Tight Structuring for Precise UML-based Requirement Specifications *

E. Astesiano and G. Reggio
DISI, Università di Genova - Italy

Abstract. On the basis of some experience in the use of UML-based use case-driven methods, we believe and claim, contrary to a recent wave for allowing almost total freedom as opposed to disciplined methods, that a tighter and more precise structuring of the artifacts for the different phases of the software development process may help speed-up the process, while obviously making easier the consistency checks among the various artifacts. To support our claim we have started to investigate an approach, that, though being compliant with the UML notation and a number of UML-based methods, departs from them both in the basic philosophy, that follows the "tight and precise" imperative, and in the technical solutions for structuring the various artifacts.
Building on some previous work concerning the structure of the requirement specification artifacts, here we complete upwards and improve our proposal, investigating the link between the analysis of the problem domain and the requirement capture and specification. Indeed, one of our assumptions, as advocated by some methodologists and backed by our own experience, is the neat separation between the problem domain and the system. To that purpose we propose a rather new way of structuring the problem domain model and then the link with the system, that encompasses the most popular current approaches to domain modelling. Then we exploit both the domain model and our requirement specification frame for capturing and specifying the requirements. From our construction we can derive rigorous guidelines, only hinted to here, for the specification tasks, in a workflow that allows and suggests iteration and incrementality, but in a way that is not just based on the single use cases and takes more care of the overall construction. The various concepts and constructions are illustrated with the help of a small running case study.

1 Introduction

In recent years we have seen the introduction and the acceptance of use-case driven approaches combined with object-oriented techniques, particularly in connection with visual notations such as UML [19]. This is the case of software development process models such as RUP (the Rational Unified Process [14]), Catalysis [6] and COMET [9]. In the last three years we have made some experiments in the use of UML-based and use case-driven techniques and of some related methods, both in teaching and by personal involvement. Those experiments were concerned especially with the early development phases, requirement capture and specification and then design.

Because of that experience we have become more and more convinced that, to be more effective both in terms of productivity and quality, those approaches need to be

* Work supported by the Italian National Project SAHARA (Architetture Software per infrastrutture di rete ad accesso eterogeneo).

improved and complemented especially in two directions. The first is a tighter and more systematic structuring of the artifacts based on precise guidelines for their building. The goal we want to achieve by that is twofold: first, cut experimentally endless discussions on the structural choices and thus making the process much faster; second, provide a better support to the consistency checks among the different artifacts. Indeed, it is well-known that consistency is one of the hot problems in multiview modelling approaches ([1, 3]). Of course tight and precise structuring is not enough for consistency checks, as long as we want (and we much need) to go beyond pure syntactic checks (see [7] also for references). Thus there is a second sense of our "precise" qualification, that does not refer to the structural aspects, but to the semantics of the single constructs. Indeed, another principle we follow is the use of constructs with an unambiguous well-defined semantics. Altogether, by the tight structuring and the use of semantically well-defined constructs, we here provide an example of what we call well-founded method, as a modern and more viable approach that still embodies the basic sound principles of the "explicitly formal" methods (see [5] for a perspective and a rationale of well-founded methods).

In this paper we continue the investigation and update the initial proposal first presented in [2]. In that paper we have outlined some new ideas about the structure of the Requirement Specification artifacts. Here we complete upwards and improve that proposal, investigating the link between the analysis of the problem domain and the requirement capture and specification. Indeed one of our assumptions, backed by our own experience, is the neat separation between the problem domain and the system (as much advocated in the work of pioneers, such as M.Jackson's [10]). To that purpose we propose a rather new way of structuring the problem domain model (PDM) and then the link with the system. Our proposal, centered on two views, the Conceptual View and the Work Case View, in a structural sense encompasses the two most popular current approaches to domain modelling, namely conceptual modelling and business modelling, and can be reduced as a specialization to each of those. Then we propose a "system placement" activity, supported by a System Placement Diagram, to relate the system to the domain and, by that, to locate the system boundary.

This paper is mainly aimed at presenting the structural aspects of our approach; we present both its rationale and the technical aspects, illustrated with the help of a small running case study. But the fact that we illustrate the structure with the end artifacts, should not induce to underestimate the relevance of the methodological aspects in the building of the artifacts. Indeed, in the development we make an ample use of iteration and feedback, as it is unavoidable in any sensible method. But, also for lack of room, we only touch that issue, just providing some methodological guidelines for the workflow, while not presenting the various iterations we have followed when handling the case study.

In the first section we present the rationale and our way of structuring the problem domain. In the second we outline the transition from the problem domain model to the requirement capture and specification, by exploiting our particular way of structuring the requirement artifacts. Then, after some methodological hints on the workflow, we discuss the relation to other and future work. Throughout the paper we illustrate our approach by means of a small case study, shown in Fig. 1.

We have to develop a system AL⊥ to handle algebraic lotteries. Our lotteries are said "algebraic" since the tickets are numbered by integer numbers, the winners are determined by means of an order *over such numbers, and a client buys a ticket by selecting its number. Whenever a client buys a ticket, he gets the right to another free ticket, which will be given at some future time, fully depending on the lottery manager decision. The number of a free ticket is generated by the set of the numbers of the already assigned tickets following some* law.
Thus a lottery is characterized by an order *over the integers determining the winners and a* law *for generating the numbers of the free tickets. To guarantee the clients of the fairness of the lottery, the order and the law, expressed rigorously with algebraic techniques, are registered by a lawyer before the start of any lottery.*
The system will be then realized as an on-line system, where the tickets must be bought and paid on-line using credit cards with the help of an external service handling them. Possible clients must register with the lottery system to play; and clients access the system in a session-like way. An external service takes care of the authentication of the clients.

Fig. 1. The AL⊥ case study

2 Modelling the Problem Domain

2.1 Method Rationale

The distinction between the (problem) domain and the (solution) system has been recognized and accepted long time ago in the software engineering community (see, e.g., [10]). The problem domain consists of those aspects of the real world that are relevant for the system to be developed for providing a solution to the problem under consideration. For instance, in the case of a system for handling a lift the relevant domain aspects concern how the lift works, that is in which way the calls can be made, whether the cabin doors are opened/closed by the users, and the most typical habits of the users (e.g., a user immediately leaves the cabin once the doors are open). Of course the separation line between domain and system depends on the way the problem is stated. For example for the algebraic lottery, the problem domain aspects concern how the clients buy the tickets, how and when the winners are drawn and so on. Instead, in our formulation of that problem, the possibilities for the clients to use Internet to access the lottery and the other of having clerks selling the tickets produced by a printer are choices to be made when devising the system and thus should appear in the requirements and not in the problem domain.

More or less, any development method requires to model the problem domain either explicitly, in a specific task, or implicitly in the requirement specification task. In our proposal, we prefer to separate the domain modelling from the requirement definition and to present the result in a specific document (the PDM), because, in our opinion,

- that separation helps get a more abstract unbiased description of the system that we denote by System;
- the resulting PDM may be reused for many different System, thus extending to the early phases of the development the MDA philosophy [12].

Currently, in the literature and also in the practice, there are two main ways to present a PDM:

as a conceptual model: the PDM is a conceptual model of the entities present in the domain, in this case it is usually represented by a (UML) class diagram, where the classes correspond to such entities, the associations to their mutual relationships

and, if allowed, the attributes to some characteristics of such entities. Sometime, some limited behavioural aspects are given by sequence/collaboration diagrams.

as a business model: the PDM is the description of a business, intended as an organized offer of functionalities (business use cases) to outside entities interacting with it (business actors), and with an internal structure (business object model based on business workers and business items). Clearly, in this case actors, workers and items correspond to entities present in the real world, and are not parts of the System to be developed. This technique has been introduced recently in the RUP development method [14].

In our opinion, the conceptual model approach is not satisfactory in the cases where the entities in the domain are highly interacting and autonomous (e.g., participants in a meeting for a system handling meetings electronically), or the most relevant aspects of the domain are naturally presented as workflows (e.g., handling an order in an invoice electronic system). The business model approach overcomes the above limits, and is quite satisfactory whenever it is possible to naturally determine the "business organization". However, it is problematic in the cases where the domain is quite static (e.g., the domain for a word processor concerning texts, paragraph, documents, layout and so on) or when trying to find the "business organization" we fix too early the boundaries of the the System to be developed.

Here we propose a somewhat more general technique trying to avoid the negative aspects of both the above approaches, and such that the two above approaches are particular subcases of our one.

2.2 Problem Domain Model: a Proposal

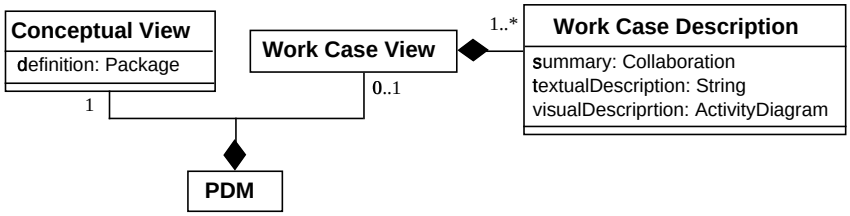
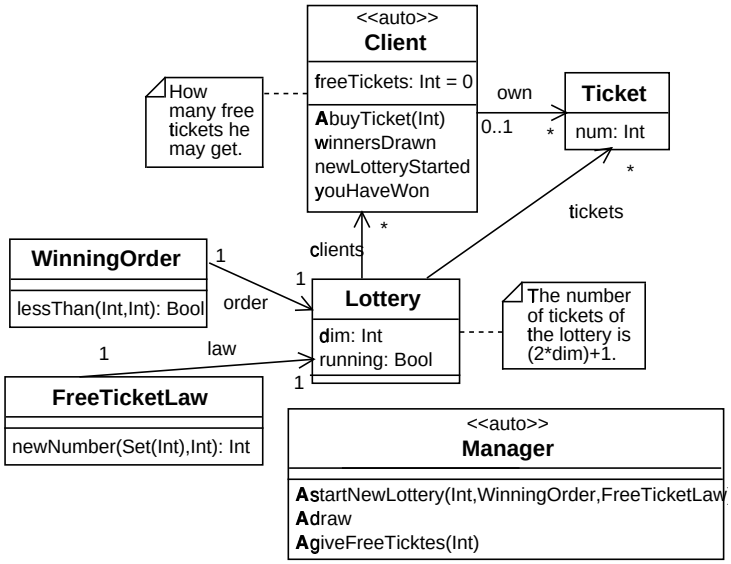


Fig. 2. PDM Structure

Overall Structure The structure of a PDM in our proposal is shown in Fig. 2. We propose to model the various entities present in the domain by the Conceptual View, a UML class diagram, but where the classes may be also active, thus with a dynamic behaviour; even more we allow to model the autonomous features of their behaviour. Then, the most relevant cooperation among such entities may be modelled in the Work Case View part that consists of a special kind of workflows named *work cases.*

Conceptual View The Conceptual View, a UML package containing at least a class diagram, makes explicit which are the entities appearing in the domain (they are modelled by objects whose classes appear in such package) and their mutual relationships, if any

(modelled by associations among the corresponding classes). The other elements of the class diagram, such as class attributes, operations and constraints, and the other diagrams in the package (as state charts defining the operations) may be used to model relevant aspects of such entities. In such package we may use the active class stereotype ≪auto≫ to indicate those domain entities capable of autonomous behaviour (i.e., they are not just reacting to external stimuli). An autonomous action of such entities is modelled by the self call of operations of the stereotype ≪A≫ (visually denoted by identifiers starting with a bold capital **A**).




```
context C: Client inv: C.freeTickets >= 0
context WO:WinningOrder inv: "x < y iff WO.lessThan(x,y)" is a total order
context newNumber(asTks,j):
 pre: {-j, ..., +j} - asTks <> {}
 post: asTks->excludes(result) and -j =< result and result =< j
context L: Lottery inv:
 All the tickets in L.tickets have different numbers and
 L.dim = 5000 * k with K >= 1 and L.tickets.num = {-L.dim ... L.dim}
Lottery.allInstances->size = 1 Manager.allInstances->size = 1
```

Fig. 3. AL_L PDM: Conceptual View

In Fig. 3 we have the diagram presenting the Conceptual View of the AL_L case study.

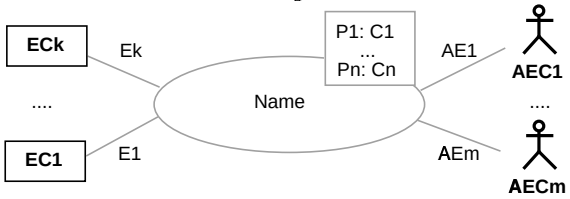
The autonomous entities appearing in such domain are the clients, which may buy the tickets, and a manager, which may start the lotteries, and decide when to draw the winners and when to give out free tickets. Four **A** operations model that such entities may perform such acts autonomously; whereas other non-autonomous activities, such as to be informed of winning a prize, are modelled by plain operations (e.g., youhaveWon). There are also passive entities, describing the current lottery and its tickets. The constraints attached to the class diagram model relevant aspects of the domain entities (e.g., there

is at most one running lottery, or the *winning orders* are total orders on the integer numbers).

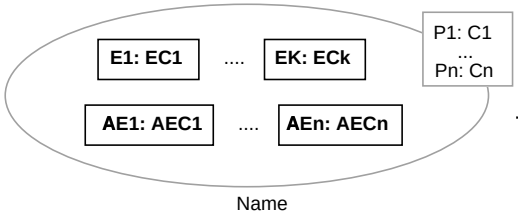
Work Case View Technically, a work case is a variant of the UML collaboration, thus it allows to represent some cooperative effort among some entities present in the domain. As a collaboration, it has a name, precisely defines (the roles of) the participants, and may have some parameters. But, we prefer to model the behaviour of a collaboration by means of an activity diagram expressing the causal relationships among actions made by the participants, instead of by a set of interactions (message exchanges). In this way we can just describe the causal/temporal relationships among relevant actions made by the participants without necessarily presenting such actions as messages sent by someone to some other one. The reason of this choice is to keep the description of a work case quite abstract avoiding to introduce spurious objects (just to have someone calling some operations) or to make particular choices about who calls who.

Notice that there is a big difference between our work cases and the RUP business use cases. Indeed all the participants in a work case are modelled by the roles of the work case (recall it is a variant of a UML collaboration), whereas in a business use case there exists a business organization, which is a special implicit participant interacting with all the other ones (business actors), and in general the latter do not interact each other.

The description of a work case consists of three parts (see Fig. 2). The main part is a, possibly parameterized, UML collaboration, where its roles corresponds to all the participants in the work case. Since we do not describe its behaviour by a collaboration diagram, we prefer to visually represent the collaboration corresponding to a work case named Name in the following way:

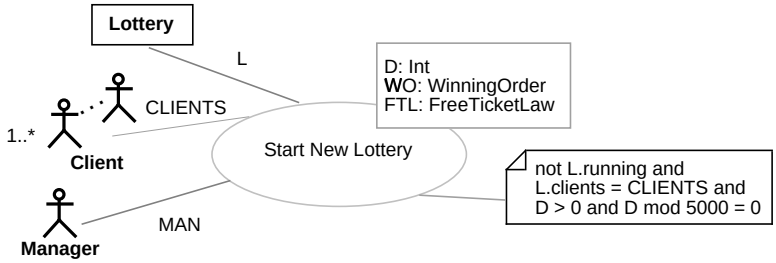


where P1, ... Pn are the parameters, AE1, ... AEm, E1, ... Ek are the roles (to be played by domain entities) of the participant in the work case; C1, ... Cn, AEC1, ... AECm, EC1, ... ECk are classes appearing in the Conceptual View. We distinguish the class of the the autonomous entities by using the icon  . Using the standard UML notation it should be represented by



Since we can attach to the collaboration icon a constraint, we can state, if any, which conditions the participants and the parameters must satisfy to take part of the work case. Then, a work case description contains a textual description made by using the natural language. It must start with a sentence of the form "When ..." expressing under which conditions the considered domain entities may take part to the work case, and

must consist of sentences where the subjects are autonomous participants and where the object complements are participants. The last part of a work case description is a visual presentation of its behaviour by means of a UML activity diagram. The action-states of such diagram can be only calls of the operations of the work case participants, and the conditions properties on the states of the work case participants.



textual When no lottery is running, the manager may start a new one giving the dimension of the lottery (a natural greater than 0 and multiple of 5000), the law for generating the numbers of the free tickets (a function which given a set of integers finds a new number not belonging to it) and a total order on integers, which will be used to find the winners.
All clients, will be informed of the new lottery.
Then, a lottery is running and is characterized by the data given by the manager, and all its tickets are available.

behaviour

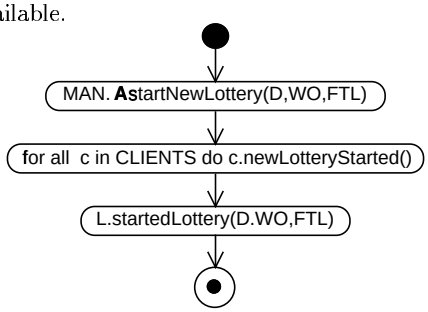


Fig. 4. AL⊥ PDM: Work Case Start New Lottery

In Fig. 4 we present one work case of the AL⊥ PDM (the remaining ones: Buy Ticket, Draw Winners and Give Free Tickets can be found in [4]). This work case is quite simple; it just say under which conditions the manager my start a new lottery and what happens when he does that (the clients are informed, and the characteristics of the new lottery are recorded by the domain entity of class Lottery.

To keep the presentations of the behaviour views of the work cases simple and quite readable, we strongly suggest to define appropriate additional operations, similarly to those used in [19] for presenting the "well-formedness rules". For example, in the work case Start New Lottery we have used the operation `startedLottery` to describe the update of the Lottery domain entity.

```
context Lottery::startedLottery(D:Int,WO: WinningOrder,FTL:FreeTicketLaw)
post: self.running  and  self.dim = D and
      self.winningOrderr = WO and  self.FreeTicketLaw = FTL  and
      self.availableTickets.num = { -D ... D }
```

3 Capturing and Specifying Requirements

3.1 System Placement

Once we have given the PDM, the next step of the development of the System is "to place it" in the domain by making precise which problem it must solve. This task consists of the following activities:

1. add a class for System to the class diagram in the Conceptual View of the PDM;
2. decide which entities of the domain will be encompassed in the System, that is if they are autonomous their activities will be realized by the System, otherwise the data that they contains will be preserved by the System; place them inside the icon of the System class;
3. decide which entities of the domain will interact with the System; connect them with the icon of the System class by a line;
4. decide if the System needs to cooperate with further external entities (not present in the domain); usually they are devices or entities offering services to support the System activity; add them as new classes to the diagram and connect them with the icon of the System class by a line;;
5. decide which work cases the System will support (clearly all their participants have to be included in those considered at points 3 and 4; for each of them place the corresponding collaboration pictures over the class diagram.

After having performed the above tasks, you have got what we call "System Placement Diagram".

Notice that placing the System includes of course the definition of its boundary, which is recognized to be an important task almost in any development method (see [17, 11]). If we consider the AL_L case study, we can see how we can place different systems in the domain described by the PDM given in Sect. 2.2. For example:

a) The System must completely automate the handling of the lottery using Internet, and taking advantage of an external authentication service and of a credit card service for the payment.

b) As for the previous case, but the System will not replace the manager deciding, e.g., when to draw the winners, and email will be used for some communications with the clients.

c) The System just helps the clerks to sell the paper tickets to the clients by showing the available tickets, printing the tickets, generating the list of the winning tickets, and printing the free tickets, which will be given by the clerks to the clients that show a paid ticket.

For what concerns the work cases all of them will be supported by the above systems. In this paper we consider case **b)**, and in Fig. 5 we show the resulting System Placement Diagram. This diagram will be the starting point to capture and specify the requirements.

3.2 Overall Structure of a Requirement Specification

In our approach the Requirement Specification artifacts consist of different views of the System, plus a part, Data View, needed to give a rigorous description of such views. Its structure is shown in Fig. 6 by a UML class diagram.

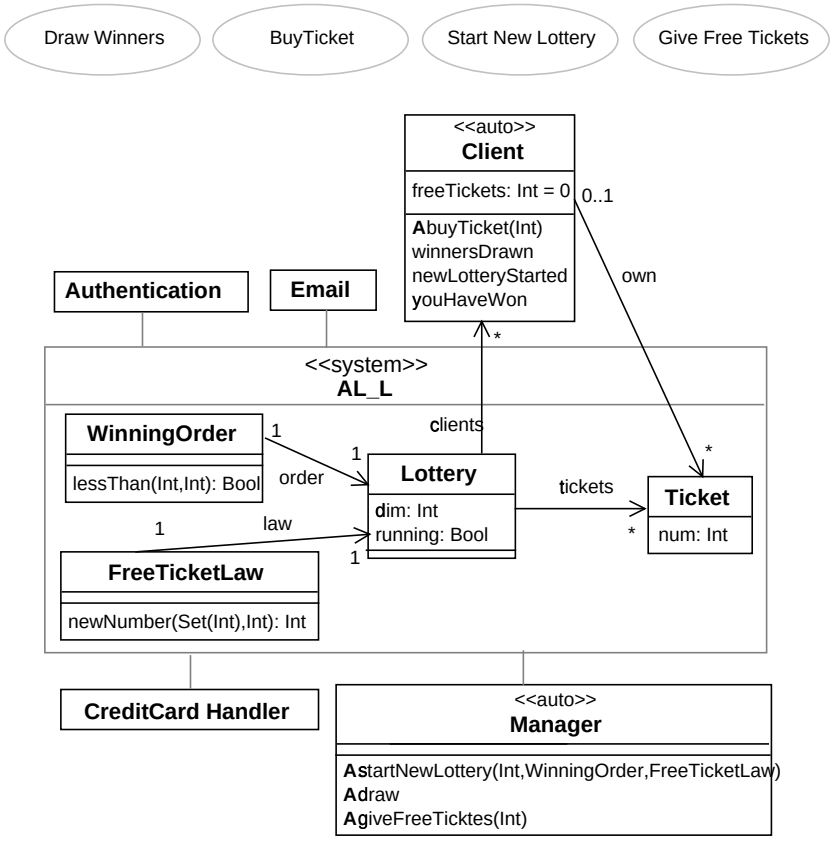


Fig. 5. AL_L Case Study: System Placement Diagram

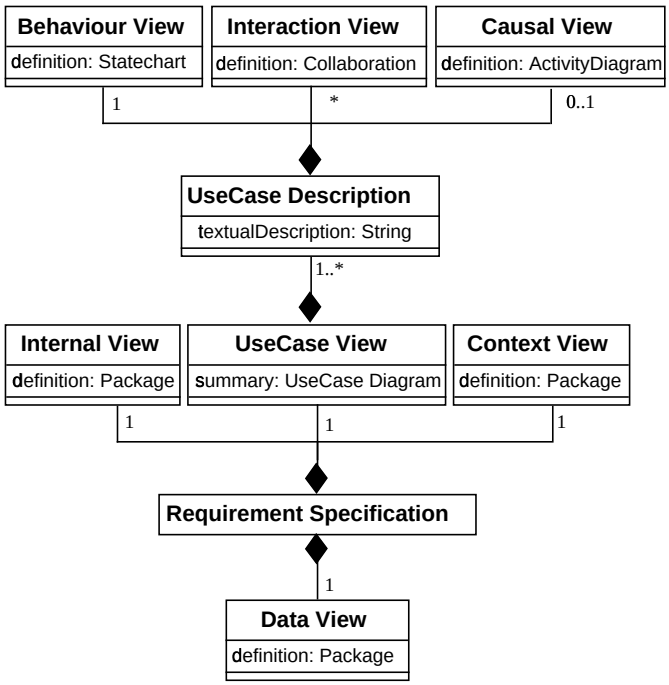


Fig. 6. Requirement Specification Structure

Context View describes the context of the System, that is which entities (*context entities*) and of which kinds may interact with the System, and in which way they can do that. Such entities are further classified into those taking advantage of the System (*service users*), and into those cooperating to accomplish the System aims (*service providers*). That explicit splitting between the System and the context entities should help avoid confusions between what exists and needs just to be precisely described (context entities) and what instead has to be developed (System) on which we have to find (capture) the requirements The further splitting between users and providers should help distinguish which context entities cannot be modified by the developer (providers), and those which may be partly tuned by the developer (users), e.g., by fixing their interface towards the System.

Use Case View, as it is now standard, shows the main ways to use the System (*use cases*), making clear which actors take parts in them. Such actors are just *roles* (*generic instances*) for some context entities depicted in the Context View.

Internal View describes abstractly the internal structure of the System, that is essentially its Abstract State. It will help precisely describe the behaviour of the use cases, by allowing to express how they read and update it. UML allows a single use case to have a proper state, but we prefer to have a unique state for all the use cases, to help model their mutual relationships (e.g., if two use cases update the same information, we are led to detect and to handle possible conflicts).

Data View lists and makes precise all data appearing in the various views of the System to help guarantee the consistency of the concepts used in such views.

Some of the above views (e.g., Internal View and Context View) are new w.r.t. the current methods for the OO UML-based specification of requirements. In our approach,

they play a fundamental role to help ensure the consistency among the various use cases and of the whole specification.

3.3 Examples from the AL_L Case Study

Here we illustrate the proposed structuring for the requirement specification artifact, showing its use in the AL_L case study.

Notice that here we present the result of an activity that includes various steps and iterations. For lack of room here we do not discuss those aspects of incremental development with feedback. We just provides a hint in Fig. 7.

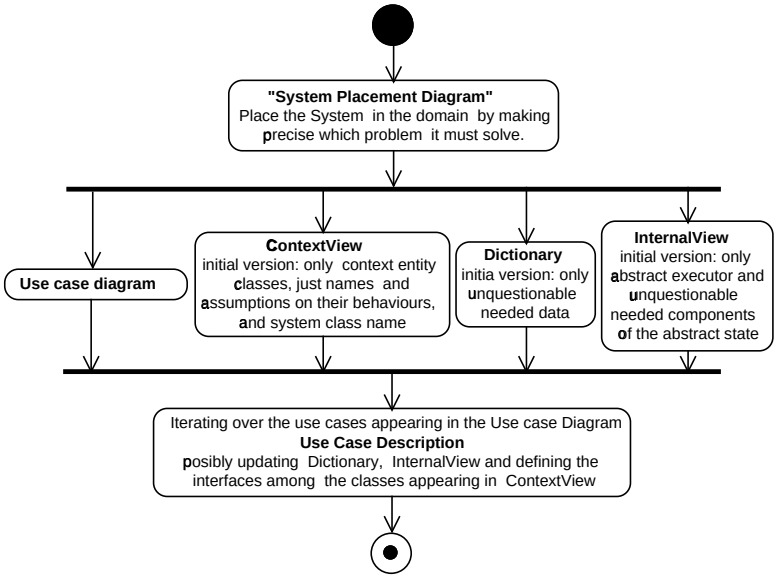
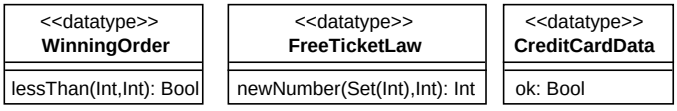


Fig. 7. Requirement Specification Tasks




```
context WO: WinningOrder inv: "x < y iff O.lessThan(x,y)" is a total order
context newNumber(asTks,j):
 pre: -j, ..., +j - asTks <> {}
 post: asTks->excludes(result) and -j =< result and result =< j
```

Fig. 8. AL_L Requirement Specification: Data View

Data View The Data View for the AL_L case is quite simple and just introduces three data types: the orders for finding the winners, the rules for finding the numbers of the tickets to be given freely, and the data needed to identify a credit card.

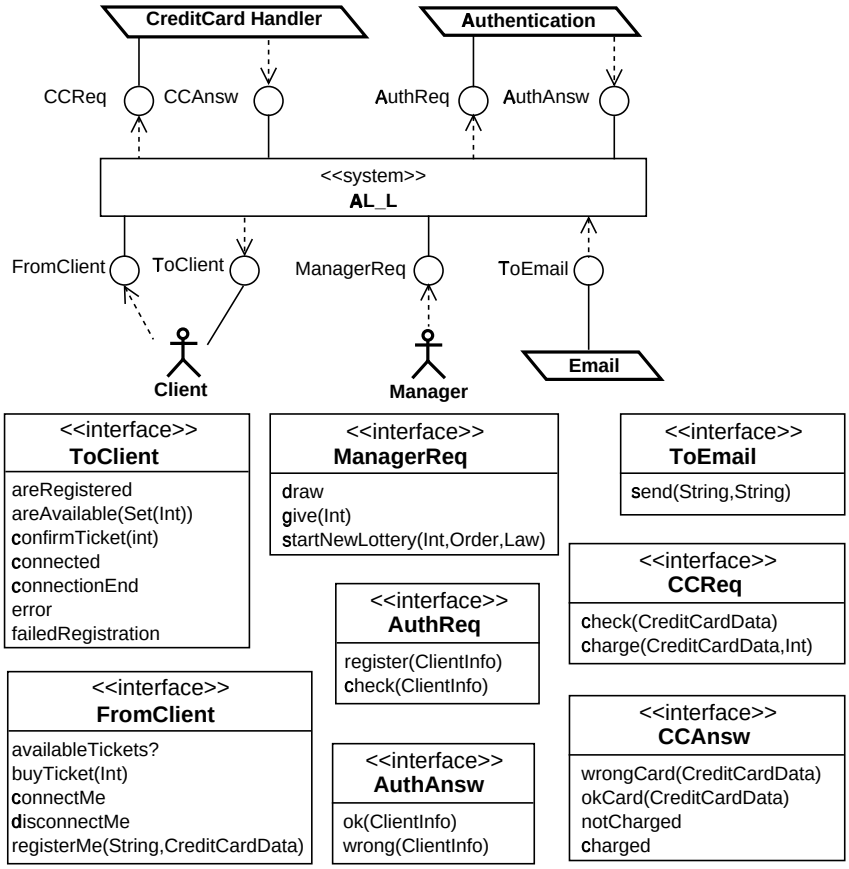
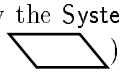
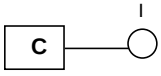
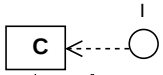


Fig. 9. AL_L Requirement Specification: Context View

Context View The Context View of the AL_L System, shown in Fig. 9, consists of a class diagram, where there is a class AL_L of stereotype ≪System≫ whose unique instance is the System, some classes of stereotype ≪SU≫ (icon ) whose instances are users of the services provided by the System (the clients and the manager); and some classes of stereotype ≪SP≫ (icon ) whose instances are providers of services used by the System (the email, the credit card service and the authentication service).

In this diagram we show the mutual interfaces among these classes, that is in which way they may interact, using the the standard UML *interface construct*. In Fig. 9, for example, we can see that the interface ToEmail of the **Email** context entity is really simple, just offering the possibility to receive request to send an email message. 

and 

visually present respectively that a class C realizes/uses an interface I. The interfaces appearing in this diagram are usually given apart (here in the bottom part of Fig. 9).

The Context View may include also some information on the behaviour of the ≪SU≫ and ≪SP≫ classes, but not of the ≪System≫ class, to model the assumptions on the behaviour of their instances on which the System relies.

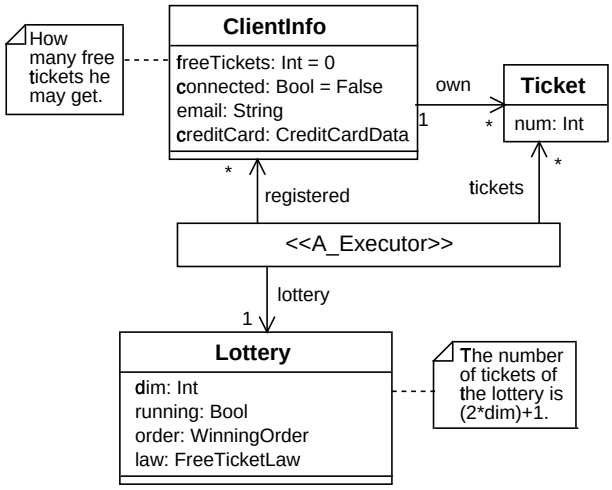


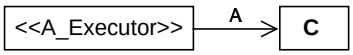
Fig. 10. AL_L Requirement Specification: Internal View

Internal View The Internal View describes at an extremely abstract level the structure (architecture) of the System. This structure consists of a unique active object able to perform the System activities (abstract executor) and by many passive objects describing the System Abstract State.

In Fig. 10 we show the Internal View of the AL_L case study. It consists in a class diagram containing exactly one class of the stereotype ≪A_Executor≫, and several passive classes defining the parts of the System Abstract State.

The instances of the class ClientInfo represent the information relative to the client context entities. Very frequently, the Abstract State must contain information about some context entities, and so we propose a standard way to treat these cases We name CENTInfo the class of the information on the context entities of class CENT, and assume that its instances are in bijective correspondence with those of CENT, and thus with the context entities. Furthermore, this correspondence is supported by an operation CENT::Info: CENTInfo that returns the information element corresponding to a context entity.

Following this approach, we avoid, on one side, models where the presence of a class named as a context entity class, say Client, requires to think about its true nature (e.g., is it a database relation ? , or a kind of interface taking care of the interactions with such context entities"?, or ...), and, on the other one, precise but too much detailed models, where the association of the information to the corresponding context entities is realized, e.g., by using codes uniquely identifying the entities.

The class diagram of the Internal View describes implicitly also the "Abstract State" of the System (technically the state of the ≪System≫ class appearing in the Context View) in the following way: for each association in the diagram from the ≪A_Executor≫ class 

the ≪System≫ class has an attribute A: Bag(C).

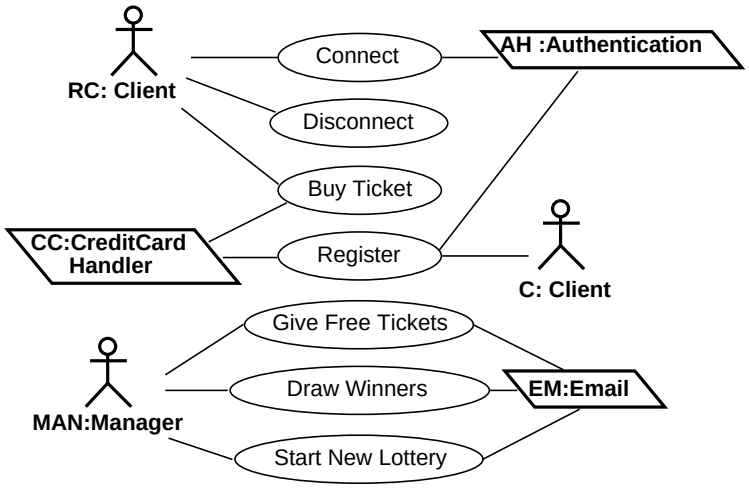


Fig. 11. AL_L Requirement Specification: Use Case Diagram

Use Case View The Use Case View consists of a UML "Use Case Diagram" and of a Use Case Description for each use case appearing in it.

But, for us the actors appearing in the Use Case diagram are possible roles for the entities outside the System interacting with it (context entities, defined in the Context View). Thus each actor will be denoted by a name, expressing the played role, and by a class, appearing in the Context View, showing which kind of context entities may play such role. Moreover, since the context entities are distinguished between users of services provided by the System and providers of services needed by the System also the actors will be distinguished in the same way. The same icons used for the context entity classes will be used for the actors (and).

The Use case Diagram for the AL_L case study is reported in Fig. 11. Notice how the client context entities may play two different roles, when interacting with the System, as registered client (RC) when playing with the system, and as normal client (C), when trying to register.

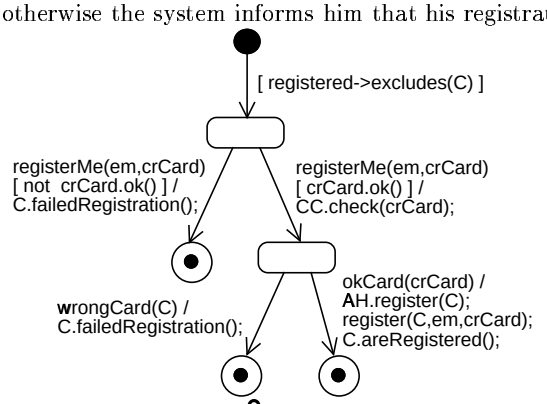
A Use Case Description, see those of two use cases of AL_L in Fig. 12 and 13 (the remaining use case may be found in [4]), consists of a textual presentation and of one or more views, of different kinds, of the use case.

The textual description should be expressed by sentences where the subject is either one of the actors or the System, and may start with a sentence of the form "When ..." expressing under which condition the use case may happen (pre-condition).

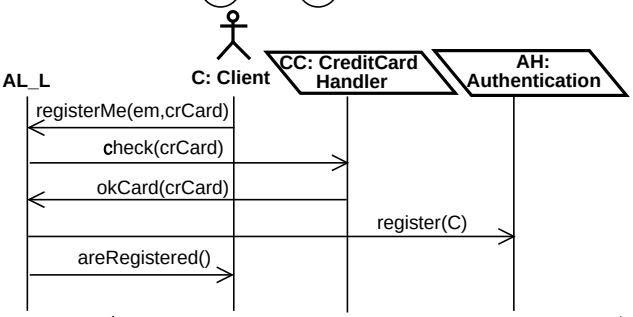
Any Use Case Description must include a Behaviour View, which is a statechart for the ≪System≫ class describing the complete behaviour of the System with respect to

textual When a client is not registered may register himself to the lottery system by giving his email and the data of a credit card. The system check the credit card data with the credit card service, if they are ok and are validated by the credit card service, then the system registers the client with the authentication service, informs him that he has been registered, and he will be registered; otherwise the system informs him that his registration has failed.

behaviour



interaction

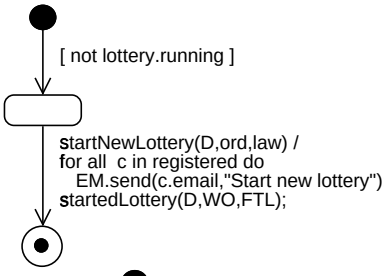



```
context AL_L::register(C:Client,em:String,crCard:CreditCardData)
post: registered->
 exists(CI | CI.email = em and CI.creditCard = crCard and C.Info = CI)
```

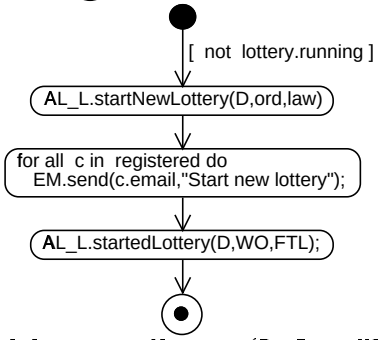
Fig. 12. AL_L Requirement Specification: Use Case Register

textual When no lottery is running, the manager may ask to the system to start a new one by giving its dimension (a natural greater than 1), winning order (an order on integers, which will be used to find the winners) and free ticket law (for generating the numbers of the free tickets, just a function which given a set of integers finds a new number not belonging to it). Then, a new lottery will be running having the dimension, winning order and free ticket law given by the manager.
The system will inform all the registered clients by an email message that a new lottery is running.

behaviour



causal




```
context AL_L::startedLottery(D: Int, WO: WinningOrder, FTL: FreeTicketLaw)
post: tickets = {} and lottery.dim = D and
 lottery.Order = ord and lottery.Law = law and lottery.running = True
```

Fig. 13. AL_L Requirement Specification: Use Case Start New Lottery

such use case. Such statechart, see, e.g., Fig. 12 and 13 has particular features. The transition from the initial state should be labelled by the "pre-condition", its events may be only call of the operations of the ≪System≫ interfaces, its conditions may test only the System Abstract State and the event parameters, and its actions may only be calls of the the operations of the actors, as defined by their interfaces, or actions updating the System Abstract State. To keep the behaviour views simple and quite readable we use appropriate additional operations, as previously suggested for the work cases.

The behaviour view is a "complete" description of what the System does that concerns the use case. In Fig. 12 we can see that the registration of a client requires some collaboration by the credit card service and the authentication service, that affects the System Abstract State, and that the use case has three possible cases (all of them visually presented in the diagram); whereas in Fig. 13 we see that the registered client will be informed by an email message of the new lottery, and that the use is really simple, not having any alternative way.

A Use Case Description may include any number of Interaction View, which are sequence (or collaboration) diagrams representing the interactions happening in a scenario of the use case among the context entities and the System. The Use Case Description in Fig. 12 has an Interaction View, whereas the one in Fig. 13 none. Any Interaction View must be coherent with the Behaviour View (that is, it must represent a particular execution of the complete behaviour of System described by such view).

We think that the Interaction View are really important showing visually who does what, but they are complementary to the Behaviour View because they cannot show under which conditions the various messages may be exchanged and their effects on the System Abstract State.

A Use Case Description may include also a Causal View (see for example Fig. 13), which is an activity diagram describing all the relevant facts happening during the use case and their causal relationships. The relevant facts (technically represented by action-states of the activity diagram) can be only calls of the interface operations of System by the actors, calls of the operations of the actors by System, UML actions producing side effects on the System Abstract State. Also the Causal View must be coherent with the Behaviour View, in the sense that the causal relationships among "facts" that it depicts may happen in the behaviour depicted by the state chart.

The various views listed above play different roles in the description of a use case and are partly complementary and partly overlapping. The choice of which of them to use depends on the nature of the considered use case. The only rule enforced by the method is that the behaviour view is mandatory, because it obliges to present all the behaviour of the use case (e.g., all possible alternative scenarios are included), even if it may be less readable than the others. However, due to the nature of the UML state chart, the behaviour view cannot be a complete description of the use case, indeed; it does not allow to express who is calling the operations to which System reacts.

4 Related Work and Conclusions

The approach that we have outlined (see [4] for an extended version with interesting and complex case studies), here limited to the early development phases, is in the line of some of the best-known methods for software development, adopting a multiview and use case approach and using the UML notation. But it departs from them, at least to our

knowledge, in some important respects, both from the methodological and the technical viewpoint.

First, on the method side, the overall major goal is to propose a more systematic and stringent approach, in the sense that the overall structure of our artifacts, both for the PDM and the Requirements, is constrained in order to tightly relate the components and have at hand the possibility of performing a number of consistency checks. This view contrasts with the almost total freedom given, for example in RUP [14], where the structure is just based on the use case description. The same freedom, just use case diagrams and use case description, is given for the Requirement Specification phase in COMET [9], in sharp contrast with the detailed structure and the many constrained guidelines and notations for Analysis and Design. That level of freedom is, on the other hand, explicitly advocated, for example in [8], on the basis that experience matters more than stringent structuring and rules. There the underlying philosophy is admittedly the same of the Agile Methods Movement (see [18], for an interesting discussion and references). However, while we do not deny that highly skilled and experienced software developers perhaps need only loose guidelines and a liberal supporting notation, from our experience we have seen that, for less experienced people, such liberality is a source of endless discussions, contrasting choices and a proliferation of inconsistencies. Moreover, we believe that our "tight and precise" imperative and the related techniques may help from one side reduce the amount and the fuzzy verbosity of some documentation and on the other provide effective guidelines for passing to the design and then the implementation phase, though we have not yet explored all the later phases.

The approach taken in Catalysis [6], that in other details shows some similar general views to ours, is not directly comparable, being an overall transformational approach based on components that are refined from business modelling to implementation units. But definitely our way of structuring requirements is not targeted to a transformational approach; we are more interested in providing a separate step preliminary to devise in a rather structurally independent manner, a model-driven software architecture of the system. Indeed, we have already performed some experiments to pass from a requirement specification in the suggested form to a design document, for which too we have proposed a more tight and precise structuring. Our approach is totally compliant with the OMG Model Driven Architecture philosophy (see [12]) and it is within that framework that we intend to explore the connection with the implementation phase, passing from Platform Independent Models to Platform specific Models and then to code. A second more specific methodological difference is the strict and explicit separation between the Problem Domain Model and the system, in the line for example of [10]. That distinction was and is somewhat blurred in some classical and Object Oriented approaches, though revisited with UML (see, e.g., [13,8]) for very recent examples) . In other approaches that distinction has been reintroduced and phrased in the distinction between Business Modelling (e.g., in [14]) and Requirements.

On the more technical side there are a number of major distinctions with the extant work, namely

- the PDM structure, encompassing conceptual modelling and business modelling;
- the System Placement activity, that encompasses the search for the system boundary;
- the use of the Context View to make explicit the distinction between the system and its environment and as a basis for defining the requirements about the interaction of the system with its context;

- the explicit use of the concept (a class) of System, both in the context diagram and in the use case descriptions, where we specify the System behavior related to a specific use case with a statechart;
- the use of a very Abstract State, instead of the many optional use case states, to allow expressing abstract requirements about the interaction of the System and the context, without providing an object-oriented structuring at a stage when such a structure is not required and can be premature.

Notice that the use of the class System is not in direct contrast with the traditional object-oriented approaches, where the presence of such a class, at the level of analysis and design, is considered a typical naive student's mistake. Still, because of the fact that those approaches also at the requirement level start with an object structure, the presence of that class is most unusual. However the danger of providing a structure not immediately needed when defining the system requirements has been remarked by many authors (notably M. Jackson, see, e.g., [10]). Even more interesting, also in Catalysis, that claims to be completely object-oriented, a class system and a context diagram is used in the preliminary phases and it appears in the sequence diagrams explaining the role of the system (see [6, p.15, fig 1.16]). Of course the context diagram with the system initial bubble was the starting diagram in the Structured Analysis approach [20].

Finally we just mention that in our approach the choice and use of the UML constructs is guided by a careful semantic analysis (see, e.g., [15, 16]), that has led us to prevent and discourage the indiscriminate use of some features that, especially in combination, may have undesirable side-effects, like interferences and ambiguities.

References

1. Consistency Problems in UML-based Software Development: Workshop Materials. Technical report, Blekinge Institute of Technology (Sweden), 2002.
2. E. Astesiano and G. Reggio. Knowledge Structuring and Representation in Requirement Specification. In *Proc. SEKE 2002*. ACM Press, 2002. Available at `ftp://ftp.disi.unige.it/person/ReggioG/AstesianoReggio02a.pdf`.
3. E. Astesiano and G. Reggio. Consistency Issues in Multiview Modelling Techniques. Technical Report DISI–TR–03–05, DISI, Università di Genova, Italy, 2003. To appear in Proc. WADT 2002.
4. E. Astesiano and G. Reggio. Tight Structuring for Precise UML-based Requirement Specifications: Complete Version. Technical Report DISI–TR–03–06, DISI, Università di Genova, Italy, 2003. Available at `ftp://ftp.disi.unige.it/person/ReggioG/ReggioEtAll03c.pdf`.
5. E. Astesiano, G. Reggio, and M. Cerioli. From Formal Techniques to Well-Founded Software Development Methods. Technical Report DISI–TR–02–50, DISI, Università di Genova, Italy, 2002. Presented at *The 10th Anniversary Colloquium of the United Nations University International Institute for Software Technology (UNU/IIST): Formal Methods at the Crossroads from Panacea to Foundational Support*. Lisbon - Portugal, March 18-21, 2002. Available at `ftp://ftp.disi.unige.it/person/ReggioG/AstesianoEtAll02a.pdf`.
6. D. F. D'Souza and A. C. Wills. *Objects, Components, and Frameworks with UML: The Catalysis Approach*. Object Technology Series. Addison-Wesley, 1999.
7. G. Engels, J.M. Kuester, and L. Groenewegen. Consistent Interaction of Software Components. In *Proceedings of IDPT 2002*, 2002.
8. M. Fowler and K. Scott. *UML Distilled: Second Edition*. Object Technology Series. Addison-Wesley, 2001.

9. H. Gomaa. *Designing Concurrent, Distributed and Real-Time Applications with UML.* Addison-Wesley, 2000.
10. M. Jackson. *Software Requirements & Specifications: a Lexicon of Practice, Principles and Prejudices.* Addison-Wesley, 1995.
11. Pfleeger S. L. *Software Engineering: Theory and Practice.* Prentice Hall, 2001.
12. OMG Architecture Board MDA Drafting Team. Model Driven Architecture (MDA). Available at `http://cgi.omg.org/docs/ormsc/01-07-01.pdf`, 2001.
13. Stevens P. and Pooley R. *Using UML: Software Engineering with Objects and Components.* Addison-Wesley, 2000.
14. Rational. Rational Unified Process© for System Engineering SE 1.0. 2001.
15. G. Reggio, E. Astesiano, C. Choppy, and H. Hussmann. Analysing UML Active Classes and Associated State Machines – A Lightweight Formal Approach. In T. Maibaum, editor, *Proc. FASE 2000*, number 1783 in Lecture Notes in Computer Science. Springer Verlag, Berlin, 2000.
16. G. Reggio, M. Cerioli, and E. Astesiano. Towards a Rigorous Semantics of UML Supporting its Multiview Approach. In H. Hussmann, editor, *Proc. FASE 2001*, number 2029 in Lecture Notes in Computer Science. Springer Verlag, Berlin, 2001.
17. J. Sommerville. *Software Engineering: Third Edition.* Addison-Wesley, 1989.
18. DeMarco T and Boehm B. The Agile Methods Fray. *Computer*, pages 90–92, 2001.
19. UML Revision Task Force. *OMG UML Specification 1.3*, 1999. Available at `http://www.rational.com/media/uml/post.pdf`.
20. E. Yourdon. *Modern Structured Analysis.* Prentice-Hall, 1989.