International Journal of Computer Science and Applications,
2008, Vol. 5, No. 3b, pp 69 - 92

A SCENARIO AND ASPECT-ORIENTED REQUIREMENTS AGILE APPROACH

JOAO ARAUJO

Faculdade de Cências e Tecnologia, Universidade Nova de Lisboa,
Caparica, Portugal
ja@di.fct.unl.pt
http://ctp.di.fct.unl.pt/~ja

JOAO C. RIBEIRO

ONI Telecommunications,
Lisboa, Portugal,
joao.ribeiro@oni.pt

Software development agile methodologies aim at promoting fast communication and incremental software delivery. The success of these methodologies in permanently evolving systems depends on how software engineers identify and structure changing requirements. Current agile methodologies fail to explicitly deal with the crosscutting nature of requirements, compromising the speed and capacity of systems to evolve. Since Aspect Oriented Requirements Engineering deals with crosscutting requirements systematically, its concepts and mechanisms can be integrated to agile methodologies. This work proposes a scenario based requirements approach – AspOrAS – that incorporates aspects.

Keywords: Aspect-oriented requirements; Agile modeling; Scenarios.

1. Introduction

The goal of software development is not just creating systems, but creating evolvable and adaptable systems where time to market is a constraint that must not be ignored. Nevertheless, traditional methodologies are not agile enough to deal with evolving systems. Software agile development [Ambler (2002)] intends to promote fast communication and incremental software delivery. The success of agile approaches to develop permanently evolving systems depends on how successfully software engineers identify and structure changing requirements.

On the other hand, systems evolution depends on how efficiently the crosscutting nature of concerns [Kiczales *et al*. (1997)] is dealt with. This is not addressed explicitly by agile methodologies. Crosscutting or aspectual requirements are requirements that influence other requirements and are scattered or repeated throughout the requirements artifacts due to the tyranny of dominant decomposition [Tarr *et al*. (1999)]. Failing to identify these requirements leads to requirements dispersion and tangling with other requirements thus compromising modularity, and making software evolution more difficult due to the requirements change management overheads. The result is the systems quality is negatively impacted.

Aspect oriented requirements engineering mechanisms [Baniassad *et al.* (2006)] deals with crosscutting requirements systematically and this can be integrated by agile software development methodologies.

This article proposes an agile requirements approach that integrates crosscutting requirements management aiming to achieve more evolvable and adaptable systems. Requirements are described through scenarios. Scenarios are systems current or desired behaviors, examples of interactions between users and systems, or stories describing the expected system usage.

This paper is structured in as follows: Section 2 introduces some background; Section 3 describes the current proposal through an example; Section 4 applies it to a real case study; Section 5 discusses some related work; finally, in Section 6, we draw some conclusions and point out directions for future work.

2. Background

2.1. Aspectual requirements

Requirements that are scattered and tangled with other requirements are named crosscutting or aspectual requirements [Baniassad *et al.* (2006)]. These compromise the requirements management as failing to deal with them leads to modeling inefficiencies, increasing the efforts for change management. Therefore, they must be identified and modeled separately.

Management of aspectual requirements consists of identification, capture, and composition [Rashid *et al*. (2003)]. In the identification step the collection of concerns (i.e. a subject of interest to the system) is defined, and some of them are crosscutting, i.e. aspectual requirements. In the capture step, requirements are reorganized to reflect an improved separation of concerns with the modularization and representation of properties identified as aspects. Composition rules specify how non-aspectual requirements are influenced or affected by aspectual requirements [Rashid *et al*. (2003)].

2.2. Agile modeling

Agile modeling is about the daily and perseverant application of a group of practices, principles, objectives and values described in the Agile Software Manifesto (http://www.agilemanifesto.org). Agile models fulfill its objectives with the minimum possible energy cost. They are only sufficiently precise, consistent and detailed to achieve its objectives, in a simple and noticeable way. Models are only expected to work (that is, transmit the message in an efficient way), they don't need to be perfect and certainly not complex.

There is an emphasis on the communication process, through the creation of something "visible", e.g. application interfaces, as fast as possible, through flexibility in executing tasks in a deliberately and collaborative way.

Many methodologies and practices were and are still being created. Among these, Extreme Programming (XP) [Beck (2000)], Crystal Methodologies [Cockburn (2004)], Scrum [Schwaber and Beedle (2002)], Feature-Driven Development (FDD) [Palmer and Felsing (2002)], Agile Modeling (AM) [Ambler (2002)] stand out.

3. ASPORAS

The AspOrAS (Aspect-Oriented Agile Scenario) approach is illustrated in Figure 1. It is organized in two major blocks: (i) Domain Analysis and Modeling, and (ii) Development, which are iteratively executed through the system lifecycle. We will integrate aspects at domain and modeling levels.

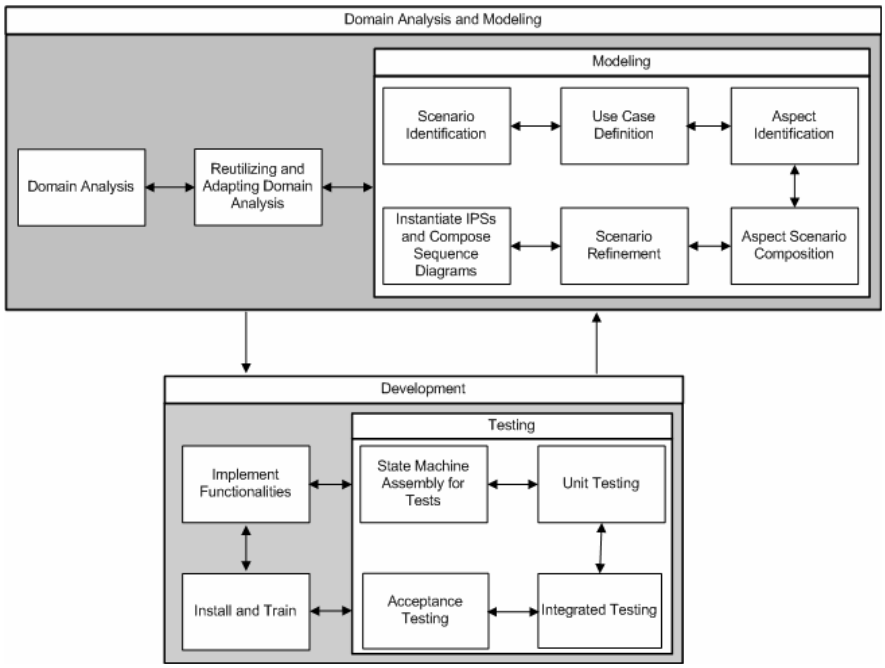

Fig. 1. The AspOrAS process model.

The proposed approach will be explained in detail in the following subsections using a parking lot example, taken from [Araújo *et al*. (2004)], here described:

"To use any parking lot system, the client must obtain a ticket from a machine by pressing a button. Then the vehicle may enter the parking lot and park in an available space. The system must control whether there are any available spaces. When the client wishes to leave the park, he must pay the previously obtained ticket, in a payment machine. The amount paid depends on the time spent in the park. After paying, the client

may leave the park by inserting the paid ticket in a machine that will open a gate. All the machines must be operational for the client to use park facilities. What we want to develop is a parking lot system that also contemplates frequent users, who may buy pre-paid parking time and enter and leave the park by inserting a card and a PIN, which automatically deducts a value from the clients account."

Note that, since our emphasis will be on requirements, we will not discuss the implementation activities.

3.1. Domain Analysis (DA)

Our approach starts the modeling process by reusing and analyzing the domain artifacts where the application will be built. These artifacts consist of the description of reusable concepts, functionalities, and scenarios, by creating two documents: a domain use case model and a domain glossary.

Domain use cases are an easy way for identification of reusable information by anyone involved in the application development as it provides a structured definition of the domain elements. The domain use cases are described by a domain use case diagram and domain use case templates. This diagram is a use case diagram [Jacobson and Ng (2004)] at domain level, that defines entities (actors) and related domain functionalities (the domain use cases). Figure 2 shows the domain use case diagram for the proposed example.

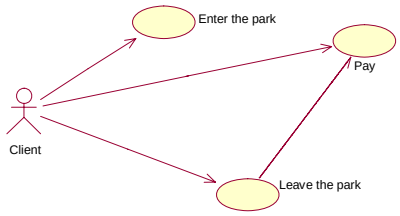

Fig. 2. Domain Use Case Diagram.

For each domain use case described in Figure 2, we fill a template with detailed information. The use cases "Enter the park" and "Leave the park" are described in the respective templates of figures 3 and 4. Note that we associate the usual non-functional requirements (NFRs) [Chung (2000)] to the domain use cases, information that will be useful in the future application.

Item	Description
Name	Enter the park
Problem	Model the process associated with a client entering the parking lot with a vehicle.
Description	To use the parking lot the client must obtain a paying ticket from a machine after

	pressing a button. Then the vehicle may enter the parking lot and park in an available space. The system must control if the park is full.	
Requirements	Functional	F1. When entering the park the client presses a button and gets a paying ticket; F2. When entering the park free available space is checked; F3. If the entry machine is broken a warning message is displayed and a message is sent to a supervisor.
	Non-functional	NF1. Response Time – The check-in machine must hand out a paying ticket and open the gate timely; NF2. Safety – After the vehicle passes through the gate it must close without hitting it.
Participants	Client	

Fig. 3. "Enter the park" use case.

Item	**Description**	
Name	Leave the park	
Problem	Model the process associated with a client leaving the parking lot with a vehicle.	
Description	To leave the parking lot the client must insert a valid paid ticket into an exit machine. Then the vehicle may leave the parking lot. The system must update the park available places.	
Requirements	Functional	F1. When leaving the park the client insert a valid paid ticket; F2. When leaving the park its available places are updated; F3. If the exit machine is broken an warning message is displayed and an alert message is sent to a supervisor.
	Non-functional	NF1. Response Time – The exit machine must validate a paid ticket and open the gate timely; NF2. Safety – After the vehicle passes through the barrier, this must close without hitting it.
Participants	Client	

Fig. 4. "Leave the park" use case.

The domain glossary consists of a list of the common terms of the domain and their respective descriptions. In our example, these terms are, for example, Vehicle, Ticket, Paying Machine, Check-in Machine, Check-out machine, Client, Park. Having all these terms defined in advance saves a lot of effort when starting to develop a new application for the same domain.

The domain analysis artifacts can be even more reusable if domain aspects are identified. In our example, by looking at the requirements we observe that some of them that are systematically repeated. For example, the requirement F3 of both "Enter the park" and "Leave the park" use cases are very similar. The only difference is the kind of machine. A comparable requirement would be discovered for the "Pay" use case that considers the case of the paying machine being broken. We can extract these requirements from both use cases and create an aspectual domain use case that is included by the base domain use cases. We can call this aspectual domain use case as "Broken Machine". The respective template is shown in Figure 5.

Aspectual Domain UC	Description
Name	Broken Machine
Problem	Model the domain aspect associated with a client entering, leaving or paying the parking lot where either machine is broken.
Description	The client cannot enter, leave or pay the parking lot if the respective machine is broken. A supervisor must be notified to solve the problem.
Requirements	A1. If the entry, exit or pay machine is broken a warning message is displayed and an alert message is sent to a supervisor;
Participants	Client

Fig. 5. "Broken Machine" aspect domain UC

The same applies to the non-functional requirements NF1 and NF2, not shown here.

3.2. Reusing & Adapting DA artifacts

The domain use cases information is generic to the problem domain. In the first meetings with the project stakeholders, generic domain functionalities are discussed but more importantly specific system and problem functionalities are identified. In the chosen example besides identifying generic domain functionalities like "Enter the park", "Leave the park" and "Pay", specific ones are identified, defined to the specific problem, such as "Get Card" and "Credit the Card" (Figure 6).

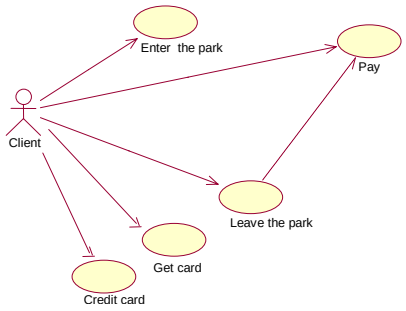

Fig. 6. Application Use Case Diagram

As in the case of the use case domain diagram, the functionality description templates are adapted to the defined specific functionalities. The new particular concepts identified in the specific functionalities definition are now listed in the glossary document. For example, we add the terms Card and PIN and complement the definitions of the entry and exit machines to allow the use of a pre-paid card. After the definition of the system domain as well as of the involved terms and concepts, and the corresponding client's approval, the modeling phases start. The modeling activity starts based on these two documents, detailing them as to let all the participants interpret and understand the processes identified.

3.3. Modeling

In this approach, modeling is carried out by scenario identification, use case refinement, aspect identification, aspect scenario composition, scenario refinement.

Scenario Identification. The starting point to scenario identification is the use case diagram and respective use case templates. These are used to build a kind of sequence diagram that gives us an overview of the dependencies between use cases through interlinked scenarios. We call it scenario identification diagram. This detailed definition is incrementally defined, starting with one or more diagrams (based on the system complexity) that refines the domain use case diagram, which are then detailed in a scenario table to contains all the possibilities of system scenarios.

The visual and graphical nature of the diagram adheres to the requirements of simplicity on communicating and understanding of the initially defined processes keeping a link with the starting point – the domain use case diagram – through use cases. Also, it gives more information than the usual UC diagram, by showing the relations between them. This diagram is mapped into a table with all the variations of the system's functionalities, where common and specific parts of information flows associated with each use case are identified. The analysis of flows into the use cases results in the identification of common and variable parts. For example, in the "Enter the park" use case, the possibility of the park having available spaces or the machine being broken occur in both situations: when the client presses the button or try to use a card and PIN. Nevertheless, after inserting the card, the client shall only enter the park if he successfully enters the PIN. Applying this technique to the parking lot system example, the diagram in Figure 7 is created. Based on this scenario description diagram, possible variations for each scenario are specified and recorded in the scenario template in Table 1.

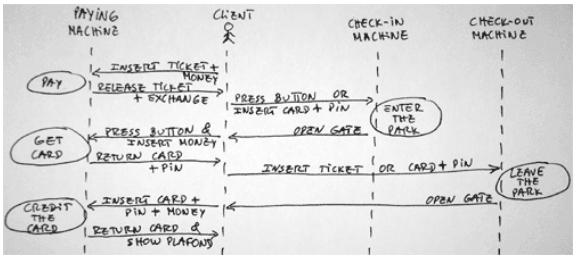

Fig. 7. Partial scenario identification diagram

Table 1. Scenario table

Scenario ID	Scenario	
	Common Part	Specific Part
UF1-S1	Enter the park	Park has free space, driver gets ticket
UF1-S2	Enter the park	Park has no free space
UF1-S3	Enter the park, driver inserts card and PIN	Correct PIN, driver gets in
UF1-S4	Enter the park, driver inserts card and PIN	Incorrect PIN
UF1-S5	Enter the park	Park has free space, machine is broken
UF2-S1	Leave the park, driver inserts ticket	Paid ticket
UF2-S2	Leave the park, driver inserts ticket	Unpaid ticket
UF2-S3	Leave the park, driver has no ticket	
UF2-S4	Leave the park	Pay and exiting grace period exceeded
UF2-S5	Leave the park, driver inserts card and PIN	Correct PIN, driver leaves
UF2-S6	Leave the park, driver inserts card and PIN	Card plafond exceeded
UF2-S7	Leave the park, driver inserts ticket	Machine is broken
UF2-S8	Leave the park, driver inserts ticket	Unreadable ticket
UF2-S9	Leave the park, driver inserts card and PIN	Incorrect PIN
UF3-S1	Pay, driver inserts ticket	Full amount paid
UF3-S2	Pay, driver inserts ticket	Unreadable ticket
UF3-S3	Pay, driver inserts ticket	Machine is broken
UF4-S1	Credit the card	Correct PIN, plafond updated
UF4-S2	Credit the card	Machine is broken
UF5-S1	Get card	Minimum amount inserted, plafond updated
UF5-S2	Get card	Machine is broken

UC refinement and aspect identification. Based on the scenario table described in the previous section we can refine the use case diagram to visualize the main actions associated with each use case and, essentially, identify the aspects. Obviously, some of them are identified when reusing and adapting the domain use cases and domain aspects.

Thus, each use case is detailed through a set of cohesive procedures that can be represented by “sub” use cases that are included by the main use case in the diagram. Figure 8 shows the use case refinement for the parking lot system. For example, the use case “Enter the park” is detailed by the included use cases “Validate PIN”, “Validate Free Space”, “Broken Machine” and “Get Ticket”. Some scenarios crosscut other scenarios, called aspectual scenarios. Figure 8 shows some use cases crosscutting other use cases. For instance, “Validate PIN” crosscuts “Enter the Park” and “Leave the Park”; “Machine is Broken” is part of “Enter the Park”, “Pay”, “Leave the Park”, “Get Card” and “Credit the Card”; “Validate Ticket” is used in “Pay” and “Leave the Park”. This use case repetition identification allows us to obtain an improved system’s modularization, facilitating the propagation of requirements change throughout the project lifecycle.

If this step is omitted or incorrectly defined, changes will be proportionally costly to the later stages when they are detected. Therefore, an early system aspect identification

allows a lighter impact of system changes and a closer alignment (without significant impact) with one of the Agile Manifesto principles – accept changes, even late ones.

Following identification of crosscutting or aspectual use cases, we can re-organize the use case scenarios table by splitting it into two tables. In the example, Table 2 lists the non crosscutting use cases scenarios; and Table 3 depicts the aspectual ones.

Aspect Scenario Composition. Scenarios may be used for existing requirements validation, as the basis for system design and test case definition [Whittle and Schumann (2000)]. As stated in our preliminary work [Ribeiro and Araújo (2005)], composition rules are defined for each UC, by composing aspectual and non-aspectual scenarios. The separation of scenarios, aspects and composition rules facilitates modularization and system evolution. The impact of change is reduced where many changes are restricted to aspects or composition rules.

Fig. 8. Use case refinement

Table 2. Scenarios.

Scenario ID	Scenario	
	Common Part	Specific Part
UF1-S1	Enter the park	Park has free space, driver gets ticket
UF1-S2	Enter the park	Park has no free space
UF1-S3	Enter the park, driver inserts card and PIN	Correct PIN, driver gets in
UF2-S1	Leave the park, driver inserts ticket	Paid ticket
UF2-S2	Leave the park, driver inserts ticket	Unpaid ticket
UF2-S3	Leave the park, driver inserts ticket	

UF2-S4	Leave the park	Pay and exiting grace period exceeded
UF2-S5	Leave the park, driver inserts card and PIN	Correct PIN, driver leaves
UF2-S6	Leave the park, driver inserts card and PIN	Card plafond exceeded
UF3-S1	Pay, driver inserts ticket	Full amount paid
UF4-S1	Credit the card	Correct PIN, plafond updated
UF5-S1	Get card	Minimum amount inserted, plafond updated

Table 3. Aspectual scenarios.

ID Aspect	Aspect
A1	Validate PIN
A2	Broken Machine
A3	Validate Ticket
A4	Update Plafond

Composition rules are defined according to previously identified scenarios. For instance, in the parking lot example, by composing the aspect A2 with the scenario UF1-S1, the following rule is defined:

```
Compose A2 with UF1-S1 where A2 OR UF1-S1
```

This means that entering the park depends on free space availability and the Check-in Machine not being broken. For the aspect A3 and the UF2-S1 scenario, a composition rule is specified using conjunction:

```
Compose A3 with UF2-S1 where A3 AND UF2-S1
```

That is, the client leaves the park after inserting a paid and validated ticket. A more complex composition rule may be created by composing two aspects with a scenario:

```
Compose A2 with A3, UF2-S1 where A2 OR (A3 AND UF2-S1)
```

In this case, the client leaves the park after inserting a paid and validated ticket, or not if the Check-out Machine is broken.

Scenario Refinement. Scenarios are refined by creating Interaction Pattern Specifications (IPSs) for aspects and Sequence Diagrams for the remaining scenarios. IPSs are defined in [France *et al*. (2004)] and describe a pattern of interactions between its participants. It consists of a number of lifeline roles and message roles which are specializations of the UML metaclasses Lifeline and Message respectively. Each lifeline role is associated with a classifier role, a specialization of a UML classifier. By instantiating IPS roles IPSs are converted into sequence diagrams. A sequence diagram conforms to an IPS instantiation if the relative order of messages and participants in the IPS are preserved in the sequence diagram, even if additional modeling artifacts, not part of the IPS, are added to the sequence diagram.

IPS definition is hereby extended (as in [Whittle and Araújo (2004)]) by allowing concrete modeling elements inclusion (apart from roles). In the figure 9 example, both

Client and Supervisor lifelines are concrete elements. This allows greater flexibility, making the instantiation process more agile as less IPS instantiations are necessary.

Applying this technique to the aspect A2, Broken Machine, the IPS on figure 9 is specified.

Non-aspectual scenarios are described through UML sequence diagrams. Separate IPS and sequence diagram modeling promotes improved separation of concerns and consequently improved evolution and modifiability. Error correction and change costs are narrowed as encapsulation reduces the change impact and consequently costs, even overdue ones.

The composition process starts with the selection of aspectual scenarios to be composed with the non-aspectual scenarios crosscut by them. Then the instantiation of IPS roles takes place. Finally the instantiated aspectual scenario is composed with the non-aspectual scenario by means of an operator. This is made easier by the previous step, where the aspect scenario composition is defined.

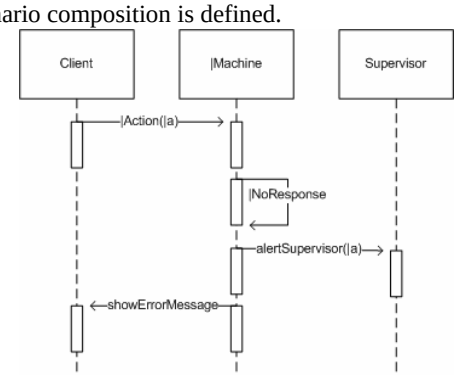

Fig. 9. IPS for the Broken Machine aspect

Instantiation. Next the IPS roles are instantiated to the specific elements in the sequence diagrams. Table 4 shows the instantiations of the A2 roles that will be composed with the scenario UF2-S1.

Table 4. Instantiation of roles to elements

Role	Instantiation
|Machine	Check-out Machine
|Action	InsertTicket
|a	B
|NoResponse	Timeout

Being this instantiation process manually defined, it must be made as agile as possible. Because there's a high probability of the same instantiations occurring in different problems of the same domain, the workload associated with the instantiation process may be reduced by instantiation reuse in a domain. This reuse is enhanced by means of a repository creation (in a database, for example). Despite instantiation reuse

not being taken care of in [Whittle and Araújo (2004)], in the parking lot example, the same instantiations occurs several times in the same domain, as is the case of the instantiation |Action to InsertTicket that occurs in the Paying Machine and in the Check-out Machine. The instantiated IPS becomes a sequence diagram that must be composed to the base scenario described next.

Composition of refined scenarios. Before composition, it is necessary to identify the composition operator to use, which was previously defined in the scenario composition step. This operator must be defined in a case by case basis, according to the current aspectual and non-aspectual scenarios. In the example, we either leave the parking lot after inserting a paid ticket or we cannot do that because of the fact that the Check-out Machine is broken; thus the suitable operator is OR – identified previously.

The result of the composition is a new sequence diagram, presented in figure 10. This diagram combines the two sequence diagrams in order to obtain features from both of them and at the same time conform with the definitions of the original aspect. The OR operator is mapped naturally to the alt fragment used in UML sequence diagrams.

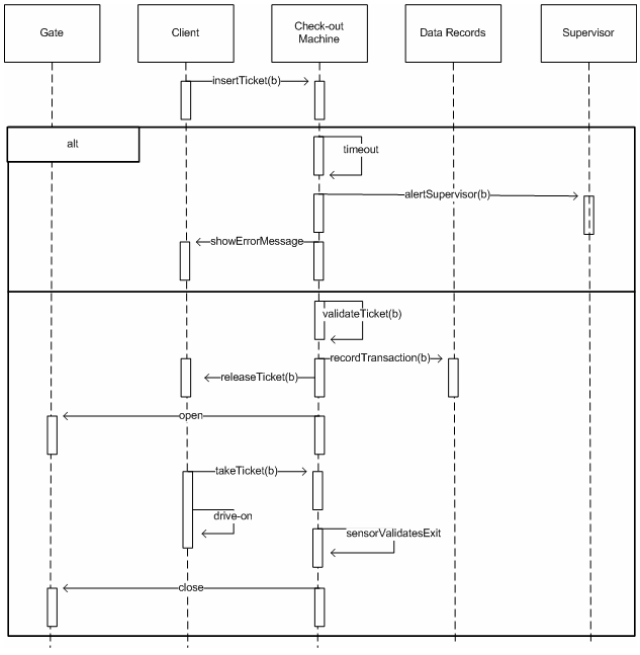

Fig. 10. Composed sequence diagram for the “Leave the park” scenario

3.4. Testing

After the IPS and sequence diagram composition, executable state machines are generated automatically using existing approaches such as the Synthesis approach [Whittle and Schumann (2000)]. First, each sequence diagram is converted into a set of state machines, one for each object involved in the interaction. Next, the individual state machines derived for each object (from different sequence diagrams) are merged into a single state machine for that object. In the first step, an individual sequence diagram is translated into a collection of finite state machines (FSMs). Messages directed towards a particular object are considered events in the FSM for that object. Messages directed away from an object are considered actions. The synthesis algorithm starts by generating an initial state for each FSM. It then traverses the sequence of messages. Messages have a unique sender object and a unique receiver object. For each message, a transition is added to the FSM for the receiver of the message where the transition is labeled with an event having the same name as the message. Similarly, a transition is added to the FSM for the sender with an action defined where the action is to send the message.

Once finite state machines have been created for the individual sequence diagrams, the finite state machines generated from different sequence diagrams for a particular object are merged together. Merging state machines derived from different sequence diagrams is based upon identifying similar states in the FSMs. Once state machines are created they are ready for execution in the test phase. The generated state machine for the exit lot machine is shown in figure 11.

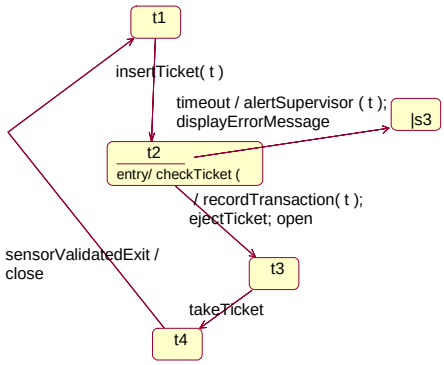

Fig. 11. Composed FSM for the Check-out machine.

3.5. Management

Some management issues must be discussed here, which are common to most of the agile approaches. Firstly, in terms of duration of the iteration, our approach opted for iterations that last between 2 and 4 weeks, as the target kind of project here is small and medium size.

Secondly, the main members of AspOrAS development team are the programmers, but, of course, the client plays a very important role, as the quality of the system depends on his feedback, and also the leader of the project, who coordinates the whole development process.

In AspOrAS maintenance considers constant restructuring. This is achieved through the use of a simple and efficient design, i.e. the simpler the design is the easier maintenance will be conducted.

Stimulus to communication should be always encouraged and the project leader has to steer this among the team members be a programmer or a client. This will determine shared responsibility by all the project members.

4. Case Study

The approach has been applied to a real case study, a system to send and receive SMS messages developed by a Portuguese telecommunications company (ONI) where the first author works. The details of the case study can be found in [Ribeiro (2007)]. This case study helped with evaluation of the approach. We will highlight the main issues involved.

4.1. Domain Analysis

Normally, reusable artifacts are identified after an application is developed. This is true, but some model elements can be easily classified as reusable. Of course the amount of reusable artifacts is limited, but useful in their construction. In the case study, we decided that it was reasonable to have at least a domain use case “Send SMS”. As new applications are developed more domain use cases are added to the domain models. Although we did not use it in the case study as it is a simple one, we believe that, in more complex systems, use case grouping into packages may be used to deal with complexity. This grouping eases information organization and the interaction with external entities, and, more importantly, creates a baseline for the system’s architecture. Figure 12 shows the domain use case diagram for the proposed example. The specification of the use case “Send SMS” is shown by in the template of figure 13. The domain glossary is specified through an alphabetically ordered list of terms to be reused. The glossary for the case study is presented in figure 14.

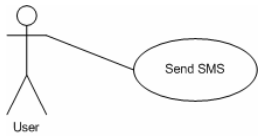

Fig. 12. Domain Use Case Diagram

Item		Description
Name		Send SMS
Problem		Model the process associated with sending a SMS.
Description		Users may use the SMS System by entering an end user phone number, a text message to send and pressing the "send" button.
Requirements	Functional	F1. Users may create a SMS to send by pressing the "new sms" button; F2. Before sending a SMS, users must fill-in the phone number to send the SMS to; F3. Users may fill-in a character number limited message to send to the recipient of the SMS; F4. After being created, the SMS is sent by pressing the "send" button;
	Non-functional	NF1. Availability – The SMS System must be available at all times to fill-in messages;
Participants		User

Fig. 13. "Send SMS" domain use case

Term	Definition
Phone Number	Telephone number of the SMS recipient.
Send date	Hour and date the SMS is sent.
SMS	Short for *Short Message Service*. It's a limited character text block, sent in a specified date and time through a telephone to another telephone specified by a telephone number.
SMS Message	Character limited text sent in an SMS message, in a pre-determined date to a phone number.
User	Person that using the SMS System creates and sends a SMS.

Fig. 14. Domain glossary for the SMS system

4.2. Reusing & Adapting the Domain Analysis

In the chosen example besides identifying the generic domain functionality "Send Free SMS", specific ones are identified, defined and adapted (from generic) to the problem specifics, like "Send Template Formated SMS", "Create Template", "Edit Template" and "Delete Template". Note that in the case study, since there was only one domain use case, it did not make sense to define any aspectual domain use case. Next, the actual application modeling phase begins having as a starting point these two documents.

4.3. Modeling

Scenario Identification. Scenarios are identified using a scenario identification diagram like the one in figure 7 (not shown here for space reasons). Based on this diagram, we build a scenario table shown in table 5.

Table 5. Scenario table for the SMS system

Scenario ID	Scenario	
	Common Part	Specific Part
UF1-S1	Send Free SMS	Phone number filled, send date specified
UF1-S2	Send Free SMS	Phone number filled, send date not specified
UF1-S3	Send Free SMS	Phone number not filled
UF2-S1	Send Template Formatted SMS, User chooses Template	Phone number filled, non-existing variables, send date specified
UF2-S2	Send Template Formatted SMS, User chooses Template	Phone number filled, non-existing variables, send date not specified
UF2-S3	Send Template Formatted SMS, User chooses Template	Phone number filled, filled existing variables, send date specified
UF2-S4	Send Template Formatted SMS, User chooses Template	Phone number filled, filled existing variables, send date not specified
UF2-S5	Send Template Formatted SMS, User chooses Template	Phone number not filled
UF2-S6	Send Template Formatted SMS, User chooses Template	Phone number filled, existing variables not filled, send date specified
UF2-S7	Send Template Formatted SMS, User chooses Template	Phone number filled, existing variables not filled, send date not specified
UF3-S1	Create Template	Minimum identification fields filled, variables non-identified and non-defined
UF3-S2	Create Template	Minimum identification fields not filled
UF3-S3	Create Template	Minimum identification fields filled, variables identified and defined
UF4-S1	Edit Template	Minimum identification fields filled, variables non-identified and non-defined
UF4-S2	Edit Template	Minimum identification fields not filled
UF4-S3	Edit Template	Minimum identification fields filled, variables identified and defined
UF5-S1	Delete Template	Confirmation message accepted
UF5-S2	Delete Template	Confirmation message non-accepted

UC refinement and aspect identification. To fasten the aspect identification we refine the use case diagram to reflect the several scenarios for each use case shown in the

scenario table. We agreed that it is not necessary to use both techniques, so it is up to the development team to choose the most appropriate one. Figure 15 shows the use case refinement for the SMS system case study. Figure 15 shows some aspectual use cases, easily identified. For instance, “Check phone number” crosscuts “Send Free SMS” and “Send Template Formatted SMS”; “Check minimum identification fields filled” is part of both “Create Template” and “Edit Template”. Tables 6 and 7 shows the refactored scenario table into aspectual and non-aspectual scenarios.

Aspect Scenario Composition. Composition rules are defined according to previously identified scenarios. For instance, in the SMS System example, by composing the A1 aspect with the UF1-S1 scenario, the following rule is defined:

```
Compose A1 with UF1-S1 where A1 IN UF1-S1
```

This means that sending a Free SMS depends on the current user role. In the case of the scenario UF3 the composition is made with three aspects:

```
Compose A5, A6, A7 with UF3 where A5, A6, A7 IN UF3
```

That is, the template is created and saved only if the minimum definition fields are filled and variables are correctly identified and defined. We assume the order which the aspects are composed is the same as the order they appear in the rule. A more complex rule may be created by reusing compositions. For instance, if we have:

```
LET UF1-S1’ = Compose A1 with UF1-S1 where A1 IN UF1-S1
```

We can compose UF1-S1’ with A2:

```
Compose A2 with UF1-S1’ where A2 IN UF1-S1’
```

Fig. 15. Use case refinement

Table 6. Refactored scenarios for SMS system

Scenario ID	Scenario	
	Common Part	Specific Part
UF1-S1	Send Free SMS	Phone number filled, send date specified
UF1-S2	Send Free SMS	Phone number filled, send date not specified
UF2-S1	Send Template Formatted SMS, User chooses Template	Phone number filled, non-existing variables, send date specified
UF2-S2	Send Template Formatted SMS, User chooses Template	Phone number filled, non-existing variables, send date not specified
UF2-S3	Send Template Formatted SMS, User chooses Template	Phone number filled, filled existing variables, send date specified
UF2-S4	Send Template Formatted SMS, User chooses Template	Phone number filled, filled existing variables, send date not specified
UF2-S6	Send Template Formatted SMS, User chooses Template	Phone number filled, existing variables not filled, send date specified
UF2-S7	Send Template Formatted SMS, User chooses Template	Phone number filled, existing variables not filled, send date not specified
UF3	Create Template	

UF4	Edit Template	
UF5-S1	Delete Template	Confirmation message accepted
UF5-S2	Delete Template	Confirmation message non-accepted

Table 7. Aspectual scenarios for SMS system

ID Aspect	Aspect
A1	Check user role
A2	Check phone number filled
A3	Verify send date
A4	Store SMS for emission
A5	Check minimum identification fields filled
A6	Check identification and definition of variables
A7	Save template

Scenario Refinement. Here we refine aspectual scenarios into IPSs and and non-aspectual scenarios (or base scenarios) into Sequence Diagrams. The aspectual scenario A1 is shown in on figure 16.

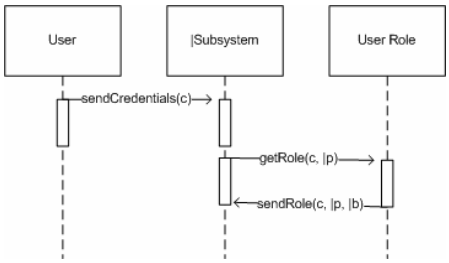

Fig. 16. IPS for the aspect “Check user role”

Instantiation. The IPS roles are instantiated to the specific elements in the sequence diagrams. For the aspectual scenario A1 and the base scenario UF1-S1 the instantiations are presented in table 8.

Table 8. Instantiation of roles to elements

Role	Instantiation
|Subsystem	Free SMS
|p	SMSFreeSender
|b	True

Composition of refined scenarios. For example, consider the aspectual scenario Check user role (A1) and the base scenario “Send Free SMS (UF1-S1), Phone number filled, send date specified”. In this case, the task of checking the user role must be executed in the process of sending free SMS, with phone number and send date specified, therefore, the operator is IN. But that was already defined previously. The resulting composed sequence diagram is presented in figure 17. The automatic generation of executable state machines was not realized as the tool was not ready to be used in an industrial

environment as some technical adjustments were not finished by the time of the application of the case study.

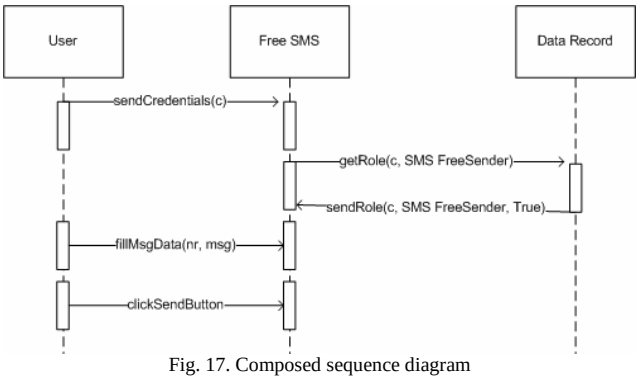

Fig. 17. Composed sequence diagram

4.4. Discussion

The development team of ONI composed by three people already had experience with agile development, but neither with domain analysis nor with aspects. They observed that agility was very much gained with:

- the potential of the reuse of domain analysis models, including the reuse of domain aspects, which can shorten the initial stages of development. They agreed that this is important even for small systems;
- the use of the scenarios identification diagram, which, by complementing the use case diagram, gives a fundamental UC interdependency perspective that facilitates the scenario identification;
- an improved separation of concerns achieved by the aspectual scenarios at the application level, where, when new requirements are introduced the impact of change is controlled and reduced;
- composing scenarios and aspectual scenarios much earlier, not having to wait to specify sequence diagrams or FSM to decide which composition operators to use. You may even decide that not all scenarios need to be refined, just their pre or post-conditions;
- last but not least, they believe that approach can be fairly adapted to software product lines development, by enriching the domain analysis with feature models (which is part of our current research).

However, it was consensual that agility was compromised due to lack of tool support, a conclusion that came without surprise to us. Also, they perceived that composition could be used for the identification of requirements ambiguities errors or omissions. These are objects of our future work.

4.5. Comparative analysis

Finally, we compared AspOrAS with FDD, XP and Crystal according to several criteria [19]. Table 9 presents a comparison between our approach and other agile approaches.

Table 9. Comparing AspOrAS with other agile approaches

Criteria	**FDD**	**XP**	**Crystal**	**Asporas**
Duration of iterations	2 weeks	1 - 4 weeks	2 weeks ; 2;3 months	2 - 4 weeks
Team size	6	10	40	3
Integration of client in the team	N	Y	Y	Y
Main members	Develop. Manager	Programmer	Analyst prog. senior	Programmer
Constant production rythm (CPR) vs Milestones (M)	CPR	CPR	M	CPR
Incremental Restructuring	N	Y	N	Y
Constant Restructuring for maintainance	Y	Y	N	Y
Stimulate Communication	Y	Y	Y	Y
Shared responsibility	N	Y	Y	Y
Well-defined and modelled functionalities	Y	N	N	Y
NFR specification	N	N	N	Y
Crosscutting requirements spec.	N	N	N	Y
Test driven process	N	Y	Y	Y
Validation with the user	Y	Y	Y	Y

Firstly, duration of iterations varies between 2 and 4 weeks. AspOrAS is situated between FDD e XP. Therefore, AspOrAS is more appropriate to small or medium projects.

In AspOrAS team size is limited to three programmers which is comparatively the smallest size, in relation to the other approaches. However, multiple teams can be formed for bigger projects. Extra effort is needed to coordinate them though.

Also, in AspOrAS the integration of the client in the team is promoted as it happens in XP and Crystal. This is fundamental for the success of the project as facilitates expression of the real needs of the client. But if this not possible, AspOrAS allows periodic meetings with the client, to guarantee that the system is being developed according to his expectations.

The main members of AspOrAS development team are the programmers as it happens with XP. This makes the programmers more responsible concerning the project. The result is a better quality of the software.

In AspOrAS, the constant rhythm of production is more important than following the milestones, similarly to FDD and XP. Thus the pressure associated with the milestone is avoided and also the relaxing that comes afterwards.

In AspOrAS, like in XP, successive restructuring is more valued than specific restructuring iterations. Successive restructuring guarantees a simpler design, better communication among other benefits.

In AspOrAS maintenance is promoted by constant restructuring similar to what happens in FDD and XP. What is subjacent is the use of a simple and efficient design.

Stimulus to communication is a common characteristic to all the approaches and should be always encouraged to help achieving the desired quality of the system. Shared responsibility by all the project members is highlighted by AspOrAS, XP and Crystal and is associated with the programmers, mainly. This is important as in the end it helps the members to make a better use of time, promoting a better team dynamics.

Modelling functionalities using templates or UML models are not explicitly promoted by XP or Crystal Crystal. But this is important as the models work as communication tools to be used together with stakeholders.

Identification and specification of non-functional requirements (NFR) is promoted by our approach, which does not occur with the other approaches. Failing to address them compromises the success or the global quality of the project. In AspOrAS NFR are associated with use cases. Catalogues can be used to help in the elicitation of NFR (e.g. the NFR framework [Chung *et al*. (2000)].

The description of crosscutting requirements is supported explicitly only by AspOrAS. Not addressing this kind of requirements compromises modularity, makes harder the identification of conflicts between requirements, and augments the effort associated with requirements change, due to requirements scattering and tangling problem. The result is slower system evolution. Conflict resolution is not addressed by AspOrAS, but it could adapt the conflict resolution mechanism present in the approach AORE [Rashid et al., 2003].

Test driven support is promoted by our approach and also in XP and Crystal. This guides the programmers to develop more reliable software. Note that automation is essential in this activity to make it more productive.

Validation with the user is common place with all the approaches, being an essential criterion to measure agility.

In summary, what mainly distinguishes our approach from these approaches is the lack of mechanisms to deal with crosscutting requirements and NFRs (which are potentially crosscutting), compromising systems evolution, making this fact our contribution for the agile requirements area

5. Related Work

As mentioned in the previous section, none of the current agile approaches address explicitly aspect modeling as AspOrAS does. In relation to the main aspect-oriented

requirements engineering approaches, they do not address agile modeling explicitly. Moreover, none of them includes domain analysis as part of their processes. But in terms of aspect modeling we have the following. The approaches described in [Araújo *et al*. (2004)] and [Whittle and Araújo (2004)] were integrated into AspOrAS, but they also benefit from it by including the domain analysis, aspect identification and composition definition much earlier.

The approaches described in [Rashid et al. (2003)] and [Moreira et al. (2005)] provide conflict identification and resolution mechanisms. AspOrAS does not have such mechanisms, but it would surely be enriched by some conflict management mechanism that can be adapted from these approaches. The composition language strategies they use are not suitable for analysis models such as interaction models.

The approach in [Jacobson and Ng (2004)] deals with aspects in use case modeling thorough a use case slice technique. Use case slices are a way of maintaining a use case-based decomposition throughout the development lifecycle. For example, in the case of state diagrams, this means that each use case maintains its own state diagram and these state diagrams are composed during late design or implementation to obtain the overall design. However, they do not adequately address how to compose use case slices during design. Their approach is to apply AspectJ-like composition operators [Kiczales *et al*. (1997)], which are not expressive enough.

[Clarke and Walker (2001)] describes composition patterns to deal with crosscutting concerns as patterns at the design level. Pattern binding is used, and sequence and class diagrams illustrate compositions. The compositions, however, are rigid as they concentrate on pattern instantiations. [Baniassad and Clarke (2004)] proposes Theme to provide support for aspect-oriented analysis through Theme/Doc. Analysis is carried out by first identifying a set of actions in the requirements documentation that are, in turn, used to identify crosscutting behaviors. An action is a potential theme, which is a collection of structures and behaviors that represent one feature. A tool is provided to create graphs of the relationships between concerns and the requirements that mentioned those concerns. However, the approach does not provide explicit composition mechanisms. Also, scalability problems related to theme identification persists.

6. Conclusions and Future Work

The proposed approach extends the initial work in [Araújo and Ribeiro (2005)] where only aspect scenario identification and composition were provided. In this paper a whole approach, ranging from domain analysis and modeling (including scenario refinement and composition) until tests and implementation was provided. The domain analysis and the reuse of its models are very useful for the stakeholders and development team system to seek for requirements, smoothing the progress of the start up process. Aspects identification (including in the domain analysis), modularization and composition have the advantage of preparing the systems to be more adaptable and evolvable. Future work will concentrate on the automation of the approach to make the agility potential to be

fully achieved. Moreover, by analyzing composition rules we may discover ambiguities, conflicts and omissions. This will be also a goal for further research.

References

Ambler, S. (2002). Agile Modeling, John Wiley.

Araújo, J., Ribeiro, J. C., "Towards an Aspect-Oriented Agile Requirements Approach", 8th International Workshop on Principles of Software Evolution (IWPSE 2005), Lisbon, Portugal, 2005.

Araújo, J., Whittle, J., Kim, D. (2004). "Modeling and Composing Scenario-Based Requirements with Aspects", 12th IEEE Int'l Requirements Eng. Conf. (RE 04), IEEE CS Press.

Baniassad, E., Clarke, S. (2004). "Theme: An approach for aspect-oriented analysis and design", International Conference on Software Engineering (ICSE), IEEE Press, Edinburg, Scotland.

Baniassad, E., Clements, P. C. , Araújo, J., Moreira, A., Rashid, A., Tekinerdogan, B. (2006). "Discovering Early Aspects", IEEE Software.

Beck, K. (2000). Extreme Programming Explained: Embrace Change, Addison Wesley,.

Clarke, S., Walker, R. (2001). "Composition Patterns: An Approach to Designing Reusable Aspects", International Conference on Software Engineering (ICSE).

Cockburn, A., (2004). Crystal Clear: A Human-Powered Methodology for Small Teams, Addison-Wesley.

Chung, L., Nixon, B., Yu, E. & Mylopoulos, J. (2000). Non-Functional Requirements in Software Engineering, Kluwer Academic Publishers.

France, R., Kim, D., Ghosh, S. & Song, E. (2004) "A UML-Based Pattern Specification Technique", IEEE Transactions on Software Engineering, Volume 30(3).

Jacobson, I., Ng, P.-W. (2004). Aspect Oriented Software Development with Use Cases. Addison-Wesley Professional,.

Kiczales, G., Irwin, J., Lamping, J. Loingtier, M., Lopes, C. V., Maeda, C. & Mendhekar, A. (1997). "Aspect-Oriented Programming", ECOOP 1997. Springer LNCS 1241, pp 220-242.

OMG (2005). UML Specification, version 2.0, August 2005, http://www.omg.org

Moreira, A., Rashid, A., Araújo, J., "Multi-Dimensional Separation of Concerns in Requirements Engineering,", 13th IEEE Int'l Requirements Eng. Conf. (RE 05), IEEE CS Press, pp. 285–296, 2005.

Palmer, S. R., Felsing, J. M., (2002). A Practical Guide to Feature-Driven Development, Addison-Wesley.

Rashid, A., Moreira, A., Araújo, J., "Modularization and Composition of Aspectual Requirements", 2nd International Conference on Aspect-Oriented Software Development (AOSD), Boston, EUA, ACM Press, pp. 11-20, March, 2003.

Ribeiro, J. C. (1997). "An aspect-oriented requirements agile approach based on scenarios", Master's Thesis, New University of Lisbon, Portugal, 2007.

Schwaber, K., Beedle, M. (2002). Scrum: Agile Software Development, Prentice-Hall, 2002.

Tarr, P., Ossher, H., Harrison, W. H. and Sutton, S. M. "N Degrees of Separation: Multi-Dimensional Separation of Concerns", ICSE 1999, ACM, pp. 107-119.

Whittle, J., Araújo, J. (2004) "Scenario Modeling with Aspects", IEE Proceedings – Software, Special Issue on Early Aspects. Editors: Rashid, A., Moreira, A., Teknierdogan, B., IEE Proceedings Software, **151**, 04, pp. 157-172.

Whittle, J., Schumann, J., (2000). "Generating Statechart Designs from Scenarios", ICSE 2000, Limerick, Ireland, IEEE CS Press, pp.314-323, June 2000.