On the Derivation of Class Diagrams from Use Cases and Logical Software Architectures

Maribel Yasmina Santos, Ricardo J. Machado
Dept. de Sistemas de Informação, Centro de Investigação Algoritmi
Universidade do Minho
Guimarães, Portugal
{maribel, rmac}@dsi.uminho.pt

Abstract—**The transformation of user requirements into system requirements models can be achieved using the 4-Step Rule Set (4SRS) method that transforms UML use case diagrams into system-level object diagrams. These diagrams represent the logical architecture of the system, integrating the system-level entities, their responsibilities and the relationships among them. The logical architecture captures the system functional requirements and its non-functional intentionalities. Although contributing to the formalization of the design of software architectures, the 4SRS method needs to be extended in order to support the design of the database subsystems that may be considered pertinent within the specified logical architecture. This paper presents the extension of the 4SRS method to support the construction of the class diagram that complements the logical architecture, and shows, through the presentation of a demonstration case, the applicability of the proposed approach.**

Keywords-system software requirements; use case diagrams; class diagrams; logical software architectures

I. Introduction

In the development of a software system, the most complex activity is probably the transformation of a requirements specification into an architectural design. The process of designing software architectures is less formalised and often is greatly an intuitive ad-hoc activity, poorly based on engineering principles. The 4-Step Rule Set (4SRS) [2] method employs successive model transformations in order to obtain a logical architecture that satisfies the previously elicited user requirements. It is based on the mapping of UML use case diagrams into UML objects diagrams [2]. The iterative nature of the method and the usage of diagrammatic models help to ensure that the obtained logical architecture reflects the user requirements. After the generation of the first logical architecture of the system, the design of class diagrams is desired for the specification of the static characteristics of the software to be produced, namely to address the design of the database subsystems that may be considered pertinent within the specified logical architecture.

In the approach described in this paper, a derivation technique is used to obtain the class diagram, using not only the use case model but also the logical architecture that results from the application of the 4SRS method. Since the 4SRS method supports a recursive approach [1] to ensure that the system functional requirements are present in the logical architecture independently of the subsystem we may need to refine, the approach proposed in this paper guarantees that the necessary classes are able to be identified. We propose an extension to the 4SRS method to derive class diagrams from the use case diagrams and from the logical system architecture. We decided not to pursue an empirically validation effort of the technique [3]. Instead, we opted for constructing one demonstration case to illustrate the systematic approach of the extension proposed here for the 4SRS method. This extension contributes for the derivation of class diagrams, which are the most frequently used component among the UML diagrams [4].

This paper is organized as follows. Section 2 introduces some related work. Section 3 gives an overview of the steps that integrate the 4SRS method. Section 4 describes the additional steps added to the 4SRS method to derive class diagrams from the use case model and the software logical architecture. Section 5 presents a demonstration case using the extended 4SRS method. Section 6 discusses the proposed approach and the obtained results. Section 7 concludes with some remarks regarding the work undertaken and with some proposals for future work.

II. Related Work

The functional requirements of a software system can be captured and documented in use cases [3], as happens in the 4SRS method. Although use case-driven approaches are frequently used to identify system classes [3, 5], there is no established technique for the transition from use cases to class diagrams [3]. In these approaches, the development process may lead to missing classes because the use cases are insufficient for deriving all necessary classes.

The identification of classes from the documented user requirements is one of the most important and difficult tasks during the analysis and the design of object-oriented systems [6]. CASE systems support the generation of class diagrams through natural language processing of the documented requirements. Giganto and Smith [6] argue that this approach has considerable problems, either in the form of overlooked and/or excess of classes. For these authors, the problem can be originated by the inherent ambiguity in written language and a general lack of conciseness and completeness in requirements specification. Based on this, Giganto and Smith proposed the identification of classes from use cases rather than directly from the specification. Use cases describe specific functionalities of the system and, therefore, the candidate objects involved in those functionalities. In this approach, the authors propose an algorithm that extract use case sentences from requirements, validate functional specifications in each sentence so that classes can be identified, and reuse previously use cases to supply missing functional specifications that may contain participating classes. In this approach, ambiguity problems are mitigated by imposing restrictions on the language that can be used for writing requirements.

The work of Liang [7] also follows a use case-driven process in which classes are identified based on the goals of use cases without descriptions. The proposed approach identifies use case-entity diagrams as a vehicle for deriving classes from use cases. Classes are identified from use cases' goals rather than use case descriptions. Classes are the entities that participate in achieving the goals in the real world. With this approach, the author avoids the identification of too many classes at one time, what usually happens in use case-driven processes.

III. Synopsis of the 4-Step Rule Set Method

The 4SRS method generates logical architectures, representing system requirements, from user requirements models. A complete description of the 4SRS method can be found in [2]. A brief description of the method is next presented.

The 4SRS is organized in four main steps which transform use cases into system-level objects. Step 1 is designated *object creation*. It automatically transforms each use case into three object types, interface, control and data. From this step on, only objects exist as design entities. Use cases continue to be used in the next steps allowing the introduction of the user requirements into the object model.

Step 2 is called *object elimination*. For each object, it must be decided which of the three type of objects must be maintained taking into consideration the whole system and not each use case in particular. These decisions are based on the use cases textual descriptions. This is the most complex step of the 4SRS; that is why it is divided in seven micro-steps.

In micro-step 2i (*use case classification*), use cases are classified as interface, data or control, or any combination of theses, transforming each use case into objects. Micro-step 2ii (*local elimination*) analyses if each object created in step 1 makes sense in the problem domain. This elimination is based on the textual descriptions of the use cases. In micro-step 2iii (*object naming*), objects that were not eliminated from the previous micro-steps must receive proper names that reflect both the use case from which they were originated and the specific role of the objects taking into account their main functionality.

Micro-step 2iv (*object description*) deals with the description of each object resulting from previous micro-step, allowing the inclusion of system requirements into the object model. The descriptions must be based on the original use case descriptions. Basically, this micro-step leads to the system requirements based on user requirements.

The most critical micro-step is micro-step 2v (*object representation*) since it supports the elimination of redundancy on user requirements and the identification of missing requirements. It constitutes a validation step that ensures the semantic coherence of the object model and that discovers anomalies in the use case model.

Micro-step 2vi (*global elimination*) is a fully automatic micro-step that eliminates objects that are represented by other ones. It must assure the generation of a coherent object model, from the system requirements point of view.

The last micro-step 2vii (*object renaming*) renames the objects that were not eliminated in the previous micro-step and that represent additional objects.

In step 3 (*object packing and aggregation*), the remaining objects should give origin to aggregations or packages of semantically consistent objects if there is an advantage in being treated in a unified way. This step supports the construction of a coherent object model adding an additional semantic layer at a higher abstraction layer.

The final step, step 4 (*object association*), supports the introduction of associations in the object model based on the information available in the use case model and generated in micro-step 2i. In the 4SRS terminology, this last version of the object diagram is called raw object diagram.

IV. Extending the 4-Step Rule Set

The logical architecture obtained from the iterative application of the 4SRS method captures the functional and non-functional system requirements. For refining the obtained architecture, the recursive approach of the 4SRS suggests the construction of a new use case diagram that captures the user requirements of the subsystem to be refined. From this new use case diagram a new raw object diagram is obtained. This process is repeated until raw objects diagrams are identified for all the subsystems to be implemented.

Given a raw object diagram for a system or for one of its subsystems (representing the system or subsystem logical architecture), the identification of the corresponding class diagram is not addressed by the current version of the 4SRS method. In this paper, these additional steps are defined and illustrated with a demonstration case. The additional steps are: Step 5 *class creation* and Step 6 *class characterization*.

A. *Step 5 – class creation*

This is a fully automatic step in which a class is created for each one of the objects present in the raw object diagram resulting from step 4. The associations among the classes are also inherited from the associations present in the raw object diagram. After the execution of this step, the classes and the relationships among them are obtained.

B. *Step 6 – class characterization*

The class diagram obtained from step 5 needs to be complemented with the attributes and methods that characterize each class. For this task, the use case diagram of the refined logical architecture, specifying the user requirements, and some intermediary results from previous steps are used. This step is divided into 2 micro-steps: micro-step 6i, *methods creation* and micro-step 6ii *attributes creation*.

In micro-step 6i (*methods creation*), each class resulting from the raw object diagram (step 5) must include a method that implements the use case that originated its creation as a system-level object (step1) and one additional method for each one of the use cases represented by this object. This information is available from micro-step 2v, where the information of the represented objects leads to the corresponding use cases. The name of the methods results from the use cases that originated the enrolled objects.

The definition of the methods is only complete when their parameters are specified. For this purpose, the analysis of the refined use cases textual descriptions is done. Besides the methods that emerge from the objects and respective use cases, additional methods are usually identified in the analysis of the refined use cases textual descriptions. These methods must also be included in the corresponding classes since they also refer to system requirements.

Micro-step 6ii (*attributes creation*) allows the identification of the attributes that must be present in each class. For this task, the textual description associated to each refined use case needs to be analysed in order to identify the data that must persist in the system.

Our approach is based on the linguistic analysis of the requirements documentation (use case textual descriptions), written in a natural language [8]. Attributes correspond to the nouns, and the operations (also known as methods or services) are related to verbs [9]. This strategy must be used with some prudence and requires strong linguistic knowledge, since it is possible to transform a noun into a verb and vice-versa.

V. The Demonstration Case

The usefulness of the new steps added to the 4SRS method will now be discussed making use of a demonstration case. This demonstration case is associated to a mobile application platform for which the service-oriented architecture was obtained applying the 4SRS [10]. In this paper, we give special attention to one of the system services, the AVAccess Service. This service is a refinement of a package obtained in the raw object diagram of the overall USE-ME.GOV platform specification [11].

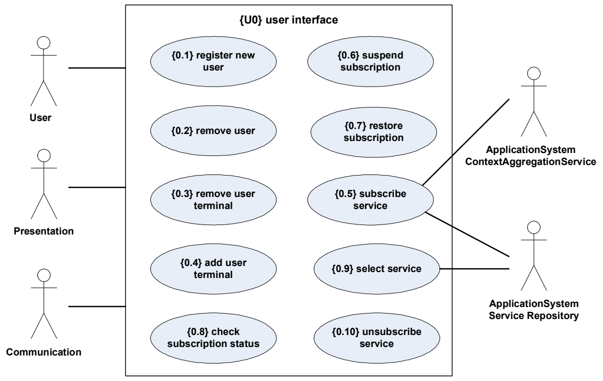
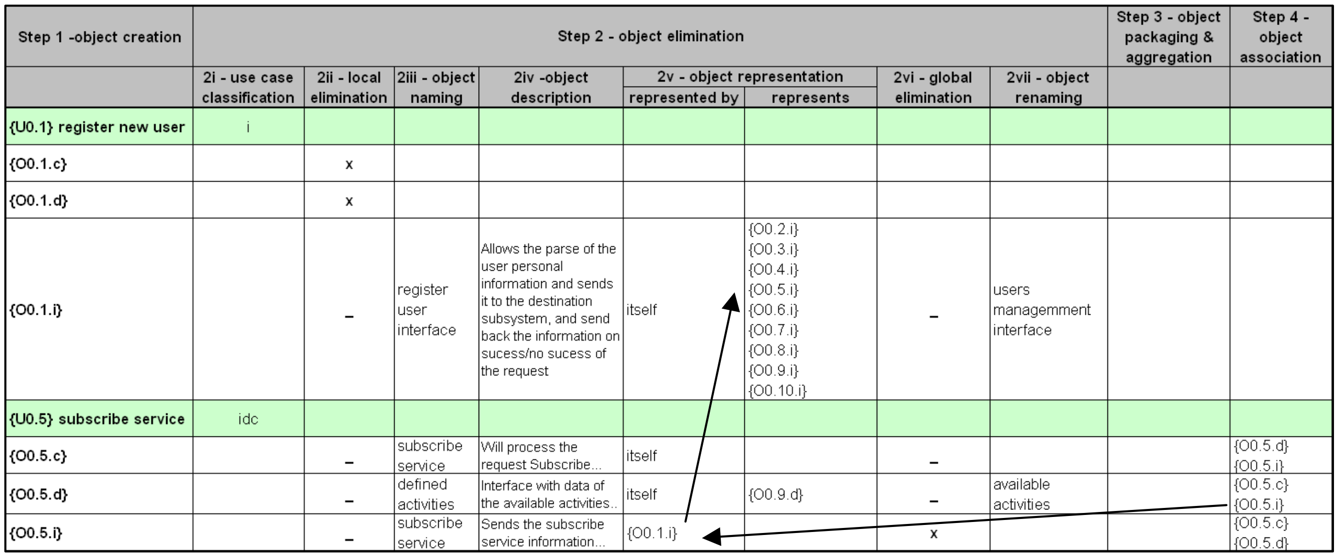

{U0} user interface
{0.1} register new user
{0.6} suspend subscription
{0.2} remove user
{0.7} restore subscription
{0.3} remove user terminal
{0.5} subscribe service
{0.4} add user terminal
{0.9} select service
{0.8} check subscription status
{0.10} unsubscribe service
User
Presentation
Communication
ApplicationSystem ContextAggregationService
ApplicationSystem Service Repository

Figure 1. Use case diagram for the AVAccess Service



Step 1 -object creation	Step 2 - object elimination								Step 3 - object packaging & aggregation	Step 4 - object association
	2i - use case classification	2ii - local elimination	2iii - object naming	2iv -object description	2v - object representation: represented by	2v - object representation: represents	2vi - global elimination	2vii - object renaming		
{U0.1} register new user	i									
{O0.1.c}		x								
{O0.1.d}		x								
{O0.1.i}		–	register user interface	Allows the parse of the user personal information and sends it to the destination subsystem, and send back the information on sucess/no sucess of the request	itself	{O0.2.i} {O0.3.i} {O0.4.i} {O0.5.i} {O0.6.i} {O0.7.i} {O0.8.i} {O0.9.i} {O0.10.i}	–	users managemment interface		
{U0.5} subscribe service	idc									
{O0.5.c}		–	subscribe service	Will process the request Subscribe...	itself		–			{O0.5.d} {O0.5.i}
{O0.5.d}		–	defined activities	Interface with data of the available activities..	itself	{O0.9.d}	–	available activities		{O0.5.c} {O0.5.i}
{O0.5.i}		–	subscribe service	Sends the subscribe service information...	{O0.1.i}		x			{O0.5.c} {O0.5.d}

Figure 2. Table that supports the 4SRS method

The AVAccess service is a point of contact with the mobile platform and redirects the user to the appropriate service. Users usually start the interaction with the mobile system contacting this component. Figure 1 presents the refined use case diagram for the AVAccess service.

The execution of the 4SRS method for this use case diagram is described in detail in [1, 2]. In this paper, it is presented an extract of the table that allows the 4SRS transformation, as this table supports the application of the additional steps of the 4SRS method to this demonstration case. This table is presented in Figure 2.

After the execution of the 4SRS approach until step 4, Figure 3 shows the obtained raw object diagram with the subsystem (AVAccess service) logical architecture. At this stage, it is now possible to apply the two additional steps for the 4SRS method in order to obtain the class diagram that gives a static characterization of the described service.

In step 5 (class creation), each object in the raw object diagram gives origin to one class in the generated class diagram. The name of each class follows the name of the corresponding object. The relationships among the classes correspond also to those existent in the raw object diagram. After the execution of step 5, the first draft of the class diagram is presented in Figure 4.

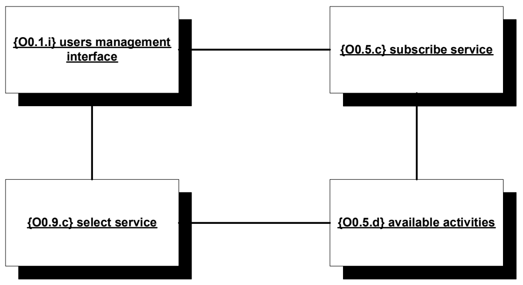

{O0.1.i} users management interface
{O0.5.c} subscribe service
{O0.9.c} select service
{O0.5.d} available activities

Figure 3. Raw object diagram for the AVAccess Service

In the execution of step 6 (class characterization), the process starts by micro-step 6i where the methods of each class are identified.

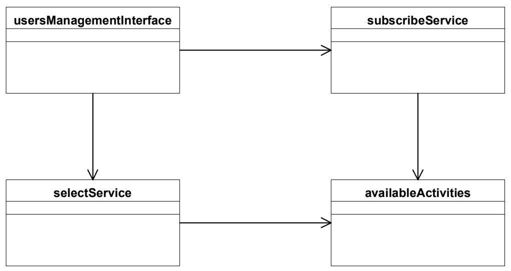


Figure 4. First schema of the class diagram

Analysing the results presented in Figure 2, object {O0.1.i} must be kept by the system and also has the representation of objects {O0.2.i}, {O0.3.i}, {O0.4.i}, {O0.5.i}, {O0.6.i}, {O0.7.i}, {O0.8.i}, {O0.9.i} and {O0.10.i}. Each one of these objects gives origin to one method of the *usersManagementInterface* class. Each method is associated with the use case that originated the corresponding object. For example, object {O0.1.i} is originated from the use case {U0.1} *register new user*, so this will result on the *registerNewUser* method. Following this procedure for the other objects, all remaining methods are identified (Figure 1 includes all corresponding use cases): *removeUser*, *removeUserTerminal*, *addUserTerminal*, *subscribeService*, *suspendSubscription*, *restoreSubscription*, checkSubscriptionStatus, *selectService*, and *unsubscribeService*.

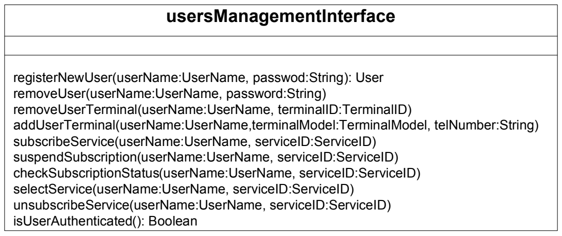
After the identification of the methods, it is necessary to complement their definition through the identification of their parameters. In this task, the refined use cases textual descriptions are analysed. As an example, let us consider the textual description associated with the use case *{U0.1} register new user*:

{U0.1} Register new user: the user provides (through communication subsystem) user personal information to the AVAccess system. Its personal information consists of ***userName, password****, and, optionally, user profile information. The AVAccess service parses user personal information and sends it to subsystem User. The AVAccess system sends back the information on success/no success of this operation. The information sent to the user is formatted by the subsystem Presentation. The system must know terminal model information.*

From this description, the mandatory parameters of the *registerNewUser* method are identified: *username* and *password*. Besides the methods that emerge from the objects, other methods are usually needed in the classes. These methods are mentioned in the use cases descriptions:

{0.2} Remove user: the user provides (through communication subsystem) user username and password information to the AVAccess system. The ***user must be authenticated****. The AVAccess system sends a request to User system to remove the user identified by username and password. The AVAccess system sends back the information on success/no success of this operation. The information sent to the user is formatted by the subsystem Presentation. The system must know terminal model information.*

In this use case textual description, we identify the parameters of the *removeUser* method (which are also the username and the password) and recognize the need for a method that answers to the question if the user is authenticated or not. Using this process, it is possible to identify all the methods that must appear in the *usersManagementInterface* class. This class is presented in Figure 5.


usersManagementInterface

registerNewUser(userName:UserName, passwod:String): User
removeUser(userName:UserName, password:String)
removeUserTerminal(userName:UserName, terminalID:TerminalID)
addUserTerminal(userName:UserName,terminalModel:TerminalModel, telNumber:String)
subscribeService(userName:UserName, serviceID:ServiceID)
suspendSubscription(userName:UserName, serviceID:ServiceID)
checkSubscriptionStatus(userName:UserName, serviceID:ServiceID)
selectService(userName:UserName, serviceID:ServiceID)
unsubscribeService(userName:UserName, serviceID:ServiceID)
isUserAuthenticated(): Boolean

Figure 5. Methods for the usersManagementInterface class

Classes can also integrate attributes. These attributes add persistency to the system, enabling the storage of information in a database system. The identification of the relevant attributes is carried out by analysing the refined use cases textual descriptions. For the class *availableActivities* originated from the object {O0.5.d} and that also represents the object {O0.9.d} (see Figure 2), the relevant attributes were identified by analysing the following two use cases descriptions:

*{U0.5} Subscribe service: the user provides (through communication subsystem) service subscription information to the AVAccess: the user gets (through communication system) activities defined in AVAccess system; The user gets **activityID**, **activityName** and **activityDescription**; The information on activities sent to the user (through communication subsystem) is formatted by the subsystem Presentation; The user provides (through communication subsystem) the activityID to the AVAccess system; The AVAccess system computes the AVService type serviceType that match the chosen activity; The AVAccess system sends back the complete list of AVServices (information on name, description, cost and ServiceID (URI)) registered in the service repository. The user must be authenticated. Service subscription information provided to AVAccess system consists of a list of serviceID. The AVAccess system sends back the information on success/no success of this operation. The information sent to the user (through communication subsystem) is formatted by the subsystem Presentation. The system must know terminal model information.*

*{U0.9} Select service: the user gets (through communication subsystem) AVService handler (URL). The user gets (through communication system) activities defined in AVAccess system. The user gets activityID, activityName and activityDescription; there are three **types of activities**: activities based on free-subscription context-aware services, activities based on subscription-based context-aware services and activities based on subscription non context-aware services. The information on activities sent to the user is formatted by the subsystem Presentation; The user sends (through communication subsystem) activityID to the AVAccess system; The AVAccess system computes the AVService type serviceType that match the selected activity;*

(1) The user selects activity based on free-subscription context-aware services: The AVAccess system sends back a list of AVServices of serviceType (information on name, description, cost and Uri) in accordance to the user context. If the system does not retrieve relevant services the AVAccess gets user selection on context information; the context-aware list of AVServices is filtered by ContextAggregatorService subsystem from the list of AVServices of serviceType registered in service repository.

(2) The user selects activity based on subscription-base context-aware services: The AVAccess system sends back a list of AVServices of that serviceType (information on name, description, cost and Uri) subscribed by the user in accordance to the user context. If the system does not retrieve relevant services the AVAccess gets user selection on context information. The user must be authenticated. The context-aware list of AVServices is filtered by ContextAggregatorService subsystem from the list of AVServices of serviceType subscribed by the user.

(3) The user selects activity based on subscription-base non context-aware services: The AVAccess system sends back a list of AVServices of that serviceType (information on name, description, cost and Uri) subscribed by the user independent on current user context. The user must be authenticated. The list of AVServices is returned from the list of AVServices of serviceType subscribed by the user.

AVService information provided by AVAccess system consists of service serviceUri, service description and service name. The AVAccess system sends back the information on selected service. The information sent to the user (through communication subsystem) is formatted by the subsystem Presentation. The system must know terminal model information.

Through the analysis of the use case {U0.5}, the relevant attributes for the *availableActivites* class are *activityID, activityName* and *activityDescription.* Relevant attributes about the services are also presented in the description and must be included in the corresponding class (in this case, they belong to another subsystem).

From the {U0.9} use case description, it is also identified the *activityType* attribute for the *availableActivities* class. In terms of functionalities, and after the analysis of both use cases, the relevant ones were underlined in the textual descriptions and are now summarized:

- The user gets the activities defined in the AVAccess system (*getActivities():Activity[]*);
- The user provides the *activityID* and gets the complete list of AVServices that match the chosen activity (*getActivityServices(activityID:ActivityID):AVServices[]*);
- The user provides the *activityID* and gets the service type that matches the chosen activity (*getActivityType(activityID:ActivityID):ActivityType*);
- The user selects activities based on the three types of activities, free subscription, subscription-base context-aware services and subscription-base non context-aware services, and gets a list of AVServices (*activitiesSubscribedServices(username:UserName, activityType:ActivityType):AVService[]*).

The *availableActivities* class integrates now attributes and methods, as depicted in Figure 6.

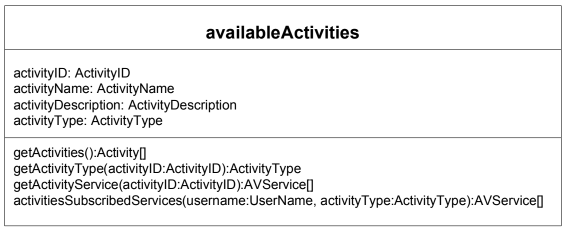

availableActivities
activityID: ActivityID
activityName: ActivityName
activityDescription: ActivityDescription
activityType: ActivityType
getActivities():Activity[]
getActivityType(activityID:ActivityID):ActivityType
getActivityService(activityID:ActivityID):AVService[]
activitiesSubscribedServices(username:UserName, activityType:ActivityType):AVService[]

Figure 6. Attributes and methods for the availableActivities class

For the *subscribeService* class (Figure 7), the analysis of the {U0.5} use case allowed the identification of:

- The user subscribes a service (*subscribeService(serviceID:SerrviceID, username:UserName)*);
- The user provides the *activityID* and gets the complete list of AVServices that match the chosen activity (*getServices(activityID:ActivityID):AVServices[]*);

- The user provides the *activityID* and gets the complete list of service id's of the subscribed services (*getSubscribedServices(activityID:ActivityID, username:UserName):ServiceID[]*).

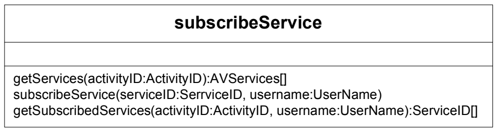

subscribeService
getServices(activityID:ActivityID):AVServices[]
subscribeService(serviceID:ServiceID, username:UserName)
getSubscribedServices(activityID:ActivityID, username:UserName):ServiceID[]

Figure 7. Attributes and methods for the subscribeService class

The last class, *selectService*, is based on the {U0.9} use case. The identified methods (Figure 8) are:

- The user selects a service (*selectService(serviceID:SerrviceID, username:UserName)*);
- The user gets an AVService handler based on the service id (*getServiceHandler(serviceID:SerrviceID):URL*);
- The user gets a list of AVServices based on the service type (*getService(serviceType:SerrviceType):AVService[]*).

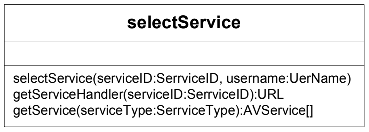

selectService
selectService(serviceID:ServiceID, username:UerName)
getServiceHandler(serviceID:ServiceID):URL
getService(serviceType:ServiceType):AVService[]

Figure 8. Attributes and methods for the selectService class

After the analysis of all objects and their corresponding use case descriptions, the resulting class diagram is shown in Figure 9.

The application of the 4SRS method in this demonstration case allowed the identification of the system logical architecture, as well as the identification of the class diagram that adds persistence to the data manipulated in the AVAccess service.

As advantages for this approach, we can point out that the identified classes properly represent the system requirements as they are identified through a recursive process embedded in the 4SRS method that ensures the elimination of redundancy and the identification of missing requirements. Comparing to the Giganto and Smith [6] approach, our approach is more robust since it complements the use of use case models with object-oriented diagrams to base the derivation of class diagrams. This complementary approach attenuates the typical problems of the natural language ambiguity.

In what concerns the lack of conciseness and completeness in requirements specification, as already mentioned, micro-step 2v constitutes a validation step certifying the semantic coherence of the object-model and discovering anomalies in the use case model.

Additionally, the recursive nature of the 4SRS method permits that several components of a system can be treated one at a time (each one with its own 4SRS execution [1]). This approach reduces the complexity of the overall system design, avoiding the construction of a global and massively complex class diagram for the whole system. Instead, we obtain a single class diagram for each system component, when the 4SRS executions adopt the recursive approach.

When compared to the existing approaches, the current version of the 4SRS method adopts a complementary approach by using both object-driven artefacts and use cases to support the complex process of identifying class diagrams from user requirements.

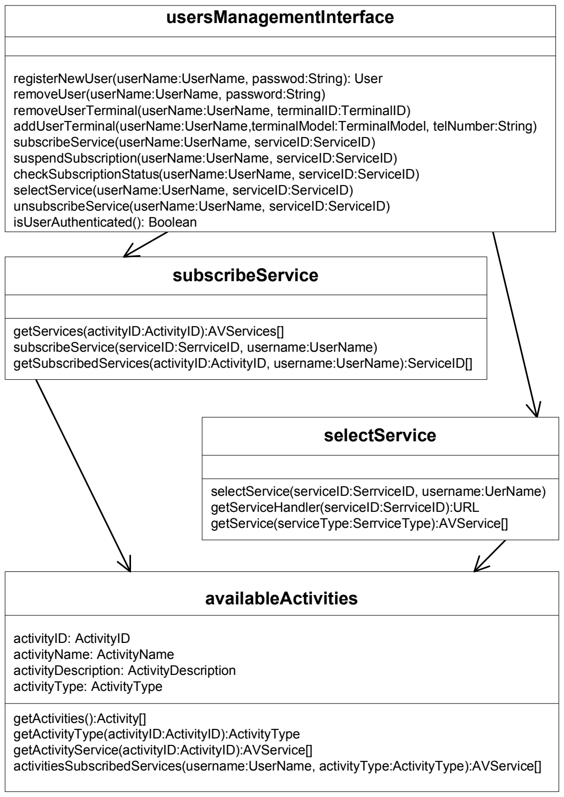

usersManagementInterface
registerNewUser(userName:UserName, passwod:String): User
removeUser(userName:UserName, password:String)
removeUserTerminal(userName:UserName, terminalID:TerminalID)
addUserTerminal(userName:UserName,terminalModel:TerminalModel, telNumber:String)
subscribeService(userName:UserName, serviceID:ServiceID)
suspendSubscription(userName:UserName, serviceID:ServiceID)
checkSubscriptionStatus(userName:UserName, serviceID:ServiceID)
selectService(userName:UserName, serviceID:ServiceID)
unsubscribeService(userName:UserName, serviceID:ServiceID)
isUserAuthenticated(): Boolean
subscribeService
getServices(activityID:ActivityID):AVServices[]
subscribeService(serviceID:ServiceID, username:UserName)
getSubscribedServices(activityID:ActivityID, username:UserName):ServiceID[]
selectService
selectService(serviceID:ServiceID, username:UerName)
getServiceHandler(serviceID:ServiceID):URL
getService(serviceType:ServiceType):AVService[]
availableActivities
activityID: ActivityID
activityName: ActivityName
activityDescription: ActivityDescription
activityType: ActivityType
getActivities():Activity[]
getActivityType(activityID:ActivityID):ActivityType
getActivityService(activityID:ActivityID):AVService[]
activitiesSubscribedServices(username:UserName, activityType:ActivityType):AVService[]

Figure 9. Final class diagram

VI. DISCUSSION

Usually, object-oriented methodologies do not pay too much attention to the object diagram, i.e., they are class-driven. When they do, the class diagram is built firstly and only later the object diagram is specified. The approach presented here reverses this order, not in a reverse engineering approach, but in an object-driven perspective. We believe that it is more important to have a good system-level object model (with logical architecture semantics), because the system is composed of objects and not by their classes. This is the main reason to first identify the system-level objects and to later classify them, that is, to select the classes to which those objects belong.

Some specialists may classify the object-driven perspective (that puts classes in an apparently secondary role and that firstly defines the objects and later the classes) as object-based rather than object-oriented. Nonetheless, the object-driven approach is somehow similar to the bottom-up discovery of inheritance, as defined in [12], to hierarchically organize the classes.

In the approach described in this paper, the identification of a class diagram for an already refined logical architecture is based on the structural characteristics of that model (refined raw object model). Currently, for discovering the methods and the attributes, we are analyzing the textual descriptions of the refined use cases. We plan to develop a study to analyze the possibility of using the textual descriptions of each system-level object to support the discovering of methods and attributes, avoiding imprecise and incomplete system requirements. This study will also make possible the conciliation of all the obtained class diagrams for each subsystem to construct a global class diagram to support the design of the database that may be considered pertinent within the specified logical architecture.

The 4SRS method relies on a model-driven development approach, which is a technique that uses models during the software development. It is executed by successively transforming models into other (more reified) models, until the final system design is obtained. Using the MDA (model-driven architecture) [13] reference framework, we consider that the extension of the 4SRS method presented in this paper transforms UML use case diagrams (that represent the original user requirements) and UML object diagrams (that represent the system requirements of the logical architecture) into UML class diagrams (to be used in future transformation steps - not covered by this paper - to derive the structure of the relational database component of the global solution).

VII. Conclusion

This paper presented the additional steps added to the 4SRS method in order to be possible the identification of the class diagram for a given logical architecture. One open issue in the 4SRS method was the transformation of the raw object diagram, with the system level entities, into a class diagram. Two additional steps were included into the 4SRS method to allow this transformation. These steps use the available raw object diagram and the refined textual use cases descriptions.

A demonstration case showed the application of these two supplementary steps of the 4SRS method. Through them, a class diagram was identified. It includes the classes, attributes and methods that emerge from the several steps that now constitute the 4SRS method.

As future work we intend to add two more steps to this process. One of them is associated with the design of sequence diagrams modelling the interactions between system entities. After that, the sequence diagrams will be used to validate the class diagram ensuring that all system requirements were considered.

References

[1] R.J Machado, J.M. Fernandes, P. Monteiro, and H. Rodrigues, “Refinement of Software Architectures by Recursive Model Transformations”, Proc. of the 7th International Conference on Product Focused Software Process Improvement - PROFES’06. 2006. Amsterdam: Springer-Verlag.

[2] R.J. Machado, J.M. Fernandes, P. Monteiro, and H. Rodrigues, “Transformation of UML Models for Service-Oriented Software Architectures”, Proc. of the 12th IEEE International Conference on the Engineering of Computer-Based Systems - ECBS 2005. 2005. Maryland, U.S.A.: IEEE Computer Society Press.

[3] B. Anda and D.I.K. Sjoberg, “Investigating the Role of Use Cases in the Construction of Class Diagrams”, Empirical Software Engineering, vol. 10, 2005, pp. 285-309.

[4] B. Dobing and J. Parsons, “How UML is used”, Communications of the ACM, vol. 49, 2006, pp. 109-113.

[5] H. Christiansen, C. T. Have, and K. Tveitane, “From use cases to UML class diagrams using logic grammars and constraints”, Proc. of Recent Advances in Natural Language Processing. Shoumen, Bulgaria, 2007.

[6] R. Giganto and T. Smith, “Derivation of Classes from Use Cases Automatically Generated by a Three-Level Sentence Processing Algorithm”, Proc. of the Third International Conference on Systems, 2008. IEEE Computer Society.

[7] Y. Liang, “From use cases to classes: a way of building object model with UML”, Information and Software Technology, 45:83-93, 2003.

[8] N. Juristo, A.M. Moreno, and M. López, “How to Use Linguistic Instruments for Object-Oriented Analysis”, IEEE Software, 17(3):80-9, May/June 2000.

[9] R. J. Abbott, “Program Design by Informal Descriptions”, Communications of the ACM, 26(11):882-94, Nov. 1983.

[10] J.M Fernandes, R.J. Machado, P. Monteiro, and H. Rodrigues, “A Demonstration Case on the Transformation of Software Architectures for Service Specification”, Proc. of the 5th IFIP Working Conference on Distributed and Parallel Embedded Systems - DIPES 2006. 2006. Braga, Portugal: Springer-Verlag, New York, U.S.A.

[11] P. Monteiro, “Model-based Transformations for Software Architectures: a prevasive application case study”, MSc Thesis, University of Minho, Portugal, 2005.

[12] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. “Object-Oriented Modeling and Design”, Prentice-Hall International, 1991.

[13] OMG Unified Modeling LanguageTM (OMG UML), Infrastructure, Version 2.2, OMG Document Number: formal/2009-02-04.