Aspects of BPM/SOA: Processes, Use Cases and Concerns

Manuel Imaz, PhD

BlendMind
Madrid. España
imaz@mac.com

Abstract. In this paper we show how BPM/SOA avoid the increasing complexities added by the aspect-oriented programming (AOP) approach, mainly in relation to functional concerns. From the beginnings of object-orientation, some difficulties derived from the uses cases model have been detected, as they are the root of scattering and tangling. This is the question that AOP addresses even if it uses its own jargon: *concerns* in place of *use cases*. The present analysis of the problem is based on concepts of Cognitive Semantics (CS) that allow to explain some odd questions, such as the way of presenting the classical UML architecture as a '4 + 1' –in place of 5– views. Some CS concepts, such as perspective, focusing and profiling help to clarify some phenomena that have been analyzed from a very general notion of *view* that needs, evidently, to be refined in order to build more useful ideas about software engineering.

Key words: Concerns, aspects, use cases, AOP, cognitive semantics.

1 Introduction

In an earlier paper [6] we have presented the central role of categorization in Software Engineering as an important cognitive process, similar to abstraction. In fact, in Software Engineering we are constantly categorizing different aspects of reality, from the initial stages –requirements elicitation– up to the final ones. During the requirements elicitation and development we try to determine the needs –the problem domain– and the functions or features –the solution domain– of the system we are going to implement. There are two useful metaphors that may be quite adequate to conceptualize what happens during the requirements development: this is a *discovery* and *invention* process, where the needs have to be discovered while functions or features need to be invented. There is a difference between discovery and invention: 'The distinction is clear even in prescientific times: Fire was a discovery; the fireplace was an invention. That fire hardened clay was a discovery; pottery was an invention". [2]

After the requirements stage, we continue with the specification of the system or application. The way we specify software has evolved through the years, depending on the style used to conceptualize it. In terms of literary analysis it is said that a narration has a content but equally importantly it has also a style. Analyzing the question from a cognitive point of view –in particular from the point of view of Cognitive Semantics (CS)– it can be stated that:

> *In viewing a scene, what we actually see depends on how closely we examine it, what we choose to look at, which elements we pay most attention to, and where we view it from. The corresponding labels I will use, for broad classes of construal phenomena, are* ***specificity****,* ***focusing****,* ***prominence****, and* ***perspective****. They apply to conceptions in any domain.* [12] p. 66 (bolds in the original)

When specifying software it is evident that styles have been changing mainly in function of the focus, that is, what we choose to look at and also in function of the perspective. In a first time, the focus has been put on the procedures, the processes performed on data. This style is known as data flow representation, where the processes are specified as circles or bubbles and data are a means of connecting those bubbles. Following this style the focus was put on data objects or simply objects that travel through the data flows, but including in these objects the specific pieces of processes –called methods– applied to them. This evolution has finally lead us to focus on a broader scene, the business processes with activities and data objects to which these activities are applied.

The difference between business processes and data flow diagrams is the way we conceptualize them, as in the former we see a spatially distributed network of fine or medium grained processes while in the latter we consider coarse grained processes or conceptual process packages. In object orientation the packages are the data objects with the pieces of processes that are applied to them. So, in data flow diagrams we consider conceptual packages of processes independently of the spatial situation of such processes in a workflow and the data objects to which they are applied, while in object orientation the different processes applied to a data object are compressed into a conceptual package.

In order to show the difference between business processes and data flow and object oriented approaches, it is also necessary to use the *perspective* dimension of language, that is, the viewpoint from which we are observing the scene. Both data flow diagrams and object orientation are observed from the inside of the system to be developed, while business processes are observed from the inside of the enterprise or organization. These distinctions are more precisely defined using the concepts of focusing and prominence, described in the next section.

2 Cognitive Semantics

There are two approaches to semantics. The classical one –or realistic– considers that the meaning of an expression is something out there in the world. The semantic of table, for example, is a matching between the word *table* and a real world object. Cognitive semantics, on the other hand, identifies meanings of expressions with mental entities. [1]

Leonard Talmy states that Cognitive Semantics is the study of the way conceptual content is organized in language. In Talmy's view, a sentence (or other portion of discourse) does not objectively represent its referent scene –it is not something out there in the world–, but it evokes in the listener a cognitive representation, defined as an emergent, compounded by various cognitive processes out of the referential meanings of the sentence elements, understanding of the present situation, general knowledge, and so on [16] p. 93, note 2.

Historically, science has tried to be consistent with the need for objectivity eliminating the subject from the scientific discourse. The same effort has been assumed by the software engineering community when using a disembodied discourse, but the failure of this intention is unmasked when analyzing in detail some conceptual structures in which the subject surreptitiously reappears, as –for example– the concept of *perspective* implies an object and a subject and the concept of *focusing* implies that the subject is using his visual capacity (as Langacker defines it: "what we choose to look at" [12]).

Another aspect of cognitive semantics is that the conceptual structure is *embodied*, that is, the nature of the human mind is largely determined by the form of the human body. But the *form* of the human body must be understood in a broad sense, meaning the human being in an environment, in a given situation –cultural, social, and so on– as some concepts of CS imply. For example, in the previous section we have mentioned the concepts of focusing, perspective and so on. It is evident that a perspective implies a subject observing a scene from a given point of view, that is, a subject in a given situation.

The concept of *perspective* allows us to make a difference between observing a software system from an internal or external point of view, and conceptualizing the internals of the software system or a general viewpoint that encompasses the business processes running in the enterprise. The concept of perspective is represented in (Fig. 1):

When focusing on the computer system we need additional concepts in order to use different categories applied to the same system. Besides what we choose to look at –focusing– we need to take into consideration which elements we pay most attention to or *prominence*, in particular one sort of prominence: *profiling*. Langacker states that:

> *As the basis for its meaning, an expression selects a certain body of conceptual content. Let us call this its conceptual* ***base****. Construed broadly, an expression's conceptual base is identified as its maximal scope in all domains of its matrix (or all domains accessed on a given occasion). Construed more narrowly, its base is identified as the immediate scope in active domains –that is, the portion put 'onstage" and foregrounded as the general locus of viewing attention. Within this onstage region, attention is directed to a particular substructure, called the* ***profile****.* [12] p. 66 (bolds in the original)

In our example, one conceptual base is the computer system and the profile may be a process or a data flow –a particular substructure– or an object in another profile. That is, the same conceptual base may be considered in terms of different profiles: data flows and processes or objects. Both ways of categorizing the computer system are different types of conceptual integrations or blends (which will be considered in the next section). On the other hand, a conceptual base such as a business process, may be profiled in terms of tasks, decision points, etc., or may be also profiled as use cases, that is, subsets of the business process in which some actors –users– interact with software components in order to achieve a goal.

3 Metaphors and Blends

Metaphor is a cross-domain mapping –conceptualizing one domain in terms of another– and is central to our thinking process. The first domain –the well known– is called the *source* domain while the new one –less known– is the *target* domain. The usual idea we have of a metaphor is that of a literary figure whereby we say something using a figurative expression. In fact, the figurative expression is the external manifestation of an underlying cognitive process: that is precisely the conceptual metaphor.

An important and well-known metaphor –in relation to ontologies– is the conduit metaphor, first analyzed by Reddy [14]. This metaphor reflects quite singularly the objectivist philosophy: the mind contains thoughts, language transmits ideas, human communication achieves the physical transfer of thoughts and feelings, etc. and it is embodied in many expressions which are manifestations of the metaphor:

You have to *put* each concept *into words* very carefully.
Try to *pack* more *thoughts into* fewer words.

Reddy's assertions regarding the underlying cognitive processes are similar to those used currently by cognitive semantics, proposing that texts are instructions to create mental spaces (patterns of thought, in Reddy's terms) which, as any active complex process, will re-create, re-enact meaning.

There is another way of conceptualizing both terms of a metaphor (source and target domains), using the concept of mental space. The concept of mental space refers to partial cognitive structures that emerge when we think and talk *'allowing a fine-grained partitioning of our discourse and knowledge structures'*. [3]

Finally, a conceptual integration or blend [4] is an operation that could be applied to a couple of input spaces, which gives as a result a blended space or blend. The blend receives a partial structure from both input spaces but has an emergent structure of its own.

One important example of blend is that of imaginary numbers, first showed up in the formulas of the sixteenth-century. The authors Cardan and Bombelli considered imaginary numbers only as notational expedients, with no conceptual basis (they were called sophistic, imaginary, impossible).

This is an interesting example not only because it is an illustration of how blends are also created in science taking sometimes many years, but also because its initial status was not ontological at all -instead, it was its practical usefulness that allowed the concept to survive- to end up, after an epistemological elaboration, as a very concrete and useful theory in mathematics. Rolando García asserts that in cases like this one, as well as in many others, *there is no ontology without an epistemology.*[5]

The important point is that the intertwined relations between both philosophical disciplines -Ontology and Epistemology- derived in a new approach called by García as Constructivist Epistemology, meaning that we need to think of scientific explanation as ascribing to the empiric relationships –to external reality– the necessary connections which are verified in the logico-mathematical structures of scientific theories. This constructivist approach to epistemology, when applied to IT domains, results in taking as existent what has been built –results or elaborations– in previous stages of the disciplines. For example, the blend built to framing a class –as in UML, with three containers for a name, the attributes, and the operations– is one of the two input mental spaces used to build a new, concrete class, as the *invoice* class.

Data-flow diagrams (DFD) are based on a metaphor. Even if one process is also categorized as a container and its structure is determined by another data-flow diagram at a lower level, the main metaphor on which the model is based is THE SYSTEM IS AN INDUSTRIAL PLANT. In such a plant, there is a collection of processes interconnected by pipes or assembly lines. The raw material for one process originates from other processes, external sources, or stores containing by-products of yet other processes. [7] pag. 89

The paradigm of object orientation has its own constitutive metaphor: THE SYSTEM IS A SOCIETY OF PEOPLE. Object orientation is full of expressions based on this metaphor. Objects have *responsibilities*, they *collaborate* with each other, they *have acquaintance* of other objects, they *communicate*, they have a defined *behavior*, and so on. [7] p. 90

In an invoice object, for example, we have a mental space that corresponds to a frame of three containers: one for a name, another for attributes and a third for operations. Another mental space corresponds to a real world entity –a piece of paper with data– but there are other mental spaces with activities performed on the invoice. So we incorporate into the blend the actions that some agents will perform on the entity we are modeling. In order to get a complete set of mental spaces, we need to analyze the different stages of the invoice in its whole business story or life cycle. So in general, there may be several other mental spaces that provide a source of behavior in terms of operations.

As a consequence of creating the blend, there will be –in a class– an emergent structure compared to the input mental spaces. We may see that an invoice class –in contrast to a real-world invoice– will generate objects capable of producing events or sending messages to other objects. This behavior is something that real, inanimate invoices, cannot do. [7] p. 95

4 Perspectives and Conceptualization

Restoring the subject in the discourse means to make visible some aspects that normally remain hidden. When talking about a software system we may adopt different perspectives that usually are implicit in the language but when making them explicit they may suggest to us interesting questions. One point is that each perspective may have different views, as it may be seen in the internal perspective.

The more frequent perspectives adopted to describe a software system are shown in the following figure (Fig. 1).

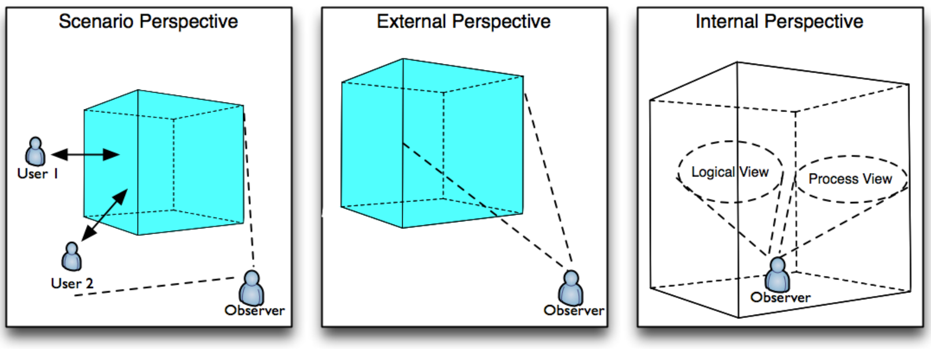

Fig. 1. Different perspectives: the internal, the external and the scenario perspectives

When using the concepts defined by Langacker: *specificity*, *focusing*, *prominence* and *perspective* we must remember that the sense of vision is not a merely passive, photographic one, but a very complex construction as shown by Francisco Varela [17], p. 332:

> *A first-group of animals [cats] was allowed to move around normally while harnessed to a yoke; their gross movements were transferred mechanically to a second group of animals conveyed in gondolas. The two groups shared the same visual experience, but the second group was entirely passive. When the animals were released after a few weeks of this treatment, the first group of kittens behaved normally, but those who had been carried around behaved as if they were blind: they bumped into objects and fell over edges. This marvelous study supports the* ***enactive*** *view that objects are not seen by the visual extraction of features, but rather by the visual guidance of action. Similar results have been obtained under various other circumstances and studied even at the single-cell level.* (bolds in the original)

So, what is implied in last resort is an embodied concept, jointly determined by a physical perception and bodily actions and with additional cognitive constructs. The sense of vision is frequently used as a metaphorical concept as when we say ‘I *see* what you mean by that’. It is in this sense that we will use the concepts defined by the CS.

Each perspective implies its own views, as in the *internal perspective*, which has been represented in Fig. 1 using two views (process and logical) of the set of views defined in UML. While in the three concepts of *perspective*, *focusing* and *specificity* the visual metaphor is quite direct, the concept of *profiling* deserves some additional comments. Langacker ([12] pp. 66-67) points out that:

> *The profile can also be characterized as what the expression is conceived as designating or referring to within its base (its conceptual referent)... In fact, it is quite common that two or more expressions evoke the same conceptual content yet differ in meaning by virtue of profiling different substructures within this common base. For instance,* ***Monday, Tuesday, Wednesday****, etc. all evoke as their base the conception of a seven-day cycle constituting a week, within which they profile different segments.* (bolds in the original)

As the Langacker’s example shows, both the structure and the substructures are conceptual constructions, based on framing a conceptual base –the week– in terms of another conceptual units –the days. So, we can choose different conceptual frames to profile the elements that make up a software product.

Thus, in the internal perspective we can choose a profiling based on different metaphors, in particular the THE SYSTEM IS AN INDUSTRIAL PLANT metaphor, which works as a frame to *see* processes and connections among them as well as data stores and even, in some cases, external interactors. These are different views when changing the *focusing* (what we choose to look at). We may also go from the analysis to the design varying the *specificity* (how closely we examine it) and go, for example, into the processes (viewed in the analysis) to find modules (viewed in the design), which are organized in an hierarchical structure.

An alternate profiling of the internal perspective is using the THE SYSTEM IS A SOCIETY OF PEOPLE metaphor, on which some blends are built, in particular classes. The blends are categorized into groups that correspond to different views when changing the focusing: considering the static, structural aspects we get the *logical view*, while when considering the dynamic aspects we get a *process view.* In this perspective and profiling there are also, when changing the focusing, other views, such as the *physical view* and the *development view.*

An interesting point in relation to views is that all of them are orthogonal or complementary, that is, none of the views may be translated into another view. The *whole* view is the addition of all the previously ones: logical, process, and so on. As each view is the result of catching a different partial sight, the whole view is necessarily the addition of all of them.

The *external perspective* implies observing the system as a whole and the way of conceptualizing it is by using a name or syntagm. When the system software matches a previously existent activity in a domain, the name used is derived from the domain as in the examples of *invoicing, general ledger, order management* or *payroll.* This way of conceptualizing brings nothing new to the system to be implemented, except the experience provided by the software engineer and explains the symptoms pointed out by Yourdon ([18], p. 360) in relation to the top-down problem: 'analysis paralysis', the 'six analyst' phenomenon, or the 'arbitrary physical partitioning'. The top-down method results from gradually changing the specificity of a perspective.

There is also a very frequent use of figurative or metaphorical names to conceptualize a software system. Names such as *broker, bus, framework* or *virus* are usual examples of this way of conceptualizing. The advantage of a metaphorical naming is that the source domain contributes with a rich set of features that may be translated into the target domain, that is, the system to be implemented.

The *scenario perspective* includes the possible interactions of users –human and not human– with the system. This perspective needs its own representation, that is –as usually occurs with scenarios– a dialog, script or description of interactions. As in the scenario we find human users, with intentions, goals or concerns about the system to be implemented, usually these goals or concerns are included in the conceptualization of the scenario. The sentence *withdraw money* is, at the same time, the description of a scenario but also the goal of the user involved in such scenario.

5 Some Problems with Use Cases considered as Object Oriented Constructs

The way of presenting in UML the '4 + 1' views already was a symptom. According to Kruchten, these views are the description of an architecture and *can be organized around these four views, and then illustrated by a few selected use cases, or scenarios which become a fifth view* [10]. This comment shows that there is something heterogeneous between use cases and the other views, even if the author call all of them *views.* The question would be: why the use cases view is different from the other views to the extent Kruchten uses a different symbol –an ellipse– in place of a rectangle?

As we have seen in the Perspectives and Conceptualization section, use cases belong to the scenario perspective and the four object-oriented views belong to another perspective –the internal perspective– of the software product, that is, the OO views and the use case view –according to Kruchten– correspond to two different perspectives of our definition (with their focus and profiles). And each perspective needs a specific representation as the focus leads to perceive different facets of the same product. That explains the heterogeneity between the first four views and the last one, and why the consideration of use cases as object oriented constructs gives rise to some problems that have required specific solutions.

Jacobson points out in his paper [8] that to achieve use case modularity it needed two mechanisms: a *separation* mechanism and a *composition* one. He focused on the separation mechanism, which allowed to keep most use cases separate, but leaving aside the composition mechanism.

For example, it has been recognized that there are basic use cases, each one being independent of the others. However, some use cases –extension use cases– depend on other, more basic, use cases to work. In terms of object orientation, the solution may be to create subtypes –using the inheritance mechanism– from a base use case resulting in an extended use case. But the solution does not allow us to modify the base use case, for this we need a new mechanism –an extension– in order to add new functionalities.

Using the extend mechanism we get extension use cases and iterating the same extension mechanism the use case continues to grow but keeping also most use cases separate all the way down to code and even to executables. But this is not a clear and simple mechanism: even recently it has been shown that *the Achilles' heel of use cases is the unclear UML semantics, in particular the definition of the extend relationship.* [11]

When dealing with object orientation, we can verify that use cases are realized in multiple classes and conversely, each class includes portions of multiple use cases. In the jargon of object orientation these characteristics are called *scattering* and *tangling* respectively. [8] These characteristics are usually represented as in the example of Fig. 2.

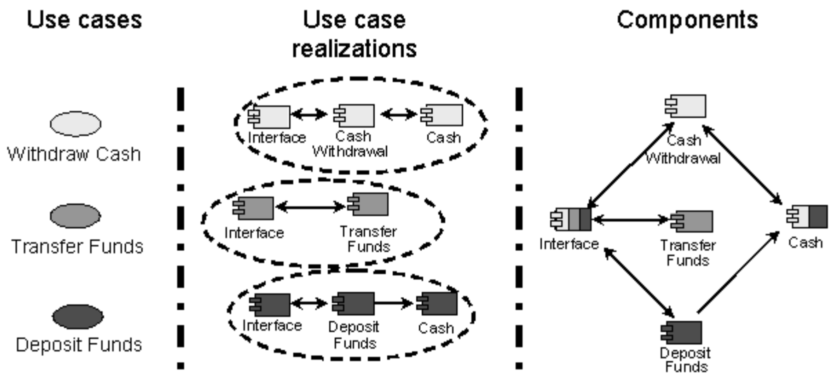

Fig. 2. Scattering and Tangling

This characteristic is known as *crosscutting*, meaning that a given concern usually spans layers and tiers of an application. The question is that at the same time that a crosscutting concern affects the entire application –implying the scattering– it should be centralized (included in a separate module) in one location where possible in order to help create a quality and maintainable software.

The question is still more complex when the paradigm of object orientation is applied to use cases, as they are not derived from a software product perspective but from an enterprise perspective. In a use case model it is possible to use object oriented concepts, for example, generalization. At such an abstract level, nothing prevents us from generalizing or specializing use cases the same way we generalize or specialize classes. But the problem arises when we try a use case realization that has to reuse a more abstract use case realization. As Jacobson explain [8]:

> *However, the extension mechanisms provided between use cases didn't make it to collaborations; I simply couldn't make a case for this since we had no mainstream programming language supporting the implementation of extensions as we now will have with AOP. Consequently, it is not possible to separate extension use cases from base use cases in design and implementation. The realization of the extension use case has to be dissolved into the realization of the base use case, and the base use case cannot be oblivious of the extension use case. So we do not have a fully seamless transition from use case modeling to design –**realization of extension use cases has to be intermingled with the realizations of base use cases.***(bolds in the original)

The difficulty comes from mixing heterogeneous conceptualizations: use cases and objets. The ideal solution would have been to get separate modules from extension use cases the same way we produce classes and subclasses as separate components. Here we need two kinds of modules: use case modules and component modules. Jacobson states that Aspect Oriented Programming (AOP) has come to the aid of these problems, allowing to create a new kind of module: use case modules.

6 Requirements and Concerns

Some definitions of a *requirement* state that it is a software capability needed by the user to solve a problem, to achieve an objective. An alternate definition refers to capabilities that must be met or possessed by a system or system component to satisfy a contract, standard, specification, or other formally imposed documentation. [13]

In other words, the requirements for a system are the descriptions of what the system should do, even if the term is not a well defined one. As Sommerville [15] points out:

> *The term requirement is not used consistently in the software industry. In some cases, a requirement is simply a high-level, abstract statement of a service that a system should provide or a constraint on a system. At the other extreme, it is a detailed, formal definition of a system function.*

There are other concepts related to requirements, such as *needs* and *features*, aimed to refine the sometimes high-level, abstract statement. The users are in a given environment and have business or technical problems that they need the software engineer's help to solve, these are the user's needs. In addition we may consider features as a service provided by the system that fulfills one or more stakeholder's needs.[13]

On the other hand, the AOP community uses the new concept of *concern*. Among the definitions of *concern* found in dictionaries we can point out these two: "To engage the attention of" and "Regard for or interest in someone or something". Both definitions are related to the CS definition of focusing, which we have been using in addition to perspective and profiling. We talk about user or stakeholder concerns referring to what can be stated as requirements, functional and non functional. The concept of concern is closely related to human intentions, goals or objectives.

But in the AOP community they have modified concern by *crosscutting concern*, indicating that programming languages decompose concerns into separate, independent entities by providing abstractions (e.g., classes, modules or procedures) that can be used for implementing these concerns. Some of these concerns defy these forms of implementation and are called crosscutting concerns because they "cut across" multiples abstractions in a program or component.

In 1986 Ivar Jacobson first formulated the concept of use cases –originally called *usage scenarios* and *usage case*– as a textual, structural and visual modeling technique. Interestingly, the term *usage scenarios* points to the idea of perspective –the scenario perspective–and the associated cognitive framework. It was a great idea to include a business perspective –which includes users and their intentions and objectives, their concerns– in addition to the software system perspective exposed in the object oriented constructs. What the previous paradigm –structured analysis and design– lacked was precisely the business perspective –with systems and users– to specify requirements and this old approach implied to directly specify an analysis view from a set of statements more or less well constructed.

The success of use cases may be attached to the chosen perspective, as the visualization of a business process is something that may be perceived more directly (recognizing that perception is not a passive sense but one that implies continuos reorganizations and incremental capacities) when observing the enterprise. The advantage of use cases compared to requirements expressed as high-level abstract statements is a more concrete profiling, as we visualize the use of a particular function or service of the application as interactions between some users and the software itself.

7 Aspects and Use Cases

In AOP the goal is to reorganize the source code in order to recompose the concerns as modules. This is fundamentally a programming task using metalanguages that allow to encapsulate fragments of code that are distributed among various components. The composition resulting from this reorganization is called an *aspect*. The reason for building aspects is a better understanding and maintenance of the software application, as each concern may be matched to a module.

The problem is explaind by Jacobson [8]:

> *We tried to specify and design them as separate units, however, when implementing the use cases, they were integrated to a mass from which it was impossible to identify which use case was being implemented by which piece of code. Or, in other words,* ***the use cases were dissolved into the code, and distilling them from the code was far from easy.*** (bolds in the original)

It is funny to observe that in AOP the original concerns –the requirements– are known as *early* concerns. This conceptualization is the result, evidently, of a given perspective: in this case, from the construction phase. AOP focus its interest in the code reorganization task, where concerns are recomposed into modules –the aspects– and so the elicitation phase is an *early* one.

Jacobson, in a book that extents his ideas about AOP [9], explains that:

> *It is well known that aspect orientation helps modularize crosscutting concerns during implementation, but there is a need to modularize crosscutting concerns much earlier, even during requirements. Use-cases are an excellent technique for this purpose. Use-cases are crosscutting concerns, since the realization of use cases touches several classes. In fact, you can model most crosscutting concerns with use-cases, and we demonstrate use-case modeling in the book.*

The emergence of BPM/SOA lead us to compare business processes and use cases. When defining a business process as a set of related, structured activities or tasks that produce a specific service –satisfy a particular goal– for a particular customer, we verify that the definition may be equally applied to use cases. A use case is usually defined as a list of steps, typically defining interactions between a role and a system, to achieve a goal. This similarity allows us to state that a use case may be an activity, a subprocess or the business process itself.

The cross-cutting phenomenon is the result of having to translate the use cases into constructs of a different perspective, that is in object-oriented constructs such as classes and components. The ideal solution would be to have the possibility of directly executing the set of interactions that make up the use cases.

When representing business processes with an appropriate language, it is possible to directly run this representation.This way, we maintain the early concern –the use case– as a module without the need of using a composition mechanism, such as addressed by AOP.

Something equivalent to the phenomenon of cross-cutting also appears in a linear narrative, whether technical or not. In a narrative about a given subject there is a linear discourse that has many references to other subjects. The traditional solution in printed articles or books has been the use of a set of mechanisms in terms of calls to the footer, to references, to other sections of text, etc. In a biography (the main concern), the linear narration of the life, for example, of an important computer science personality is cross-cut by multiple areas of interest: childhood and youth, university and work on computability, cryptanalysis and so on.

The idea of hypertext has enabled the possibility of showing the main text –the concern– with other areas of interest traversing it. In the case of a biography the text is usually embedded –in a mobile device, for example– with a number of icons to insert the text corresponding to the multiple specific areas (childhood and youth, cryptanalysis, etc). Each insertion corresponds to a new level of specificity. We may have the global picture –the whole concern– and gradually insert different cross-cut areas of interest. This form of presenting the biography allows a good maintainability of the whole concern and the specific areas of interest that cross-cut the concern.

This hypertext mechanism would also allow a similar easy maintainability of software, provided that the features of import/export would be included in the programming languages in order to see the specific components (for example, attributes and methods of classes, components of composite components) that realize the main concern (use case) importing and including them in the main text.

8 BPM/SOA and Concerns

In Fig. 2 we have a representation of a group of use cases, which are realized as collaborations and, finally, those collaborations realized as a set of components. The usefulness of use cases is due to its way of representing concerns (requirements). In terms of AOP, we can represent early concerns as use cases and finally –at the implementation time– represent the same concerns as aspects. But the advantages of use cases as concern representations disappear when they are scattered into groups of components. There is a hard work to represent, then lose in translation and finally recompose, at implementation time, the concerns.

There is no ideal solution to the problem, but the way of representation of concerns as business process diagrams is a great advantage over the classical representation of use cases. Business process diagrams, when created with a Business Process Management (BPM) tool and an adequate notation as BPMN 2.0, do not vanish as use cases do in translation, but remain intact and are executables as such until a new version is created.

At this point it is important to make the difference between BPM solutions that, as the classical code generation tools, generate all the code necessary to execute the process, and the BPM/SOA architecture where the process activities are associated with services. The services are implemented with pieces of software derived from legacy systems or software built specifically with this purpose.

On the other hand, there are also SOC solutions, that is Service Oriented Computing. The question is that this approach aims at implementing distributed applications based on the interactions of services, as an assembling of services that enhance the reusing of components. But SOC is not based on business processes and the concerns must be treated as AOP proposes to get aspects.

A use case may be a task, a subprocess or a whole business process. The aim was to indicate the usefulness of use cases as a result of adopting a different perspective. But business processes are better understood by users and they are persistent in the same representation, the BPM language, (and portable to other platforms, for example) through all the stages of the development.

But the question of scattering remains. The difference is that each task in the business process may be implemented as a service, that is, a component and then it is not necessary to recompose the concern in order to ensure a good understandability and maintenance. The service may be implemented as a component and the component may be composed, in turn, as a collaboration of other components, for example classes. In this case, the service may be allocated to different components, each one containing code fragments of other services.

There is a granularity size difference between concerns –that may cover a whole business process– and services and so the complexity of the underlying components is also decreased. Some solutions have been suggested such as partial classes in order to spread classes over separate files and matching each file to a different service, for example. The management of services greatly simplifies the maintenance of the whole business process that is not longer implemented as a whole block of software.

9 Conclusions

We have seen that, in relation to the scattering and tangling phenomena, the heterogeneity of the four object-oriented views and the use cases view –as belonging to different perspectives– and their representations is the cause of the cross-cutting phenomenon and the solutions proposed by AOP.

This is the main reason why use cases crosscut the other representations (classes, components and so on): use cases must be translated into other representations, belonging to a different perspective. Different representations in the same perspective do not crosscut as the practice of UML confirms because they are complementary: we depict the structure of objects in the logical view and afterwards the behavior of them in the process view. Or, after depicting the components and its behavior, we can show where they will be executed in the physical view. The representations of the same perspective are additive: the whole view –what has been called the *architecture*– is the integration of all representations.

As a way of avoiding the increasing complexities of developing software systems using AOP, the BPM/SOA approach greatly simplifies the development as it eliminates the need for creating aspects from the functional concerns. The concerns are directly represented as business process diagrams that remain –in contrast to use cases– throughout the entire process of development and eventually are executed. The business process representation has languages and tools that simplify the maintenance and the visibility of processes, with the ability to see the components of activities and even the execution of processes and activities.

The question of non-functional concerns (as security) remain but in a service oriented approach these concerns may be encapsulated in specific services with which the business services will interact. The separation of concerns is realized as a set of independent loose coupled services, greatly decreasing –or even eliminating– the need to use AOP and increasing the reusability because they are reusable business services that comprise people, processes, and systems and not merely technical ones.

Acknowledgements The author is grateful to Mauricio Milchberg for his revision of the manuscript and valuable comments.

References

1. Allwood, J. and Gärdenfors, P. (eds.): Cognitive Semantics: Meaning and Cognition. John Benjamins Publishing Company. Amsterdam/Philadelphia. (1999)
2. Burton, R.: Hedy Lamarr: The Most Beautiful Woman in Film. The University Press of Kentucky. (2010)
3. Fauconnier, G.: Mappings in thought and language. Cambridge, Cambridge University Press. (1997)
4. Fauconnier, G. and Turner, M.: The Way We Think: Conceptual Blending and the Minds Hidden Complexities. Basic Books. (2002)
5. García, R.: El conocimiento en construcción. De las formulaciones de Jean Piaget a la teoría de sistemas complejos. Gedisa Editorial. Barcelona. Spain. (2000)
6. Imaz, M.: Abstracción y Conceptualización: Un Enfoque Cognitivo. 40 JAIIO, Jornadas Argentinas de Informática, 29 de Agosto al 2 de Septiembre de 2011, Córdoba, Argentina. Anales 40 JAIIO /CD40JAIIO/T2011/ASSE/603.pdf (2011)
7. Imaz, M. and Benyon, D: Designing with Blends. MIT Press. (2007).
8. Jacobson, I.: Use Cases and Aspects – Working Seamlessly Together, in Journal of Object Technology, vol. 2, no. 4, July-August 2003, pp. 7-28. http://www.jot.fm/issues/issue_2003_07/column1/
9. Jacobson, I. and Ng, P.: Aspect-Oriented Software Development with Use Cases. Addison Wesley Professional. (2005)
10. Kruchten, P.: Architectural Blueprints – The 4+1 View Model of Software Architecture. IEEE Software 12 (6) November 1995, pp. 42-50. (1995)
11. Laguna, M., Marqués, J., and Crespo, Y.: On the Semantics of the Extend Relationship in Use Case Models: Open-Closed Principle or Clairvoyance? CAiSE 2010, LNCS 6051, pp. 409–423, Y. B. Pernici (Ed.), Springer-Verlag Berlin Heidelberg. (2010)
12. Langacker, R.: Cognitive Grammar: A Basic Introduction. Oxford University Press, Inc., New York. (2008)
13. Leffingwell, D. and Widrig, D.: Managing Software Requirements: A Use Case Approach, Second Edition. Addison Wesley. (2003)
14. Reddy, M.: The conduit metaphor: A case of frame conflict in our language about language. In Metaphor and Thought . Ortony, A. (Ed.). 2nd. edition. Cambridge University Press. (1993).
15. Sommerville, I.: Software Engineering. Ninth Edition. Addison Wesley. (2011)
16. Talmy, L.: Toward a Cognitive Semantics. Vol. 1, Concept structuring systems. Cambridge, MA: MIT Press. (2000)
17. Varela, F.: The Reenchantment of the Concrete. In Incorporations, pp. 320-39. Crary and Kwinter (Eds). Zone Books. (1992)
18. Yourdon, E.: Modern Structured Analysis. Englewood Cliffs, NJ: Prentice Hall. (1989)