as FRs, while restrictions to the possible solutions are determined by NFRs. But at the same time, non-functional requirements are at some point in the development cycle converted i. e. translated into functional specifications. Meaning that they are translated into quantifications or behavior. As an example, consider a NFR such as "Fault Tolerance." This requirement can in subsequent phases of development be translated into functional specifications that guarantee data persistence in view of a system failure.

A software specification is understood as the formal or semi-formal description of the behavior that a system or subsystem has to fulfill.

Based on the distinction between Non-functional and functional requirement we identify two sources of crosscutting in the coming subsections.

3.1 Crosscutting Due to Inherent Limitations in a Decomposition Paradigm

One source of crosscutting is due to limitations of existing programming as well as design paradigms. As example consider object-orientation. This paradigm decomposes software into modules, in this case classes. Some desired behavior might be common to different classes and therefore ends up spread among several classes. Systemic concerns or concerns that relate to a group of classes, such as

[6] What a system should be able to do, the functions it should perform.

[7] Requirements on the system's performance. For instance, the amount of users that have to be attended in a given time frame.

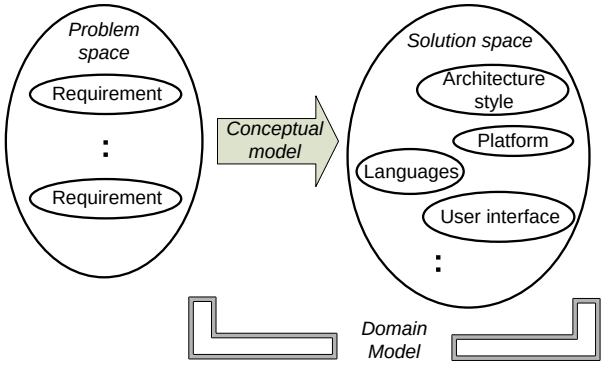


Fig. 2. Mapping between problem space and solution space

some security concerns, which cannot be encapsulated in a single unit and are implemented in several classes.

The role of programming languages in shaping the abstractions by which software designers and programmers apprehend and organize software cannot be underestimated. This applies for requirements engineering as well. The abstractions that ultimately shape a software are influenced by the underlying modeling or implementation paradigm, like the class/object concept.

The object abstraction along with its composition mechanisms comprise limitations. These limitations have already been discussed by [22] and more in-depth by S. Clarke in [23]. S. Clarke clearly demonstrates that the units of modularization in the OOP are structurally different from the units of modularization in requirements specification. This result can be generalized to other modularization paradigms. This reasoning is expressed in Definition 5.

Mathematically expressed, we may write it the following way.

Definition 7. ***(Aspect)*** *Let ρ be a problem space (Definition 2), $SOLSP = \bigcup_{\rho} SolSp(\rho)$ be a solution space (Definition 3), $\mathbb{RE}$ be a set of Requirements (Definition 1) in the problem space, $\mathbb{M}$ be a set of Modularization Units in the solution space.*

We define an aspect $\mathcal{A}$ coming forth through the mapping from problem to solution space (Definition 4) by

$\mathcal{A} =_{def} \{r \in \mathbb{RE} \mid \exists\, o_1, o_2 \in \mathbb{M} :$
$Implements_{\varphi}(r, o_1) \wedge Implements_{\varphi}(r, o_2) \wedge o_1 \neq o_2\}$
where
$Implements_{\varphi}(r, \theta) \Leftrightarrow \forall\, \varphi' \subseteq \varphi.(\varphi' \models r \Rightarrow \theta \subseteq \varphi')$
$\varphi \models r$ *means* φ *satisfies* r
$r \subseteq \bigcup_{\rho} SolSp(\rho)$

Definition 7 means that given a problem space, understood as set of requirements artifacts, an aspect is the representation in the solution space of a requirement

(problem space) that under every possible translation to the solution space is expressed in more than one modularization unit e. g. class, component, function; depending on the underlying architectural framework of the solution space.

3.2 Crosscutting as a Result of Transforming Non-functional Requirements into a Functional Proxy

Another source of crosscutting lies in the translation of NFR into a functional implementation i. e. a functional proxy of the corresponding NF specification. These are also considered in Definition 7 through the *Implements* relation. For instance, in [24] we present the translation of the security constraint *Keep transaction secure* in the early requirements stage into a number of security (sub-)constraints such as *keep transaction private, keep transaction available* and *keep integrity of the transaction* in the late requirements stage. These (sub-)constraints represent the functional proxy of the more abstract security requirement.

As already mentioned, aspects come forth in the mapping from the problem space, where requirements are elucidated, to the solution space, where the architecture, deployment, etc. are defined. Take for instance, fault tolerance. Guaranteeing data persistence despite unanticipated faults may require a set of functions that are implemented in several modularization units (components, classes, services).

Design requirements relate to decisions regarding architectural style, user interface style, and decisions that constrain the set of options available in the domain model. The domain is also referred as *Universe of Discourse* (UoD) in the literature. It is used to mean the part of the world to which the data manipulated by the *System under Development* (SuD) [8] refers. We consider adaptability and maintainability as design requirements.

Performance requirements, usually considered non-functional, are namely translated in specifications that precisely quantify what performance actually means for the *SuD*. Same in the case of security requirements. Components performing security protocols such as encryption, authentication, etc. can be considered proxies for more abstract security requirements e. g. data integrity. These proxies not only act upon several modularization units in the resulting system, but are implemented in several modularization units.

We do not consider scattered and tangled code resulting from an erroneous modularization i.e. an incompetent design or bad programming techniques. Therefore code clones, code smells and the like are out of the scope of this work.

We would therefore propose the following:

1. Crosscutting can be described in terms of a tracing relation from requirements to system design as well as from this to code. It is a transitive relation.

[8] "Beacuse computer-based systems manipulate data, there are two systems we can specify: the computer-based system and its UoD. In a development process, the computer-based system is also called the **system under development** or SuD" [20].

2. A crosscutting concern is any software artifact that can be described by the crosscutting relation
3. An aspect is a kind of crosscutting concern that can be described by a special refinement function, namely communication refinement.
4. Crosscutting is an outcome of the translation from a higher to a lower abstracion level starting with requirements and ending with code.
5. Crosscutting is due to the available modularization concepts and will always be provided.
6. It follows that crosscutting is determined by the modularization technique and is given after requirements translation to further steps in the development process.
   (a) Also assuming that existing requirements engineering techniques suffice to design software, we consider that
7. crosscutting should be managed in design and code.

4 A Classification of Aspects

As already mentioned, we classify aspects by the nature of the crosscutting with respect to requirements.

In order to introduce our classification we consider two examples. On the one hand, the problem of multiple views on a software system as introduced in [25].

On the other hand, the introduction of security mechanisms on a given software system.

The problem can be illustrated as follows. In [26] we presented a prototype for an E-Learning Support System (ELSS). The system supports the evaluation of students in a university's e-learning system. Students are supposed to attend a given number of hours for self-learning and evaluate their advance using the system. The professors define the tests to be applied via the system and are also able to have an overview of their students' advancement through this system.

The system is designed for the following users: `Professors` and `Students`. We consider a view as a set of requirements whose grouping criteria is that they have been requested by a given user of the system to be. Views are related to a specific user i. e. interested party.

We define two views on the system and based on these views restrict access to evaluation and examination -related methods. Restricting access based on the user is a functional requirement that under an object-oriented design would be implemented in different classes. We define two views on the system, `studentView` and `profesorView`. Table 2 shows the access critical methods that are assigned to each view.

We recall that in object-orientation the class abstraction encapsulates behavior related to each of the classes while leaving behavior related to multiple classes in several modularization entities.

Example 1. As a first example, assume that we decide to modify the ELSS and add a new set of requirements to the system. Say this set of requirements comes from the *university exams administration office*. This additional view is supposed

Table 2. Methods per view

Student	Professor	Exams Office
readTest()	enableTest()	enableProf()
answQuestion()	defineAnswer()	enableStudent()
checkOwnAnsw()	evaluateStudent()	allowAccessTest()
answTest()	groupAverage()	
	eraseTest()	

to enable authorized students to take an examination and enable authorized professors to define or apply a test. Implementation of the above requirements might be performed by the methods in Table 2 under the view `Exams Office`. These methods relate to more than one class, namely to classes `Student` and `Professor` considering Fig. 3.

Example 2. As a second example, consider the same diagram from Fig. 3 and assume users may access the system remotely. A related requirement is that the information from the exams shall be kept secret to third parties through the communication channel. A possible solution is adding an encryption protocol to every information sent from and to the following classes `student`, `professor`, and `exam`.

The two examples presented above can be considered representative of the aspects found in the literature, from synchronization policies, all the way to composition aspects as multiple views, considering as well logging, security, persistence, which are system properties involving more than one functional component.

Table 3. Classification

Non-Functional	Semi-localized	NFSL
	Systemic	NFS
Functional	Black-box	BB
	Clear-box	CB

4.1 Non-functional Aspects (NFA)

These are constraints that characterize *how* the SuD will perform, just as non-functional requirements do, however, NFA affect several modularization units in

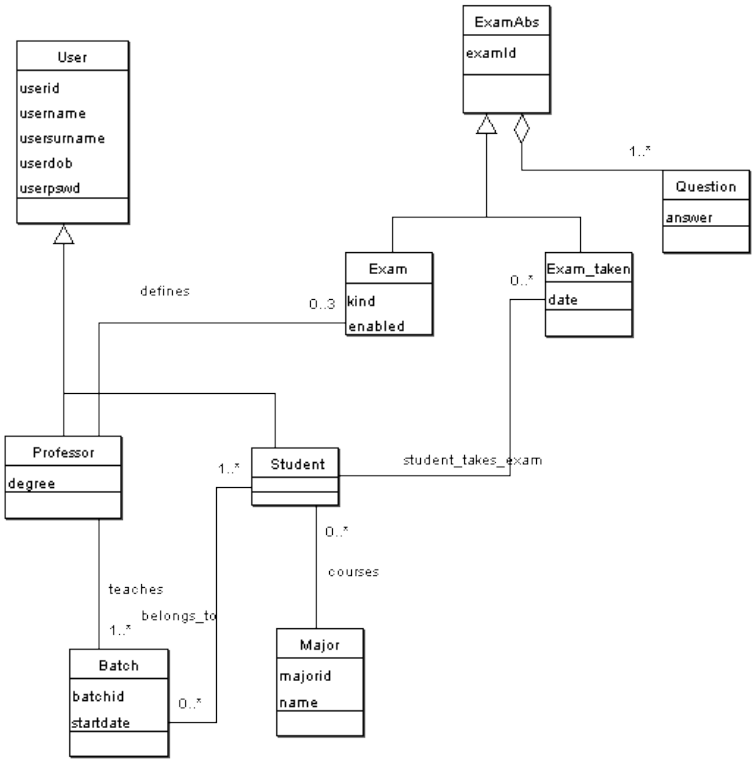


Fig. 3. Class diagram ELSS

the solution space. We further distinguish two types of NFA depending on the scope of the modularization units affected.

Non-functional Semi-localized aspects (NFSL) influence the decisions shaping or modifying the architecture of a system. These aspects can be seen as views on the system in the sense explored in Example 1.Consider also for instance, the requirements formulated in Appendix A.4.

Non-functional Systemic aspects (NFS) relate to qualities expected on the system, such as adaptability, maintainability, quality of service, performance, etc. See for instance part 3 of Appendix A.5.

We do not explore here the transformation from requirements to systems design and implementation. It is outside the scope of this work, it belongs to more extent research that may be performed in a future time. We just recall that different technology solutions (e. g. languages)to the problems posed by aspects

exist. Non-functional semi-localized aspects may be solved by using Design Patterns, AspectJ, or Compose*, for instance. Functional aspects by composition mechanisms as Hyper/J or Compose* as already mentioned. Yet, there is still no aspect-language technology capable of dealing with NFS aspects to the best of our knowledge. We consider these aspects have to be managed in modeling and design, rather than at the language level.

4.2 Functional Aspects (FA)

Concerns which are described in terms of behavior and affect several modularization units fall in this category. FAs can be implemented at the level of the methods in the classes, see for instance, Example 2.

Black-Box aspects relate to the public interfaces of components like functions, object methods, and communication channels. The behavior affecting several modularization units can be related to these via some wrapping over the communication channels or the external interfaces. Such as those implementable by frameworks as Composition Filters.

Clear-Box aspects relate to the internal structure of the classes or components. The behavior of the affected units is redirected or appended and the aspects can be seen as alien units composed inside the base units. I. Jacobson discusses this kind of aspects which he named (years before AOP) as "Existion and Extensions" in [1].

By classifying aspects in Table 3 according to the accepted division between Functional and Non-functional requirements we may see that *aspectual* and *non-aspectual concerns* are essentially the same. The difference between them results later on in the development process. We may therefore use either traditional (i. e. established) approaches like Object-orientation together with novel approaches and techniques as the ones proposed by aspect-orientation.

5 Related Work

A review of other classifications reflects a preponderating role of the programming level approach toward aspects. See for instance the work of [27]. The author proposes that aspects be categorized into two sorts: "spectators" and "assistants" with relation to the behavior of the code that they advise. This classification is certainly of interest for programmers. Moreover, [28] classify aspects based on the interaction between advice and method. Their classification helps programmers understanding the possible interactions of a given aspect and its design implications.

There is a categorization of aspects at the programming level related to classes of temporal properties in [29]. The author defines the following classes of

aspect: Spectative, regulative, and invasive. The categorization provides a basis to later analyze aspect interaction with the base model at the code level.

We aim at a more comprehensive and top-down aspect classification. In this way, though still very much at the language level, the work of [2] is quite comprehensive. The authors classify aspect-oriented systems based on type of join point, selection criteria, and the adaptation mechanism. They classify the main aspect-oriented languages based on the above criteria. Their work helps selecting the aspect language i. e. mechanism more suitable for the implementation of a given crosscutting concern. [30] propose a framework based on crosscutting concern sorts intended for aspect mining techniques. These sorts are defined as atomic descriptions of crosscutting functionality, classified based on "intent" and its relation to an aspect mechanism (at the programming language level). We agree that the authors present a classification that supports aspect mining though very much influenced by the actual programming level constructs. This means, the elements they mine might not necessarily cover aspects in general, though certainly those that they *a priori* define as crosscutting concern sorts. In contrast, we define crosscutting and aspect, and provide a classification that is independent of language mechanisms.

We presented early work on a taxonomy of aspects in December 2005 at the University of Twente. As a result of the discussion it occurred that the author's proposal was justified in terms of "requirements traceability." At that time, we had already published a definition of aspect and its relation to a formal theory for modeling aspects (see [13]). In that work we relate aspects and requirements to traceability. Similar work from [31] discusses crosscutting though in terms of *tangling* and *scattering*, yet not related to requirements traceability. Later on [32] relate their definitions to requirements traceability, though they propose no classification and focus on identification of crosscutting to the early phases of software development.

The reader may also refer to [33] for a survey on current definitions of aspect.

6 Conclusions

This paper has provided a systematic study of aspect-orientation from a *top-down* perspective. We explored where and how aspects are originated. Moreover, as a consequence of relating their causation to the transformation from a problem to a solution space, we group them based on their *nature*. In other words, we may then classify them based on a the classification of requirements. We enrich this first level of classification with the characteristics of the aspects in each group, namely black-box, glass-box in the case of functional and systemic or semi-localized in the case of non-functional. These constitute the second level of the classification.

The definitions presented here and the classification of crosscutting concerns in functional and non-functional may help identifying the technology at hand for dealing with each type of aspect. This is future work, and will be performed by proposing a modeling approach for functional aspects. We are in complete

agreement that further studies are needed to relate the type of aspect to the corresponding design and implementation technology in a more integral way. The results of this work should assist identifying the software processes at which aspects actually represent an improvement and discard those at which it makes less sense. Furthermore, this work might provide more clarity for the coming phases of *aspect-orientation* as it moves from being a new kind of maverick idea to a more consolidated field of research.

Acknowledgements. Thanks to Florian Deißenböck for his critical opinions which made me dwell on my arguments at times. To the reviewers of TEAM 2006 for their comments on an earlier draft, regarding the importance of keeping a big-picture perspective. Also to Maria Victoria Cengarle and Andi Bauer for making me aware of some mistakes (which I then tried to prevent).

References

1. Jacobson, I.: Use cases and aspects -working seamlessly together. Journal of Object Technology (2003) 7–28
2. Hanenberg, S., Stein, D., Unland, R.: Eine taxonomie für aspektorientierte systeme. In Liggesmeyer, P., Pohl, K., Goedicke, M., eds.: Software Engineering. Volume 64 of LNI., GI (2005) 167–178
3. Kiczales, G., Mezini, M.: Aspect-oriented programming and modular reasoning. In: ICSE '05: Proceedings of the 27th international conference on Software engineering, New York, NY, USA, ACM Press (2005) 49–58
4. Institute of Electrical and Electronics Engineers: IEEE standard computer dictionary: A compilation of IEEE standard computer glossaries (1990)
5. Harel, D., Pnueli, A.: On the development of reactive systems. In Apt, K., ed.: Logics and Models of Concurrent Systems. Volume F-13 of Springer NATO ASI Series. Springer-Verlag, New York, NY, USA (1985) 477–498
6. Szabó, Z.G.: Compositionality. In Zalta, E.N., ed.: The Stanford Encyclopedia of Philosophy. (Spring 2005)
7. Elrad, T., Aksit, M., Kiczales, G., Lieberherr, K., Ossher, H.: Discussing aspects of aop. Commun. ACM **44** (2001) 33–38
8. Rashid, A., Sawyer, P., Moreira, A., Araujo, J.: Early Aspects: A Model for Aspect-Oriented Requirements Engineering. In: IEEE Joint International Conference on Requirements Engineering, IEEE Computer Society Press (2002) 199–202
9. Aksit, M., Kiczales, G., Lieberherr, K., Ossher, H.: Discussing aspects of aspect-oriented programming. Communications of the ACM **44** (2001) 33–38
10. Czarnecki, K., Eisenecker, U.W.: Generative Programming - Methods, Tools, and Applications. Addison-Wesley (May 2000)
11. Rashid, A., Moreira, A., Araujo, J.: Modularisation and composition of aspectual requirements. In: AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development, New York, NY, USA, ACM Press (2003) 11–20
12. Kiczales, G., Hilsdale, E.: Aspect-oriented programming. In: ESEC/FSE-9: Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering, New York, NY, USA, ACM Press (2001) 313

13. Fox, J.: A formal foundation for aspect-oriented software development. Research on Computing Science, CIC-IPN, ISSN: 1665-9899 **14** (2005) 241–251
14. Kiselev, I.: Aspect-Oriented Programming with AspectJ. Sams, Indianapolis, IN, USA (2002)
15. Elrad, T., Filman, R.E., Bader, A.: Aspect-oriented programming: Introduction. Communications of the ACM **44** (2001) 29–32
16. Rashid, A., Sawyer, P., Moreira, A., Araújo, J.: Early aspects: A model for aspect-oriented requirements engineering. In: Proceedings of IEEE Joint International Conference on Requirements Engineering (RE 2002), IEEE Computer Society, pp. 199-202. (2002)
17. : Merriam webster online dictionary. (http://www.m-w.com)
18. Araùjo, J., Coutinho, P.: Identifying aspectual use cases using a viewpoint-oriented requirements method. In: Early Aspects 2003: Aspect-Oriented Requirements Engineering and Architecture Design, Workshop of the 2nd International Conference on Aspect-Oriented Software Development, Boston, USA, 17th March. (2003)
19. Videira-Lopes, C.: AOP: A Historical Perspective. In Filman, R., Elrad, T., Clarke, S., Aksit, M., eds.: Aspect-Oriented Software Development. Addison-Wesley (2004) 97–122
20. Wieringa, R.J.: A survey of structured and object-oriented software specification methods and techniques. ACM Comput. Surv. **30** (1998) 459–527
21. Abadi, M., Lamport, L.: Composing specifications. ACM Trans. Program. Lang. Syst. **15** (1993) 73–132
22. Aksit, M.: Composition and separation of concerns in the object-oriented model. ACM Computing Surveys **28A** (1996)
23. Clarke, S.: Composition of Object-Oriented Software Design Models. PhD thesis, Dublin City University (2001)
24. Mouratidis, H., Jürjens, J., Fox, J.: Towards a comprehensive framework for secure systems development. In Eric Dubois, K.P., ed.: Proceedings of the 18th International Conference, CAiSE 2006. Volume 4001., Springer Berlin / Heidelberg (2006) 48–62
25. Bergmans, L., Aksit, M., Tekinerdogan, B.: Aspect composition using Composition Filters. In Aksit, M., ed.: Software Architectures and Component Technology. Kluwer Academic Publishers (2001) 357–382
26. Fox, J.: E-learning support system. Technical report, National Institute of Small Industry Extension Training (nisiet), Hyderabad, India (2002)
27. Clifton, C.: A design discipline and language features for modular reasoning in aspect-oriented programs. PhD thesis, Iowa State University (2005) ISU TR 05-15.
28. Rinard, M., Salcianu, A., Bugrara, S.: A classification system and analysis for aspect-oriented programs. (2004) 147–158
29. Katz, S.: Aspect categories and classes of temporal properties. In Rashid, A., Aksit, M., eds.: T. Aspect-Oriented Software Development I. Volume 3880 of Lecture Notes in Computer Science., Springer (2006) 106–134
30. Marin, M., Moonen, L., van Deursen, A.: A common framework for aspect mining based on crosscutting concern sorts. Technical Report TUD-SERG-2006-009, Delft University of Technology (2006)
31. van den Berg, K., Conejero, J.M.: A conceptual formalization of crosscutting in AOSD. In: DSOA'2005 Iberian Workshop on Aspect Oriented Software Development. Technical Report TR-24/05, University of Extremadura (2005)

32. van den Berg, K., Conejero, J.M., Hernàndez, J.: Analysis of crosscutting across software development phases based on traceability. In: Early Aspects at ICSE2006: Workshop in Aspect-Oriented Requirements Engineering and Architecture Design, Shanghai (2006) May 21, 2006.
33. Fox, J., Jürjens, J.: Introducing security aspects with model transformation. In: Model Based Development (MBD) Workshop in Proc. 12th Annual IEEE International Conference on the Engineering of Computer-Based Systems (ECBS 2005), Greenbelt, Washington, 4-5 April, IEEE Computer Society (2005) 543 – 549
34. Pawlak, R., Duchien, L., Florin, G., Legond-Aubry, F., Seinturier, L., Martelli, L.: A uml notation for aspect-oriented software design. AO modeling with UML workshop at the 1st AOSD conference (2002)
35. Stein, D., Hanenberg, S., Unland, R.: A uml-based aspect-oriented design notation for aspectj. In: AOSD. (2002) 106–112

A Requirements: E-Learning Support System

A.1 Problem description

During the spring of 2002 in the framework of a Data Warehousing and Mining training program in Hyderabad, India, we came to the idea of proposing an E-Learning Support System (ELSS) for a Mexican university that had actually no distance learning support system at that time.

The university is located at the heart of Mexico City. Many students spend a considerable amount of time travelling to and from the university and their homes. Students come to the university from different parts of the city and its metropolitan area or from neighbouring cities.

During the training program in India the requirements´ analysis was not documented. The focus then was on developing a prototype.

The ELSS will be used as a brief case study for requirements. For that reason we design an informal specification of requirements in the following section.

In the following lines the different parties that have a stake in the system will be described. After that a the requirements are outlined in the form of an interview with the stakeholders.

A.2 Stakeholders

Academic personnel Under academic personnel we consider the teachers of the language center as well as the Director of the center. The academic personnel give lectures and examine the students regularly based on the university's calendar and the course's learning objectives. The Director of the Language Center coordinates the learning process, resolves conflicts stemming from group allocation, teacher-student interaction, supervises the evaluation process, reports advances to the academic direction of the university, and usually teaches.

Students Students are the individuals registered at the university in a given course or educational program. Their advance is regularly evaluated in terms of mastering a given language skill. Students of the language center at UDLA come from different academic faculties and share different professional and cultural interests.

Administrative personnel The administrative personnel are in charge of registering students, printing out certificates and giving support to the activities of the language center director.

A.3 Shaping the Requirements. Interviews

Simulated interviews with the interested parties will be used as a way of introducing the requirements in textual form. We consider that some interested parties also provide us with print-outs, reports and miscellaneous artifacts. A simulated walk-through is also considered. Interviews and artifacts have been designed based on an expert´s knowledge of the domain.

Miss Sylvia (Teacher): "above all, I would very much like to be able to grade examinations on a periodical basis and obtain the evaluations' results in a clear and understandable way. I would like to be able to enter the correct answers of a particular evaluation in the system and later on check the number of the correct answers per student and group. This is usually a hard work for the teachers, but we carry it out joyfully. Automatically obtaining comparisons among groups and identifying questions in which the students had more difficulties for answering would be helpful. This would certainly help us improve the learning process."

Miss Sylvia kindly gives us a copy of an actual examination. Examinations or "quizzes" are performed periodically. There is also a final examination for the students at the end of the semester.

Miss Geraldine (Director of the Language Center): "There is a lot of work to do here. Sometimes, especially at the end of the semester, we need to correct exams, send the final notes to the registrar, and send reports to the Dean. During the semester I also need to keep track of the evolution of the groups. At the beginning of the semester I am in charge of assigning teachers and groups, according to availability, it is no easy task, you know. Perhaps the system should leave such a complex problem for a second version of it. Now, let us focus on the evaluation aspect of the problem."

She also told us that the typical controls and final exams very rarely exceed a number of forty multiple choice questions. She allowed us to visit an examination session which provided us with valuable input and insight into the university's procedures.

David (Student): "we'll I'd like to review additional material at home, check my advance at my own pace. As I live outside the city presenting exams home would be nice. I'd also like to read lecture notes on the web."

Sara (Student): "this is such a great opportunity to allow us to check additional material on the web that reinforces our learning process and then be

able to present the controls and final exam under a controlled environment. I would feel so happy if I could present the controls in advance, because sometimes I am ready to present an exam weeks before the examination date."

A.4 Design Requirements

The administrative personnel has a software policy and it states that the software should be composed of the following three parts:

1. A relational database (DB) implemented on MySQL as Back-end. This will be further referred to as DB.
2. A front-end program in Java for applying exams and automatically obtaining students' results based on the correct answers stored in the DB. This is referred to as Exams module. The front-end will also contain an administration module.
3. A web-site with course supporting material for students.

A.5 What the ELSS shall do

We obtain the following requirements from the interviews above.

1. Functional Requirements
   (a) The system shall allow students to study course related support material on the internet.
   (b) Students shall also be able to attend distance education lessons.
   (c) Students shall be able to take exams in a computer at the campus under a supervised environment.
   (d) Students shall be able to take exams at their own pace and whenever they feel confident enough to do it. The former has to be carried out according to the university's semester calendar.
   (e) Once students feel confident enough they may go to campus and in the computer lab take a quiz to review their progress. Students will present the exam in the computer lab on given dates for the midterm or final exams.
2. Additional Functional Requirements
   (a) Access to the system shall be granted only to authorized users.
   (b) Identity of users shall be confirmed.
   (c) Only Academic Personnel is allowed to create, modify, or delete tests or quizzes, in the courses they are in charge.
   (d) Only Administrative Personnel may register a Student for a course and assign courses to Academic Personnel.
3. Non-Functional Requirements
   (a) Availability: the system should be available 24 hours a day, 7 days a week, though the system might stay off-line until any fault is fixed.
   (b) Performance: the system shall be able to attend at least 20 users simultaneously

B Weaving through Communication Refinement

These definitions are unpublished joint work with Professors Herzberg and Broy.

For a formal treatment of aspect-orientation, we need to explicitly highlight the means of communication in form of a *communication service*. A communication service might simply provide a static connection-oriented service, thereby reflecting an ideal or non-ideal connector. On the other hand, the communication service might also provide a connectionless means of communication, representing for example a message router. A communication service connects only a subset of the input and output channels of one component with those of another one.

Definition 8 (Communication Service). *Let $S_1 \in (I_1 \rhd O_1)$ and $S_2 \in (I_2 \rhd O_2)$ be given components. A* communication service *C for S_1 and S_2 is a component in $(I_1' \cup I_2' \rhd O_1' \cup O_2')$ where $I_1' \subseteq I_1$, $I_2' \subseteq I_2$, $O_1' \subseteq O_1$, $O_2' \subseteq O_2$. Its composition is defined by*

$$\exists\, I_1', I_2', O_1', O_2' : [\![S_1]\!] \wedge [\![C]\!] \wedge [\![S_2]\!]$$

and we then write

$$S_1 \leftarrow C \rightarrow S_2$$

□

For simplicity we assume here and in the following that all channels are named in a way such that there are no name conflicts. This form of inserting a communication service between two components is basically a generalization of composition with mutual feedback. The notation presented is just a way to semantically highlight that C represents the communication means and is not supposed to be a "usual" component.

Communication refinement relates two specifications of communication services C_1 and C_2 written at different levels of abstraction. Two other specifications, R and A, adapt the interfaces of the communication services and mediate between the abstract specification C_1 and the more concrete specification C_2. R is called the *representation* whereas A is called the *abstraction*.

Definition 9 (Communication Refinement). *Let C_1, C_2, R, and A be specifications such that*

$$C_1 \in (I_1 \rhd O_1) \wedge C_2 \in (I_2 \rhd O_2) \wedge R \in (I_1 \rhd I_2) \wedge A \in (O_2 \rhd O_1)$$

The relation of communication refinement from C_1 to C_2 is defined as

$$C_1 \rightsquigarrow (R \otimes C_2 \otimes A)$$

□

C Generic Solutions with AspectJ: an exploratory case

In this case study our interest is to learn to what extent AspectJ technology allows for a better separation of concerns and also for developing generic *concern-solutions*. By *concern-solution* we understand an aspect or a set of aspects that represent cross-cutting concerns, are woven with a system's core modules and produce a new system. Therefore changing the original system's behavior and structure in a desired way. In this case, we may consider an existing system as well as one under development. The motivation is to explore up to what extent existing AOP technology allows for reusing aspects.

So, the following problem is proposed. Suppose we need to insert a more or less generic undo() method in every class of a given object-oriented program. The `undo()` method should allow for a way to reverse the changes carried out in the objects of the program. In object-orientation adding such a method might imply major modifications of the classes in behavior and structure. The problem presented here may be seen as a typical cross-cutting concern problem. The cross-cutting concern presented here is the `undo()` method. We outline a solution that helps us explore the gap between the promise of AOP and the actual language tools. Existing aspect modeling approaches as the ones from [34] and [35] are used to illustrate the solution.

C.1 Proposal

This generic undo solution alters a class structure using an aspect and the concept of cloning, which is implemented in Java as an interface. In this solution the original program structure is preserved and the aspects are woven in a non-intrusive manner. We intend to create a framework for preserving an object's history, thus allowing for restoring, if required, an object's previous status after an operation. We assume that the object encapsulation is not violated, in other words, we may only alter an object's content through its set methods. Still we have to guarantee that only one method may actually perform a `set` operation on the concerned object. In summary, it is expected to keep track of methods that change the contents of an object. The solution is outlined in Fig. 4. In short, whenever a set method from a given object be called, the call shall be intercepted and a clone of the object be created as represented by step 1 in the figure, after this the intercepted method will be allowed to continue as shown in step 2. At this point, we may decide whether to follow with the changes on the object or discard them as seen in step 3. The rest of the program should remain unchanged.

The `undo()` is shaped as a clone or copy created every time the contents of an object are changed. Implementing it over existing classes is possible as an aspect that involves, on the one hand, inserting a cloning method to the target set of classes. On the other hand, requires specifying the point in the flow of the program to perform the copy of the object and then continue with the intercepted method.

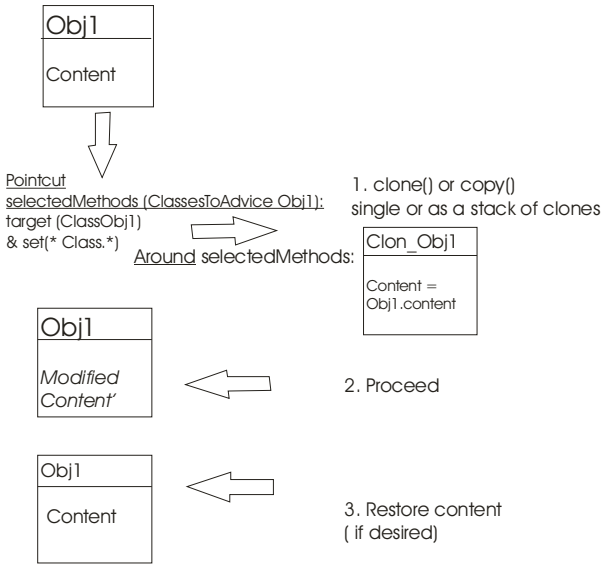


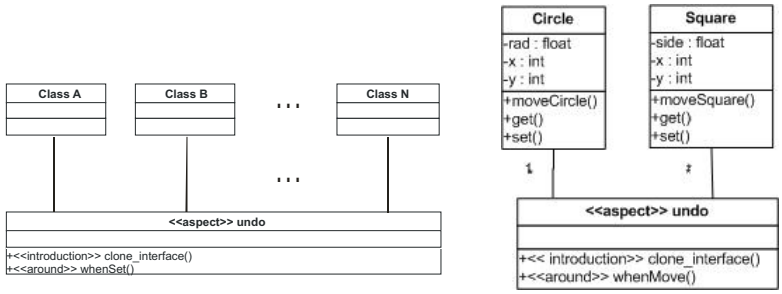
Fig. 4. General schema *undo()*

Pawlak et al [34] model an aspect as a class called "aspect-class" as shown in Fig. 5(a) and specify it with the stereotype *aspect*. This example allows us to portray one of the limitations of this modeling approach. In case we had another aspect altering the same set of classes or even a set of classes including ones from another aspect, then the class diagram would not necessarily be clearly understandable and we may need to model aspects as perhaps another layer of abstraction.Back to our example, it inserts an introduction with the clone method on every related class and specifies an advice which we may name *whenSet* (see Fig. 5(a)). The advice is called around the execution of a pointcut which consists of every set method called at any of the related classes.

C.2 Implementation

In order to illustrate that our ideal solution is feasible we define a class Circle as shown in figure 3, and the aspect following from the previous explanation. Because of its wide acceptance as an aspect language the implementation is built using AspectJ and consequently Java.

Static crosscutting From Fig. 5(b) the actual implementation should follow smoothly. Classes Circle and Square are the core modules and the crosscutting concern to be weaved on them is the undo aspect with the clone method and the actual implementation of the advice named whenMove().



(a) Generic solution (b) Particular solution

Fig. 5. Class diagram of the *undo()* aspect

Listing 1.1. Hyper-cutting i. e. Cloning interface added to class Circle

```
public aspect addCloneMethod {                                    1
   declare parents: Circle implements Cloneable;
   declare parents: Square implements Cloneable;
   public Object Circle.clone() {
   try {
    return super.clone();                                         6
   }
   catch (CloneNotSupportedException e) {
    throw new InternalError(e.toString());
   }
  }                                                               11
   public Object Square.clone() {
   try {
    return super.clone();
   }
   catch (CloneNotSupportedException e) {                         16
    throw new InternalError(e.toString());
   }
  }
}
```

The actual clone interface has to be specifically inserted to each class as illustrated in lines 2 and 3 of listing 1.1. Moreover, the method to restore the object to its previous values also needs to be specified for each affected class, because the clone method does not substitute the desired object, just substitutes it with a new one, and we use the clone just as a repository of the information, we need to define a function restore() to copy one by one the contents of the clone into the original object, otherwise we get two distinct objects after the advice in the rest of the program.

Dynamic behavior The dynamic behavior of the advice can be described using a sequence diagram based on the aspect modeling framework in [35].

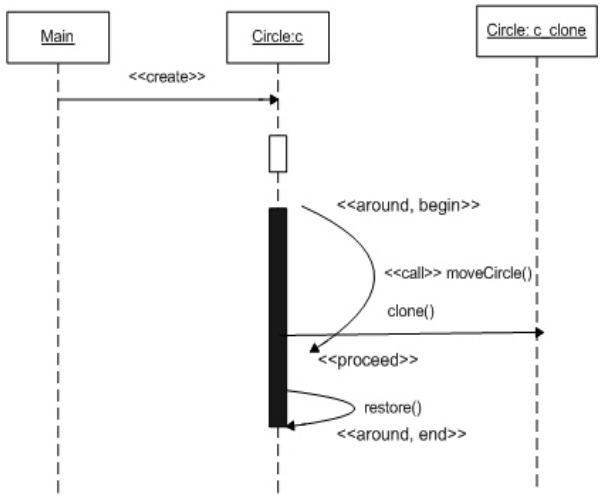


Fig. 6. Sequence of the **undo()** method

We focus on the class circle and the interception via the advice of a call to the method moveCircle().

A simple version of the pointcut and associated advice for the class Circle is found in listing 1.2 (for the Square is just about the same, just changing the names).

Listing 1.2. Pointcut

```
pointcut whenMoveCircle(Circle c) : target(c) &&  ↩
    call(* moveCircle(..));
```

In this case the pointcut was chosen in order to select every call to the method named moveCircle() where the target is an object of type Circle.

In case we would like to select every call to "set" methods we would have to add modify the pointcut definition and also add the proper restrictions to it in order to avoid undesired recursion. The solution is nevertheless not as generic as we would like it to be. So, it is safe and clear to specify every method to be intercepted specifically by its name. The same applies in the case of class Square, and so would have to be done with every class we would like to enhance with the `undo` concern.

Focusing ourselves on the needed restore() function, we would discover that it is necessary to refer to the attributes of each class and then copy them from the clone into the object in order to reverse the changes as outlined in figure 1 step 3. For that reason, it is not as generic as we originally intended it to be. The discussion of the gap between the promise and the reality of AOP present technology will be explored in the following lines.

C.3 Remarks

As we have seen, the introduction of the clone interface has to be performed class by class. Even if classes are grouped in packages it is not possible to introduce the interface to Java packages in AspectJ and make it work. Pointcuts need also to be specified in relation to selected methods, directly specifying the affected classes, instead of simply mentioning every "set" action as was our original intention. Otherwise, as we need to pass the object as parameter to the advice the selection of the corresponding clone and restore methods results unclear.

When using AspectJ the specification of join points requires careful adaptation and planning in order to avoid undesired recursion and stack overflow.

Also, the function that essentially achieves the restoring of data to the object has to be tailored for each class, in this case once for Circle and once for Square, if there were more we would certainly need corresponding restore functions for each. Otherwise we would have to explore using reflection classes that are out of the scope of the present work. Tailoring the solution for two classes is somehow undemanding, but in case of applying it to a number of classes it is certainly better than modifying each class in an intrusive manner, but is still far from being clear-cut.

This case shows that AOP shows the way to new and creative solutions toward software; allowing for a better separation of concerns and modifying existing code, though current tools or languages still lack the capability for straightforwardly shaping a generic solution and apply it on different pieces of code. It is still required to customize the solutions on a case by case basis, which posses a limit to one of the alleged goals of AOP that is encapsulation of concerns and reusability. Nevertheless, as has already been mentioned, this technology brings us closer to building software with new possibilities. This example allows us also to conceive aspects as a set of static and dynamic elements that convey specific behavior that has or may have an effect on vast sections of a given program. Therefore aspects are able to abstract not only crosscutting existing requirements, but also requirements added in a later stage in the software development lifecycle. This case supports our view that we need to go beyond the existing aspect technology and current limited approaches in order to allow for more ambitious and adequate means for aspect modeling, as well as more powerful, easy to use, and learn language mechanisms that allow us to create generic solutions and straightforwardly translate them into code.