# Knowledge Structuring and Representation in Requirement Specification

Egidio Astesiano - Gianna Reggio

DISI-Università di Genova, Italy

astes,reggio@disi.unige.it

## Abstract

On the basis of some experience in the use of UML-based methods, we believe that a more refined and stringent structuring of the knowledge in the Requirement Specification may help the specification process and make easier the consistency checks among the various components. Thus we propose a way of structuring and representing the Requirement Specification artifacts that presents a number of novelties w.r.t. the best-known current methods. Our proposal is multiview, use case-driven and UML-based; thus also object-oriented. However, also benefitting of some earlier work, notably in the Structured Analysis, we take a rather abstract view, trying to avoid a preemptive decision on the classes structuring the system to build; that is achieved not only making a sharp distinction between business/domain modelling and the system, but also dealing with the system at the requirement level as a black box, providing only the minimal structure needed to express the interactions with the context.

## 1 Introduction

Requirement capture and specification has been recognized as a paramount activity in every significant software development process model. 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 [9]. This is the case of software development process models such as RUP (the Rational Unified Process [5]), Catalysis [1] and COMET [3].

Our work tries to refine and complement those proposals especially in the way we structure the requirement specification activity and the resulting artifacts. Indeed, after some serious experiments in teaching and using those approaches, we are convinced that a more systematic and stringent structuring can help in two directions: make the process faster, cutting sometimes endless discussions, and, because of the tighter connections among the artifacts, support the consistency checks of the overall specification.

The method we propose, rigorously multiview, use-case driven and UML-based, presents some novelties, sometimes departing from traditional object-oriented dogmas and sometimes incorporating some good, but perhaps lost, ideas found in well-known methods, such as Structured Analysis [10], and in the work of some pioneers, such as M. Jackson's [4].

Indeed, central to our approach are the following three concepts:

- the total separation of the Domain Model from the System, a distinction somewhat blurred in many object-oriented approaches;

- the distinction between the System and the environment, formalized in a Context View, that will be the basis for the definition of the requirements about the interaction between the System and the context, in conjunction with the use case diagram;

- 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; of course, in our view, such System concept would disappear in the following Design stages (first Model-Driven and then Technology-Oriented).

Finally we also mention that our proposal relies on the use of a well-chosen subset of UML that can be given a rigorous semantic foundation, though we will not discuss that issue in this paper.

In the first section we outline our method, putting it into context and anticipating its novel features; moreover we introduce a small but significant running example on which we will illustrate its use. In the second

**PDM**

definition: Package

1

operates on

1..*

**(System) Requirement Specification**

1

realizes

1..*

**Model-Driven Design**

1

realizes using
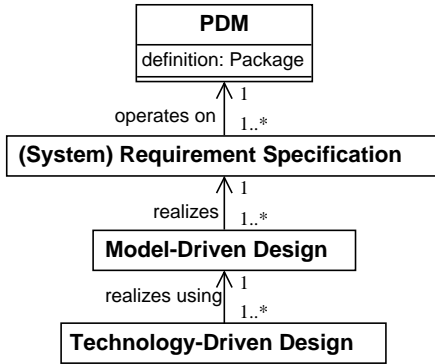
1..*

**Technology-Driven Design**

Figure 1: Artifacts

we will present in more detail the main features of the approach, with sample application to the running example. Finally we will conclude with comments and comparison to extant work.

Throughout the paper we assume a basic knowledge of the UML [9] notation, as can be found in [2].

# 2 Outline

The context of our work is sketchily represented in Fig. 1, where we present some essential steps (artifacts to be produced) in a modern software development process. We intend the Requirement Specification activity built over the Problem Domain Modelling (PDM) and preliminary to Model-Driven Design, followed by Technology-Driven design. As currently widely advocated, by Model-Driven Design we intend the activity of providing a solution of the problem in terms of Model-Driven Architecture (see, e.g., [8]),namely an architecture based on the abstract modelling and independent of the implementation platform, to which is targeted the Technology-Driven Design.

We do not deal here with those activities, nor with the Domain Modelling. Still we intend to stress that in our approach Requirement Specification is the first activity in which the (Software) System is taken into consideration. Currently Domain Modelling is covered in most methods by the so-called Business Modelling activity (as, e.g., in RUP [5]), but we prefer to keep the name Problem Domain Modelling to stress that for us Domain Modelling does not include any aspect of the System.

The Requirement Specification structure and activity we propose assumes that the Problem Domain Modelling produces as an artifact a UML package, thus describing in an object-oriented fashion the part of the real world that concerns the System, but without any reference to the System itself nor to the problem to

**UseCase Behaviour View**

definition: Statechart

**UseCase Interaction View**

definition: Collaboration

**UseCase Causal View**

definition: ActivityDiagram

1 * 0..1

**UseCase Description**

textualDescription: String

1..*

**Internal View**

definition: Package

**UseCase View**

summary: UseCase Diagram

**Context View**

definition: Package

1 1 1

**Requirement Specification**
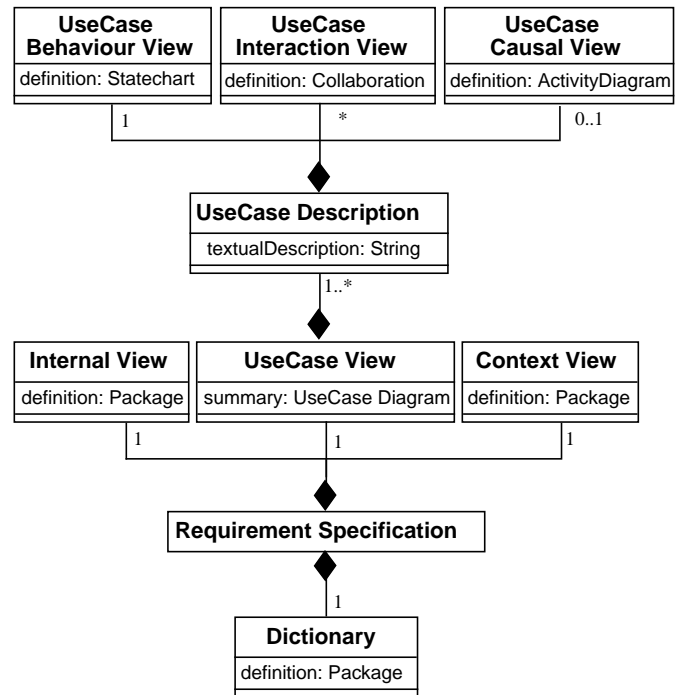
1

**Dictionary**

definition: Package

Figure 2: Requirement Specification Structure

which the System provides a solution.

In our approach the Requirement Specification artifacts consist of different views of the System, plus a part, Dictionary, needed to give a rigorous description of such views. Its structure is shown in Fig. 2 by a UML class diagram.

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*).

In our approach 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 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 in which way some info is sent or received by them.

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 de-

picted 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).

Dictionary lists and makes precise all entities 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.

The guidelines, summarized in Fig. 3, of our method for capturing the requirements and giving their specification following the above schema are as follows.

– Give the Use Case Diagram

– Give an initial version of Dictionary specializing PDM and including all the data whose need is unquestionable.

– Give an initial version of the Context View (only class names and associations).

– Find what you know on the context entities (operations, assumptions on their behaviours,...) and specify that by extending the Context View.

– Give an initial version of the Internal View.

– For each use case in the Use Case Diagram Give its views, possibly extending and updating the Dictionary, the Internal View and the Context View.

**A running example**   *We will illustrate our method by means of a simple, but significant enough, example: the Algebraic Lottery.*

*Our lottery is said "algebraic" since the tickets are numbered by integer numbers, the winners are determined by means of an* order *over such numbers, and a player buys a ticket by selecting its number.*

*Whenever a player 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*
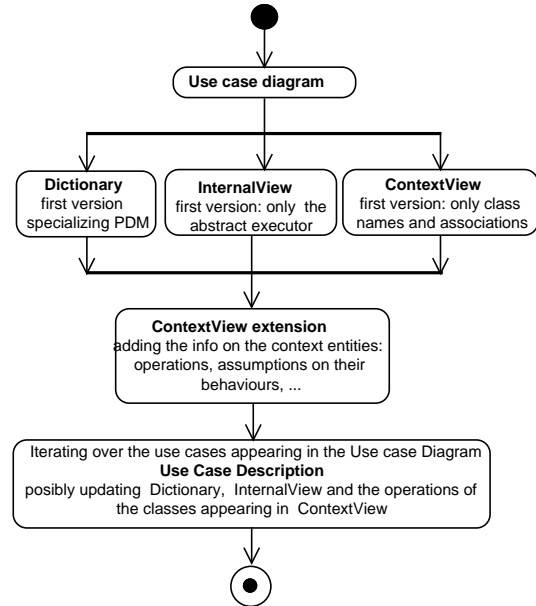


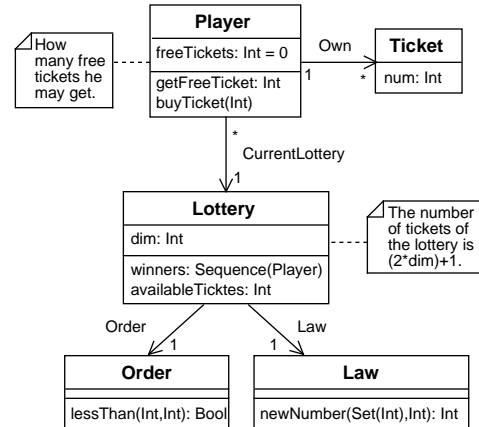Figure 3: Requirement Specification Tasks



Figure 4: Algebraic Lottery: PDM

*players 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 the lottery.*

*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, becoming players; and players access the system in a session-like way. An external service takes care of the registration of the players and of the distribution of the session keys.*

In the case of our example, the result of the PDM activity is quite simple (just the rules governing the lottery), but it serves to illustrate the distinction w.r.t.

the System and introduces some terminology that will be included in the Dictionary package. Instead the System has to support the players in accessing the lottery and buying the tickets, and the "lottery manager" in doing the administrative tasks, as drawing the winners, and launching new lotteries.

The PDM consists of a class diagram shown in Fig. 4 and of some constraints, expressed in OCL, defining the various operations, As an example let us consider the constraint that expresses under which condition a free ticket may be given out to a player (he has the right to receive it, at least half of the tickets of the lottery are already been assigned, and there is still some non-assigned ticket), and what are precisely the effects of that.

```
context getFreeTicket(): Int
pre:
   self.freeTickets > 0 and
   self.CurrentLottery.availableTicktes() <
      self.CurrentLottery.dim  and
   self.CurrentLottery.availableTicktes() > 0
post:
let existingTickets = Ticket.allInstances@pre in
   self.freeTickets = self.freeTickets@pre-1 and
   Ticket.allInstances->exits{ T |
      existingTickets->excludes(T) and
      T.num = self.CurrentLottery.Law.newNumber(
               existingTickets.num,
               self.CurrentLottery.dim) and
      self.Own->includes(T)  and
      result = T.num }
```

Similarly, the constraint below states that a ticket may be sold only once, and that a player who buys a ticket gets the right to a free ticket.

```
context buyTicket(N: Int):
pre:
   Ticket.allInstances.num->excludes(N)
post:
   self.freeTickets = self.freeTickets@pre+1 and
   Ticket.allInstances->exits{ T |
      Ticket.allInstances@pre->excludes(T) and
      T.num = N and
      self.Own->includes(T)}
```

# 3 Requirement Specification: Tasks and Artifacts

In this section we describe in some details the tasks and the artifacts of our method (see Fig. 3 and 2) and show how they are applied to the running example case.
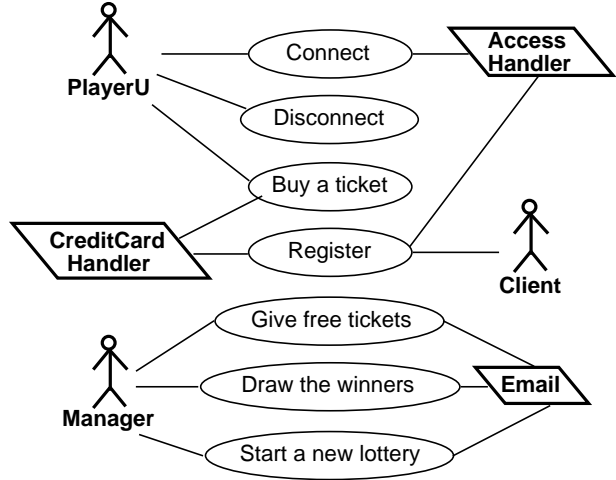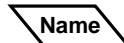


Figure 5: Algebraic Lottery: Use Case Diagram

## 3.1 Use Case Diagram

Our method is Use Case Driven; indeed our requirement specification includes a Use Case View consisting of a Use Case Diagram and of the descriptions of the various use cases. But, for us the actors appearing in the Use Case diagram are roles for **any kind** of entities outside the System interacting with it; moreover they are of two kinds represented by the stereotypes <<SU>> (User of the Services provided by the System) and <<SP>> (Providers of the Services needed by the System). We think that in this way we can avoid long discussions about who/what are the actors.

An <<SU>> named Name  will be visually represented by ![] ; whereas an <<SP>> named Name will be visually represented by ![Name] .

The Use Case Diagram for the lottery case is shown in Fig. 5. It depicts that our System will use external services for handling the credit cards and the accesses to the system. Furthermore, it will use the email to communicate with the players.

## 3.2 Dictionary (First version)

The Dictionary consists of a UML package describing all the entities needed to present the use cases, the context and the internal structure of the System. This global description helps avoid that the same entity is modelled differently in different views or in different use cases.

The Dictionary specializes PDM, since we think that the conceptual entities found in the problem domain should be used to describe what the System should do.
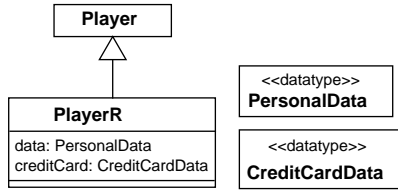
Figure 6: Algebraic Lottery: Dictionary version 1

Notice that this does not mean that all classes appearing in it will be used in the requirement specification. Here, for example, the class Player is not used anymore, since it corresponds to the abstract concept of the player. Instead, in the requirement specification we have two other classes PlayerU, to model the player context entities, and PlayerR (for player Record) to model the information on a player kept by the System to perform its activities. In Fig. 6 we show this initial version of Dictionary, where we state that to specify the System requirements we surely will use personal and credit card data.

## 3.3 Context View (First version)

The Context View is a UML package importing the Dictionary containing at least a class diagram, where all classes are all of the following three stereotypes

<<**System**>> that stands for the System (exactly one class in the diagram with this stereotype);

<<**SU**>> that stands for a user of the services provided by the System;

<<**SP**>> that stands for a provider of services used by the System.

Moreover, in such class diagram there is one binary association from the System class into each other class, and these are all the associations.

Because the actors in the Use Case Diagram are just generic instances of <<SU>> and <<SP>> classes appearing in this view, they should be in accord.

The operations of these classes model their mutual interfaces, that is in which way they may interact (being in an OO setting, an interaction is just the call of an operation).

<<SU>> and <<SP>> classes, differently from the <<System>> class, may have also attributes, constraints and a description of the behaviour. Such features represent the assumptions on the behaviour of their instances on which the System relies.

Initially, we give an initial simplified version of the Context View including only the class names and the
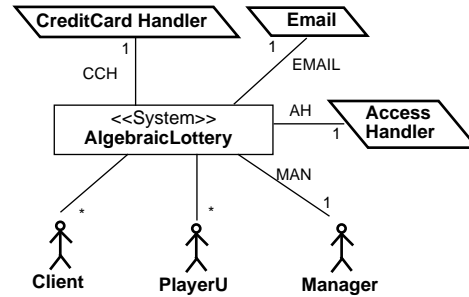


Figure 7: Algebraic Lottery: Context View version 1

associations. Such initial version of the Context View for the lottery case is reported in Fig. 7.

## 3.4 Internal View (First Version)

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 abstractly describing the System Abstract State.

We represent such structure by a class diagram containing exactly one class of the stereotype <<A_Executor>> (without attributes and whose operations are in bijective correspondence with those of the <<System>> class appearing in the Context View), and several passive classes defining the parts of the System Abstract State. For what concerns the associations, it contains only a binary navigable association between the <<A_Executor>> class and each of the passive classes.

This class diagram describes implicitly also the "Abstract State" of the <<System>> class appearing in the Context View in the following way: for each association in the diagram | <<A_Executor>> | —A→ | C | such class has an attribute A: Bag(C).

Technically, the Internal View is a UML package importing the Dictionary and containing only a class diagram.

In the algebraic lottery case the initial version of the Internal View consists only of the abstract executor, and we do not show it since it is trivial.

## 3.5 Context View (Extension)

This task requires to find out what we know on the context entities (assumptions on their behaviours, on how they interact, ...) and to specify that by extending the initial version of the Context View. Notice that to this aim we can also partly define the operations of the class <<System>>, since this is the only way to
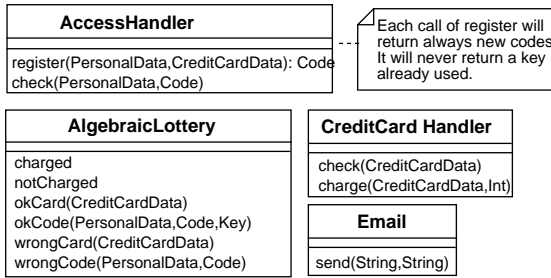
5

| **AccessHandler** |
| register(PersonalData,CreditCardData): Code |
| check(PersonalData,Code) |

Each call of register will return always new codes. It will never return a key already used.

| **AlgebraicLottery** |
| charged |
| notCharged |
| okCard(CreditCardData) |
| okCode(PersonalData,Code,Key) |
| wrongCard(CreditCardData) |
| wrongCode(PersonalData,Code) |

| **CreditCard Handler** |
| check(CreditCardData) |
| charge(CreditCardData,Int) |

| **Email** |
| send(String,String) |

Figure 8: Algebraic Lottery: Context View (Extension)

model that, e.g., a <<SP>> will answer in some way to a request.[1]

It is important to do this task before to describe the use cases, since the context entities are not under the responsibility of the System developer, but they are already existing.

This extension in the case of the lottery, see Fig. 8, defines the operations of the <<SP>> classes, and states some property on the behaviour of AccessHandler. Moreover, here we have added also some operations to the class AlgebraicLottery, just those corresponding to receiving some communications by the <<SP>> context entities.

## 3.6   Use Case Description

A Use Case Description consists of a textual presentation of the use case, and of one or more use case views, which may be of the three kinds: behaviour, interaction and causal.

**Use Case Behaviour View**   Such view, that is mandatory for each use case, is defined by a statechart for the <<System>> class describing the complete behaviour of the System with respect to such use case.

A statechart defining a behaviour view must be such that

- the conditions on its transitions may test only the System Abstract State given in the Internal View;

- the actions appearing on its transitions may include only calls of the the operations of the context entity classes, as defined in the Context View, and may update the System Abstract State;

---

[1] This is a problem of OO, where a class is characterized in some sense only by what can receive, but there is no way to express what it can output.

- and the events on its transitions may be only call events of the operations of the <<System>> class, as presented in the Context View.

The behaviour view is a "complete" description of what the System do in the use case. The UML state chart will describe how the System reacts to all the possible communications (concerning the use case) received by the context entities; moreover, also the conditions (on the Abstract State of the System) under which a reaction is possible and its effect (on the Abstract State of the System and on the context entities) is precisely described. These last points are just the condition and action part of the state chart transitions. This is quite differently from other approaches, e.g., COMET [3], where a use case is described by collaborations, which correspond to some scenarios, and where there is no way to make explicit under which condition some thing is done.

**Use Case Interaction View**   Such view is defined by a sequence (or collaboration) diagram representing the interactions happening in a scenario of the use case among

- the context entities

- the System

- the internal abstract constituents of the System, as presented in the Internal View (the abstract executor and the passive components).

Technically, a Use Case Interaction View is a standard UML interaction diagram, but the object lives appearing in it correspond to objects in two distinct worlds: inside the System (the abstract executor and the passive components), and outside the System (the context entities).

In the visual representation we split the diagram in two swimlanes, by a dashed line, to show what is happening inside and outside the System, and the lifeline corresponding to the abstract executor will be marked by a big **E**.

There are no restrictions on the number of the diagrams appearing in this view, but they must be coherent with the behaviour view (that is, they must represent particular executions of the complete behaviour described by such view).

**Use Case Causal View**   Such view is defined by 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 operations of **System** by the context entities,

- calls of the operations of the context entities by **System**,

- UML actions producing side effects on the **System Abstract State**.

The guards on such diagram can be only conditions 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.

Recall that the operations of **System** and of the context entities are described in the **Context View**, and that the **System Abstract State** is described in the **Internal View**.

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 operation of which **System** reacts.

As a guideline, in general it is better to start giving some interaction views (e.g., the one for the basic non-problematic scenario) whenever the use case requires the participation of many context entities or the causal view, if instead the use case mainly concerns the components of the **Abstract State** of **System**, and after give the behaviour view.

We suggest to describe first the use cases in which <<SU>> actors take part, and to consider for each actor one after the other those use cases to which it participates.

We illustrate the **Use Case Description** on two use cases of the lottery example, **Register** and **Give free tickets** (see Fig. 9 and 10).

**Register** The description of the use case **Register** is shown in Fig. 9. We have given the mandatory behaviour view and an interaction view corresponding to the standard scenario (i.e., where the data given by the client are correct). While giving that description, we have modified the **Dictionary** (adding the class **Code**,

**textual** A client may register by giving his personal data and those of a credit card. If his data are correct and those about the credit card are accepted by its handler, then he will receive a code, determine by the access handler, and will be considered registered; otherwise he will be informed that his registration has been refused.
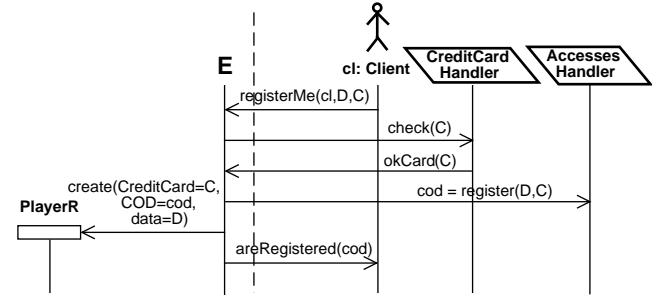
**behaviour**
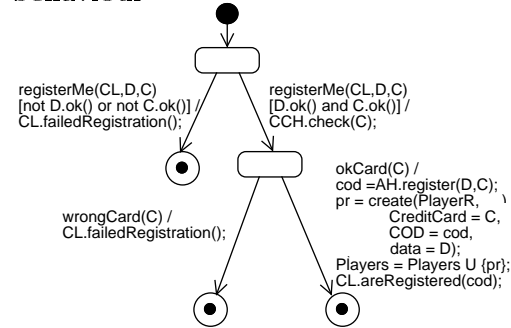


**interaction**



Figure 9: Use Case **Register** Description

the Ok operation to classes CreditCardData and Person-alData and the attribute COD to the class PlayerR), the Context View (by adding to the class AlgebraicLottery and Client the operations corresponding to the messages exchanged in this use case), and Internal View by adding the Abstract State component made by the records about the registered players.

**Give free tickets** For the description of the use case Give free tickets (see Fig. 10), together the behaviour view we give also a causal view; in this case we do not give any interaction view because it is not simple to model the complex activity of <<System>> while cycling to give one ticket after the other, whereas the causal view is quite clear and readable.

In the appendix we show the descriptions of the remaining use cases and the final version of the Dictionary, Internal View and Context View.

# 4  Conclusion and related Work

Our contribution, limited to the specific issue of Requirement Specification, 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, and can be seen as a variation or, better, a complement to some of them. Still, it departs from them, at least to our knowledge, in some important points.

First, the overall major goal is to propose a more systematic approach, in the sense that the structure of the overall specification artifact is constrained in order to tightly relate the components and have at end a number of consistency checks. This contrasts with the almost total freedom given, for example in RUP [5], where the structure is just based on the use case description. That level of freedom is, on the other hand, explicitly advocated, for example in ([2]), on the basis that experience matters more than stringent structuring and rules. However, while we do not deny that highly skilled and experienced software developers perhaps need only loose guidelines and a liberal supporting notation, form our experience we have seen that, for less experienced people, such liberality is the source of endless discussions, contrasting choices and a proliferation of inconsistencies. The same freedom, just use case diagrams and use case description, is given for the Requirement Specification phase in COMET [3], in sharp contrast with the detailed structure and the many constrained guidelines and notations for Analysis and Design. The approach taken in Catalysis [1], that in other details shows some similar general views

**textual** The manager checks how many tickets are still available. If they are less than half of the tickets of the current lottery, then he may distribute some free tickets among the players. During each distribution a player may receive at most one ticket. The numbers of the free tickets are determined by the current "law" applied to the set of the numbers of the tickets already given out.
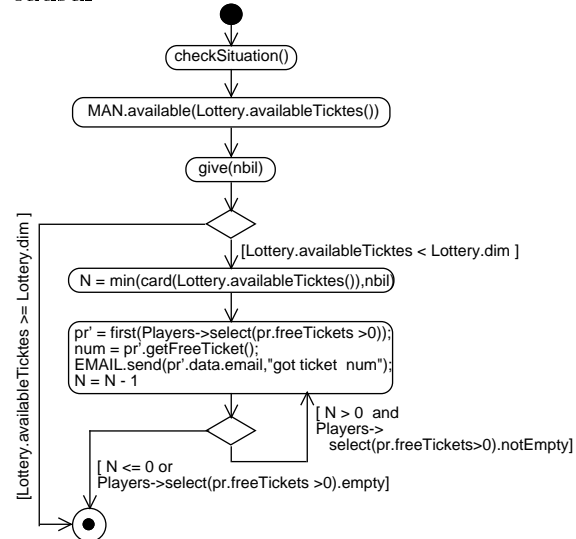
**behaviour**
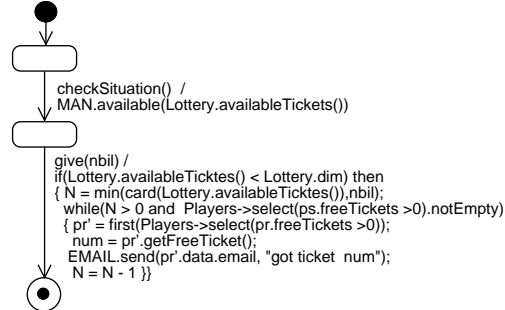


**causal**



Figure 10: Use Case Give free tickets Description

to ours, is not directly comparable, being based on 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.

A second major distinction is 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. This is not in direct contrast with the traditional object-oriented approaches, where the presence of such a class is considered a typical naive student's mistake, but at the level of analysis and design. 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., [4]). 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 [1, p.15, fig 1.16]) of course the context diagram with the system initial bubble was the starting diagram in the Structured Analysis approach [10].

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., [6, 7]), that has led us to prevent and discourage the indiscriminated use of some features that, especially in combination, may have undesired side-effects, like interferences and ambiguities.

# References

[1] D. F. D'Souza and A. C. Wills. *Objects, Components, and Frameworks with UML: The Catalysis Approach*. Object Technology Series. Addison-Wesley, 1999.

[2] M. Fowler and K. Scott. *UML Distilled: Second Edition*. Object Technology Series. Addison-Wesley, 2001.

[3] H. Gomaa. *Designing Concurrent, Distributed and Real-Time Applications with UML*. Addison-Wesley, 2000.

[4] M. Jackson. *Software Requirements & Specifications: a Lexicon of Practice, Principles and Prejudices*. Addison-Wesley, 1995.

[5] Rational. Rational Unified Process© for System Engineering SE 1.0. 2001.

[6] 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.

[7] 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.

[8] R. Soley and OMG Staff Strategy Group. Model Driven Architecture. Available at `ftp://ftp.omg.org/pub/docs/omg/00-11-05.pdf`, 2000.

[9] UML Revision Task Force. *OMG UML Specification 1.3*, 1999. Available at `http://www.rational.com/media/uml/post.pdf`.

[10] E. Yourdon. *Modern Structured Analysis*. Prentice-Hall, 1989.

# Appendix

**PDM Constraints**

```
context P: Player inv:
   P.freeTickets >= 0

context Ticket inv:
   There do not exist two tickets with the
   same number

context O: Order inv:
   "x < y iff  O.lessThan(x,y)" is a total order

context newNumber(ins,j):
pre: {-j, ..., +j} - ins <> {}
post:
   ins->excludes(resut) and
   -j =< result  and  result =< j

context L: Lottery inv:
   Lottery.allInstances->size = 1 and
   L.dim = 5000 * k con K >= 1  and
   Ticket.allInstances.num->
      forall{ N | -L.dim =< N and N =< L.dim}
```

9

```
context winners():
pre: self.availableTicktes() = 0
post:
    the result is the list, also with repetitions,
    made by the h players (h = integer part of
    self.dim/5000) owing the tickets whose numbers
    are the first h with respect to self.Order

context availableTicktes():
post: ((self.dim *2)+1) - Ticket.allInstances->size
```
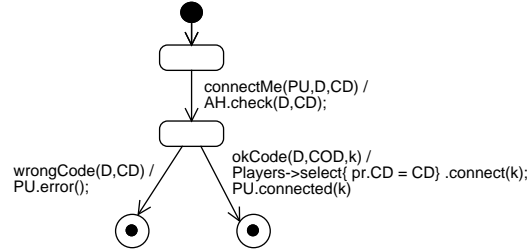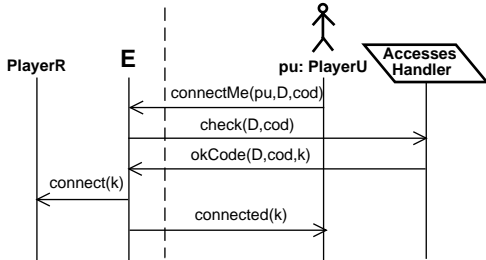
## Use Case **Connect** Description

**textual** A registered player may connect himself to
the lottery system to play by giving the code received
at the registration time.

**behaviour**



**interaction**



## Use Case **Disconnect** Description

**textual** A connected player may disconnect from the
lottery system in any moment; from then he cannot
play till the next connection.

**behaviour**
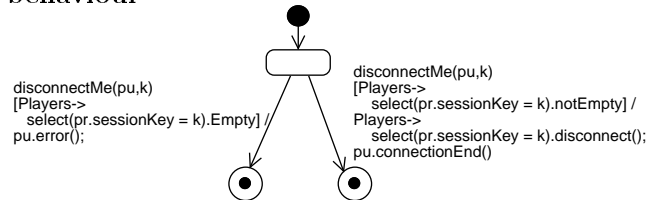


**interaction**
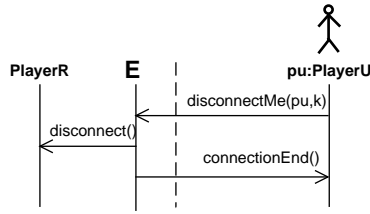


## Use Case **Buy a ticket** Description

**textual** A connected player, after having examined
the available tickets, buys a ticket by choosing its num-
ber, which must identify an available ticket. The price
of the ticket will be charged on the credit card given
at the registration time. By buying a ticket the player
gets the right to receive another free ticket at a certain
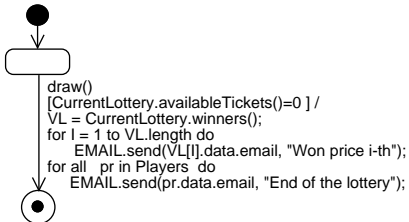time in the future depending on the manager decision.
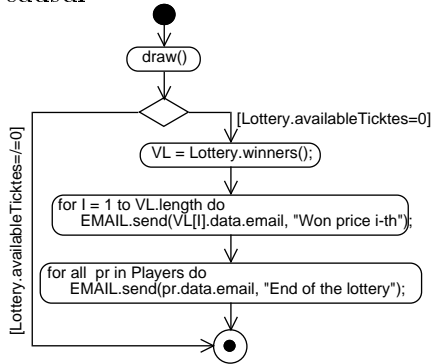
**behaviour**
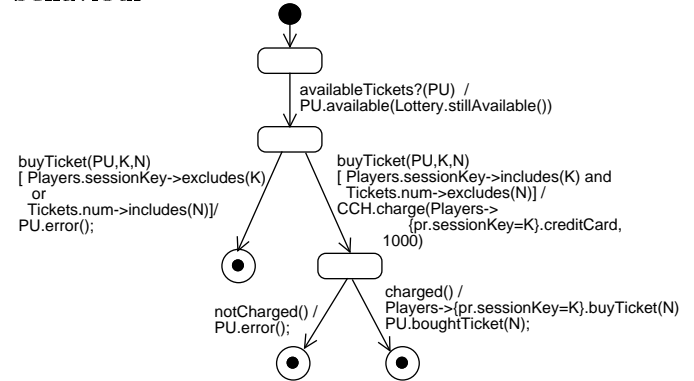


**interaction**
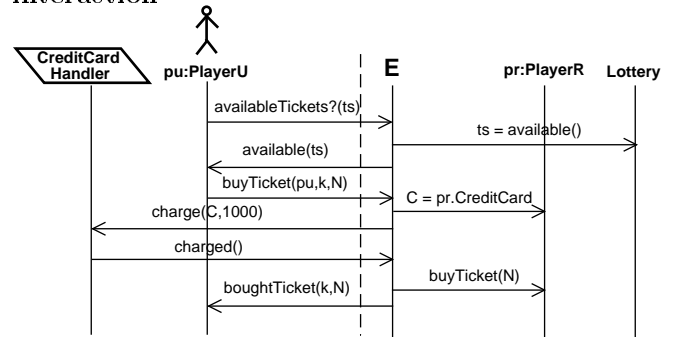


## Use Case **Draw the winners** Description

**textual** When all the tickets of the lottery have been
given out, the winners are drawn; they are precisely
the owner of the first k tickets (where k = integer part
of the lottery dimension divided by 10000) using the
current order. The owner of the winning tickets will
be informed by using the email, while all the registered
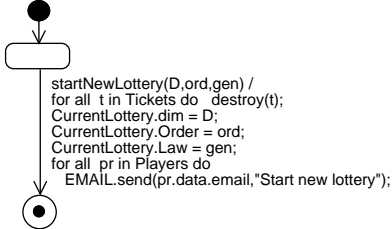players will be informed of the drawn.
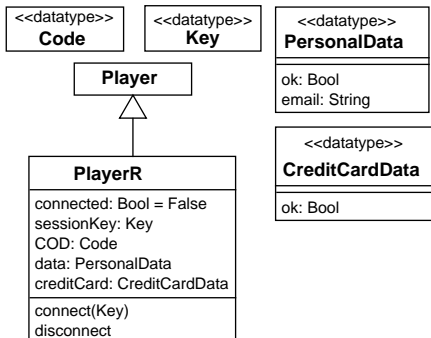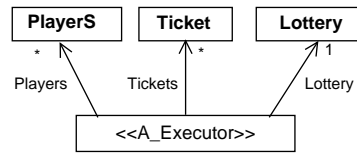
**behaviour**

causal



```
context connect(K: Key):
pre: self.connected = False and
     non exists a player P s.t. P.sessionKey = K
post: connect(K: Key):
     self.connected = True and self.sessionKey = K

context  disconnect:
pre: self.connected = True
post: self.connected = False
```

## Internal View Final version



## Context View Final version



## Use Case **Start a new lottery** Description

**textual** When no lottery is running, the manager may start a new one. First, he determines the dimension of the lottery (a natural greater than 1), 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 an order on integers, which will be used to find the winners. All the registered players, will then be informed of the new lottery by an email message.

**behaviour**



## Dictionary Final version