A Requirements Engineering Approach for Object-Oriented Conceptual Modeling

Information Systems and Computation Department
Valencia Polytechnic University

by Emilio Insfrán Pelozo

A thesis submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Science

October 2003
OlivaNova Model Execution is a trademark of CARE Technologies S.A.
ONME is a trademark of CARE Technologies S.A.
OlivaNova STAR is a trademark of CARE Technologies S.A.
OASIS is a trademark of CARE Technologies S.A.
OMG is a trademark of the Object Management Group.
Thesis Data

Thesis title: A Requirements Engineering Approach for Object-Oriented Conceptual Modeling

Presented by: Emilio Insfrán Pelozo (einsfran@dsic.upv.es)

Supervised by: Dr. Óscar Pastor

Institution: Valencia Polytechnic University. Spain

Department: Information Systems and Computation


Thesis Committee:

President: Dr. D. Antoni Olivé
            Full Professor. Dept. de Llenguatges i Sistemes Informàtics.
            Universitat Politècnica de Catalunya. Spain.

Secretary: Dr. D. Juan Sánchez Díaz
            Associate Professor. Information Systems and Computation Department.
            Valencia Polytechnic University. Spain.

Vowels:
            Dr. D. Roel Wieringa
            Full Professor. Department of Computer Science.
            University of Twente. The Netherlands.

Dr. D. Stephen Liddle
            Associate Professor. School of Accountancy and Information Systems.
            Brigham Young University. USA.

Dr. D. Hans-Dieter Ehrich
            Full Professor. Institut für Informationssysteme.
            Technical University of Braunschweig. Germany.

Summa cum laude
European PhD distinction Ref. Num.: DE-0017-43424.
Abstract

Many different attempts have been made out over the last three decades to tackle the problems in software engineering that directly or indirectly lead to the symptoms of software crisis. These attempts have mostly focused on different specific fields, such as improved programming languages, improved modeling techniques, introduction of analysis and design methods, formal specifications, CASE tools, etc. In spite of the various attempts to develop high quality software systems, a difficult task still remains to be able to provide a solution to this classical problem: how to go from the problem space (user requirements) to the solution space (design and implementation) with sound methodological guidance.

This PhD thesis is developed in the context of OO-Method (an object-oriented method for conceptual modeling and code generation) providing an approach to undertake this problem by extending the OO-Method Conceptual Model with a Requirements Model. The two cornerstones of this approach are the Requirements Model, which provides mechanisms to identify and specify functional user requirements at different abstraction levels, and the Requirements Analysis Process, which establishes conceptual links between the requirements and the structure and behavior of an OO-Method Conceptual Schema. These provide a smooth transition from a requirements specification to a conceptual schema in a traceable way. An industrial Requirements Specification Environment is being developed as a supporting tool. The integration of this tool with the automatic Conceptual Model Execution Environment provided by the OO-Method Case Tool (OlivaNova Model Execution Software®) will bring an achievable dream closer to reality, that is, to be able to go from requirements to code.
Resumen

En las tres últimas décadas se han realizado muchos intentos para minimizar los problemas en la Ingeniería del Software que, directa o indirectamente, producen los síntomas de la llamada crisis del software. Estos intentos se centran en su mayoría en áreas específicas del proceso de desarrollo de software, tales como, mejoras en los lenguajes de programación, mejoras en las técnicas de modelado, la introducción de métodos de análisis y diseño, especificaciones formales, herramientas CASE, etc. A pesar de todos estos intentos, la difícil tarea de pasar del espacio del problema (requisitos de usuario) al espacio de la solución (diseño e implementación) con una guía metodológica clara y bien definida sigue siendo una tarea pendiente.

En esta tesis, este problema se trata en el ámbito de OO-Method (un método de modelado conceptual orientado a objetos y generación automática de código) proporcionando una aproximación metodológica que extiende el Modelo Conceptual de OO-Method con un Modelo de Requisitos. Las dos piedras angulares de esta tesis son el Modelo de Requisitos, propiamente dicho, que provee los mecanismos necesarios para identificar y especificar requisitos funcionales de usuario a distintos niveles de abstracción, y el Proceso de Análisis de Requisitos, que establece vínculos conceptuales entre los requisitos especificados y la estructura y comportamiento de un Esquema Conceptual de OO-Method. De esta forma la transición desde la especificación de los requisitos funcionales de usuario a elementos del modelado conceptual se define con un proceso guiado y automatizable.

Para dar soporte al método propuesto se está desarrollando una herramienta CASE comercial de forma conjunta entre la UPV y CARE Technologies S.A. La integración de esta nueva herramienta con el entorno de generación automática de código proporcionado por la herramienta CASE de OO-Method (OlivaNova Model Execution Software®) ayudará a dar soporte al proceso de transformación de requisitos funcionales de usuario en código ejecutable.
Resum

En les tres últimes dècades s'han realitzat molts intents per a minimitzar els problemes en l'Enginyeria del Programari que, directa o indirectament, produeixen els símptomes de l'anomenada crisi del programari. Aquests intents se centren en la seua majoria en àrees específiques del procés de desenvolupament de programari, tals com, millores en els llenguatges de programació, millores en les tècniques de modelatge, la introducció de mètodes d'anàlisi i disseny, especificacions formals, eines CASE, etc. A pesar de tots aquests intents, la difícil tasca de passar de l'espaie del problema (requisits d'usuari) a l'espaie de la solució (disseny i implementació) amb una guia metodològica clara i ben definida continua sent una tasca pendent.

En aquesta tesi, aquest problema es tracta en l'àmbit d'OO-Method (un mètode de modelatge conceptual orientat a objectes i generació automàtica de codi) proporcionant una aproximació metodològica que estén el Model Conceptual d'OO-Method amb un Model de Requisits. Les dues pedres angulares d'aquesta tesi són el Model de Requisits pròpiament dit, que proveeix els mecanismes necessaris per a identificar i especificar requisits funcionals d'usuari a distints nivells d'abstracció, i el Procés d'Anàlisi de Requisits, que estableix vincles conceptuels entre els requisits especificats i l'estructura i comportament del Model Conceptual d'OO-Method. D'aquesta forma la transició des de l'especificació dels requisits funcionals d'usuari a elements del modelatge conceptual es defineix amb un procés guiat i automatizable.

Per a donar suport al mètode proposat s'està desenvolupant una eina CASE comercial de forma conjunta entre la UPV i CARE Technologies S.A. La integració d'aquesta nova eina amb l'entorn de generació automàtica de codi proporcionat per la eina CASE d'OO-Method (OlivaNova Model Execution Software®) ajudarà a donar suport al procés de transformació de requisits funcionals d'usuari en codi executable.
Acknowledgements

A journey is easier when you travel together. This thesis is the result of many years of work during which I have been accompanied and supported by many people. It is a great pleasure for me to now have the opportunity to express my gratitude to all of them.

The first person I would like to thank is my advisor. I joined his project in 1995 and since then, we have been involved in many projects, trips, meetings, and so on.

This research was supported by various organizations including CARE Technologies S.A. and Consoft S.A. I would like to thank them all: Siegfried Borho, Emilio Iborra, Pepe Iborra, José Miguel Barberá, Juan Carlos Molina, Pedro Juan Molina, Sofía Martí, Elena Tejadillos, Margarita Burbano and many others.

I would also like to thank the following:

Roel Wieringa, who arranged a ten-month stay at the University of Twente (the Netherlands) in 1999. Taking into account his overloaded schedule, the almost weekly meeting, and his advise are really appreciated. Stephen W. Liddle, who arranged a three-month stay at Brigham Young University (USA) in 2001. He and his family made me feel at home. Some chapters of this thesis were written during my stay there.

Pele, a colleague and friend, for sharing many constructive discussions, for all the late nights writing papers, and for his advise. Isidro Ramos, who gave me the opportunity to join the research group when I arrived to Valencia, and for all his careful advise. All my friends and colleagues from the OO-Method research group: Juan, Juanjo, Manoli, Jorge, Pedro, Marta, Hugo, Alicia, Isabel, Vicky and Nelly for their collaboration. My Final Course Project students, who helped me defining some aspects of this thesis: Enric, Elena, Carlos and Silvia Belda.

The administrative staff of my department: Ana, Toni, Amparo and Rosa who always efficiently manage my trips, projects, grants, etc.

My wife, colleague, and friend Silvia for being there… always. Her day-by-day support and friendship make life easier, thanks!

My family, that shares my happiness and sadness from the distance: Mom, Carol and Aníbal, thanks!

And finally, my father who is no longer with us.
# Contents

1. Introduction ................................................................................................................. 17  
   1.1. Context of this Thesis ......................................................................................... 19  
   1.2. Objectives ........................................................................................................... 21  
   1.3. Structure of the thesis ......................................................................................... 21  
2. Information Systems Development ............................................................................. 25  
   2.1. Information System Concepts ............................................................................. 26  
      2.1.1. Definitions ...................................................................................................... 27  
      2.1.2. Models, Systems, and Information Systems ................................................. 29  
   2.2. Requirements Modeling ....................................................................................... 33  
      2.2.1. Relationships between the Subject and System Worlds ......................... 34  
      2.2.2. Relationships between the Usage World and the System World... 35  
3. Requirements Model .................................................................................................... 45  
   3.1. Foundations ........................................................................................................ 45  
      3.1.1. External Interaction ....................................................................................... 46  
      3.1.2. Internal Decomposition ............................................................................... 47  
   3.2. Requirements Model Architecture ...................................................................... 50  
      3.2.1. Mission Statement ....................................................................................... 50  
      3.2.2. Function Refinement Tree ........................................................................... 51  
      3.2.3. Use Cases ..................................................................................................... 52  
   4.1. Conceptual Model ............................................................................................... 61  
      4.1.1. Object Model ............................................................................................... 62  
      4.1.2. Dynamic Model ........................................................................................... 64  
      4.1.3. Functional Model ........................................................................................ 66  
      4.1.4. Presentation Model ...................................................................................... 67  
   4.2. OASIS Formal Specification ............................................................................... 68  
      4.2.1. The Formal Specification as the System Repository ................................. 73  
   4.3. Execution Model ................................................................................................. 74  
5. Requirements Analysis Process .................................................................................. 77  
   5.1. Identification and Allocation of Responsibilities ............................................... 77
7.1.2. OlivaNova STAR ................................................................. 145
7.2. RETO Metamodel ............................................................... 146
7.3. RETO Architecture ........................................................... 148
7.4. RETO User Interfaces ......................................................... 149
  7.4.1. Form-based user interface ............................................... 149
  7.4.2. Icon-based user interface ................................................ 153
8. Conclusions ......................................................................... 157
  8.1. Contributions ..................................................................... 159
  8.2. Related Publications .......................................................... 160
  8.3. Future and Current Research ............................................... 164
List of Figures ........................................................................... 167
List of Tables ............................................................................. 171
Bibliography .............................................................................. 173
Appendix A. Traceability Rules Catalog ..................................... 183
Appendix B. Case Study: Car Rental .......................................... 193
1. Introduction

The introduction of the first digital computers in the 1940s may be considered as the initiation of the history of software development. In these early days, the first software programs were written in machine language and were basically developed for numerical calculations in military projects. Later, with the introduction of the first programming languages in the late 1940s and the early 1950s, software development has undergone several evolutionary changes, which provided opportunities for building larger and more complex software intensive systems. This increased potential was soon followed by the realization that software was difficult to deliver on time, within the available budget and with the required quality factors such as reliability, stability, and adaptability. To cope with this so-called software crisis, an engineering approach to software development was proposed at the NATO Conference on Software Engineering [1] in 1968. The inspirational term of engineering implied that software development should be based on the conceptual foundations on which the other engineering disciplines rely.

Many different attempts have been made in the last three decades to tackle the problems in software engineering that directly or indirectly lead to the symptoms of the software crisis. These attempts have focused on many different fields, such as improved programming languages, improved modeling techniques, introduction of analysis and design methods, formal specifications, CASE tools, etc. In spite of the various attempts to develop high quality software systems, a difficult task still remains to be able to provide an end to this classical problem: how to go from the problem space (user requirements) to the solution space (design and implementation) with sound methodological guidance. To do this properly, three main aspects must be considered:

- How to assure that user requirements are captured in a precise way
- How to represent them properly at the problem space level in a correct conceptual schema
- How these conceptual constructs are properly translated into their corresponding software representations at the solution space level, thus obtaining the desired final software product.

In order to consider the software production process as a true engineering activity, a comprehensive view of the set of phases involved must be introduced. This view should be global and complete. Global, in the sense that a clear relationship between the different steps must be defined, and complete because all the involved phases must be taken into account.

This thesis is concerned with how to improve the system development process in order to solve the issues [2] detailed below:

- Moving away from the chaotic level associated with conventional methods, where contradictions, ambiguities, incompleteness, and mixed levels of abstractions occur in a natural way.

- Improving understanding at the problem space level.

- Designing flexible methods and better CASE (Computer-Aided Software Engineering)/CARE (Computer-Aided Requirements Engineering) support, with the objective of reaching the solution space with a final software product which is compliant with the problem space description.

- Providing guidance in order to be able to deal with the requirements specification and conceptual modeling process in a structured way.

To overcome these shortcomings, we should move from traditional systems engineering to advanced Requirements Engineering, where the distinction between the problem space (what the problem is) and the solution space (how it is to be implemented in a particular software development environment) must be clearly established.

It is obvious that the problem space imposes strong requirements on the software production process. Taking into account that the Requirement Engineering process is complex, knowledge-intensive, experience-based, and requires highly intellectual and creative activity, it is essential to provide a clear way of capturing those requirements that are considered to be relevant in order to build a correct conceptual schema. This is done in an informal but structured manner and the drive is not technology-oriented but usage oriented. Finally, the requirements specification should reflect the views of the users and stakeholders rather than those of the system analysts.

With respect to conceptual modeling, an important decision is to determine the model to be used for dealing with the structure, functionality, behavior, and presentation modeling process. From our point of view, the object-oriented model is a good choice, due to its proximity to human cognitive mechanisms, due to the encapsulation of structural and behavioral aspects provided by the object notion, and due to the fact that the transition from the problem space to the solution space is smooth (we can specify objects in a conceptual schema and implement objects in the final software representation making traceability more structured). Another benefit of using the object-
oriented model is that it introduces modularization into the problem space, making reuse a good practice in the conceptual modeling process.

In this context, in order to deliver accurate, valid and complete specifications, this thesis has to address three main issues:

- Extending conceptual modeling in order to establish a conceptual link between requirements and system structure and behavior. As a result, the purpose of systems will become evident. This will ensure that purposeful systems will be built all the time.

- Deriving system functionality from the user point of view so that systems are adapted to the use that stakeholders envision for them. This will lead to better system acceptability in organizations.

- Defining a requirements engineering process in order to provide the methodological guidance for requirements modeling, as well as to facilitate the transformation of requirements into system specifications.

1.1. Context of this Thesis

This thesis was developed in the context of the Research Group “Logic Programming and Software Engineering” in the subgroup Object-Oriented Methods for Software Development Group (OO-Method Group) of the Valencia Polytechnic University (UPV – Universidad Politécnica de Valencia) in close collaboration with the company CARE Technologies S. A. (CARE Tech.).

The work presented in this thesis arises from the efforts of researchers at the OO-Method Group and the Requirements Engineering Research Group at CARE Tech., in which the author has actively participated since its very beginning. The results obtained from these efforts are being applied in case studies in academia and in real project developments. Currently, there is an important investment being made in developing tools to incorporate the technology in commercial software developments products through R&D contracts between UPV and CARE Tech.

The works that have made the development of this thesis possible are in the context of R&D contracts and R&D government projects.

The R&D contracts are the following:

- “Hyper-Dictionaries for Requirements Elicitation” CARE Tech, UPV, and the University of Twente (the Netherlands). From April to November of 2003.


The R&D government projects are the following:

- “Web Environment Engineering – Task #2: Requirements Engineering” - (Ingeniería de Ambientes Web).
  

  

- “Advanced Modeling and Specification of Distributed Information Systems (ASPIRE)”
  

- “Requirements Engineering and Automatic Code Generation” - (Ingeniería de Requerimientos y Generación Automática de Software)
  

- “MENHIR: Methods, Environments and Tools for Requirements Engineering” - (Métodos, Entornos y Herramientas para la Ingeniería de Requisitos)
  
1.2. Objectives

The main goal of this thesis is to provide a Requirements Engineering Environment with a sound, flexible, and precise transition to an OO-Method Conceptual Schema. This will comprise a Requirements Model that includes mechanisms to identify and specify user requirements, and a Requirements Analysis Process that establishes the traceability bridge between requirements and an OO-Method Conceptual Schema.

This goal will be satisfied by dealing with the following subgoals:

- Studying and defining an ontological background of terms where the system to be developed and its properties will be specified and analyzed.
- Studying the different approaches that deal with requirements specification, especially those based on goals and scenarios due to their growing relevance in the scientific literature for coping with the requirements modeling phase.
- Identifying the main properties of external interaction of systems in order to define a Requirements Model where user requirements must be captured at different abstraction levels.
- Providing a technique for reasoning about requirements as well as identifying internal responsibilities of the system according to an internal decomposition that could satisfy the intended requirements.
- Defining a Traceability Rules Catalog that could be used to systematically translate the specified and analyzed requirements into an OO-Method conceptual schema.
- Building a Requirements Specification Environment where the ideas proposed in this thesis could be tested and improved with the experience obtained from the development of different case studies.

1.3. Structure of the thesis

The presentation of this thesis is organized in the following chapters:

- Chapter 1. Introduction

This chapter presents a brief introduction to the evolution of software development and the different attempts that have been made to tackle the problems that directly or indirectly lead to the symptoms of the so-called software crisis. In this context, the efforts of this thesis are oriented towards this problem and the main issues that will be addressed are presented. We explain the context in which this thesis was developed and present the goals of this work.

- Chapter 2. Information System Development
This chapter introduces the main concepts of information systems and the relationships between them. These concepts and relationships are a subset of those provided by the Framework of Information Systems Concepts (FRISCO) and give us a context and a vocabulary to define our proposal. Two main approaches for dealing with Requirements Engineering are introduced: goal-driven approach and scenario-based approach. Both approaches are explained and contrasted, and the state-of-the-art is presented for each one of them.

- Chapter 3. Requirements Model

This chapter presents the foundations of our Requirements Model where the distinction of External Interaction from Internal Decomposition is fundamental. Both concepts are introduced and their relationships explained. Then, the Requirements Model Architecture proposed in this thesis is introduced. This architecture presents three different but complementary techniques, which are organized at different abstraction levels where the key concept is that of refinement. The Use Case Model, which is the last technique used when building a Requirements Model, is found at the lower level of abstraction.

- Chapter 4. Conceptual Model: The OO-Method Approach

This chapter introduces the OO-Method approach. First, the Conceptual Model is described where each one of the four models: Object, Dynamic, Functional, and Presentation models are presented. Later, a brief introduction of the OASIS formal specification language is explained. This is the basis of the Conceptual Model and what gives to it its formal foundation making the automatic code generation process possible. This is done with the Execution Model, which is also explained in this chapter.

- Chapter 5. Requirements Analysis Process

This chapter presents the Requirement Analysis Process, which is the core of this thesis. It represents the bridge between the Requirements Model and the OO-Method Conceptual Model, which is one of the most important contributions of this work. This process starts with the realization of Use Cases by means of Sequence Diagrams and ends with their translation into elements of the conceptual schema.

This chapter also includes the notation used for Sequence Diagrams (including an extension proposed in this thesis, which is the stereotyped message) and a traceability classification that categorizes the relationships of different elements at different abstraction levels.

- Chapter 6. Traceability Rules Catalog

This chapter introduces the traceability rules catalog, which is another important contribution of this thesis. It represents the set of relevant rules that makes an explicit interpretation of partial behavior descriptions
(scenarios as Sequence Diagrams) into a coherent but partial conceptual schema.

- Chapter 7. The Requirements Engineering Tool
  This chapter introduces the main characteristics of our prototype environment for requirements modeling.

- Chapter 8. Conclusions
  This chapter presents the main contributions of this thesis. Current and future research work and the most relevant publications are also presented.

- Appendix A. Traceability Rules Catalog
  This appendix summarizes the rules for traceability introduced in Chapter 6.

- Appendix B. Case Study: A Rent-a-Car System
  This appendix illustrates a case study developed for a hypothetical Rent-a-Car company, where requirements are initially explained in natural language (as usual). Then, a complete Requirements Model is explained and finally, the obtained conceptual schema is shown.
Chapter 2

2. Information Systems Development

A system is an assemblage of parts forming a complex or unitary whole that serves a useful purpose [3]. In this wide sense, organizations, computers, cars, and software products are examples of systems. It is crucial that the parts of a system interact in such a way that they cause the system as a whole to have a useful function for one or more entities in the environment of the system. An arbitrary collection of items with some interactions between the items is not necessarily a system. In order for a collection of interacting items to be a system, the interactions between the items must produce a coherent and useful overall behavior. In systems and software engineering, one often reserves the term “system” for the entire constellation of equipment, software, and human procedures to be developed and refers to the software components of the system as “software products”. In this thesis, only systems that interact with their environment and, specifically, computer-based information systems are considered. The characterizing feature of information systems is that they store and manipulate large amounts of data usually held in some kind of database [4]. They are implemented using standard computer hardware (e.g. mainframe computers, workstations, PCs) and they are built on top of commercial operating systems (Windows, Unix, etc.). A point-of-sale system, a reservation system for an airline tickets, or a car rental system are examples of information systems.

The characteristic of all these systems is that they have a set of emergent properties. To specify a software system is to represent its desired properties in a given language. A good software specification should allow us to evaluate whether the final implemented software system solves the original purpose. This goal is related to validation and verification activities. To validate a specification is to check whether it will solve its purpose and to verify a developed software system is to check whether it satisfies its specification. To succeed in verification, a good strategy is to use a set of well-defined methods and techniques throughout the entire software development process.
In an object-oriented approach, there is usually a complex relationship between requirements modeling, conceptual modeling, and design. Some authors [5], [6] suggest that requirements modeling and conceptual modeling are quite separate activities. Requirements are mostly concerned with the problem to be solved; conceptual modeling is how we represent what has to be done. Wieringa in [4] and [7], and Liddle in [8] and [9] state that requirements modeling and conceptual modeling are interlaced activities where the abstraction level is lower in conceptual modeling. Design is even a lower level of abstraction where implementation properties of the desired system are specified according to the development and deployment environment that will be used if we follow a software engineering process.

This chapter introduces the main concepts of information systems which are based on the Framework of Information Systems Concepts (FRISCO Report) [10]. The purpose of introducing these concepts is to give us a context where the state-of-the-art will be explained, and to position the contributions of this thesis. Then, an overview of the state-of-the-art in approaches for dealing with Requirements Engineering is presented.

### 2.1. Information System Concepts

*Information Systems* concerns the use of *information* by persons or groups of persons in organizations, in particular through computer-based systems. The concepts presented in this section are based on the results of the IFIP WG 8.1 Task Group FRISCO (FRamework of Information System COncepts) [10]. While fashionable methodologies consider a method for information system development as a compendium of a notation together with a process model (see e.g. UML-RUP [11, 12], the FRISCO authors have emphasized the importance of a theoretically consistent and pragmatically sound foundation of the information systems field. In this thesis, the concepts introduced by FRISCO were taken into account to develop the method proposed.

The FRISCO approach to bridging the gap between “reality” and its modeling concepts is based on semiotics, i.e. the theory of symbols, their form (syntax), meaning (semantics), and effect (pragmatics). The entire framework of FRISCO definitions is anchored in the *semiotic triangle* which was extended by FRISCO to a tetrahedron placing an “actor” in its center (see Figure 2-1), which is based on the work of Peirce [13].

![Figure 2-1. The semiotic tetrahedron of FRISCO](image-url)
There is a domain (also called Universe of Discourse - UofD) consisting of phenomena (the referents - the lower left hand corner of the triangle) observed by a person called the actor. As a result of physical and mental activities (namely, perception and interpretation), this person forms so-called conceptions (the top corner) and decides to treat these as individual, separable and identifiable “things”. He or she may then represent them by physical symbols (the representation - the lower right hand corner of the triangle). This overall subjective construction process is “objectified” (or better, socialized) by subsequent human communication processes. Whenever a social group or community (maybe, after some negotiations or even disputes) agrees to treat a certain phenomenon as a “thing”, it becomes a thing (by social construction) and is treated as such as long as it is not forgotten or made obsolete by other conflicting constructions.

The role played by the representation of a conception (i.e. of a thing that has a physical or imagined counterpart in some domain) is that of a symbol, that is to say, a collection of tokens (or symbols) that stand for the domain in question (as conceived by the actor). Thus in the context of Internet communication, the sequence of tokens :-] [which jointly constitute a well-recognized icon] stands for someone’s conception like “smile” referring to a certain effect (the “referent”) in the real world domain such as expressing agreement, favor, or sympathy.

The semiotic triangle is a helpful tool for illustrating the differences between representations (physical, symbolic entities), their meaning or intention (mental, abstract entities) and their counterpart or effect in the physical world (physical entities, actions or processes). For example, a physical person (client Brown - the referent) can be conceived as a collection of aspects (a conception) and be represented by some database entry (a representation). The central role in these processes is taken by a person who is responsible for linking referents, conceptions and representations with each other. Because of the combined roles of interpreter and representer, we shall refer to that person as the observer (see Figure 2-2).

**Figure 2-2. Example of the use of the semiotic tetrahedron of FRISCO**

### 2.1.1. Definitions

With the previous concepts in mind, we present some definitions that are fundamental to be able to understand, represent, and reason about systems.
- **Thing.** A *thing* is any part of a conception of a domain.

- **Transition.** A *transition* is a special binary relationship between two (partially or totally) different composite *things*, called the *pre-state* and the *post-state* of that transition, whereby at least one *thing* is an element of the *pre-state*, but not of the *post-state*, or vice-versa.

- **State.** A *state* is a composite *thing*, which is involved as a *pre-state* or as a *post-state* in some *transition*. No element of a state may be a *transition* itself.

- **Pre-state.** The *pre-state* of a *transition* is the *state* that is valid before that *transition* and is characterized by a special predicate “before”.

- **Post-state.** The *post-state* of a *transition* is the *state* that is valid after that *transition* and is characterized by a special predicate “after”.

- **Actor.** An *actor* is a special *thing* conceived as being “responsible” or “responsive” and as being able to “cause” transitions. It is, therefore, part of the transitions’ pre-states, and if not “destroyed” or “consumed” by the transitions, is also part of their post-states. An actor acts as a “world-observing” subject to produce conceptions that might then imply (mostly, other) actors as objects (“actands”) of this conceiving action.

- **Action.** An *action* is a *transition* involving a non-empty set of *actors* in its *pre-state*, and, if not “destroyed” or “consumed” by the action, in its *post-states* as well. It involves a non-empty or empty set of other *things* (actands) as a part of its *pre-state* and has a non-empty or empty set of other *things* (actands) in its *post-state*.

- **Actand.** An *actand* is a *thing*, which is involved in the *pre-state* or *post-state* of an *action*, but is not considered an *actor* for that *action*.

- **Input actand.** An *input actand* is an actand that is a part of the *pre-state* of an *action*, without including the *actors*. The *pre-state* of an *action* (i.e. the union of the set of actors and the set of input actands of that action) is called its *resources*.

- **Output actand.** An *output actand* is an actand that is a part of the *post-state* of an *action*, without including the *actors*.

- **Domain.** A domain comprises any “part” or “aspect” of the “world” under consideration.

- **Domain component.** A *domain component* is any “part” or “aspect” of that domain.

- **Domain environment.** A *domain environment* is the “world” without that domain.

- **Human actor.** A *human actor* is a responsible *actor* with the capabilities and liabilities of a normal human being. It is particularly capable of performing *perceiving actions, conceiving actions, and representing actions*. 
• **Perception.** A *perception* is a special *actand* resulting from an *action* whereby a human *actor* observes a *domain* with his or her *senses*, and forms a specific pattern (visual, auditory or other sensations of it) in his or her mind.

• **Perceiving action.** A *perceiving action* is a special *action* of a human *actor* having a *domain* as *input actand* and a *perception* as *output actand*.

• **Perceiver.** A *perceiver* is a human *actor* involved in a perceiving *action*.

• **Conception.** A *conception* is a special kind of *actand* resulting from the *action* whereby a human *actor* aims at interpreting a *perception* in his mind.

• **Conceiving action.** A *conceiving action* is a special kind of *action* of a human *actor* having a *perception*.

• **Conceiver.** A *conceiver* is a human *actor* involved in a conceiving *action*.

• **Interpreting action.** An *interpreting action* is a sequence of a *perceiving action* performed on a *domain* resulting in a *perception* of that *domain*, followed by a *conceiving action* performed on that *perception*, resulting in a *conception*.

• **Interpreter.** An *interpreter* is a human *actor* performing an *interpreting action*.

• **Representation.** A *representation* is a special *actand* describing one or more *conceptions* in a language, resulting from an *action* whereby a human *actor* aims at describing his or her *conceptions*.

• **Representing action.** A *representing action* is a special *action* of a human *actor* having a *conception* and possibly some action context as *input actand(s)* and a *representation* as *output actand*.

These definitions help us to better understand, represent, and reason about models, systems and information systems, that are introduced below.

### 2.1.2. Models, Systems, and Information Systems

In the context of organizations, an important aspect is that clear, precise, and unambiguous conceptions of some perceptions of some organizational domain are established. These special conceptions are called *models*.

A *model* is a purposefully abstracted, clear, precise, and unambiguous conception. A *model denotation* is a precise and unambiguous representation of a model, in some appropriate formal or semi-formal language. An example of a model is an Entity-Relationship diagram (graphical representation) that may be used to denote a model of the information-oriented aspects of an organization. A Petri Net (graphics or algebraic expression) is often used to denote a model of the dynamics (or behavior) of
an information system. A Use Case diagram (graphical) is used to represent a model of the communication between actors and Use Cases. The documentation accompanying an application program (natural language + diagrams) provides a stylized description of its functionality, but usually does not constitute its model in any true sense.

A model may be denoted in many different ways using different languages. However, one specific model denotation is expressed in exactly one language (see Figure 2-3). A *modeling action* is a special composite action comprising a *perceiving action* and a *conceiving action* as well as a *representing action* by a modeler.

\[
\text{Model} \rightarrow \text{Conception} \rightarrow \text{Representation} \\
\text{is-represented-as} \quad \text{represents} \quad \text{is-a}
\]

**Figure 2-3. Models and model denotations**

As mentioned above, a model denotation is expressed in a language. The language, in turn, is described in what is called the meta-language. That meta-language will be described in the meta-meta-language, and so on. In this way, a meta-level hierarchy is formed, which begins with the original model and its model denotation and is open-ended towards any higher levels. We refer to this origin as the *base model* and the *base model denotation*, respectively. The meta-level hierarchy continues upward until, at some level, a self-descriptive language is used (i.e. a language being sufficiently expressive to be used to formulate its own rules). For practical reasons, this is done mostly at level 2, but it might also be done at level 1 instead (see Figure 2-4).

\[
\begin{array}{c}
\text{Base model} \\
\text{Language} \\
\text{Meta-language (self-descriptive)} \\
\end{array} \rightarrow \\
\begin{array}{c}
\text{Base model denotation} \\
\text{Language representation} \\
\text{Meta-language representation} \\
\end{array} \\
\text{Meta-level 0} \quad \text{Meta-level 1} \quad \text{Meta-level 2}
\]

**Figure 2-4. Models, model denotations and meta-levels**

The notion of system is not uniquely defined in the literature. Many definitions can be found, such as:
• A collection of interrelated parts characterized by a boundary with respect to its environment [14];

or simply:

• A set of objects with a set of relations [15].

These definitions come from the “classical” general system theory [16], [17], [18], the essence of which is the interplay of known or presumed components. In this view, a system is an abstract construct that might be contemplated in its own right. As such, it is described as a formal expression for that set of components, together with the way in which they are connected. From the properties of the elements, one may derive the overall behavior and characteristics of the system. This kind of description also permits an enumeration of all allowable states and (often) also of any transition that might occur. Analytically, therefore, it is useful to describe a system as a set of known components and to add an explicit or implicit states-and-transitions description.

A system is a special model, whereby all the things contained in that model (all the system components) are transitively coherent (i.e. all of them are directly or indirectly related to each other forming a coherent whole).

Since a system is a special model and a model is a special conception, all the considerations about conceptions and models apply here as well (see Figure 2-5)

A special kind of system is an organizational system. An organizational system is an abstraction of how an organization appears to manifest itself in the “real world”. The groups of actors and actands constitute the components that form a whole, and there are the systemic properties of (a) having an organizational goal or some other factor of coherence, and (b) operating on the basis of “action-enabled-by-information”. These are two qualifications that are neither possessed by the environment, nor by components making up the organizational system.

The practical purpose of describing an organization as a system may be to prepare for organizational change and/or for planning a computerized support system. In the Requirements Engineering field, the study, conception and representation of organizational systems is known as business modeling.
Nowadays, practical computerized systems deal only with restricted parts of overall information flows and usage in the organization. Thus, they may be considered specializations of the more general information-providing arrangements one comes across as part of organizational actions. The full extent of the informational issues can also be viewed as a system with elements such as messages, message senders/receivers, message media, and any traditional equipment (telephones, writing pads), together with advanced aids (computers, networks). The systemic property, however, does not coincide with, nor is it even part of that of the organizational system. In this instance, it is the capability of providing information. Such a system—an information system in the most general sense—may be defined as follows: an information system is a subsystem of an organizational system, comprising the conception of how the communication and information-oriented aspects of an organization are composed and how these operate. In this way, the system describes the (explicit and/or implicit) communication-oriented and information-providing actions and arrangements existing within that organization.

Organizational systems and information systems are instances of the general type “system”. Any specific information-providing arrangements are clearly part of all that is comprised by the organizational system. They form a system themselves, but the systemic properties are so different that one cannot say that an information system is a specialized instance of the organizational system it is embedded in. One can only say that a system-subsystem relationship exists.

Any actual instance of an information system comprises all informal and formal informational actions and all knowledge and data-processing actions within the organization in question. If one considers only the formally defined information-providing actions set up and/or planned in an organization, one likewise observes a systemic arrangement (a set of interacting elements with an overall purpose, i.e. a systemic property). Since these elements are restricted to what is or may be computerized (or at least so described), an appropriate name is “computerized information subsystem”.

A computerized information subsystem is a subsystem of an information system, whereby all actions within that subsystem are performed by one or several computers.

One may therefore conclude that the following specialization (“is-a” subtype) and containment (“is-subsystem-of”) relationships hold:

- An organizational system is a system.
- An information system is a system; a specific information system is also a subsystem of some specific organizational system.
- A computerized information subsystem is an information system; it is also a subsystem of a specific information system (and, hence, also a subsystem of an organizational system).
- A computerized application is a system; it is also a subsystem of a specific computerized information subsystem (and, hence, also of the corresponding information system and organizational system).
These relationships, which are consistent with our previous definitions, are illustrated in Figure 2-6.

![Diagram of relationships between System, Organizational system, Information system, Computed information sub-system, Library information system, and uses of the system](image)

**Figure 2-6. Organizational and information systems and their denotations**

### 2.2. Requirements Modeling

In the Requirements Engineering area, it is well accepted that to obtain a representation of the requirements (or to *specify requirements*) two sources must be considered: *users* and the *domain*.

The first source provides informal statements of goals and user intentions expressed in natural language. The second source provides requirements reflecting real-world facts and constraints on the designed system that are implied by laws of physics independently of any user need or wish. Hence, requirements may occur in two subtypes:

- **User-defined requirements**, which arise from people in the organization and reflect their goals, intentions and wishes,

- **Domain-imposed requirements**, which are facts of nature and reflect domain laws.

This implies that the Universe of Discourse (UofD), or domain, has to be partitioned into two: the *usage world* and the *subject world* [19]. The *usage world* describes the tasks, procedures, interactions, etc. performed by actors and how systems are used to achieve work, including stakeholders who are either system owners or direct or indirect users. It can be looked upon as containing the objectives that are to be met in
the organization and which are achieved by the activities carried out by actors. Therefore, it describes the activity of actors and how this activity leads to useful work.

The second part of the UofD, the subject world, contains knowledge of the real-world domain about which information the proposed system has to provide. It contains real-world objects that are to be represented in the conceptual schema.

There is a third world, the system world which is the world of system specifications and in which the requirements arising from the two worlds must be addressed. The system world holds the modeled entities, processes, Use Cases, functions, etc. of the subject and usage worlds with some mapping from these conceptual specifications to the design and implementation levels of the software system.

All these worlds are interrelated as shown in Figure 2-7. User-defined requirements are captured by the two relationships between the usage and the system world, namely the intentional relationship and the usage-fit relationship. Domain-imposed requirements are captured by the domain genericity relationship.

Finally, note that there is a representation relationship between the subject world and the system world, which relates the domain to its representation in the system. Historically, this relationship has been the only focus of conceptual modeling, whereas requirements engineering highlights the importance of the three other relationships, namely the intentional, usage-fit, and domain genericity relationships.

2.2.1. Relationships between the Subject and System Worlds

There is a representation relationship between the subject world and the system world which was pointed out a long time ago by researchers in information systems [20], [21] and temporal databases [22], [23], [24]. However, the focus has been on objects, events, and operations, i.e. on the functional aspects of the information system.
Non-functional quality criteria such as confidentiality, performance, accuracy, and timeliness of information can also be attached to this representation relationship. Non-functional requirements modeling is a desirable feature of requirements engineering methods although it is rarely integrated in current methodologies. Some works in this area are [25], [26], [27].

There is another modeling concern captured through the domain genericity relationship. This is the role and impact of domain knowledge [28]. Many new applications share the same requirements with well-known problems, so one possibility is to create generic domain models of such problems as templates for requirements of certain classes of applications. Approaches that refer to this relationship between the subject and the system worlds attempt to add reuse to requirements engineering by providing sets of predefined generic requirements for developing a system requirements specification. These sets can be reused to elicit the requirements of a specific application or to validate a requirements specification. The hope is to increase the efficiency of requirements engineers while creating a better quality specification.

Dependencies between systems and their environments have been analyzed in detail by Jackson [28], and Jackson and Zave [29]. Jackson points out that domains impose obligations on the required system. He formalizes event dependencies between a system and its environment that are inherent to the laws of physics, e.g. obligations for the required system in avionics and other real-time applications, and events that may arise in human failure, both of which impose requirements on the system being designed.

The separate consideration of the subject world allows the development of so-called domain ontologies which consider typical classes of object and activity abstractions as reusable modeling patterns which can significantly reduce the requirements engineering effort [30], [31]. For example, a model library for the subject world has been developed in the NATURE project [32], [33], [34]. A model is a problem abstraction that defines in generic terms the structure and behavior of the problem space. It is a unit of abstraction that aggregates objects linked by a purpose.

Several authors have proposed generic knowledge for requirements engineering e.g. clichés in the Requirements Apprentice [35] and generalized application frames in the ITHACA project [36]. In software engineering, abstraction libraries have been described at the level of design architectures [37] and reusable functions, although these have limited success in practice [38], [39]. Similar abstract classes have also been proposed in Artificial Intelligence [40], [41]. Knowledge engineering methods, notably KADS [42] have also adopted the use of generic domain classes, although only a small number of classes has been described in detail.

2.2.2. Relationships between the Usage World and the System World

The usage world consists of individuals, social groups, and organizational settings in which the system is intended to function [43]. The pragmatic relationship with the system world is provided by the usage-fit relationship whereas the semiotic
relationship is provided by the *intentional relationship*. The usage world provides the rationale for building a system.

The purpose of developing an information system is to be found outside the system itself, in the enterprise, or in other words, in the context in which the system will function. The relationship between the usage and system worlds addresses the issue of the system purpose and relates the system to the organization goals and objectives. This relationship explains why the system is developed. Modeling this establishes the conceptual link between the envisaged system and its changing environment. This suggests an augmentation of conceptual modeling to deal with the description of the context in which the system will function and will be used.

In the area of requirements engineering, *goal-driven approaches* have been developed, which directly model organizational objectives and relate them to system functions.

As stated above, the *usage world* is the world of the system users who will individually work with the system to meet the objectives assigned to them by the organization. Additionally, each of them has his/her own set of desired functionality regarding the system to be constructed. Taking these into account helps in the construction of relatively more acceptable systems. This suggests another augmentation of conceptual modeling to deal with the specification of the system functionality from the integration of desired functionality.

In the area of requirements engineering, *scenario-based approaches* take this into account (scenario modeling or Use Case specifications).

As indicated earlier, *goal driven approaches* model organizational objectives to relate them to the functions of the system. In this sense, they aim at the conceptualization of purposeful systems only. They contribute further information for the interpretation of requirements before they become understood and can be transformed into system function specifications and, therefore, aim to support conceptualizing purposeful systems.

*Scenario based approaches*, by focusing on the desired functionality, helping modeling purposeful system usage from which useful system functions can be derived.

Therefore, *scenario based approaches* provide dynamic meaning to the *goal driven approaches*, whereas the *goal driven approaches* provide the intentional setting within which the usage world finds meaning.

### 2.2.2.1. Goal-Driven Approaches

Goal driven approaches go beyond the classical conceptual schema describing system functionality. They include enterprise modeling which more formally represents the *why* part of system requirements. This complements the *what* part provided by conceptual functionality modeling.

Enterprise modeling has been developed for example, in the F3 project [44], [45] to provide a set of models for understanding the requirements and bridging the gap
between ill-defined problems and application situations on the one hand and specification of the formal, precise definition of requirements of the information systems on the other hand. The requirements specification is represented as a structured description of five interrelated submodels (see Figure 2-8), which provide the context within which the requirements are elicited. Each submodel represents a particular concern or view in requirements acquisition, and these submodels help in separating the different concerns in a workable way. The submodels are not developed in a linear, sequential manner. Although the process usually starts with an objectives model and progresses through actor and activity models to information systems requirements, this is not always the case. For instance, with an existing system, the activity and concepts models may be developed first by reverse engineering previous designs.

![Figure 2-8. The submodels of the F3 approach](image)

The objectives submodel describes the why component of a requirements specification. It is a graph with components such as goals, problems, opportunities, and weaknesses depicted as nodes connected through relationships of the type motivates. The objectives submodel is related to rationale models such as IBIS [46] but it contains a goal decomposition hierarchy close to other proposals such as [47], [48].

The concept submodel is used to define the UoD that concerns requirements engineers. It may serve as a dictionary of user and customer defined concepts.

The actors submodel is used to define the actors in the domain and their relationships with activities and objectives. Actors may be individuals, groups, roles, organizational units, systems, etc. Actors are related to goals in the objectives submodel and, therefore, represent stakeholders who are responsible for achieving goals through activities described in the activities submodel.

The activities submodel describes the organizational activities, i.e. the processes and tasks of the enterprise. Components in this submodel are created to achieve goals in the objectives submodel, referring to components of the concepts submodel and resources required to carry out these activities described in the actors submodel.
The information system requirements submodel is meant to be derived from the other models. It includes both functional and non-functional requirements. The former requirements typically indicate the need for establishing objects, defining operations and services in the object-oriented terminology or functions in top-down decomposition such as Structured Systems Analysis [49]. The latter requirements are related to the environment, performance, and quality of the required system.

Enterprise modeling offers a set of interrelated models, each of which is constructed with a set of predefined component types and relationships to address the why question and to understand where the what requirements come from. The semantic links from the set of interrelated submodels and the information system requirements model are established for reflecting the rationale, the motivation, and for designing a specific information system. Enterprise modeling was one of the first attempts to understand goals, the primary needs expression of users in the context of a problem domain. Experience has shown [50], [51] that the use of a goal-driven approach leads to improved understanding of the problem realm for decision makers, requirements holders, customers, and developers.

Since then there has been convergence on the view that goal modeling is an effective way to identify requirements [52], to elicit high-level goals to be achieved by the envisioned system [53], [44], and to help in the refinement of these goals [48], [54] and their operationalization into system requirements specifying how goals should be accomplished by the proposed system [53].

Enterprise modeling was further refined in the EKD method to support change management [55], [56], [57]. The focus here is on the evaluation of alternative scenarios for change and the selection of the most appropriate one. In the KAOS approach [58], the emphasis is directed towards supporting formal refinement of high level goals into system constraints meant as functional requirements. Although generic models are advocated, goal modeling and refinement have supplied simple guidance via heuristics [59]. The i* approach [48], [51] creates models of the environment of the system that emphasizes modeling agents and their relationships. Their strategic dependency and rationale models allow tracing of dependencies between agents, the goals, and tasks, and support reasoning to identify the trade-off between functional requirements and non-functional requirements (referred to as soft goals) [27].

Similarly, emphasis on agents and their relationships is the basis for the Albert language aiming at formally specifying requirements [60] in order to ease their validation through scenario generation.

The GBRAM method [53] that is built over the Inquiry Cycle [61] draws our attention to the nature of the elicitation process that is viewed as a deliberation process among the various stakeholders involved with requirements engineering.

**Problems with Goals usage**

Although goal modeling has proved to be useful for specifying purposeful systems, practical experience tends to show that there are still a number of difficulties.

- Goal discovery is not a simple task: It is even often assumed that systems are constructed with some goals in mind [62]. In reality [53], goals are not given
and, therefore, the question as to where they originate from acquires importance.

- Expressed corporate goals do not provide a good starting point for Requirements Engineering: Enterprise goals, which initiate the goal discovery process, do not reflect the actual situation but an idealized environmental one. Therefore, proceeding from this may lead to ineffective requirements. The elimination of uninteresting and incorrect goals is necessary and difficult [52]. Thus, goal discovery is rarely an easy task.

- Goal reduction is not straightforward: It has been shown [53] that the application of goal reduction methods [58] to discover the component goals of a goal is not as straight-forward as the literature suggests.

- Finally, the fuzzy concept of a goal seems to be difficult to deal with. This led to some formalization of goal formulation [63], [64].

   It is thus evident that help has to be provided so that goal modeling can be meaningfully performed. This help must facilitate the work of the domain expert by getting over the problem of the fuzzy nature of goals, help discover goals, and aid in the task of goal reduction.

2.2.2.2. Scenario-Based Approaches

Independently of goal modeling, an alternative approach to Requirements Engineering, the scenario-based approach, has been developed. By capturing examples, scenes, narrative descriptions of contexts, Use Cases, and illustrations of agent behaviors, scenarios have proved useful in requirements elicitation in a number of ways: to elicit requirements in envisioned situations [61], to help in the discovery of exceptional cases [61], [54], [65], to derive conceptual object-oriented models [66], [6], [67], to understand needs through scenario prototyping [68] and animation [69], to reason about design decisions [70], to create context for design [71], and so on. The underlying reason for the popularity of scenario-based approaches seems to be that people react to descriptions of real events and real things. This reaction helps in clarifying requirements expected of systems.

   The scenario school argues that typical scenarios are easier than goals to get in the first place. Goals can be made explicit only after deeper understanding of the system has been gained. The industrial-practice survey conducted by the CREWS consortium [72] confirms that scenarios are useful in particular when abstract modeling fails [73].

   In addition, since scenarios describe concrete behaviors, they capture real requirements. However, because they deal with examples and illustrations, scenarios are inherently partial and only provide restricted requirements descriptions that need to be generalized to obtain complete requirements.

   In [74] Rolland presents a classification for scenarios developed from a comprehensive literature survey. This classification considers scenarios as being developed for different purposes with different contents, expressed in different levels of abstraction, with different notations and with different life cycles (see Figure 2-9).
Insofar as their purpose is concerned, scenarios can be descriptive, explanatory, or exploratory.

- **Descriptive scenarios** [61] capture requirements by enabling the software engineer and users to walk through a process and understand its operations, actors, the events triggering the process, etc. Thus, descriptive scenarios aid in the clarification of how a process performs, who are the involved parties, and how the process is activated as well as the conditions under which it is activated.

- **Explanatory scenarios** [75] raise issues and provide rationale for these issues. They identify why something happens in the real world, what leads to it, what are its causes, what are commonly occurring events that require handling etc. In this way, the attempt of explanatory scenarios is to describe the desirable features of the system to be developed.

- **Exploratory scenarios** [76] are useful when different possible solutions exist for satisfying a given system. These solutions are to be examined and evaluated to arrive at the right solution. Such scenarios establish a direct link between requirements and desired solutions.

As mentioned above, scenarios have different contents. This can be behavioral information identifying the actions, activities, and events carried out in the usage world; a description of the objects of the subject world together with their attributes; events and event histories; organizational information like the structure of the company, the groups, departments, and agents found in it; or even stakeholder information including the characteristics of people, their views and aspirations. However, by and large, scenarios concentrate on the functional features required of a system [77], [61], [78], [79].

Scenarios have been expressed at three different **levels of abstraction**: instance, type, and mixed.
- **Instance scenarios** [70], [61] use specific names or events with real argument values. These scenarios describe particular instances of use, which can form the basis for discussion of what happens, why, and how.

- **Type scenarios** [68], [67] do not use individual entities but entity types. Thus they do not refer to “Smith” but to “customers”. Each execution of a type scenario is an instance scenario.

- **Mixed scenarios** [54] are those that have some parts at the instance level and others at the type level.

Scenarios have been expressed in different notations ranging from the informal and semi-formal to the formal.

- **Informal scenarios** use natural language [64], [76], videos [80], [81], story descriptions, etc. and are valuable in those cases where the user community is unwilling or unable to deal with formal notation. UML [11] is also used when describing scenarios. It uses the Use Case Model to represent scenarios including relationships between them and communication with actors. For scenario description (Use Case description) there is not a standard representation (it can be natural language, tables or some other form).

- **Semi-formal scenarios** use a structured notation like tables [61] or scenario scripts [82] in capturing real activities.

- **Formal scenarios** are expressed in modeling languages based on regular grammars [83] or statecharts [84]. They are useful for running simulations in order to present a vision of what the future system will look like and to gauge user reactions to it.

From the *life cycle* point of view, scenarios are considered artifacts that exist and evolve in time through the execution of operations. In other words, the framework for scenarios presented here takes the position of looking upon scenarios from both a dynamic and a static perspective.

- In the *dynamic perspective*, we have transient scenarios. They are meant to be a support for some requirements engineering or design activity and are thus discarded after being used. In object-oriented methods, scenarios are typically used for capturing requirements [6]. In other approaches, they are facilitators for requirements elicitation [85], whereas in others they are a temporary support for requirements validation [69], [68].

- In the *static perspective*, persistent scenarios exist as long as the documentation of the project they belong to exists. There are two reasons for scenarios to be persistent. The first is when scenarios are considered part of the requirements specification, the second is when the project documentation keeps track of the scenarios used. The stepwise approach proposed by Kyng [71] is an example of the first case. Indeed, in this approach, the requirements specification is incrementally constructed by integrating scenarios. The Inquiry Cycle approach [61] is an illustration of the second case. Scenarios are there and act as communication support for acquiring,
eliciting, and validating requirements; they differ from requirements but are stored as artifacts that can be accessed at any time through the hypertext facilities provided by the tool environment.

**Problems with Scenario usage**

Scenarios have become key artifacts in systems engineering, but their management is poorly understood. Scenarios are a good communication base with customers and other non-technical people and support early validation of requirements at a low abstraction level. However, the full potential of scenarios is not always fully exploited because of problems including the following:

- Modeling the behavior of a whole system with scenarios requires a great multitude of scenarios.
- Scenarios are highly redundant, because some parts of scenarios are common to many scenarios.
- The redundancy in the scenarios makes changes tedious and can cause inconsistencies between different models.
- UML interaction diagrams (which are often used to describe scenarios) lack adequate expressiveness and semantic foundation.
- Misunderstanding can happen because of unclear semantics.

**2.2.2.3. Combining Goals and Scenarios**

In order to overcome some of the deficiencies and limitations of goal-driven and scenario-based approaches used in isolation, some proposals have been made to couple goals and scenarios together.

In [66], [86], [87], and [88], goals are considered to be contextual properties of Use Cases, whereas Cockburn in [89], uses goals as a means to structure Use Cases.

The goal and scenario combination has been used to operationalize goals [53], [76], [61], and [54], to check whether the current system usage captured through multimedia scenarios fulfills its expected goals [81], to infer goals specifications from operational scenarios [90], and to discover new goals through scenario analysis [54].

In this chapter, we have introduced a general view of Requirements Engineering highlighting goal-driven and scenario-based approaches. Many authors start the requirements modeling process following one of these approaches.

In this thesis, we follow a more conservative approach. It is similar to Cockburn [89] where goals can be used to structure Use Cases. The Requirements Model (that is introduced in the next chapter) uses a Function Refinement Tree that can be used to represent and reason about goals, specifically to represent and reason about *user goals*.

A *user goal* is the goal of the primary actor trying to get work done. This corresponds to what might be called “user task”, “elementary business process”, or
“elementary function”. We found that a user goal often corresponds to a transaction in transaction systems (completing a transaction to the database). “Register a new customer” is likely to be a meaningful user goal, and in the other hand, “Log on” is not considered a meaningful user goal (maybe it is one of the steps to fulfill a user goal).

Finally, grouping these goals in a multi-layer structure give us a tree structure that can be also useful for project management, project tracking, and business process engineering. Although goals discovery is easier looking for user goals, it is also true that the relationship between goals and users can be changed in time. For this important reason, the goal structure is organized in a user-independent way. We do this by means of the Function Refinement Tree, which is explained in the next chapter.
3. Requirements Model

In this chapter we introduce the Requirements Model. This model provides the means to reason about the system to be developed and to represent and organize information related to the Universe of Discourse (UofD).

The purpose of the Requirements Model is to understand what is to be built and to provide techniques to accurately capture the desired properties for it. Furthermore, the purpose is to build a model of these requirements in a manner that people without formal training in the notation could understand and review. The notation is nevertheless precise enough so that the model it describes can be the source to derive the architecture (building blocks) of an OO-Method conceptual schema.

In this chapter, we first introduce the foundations for the Requirements Model proposed in this thesis, and later the three-level architecture of the model.

3.1. Foundations

The approach for requirements modeling presented in this thesis is based on the Framework of Information Systems Concepts (FRISCO Report) [10], and on the work of Wieringa [7], [91], where the kinds of properties of software systems that a software engineer might want to specify are classified. The two basic dimensions of this classification (see Figure 3-1) are those of external interactions and internal decomposition.
Each system interacts with its external environment and is viewed as part of an aggregation hierarchy, in which higher-level systems are composed of lower-level systems. External interactions and internal decomposition are orthogonal in the sense that design decisions about these two dimensions of a system can be separated.

### 3.1.1. External Interaction

The external interactions of a system should be useful for other systems (people, hardware, or software) that are in its external environment. This means that we should always be able to partition external interactions into chunks of useful interactions that are called external functions. These chunks may be atomic from an external point of view (i.e. they are external transactions), or they may be complicated dialogs between the system and some external entities. They are not to be confused with mathematical functions or with functions written in a programming language. They are similar to Jacobson's [67] Use Cases: pieces of external behavior that have some use for an external agent.

Of the many properties that external functions can have, two kinds are highlighted:

- the ordering of functions in time, called behavior, and
- the ordering of functions in "space", called communication.

*Communication* and *behavior* should be treated as orthogonal properties of functions. Communications are the way functions are ordered in space, whereas behavior is the way functions are ordered in time.

An external function is an external interaction, and each external interaction involves communication with one or more external entities. Moreover, external interactions are usually governed by rules of temporal precedence, which leads to the concept of behavior\(^1\).

Functions, communications, and behavior are system properties. There are other kinds of system properties, such as security and user-friendliness, also called non-functional requirements [27], but we do not cover these explicitly in our model.

---

\(^1\) The distinction between behavior and communication is the same as the classification of process operators in CCS into dynamic and static ones.
3.1.2. Internal Decomposition

Once we have collected all the external interactions of a system the result is a requirements model. This model should be analyzed in order to obtain an internal decomposition of the desired system. This analysis results in a set of "building blocks" that constitute the architecture of the conceptual schema.

Taking a systems engineering view, we can represent the allocation and flowdown of external functions to components by means of a Function Decomposition Table, also called a Traceability Matrix. In Figure 3-2, the top row lists all external functions of the system (at a certain level of refinement) and the leftmost column represents all components of the system (at a certain level of aggregation). An entry of the table represents the functions that a component must have in order to realize an external function. A column of the table represents all components that collaborate together to realize an external function.

<table>
<thead>
<tr>
<th>component 1</th>
<th>...</th>
<th>component n</th>
</tr>
</thead>
<tbody>
<tr>
<td>function 1</td>
<td></td>
<td>function n</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 3-2. Function Decomposition Table (Traceability Matrix)

This table represents the flowdown of functions to components. Object-oriented methods recognize only one kind of component: objects, which encapsulate the aspects of data stores, data processes, and control processes. This is represented by means of a class-relationship diagram.

This completes the introduction of the basic ideas of the foundations of the Requirements Model and its relationship with the Conceptual Model. Now, it is important to point out some special features to be considered and show how they can be applied to the development of software systems.

First, observe that each component of a system is itself a system, which has an internal decomposition and interacts with other components and external entities of the system. In fact, each interaction of the entire system is realized by interactions of its components. In terms of specification techniques, this means that we can use the same technique to specify interaction of systems at different levels in the aggregation hierarchy.

Next, it is possible to specify a system's external interactions at several levels of refinement, where higher levels of refinement correspond to more detail and less abstraction. Software engineers can also specify the system components, components of those components, etc. leading to an aggregation hierarchy. Some techniques are more suitable for partitioning components into this hierarchy.

The orthogonality of external interaction and decomposition implies that interaction refinement and system decomposition are orthogonal. This is visualized in Figure 3-3, called the magic square by Harel and Pnueli [92]. Orthogonality means that decisions about interactions can be intertwined with decisions about decompositions in
any way [93]. There are many strategies to deal with these activities: top-down, bottom-up, incremental, etc., (an explained example can be found in [7]); however, in whatever way this is done, the result of these decisions must be justifiable as if they were taken by means of a rational design process [94].

So far, it is important to remark that each software system interacts with its environment by exchanging symbol occurrences with its external entities. A symbol occurrence is a physical item to which people have assigned a meaning. Therefore, for these people it refers to part of the external world. The part of the world referred to by the external interactions of a software system is called the subject domain of the system (another term often used is Universe of Discourse). The subject domain is itself a system and may itself be under development by another design team. Therefore, the framework of Figure 3-1 is applicable to it. To understand how techniques are used in one method, it is important to understand what they are used for to specify the subject domain or to specify the software system.

Another important issue in the specification of software systems is the identification of the essential level of aggregation. Given a specification of external functions, behavior, and communications of a software system, it is possible to design a decomposition of this system that would be optimal for this specification of external properties, and that ignores the properties of underlying implementation layers. This is called the essential decomposition of the software system.

The only decomposition criteria that can be used for the essential decomposition are derived from the external environment of the system, such as its external functionality, external communications, external behavior, or its subject domain. The concept of essential decomposition arose with McMenamin and Palmer [95].

All other decomposition levels of a software system are designed by taking aspects of the underlying implementation environment into account. For example, in a distributed system, the essential decomposition must be allocated to processors in the network, and at each processor, essential components must be mapped to schedulable sequential processes. These decomposition levels are called implementation-oriented.

A decomposition of a system into interacting components is shown in Figure 3-4. This illustrates the fact that the external interactions of a system are realized by the
communications of its subsystems. More generally, all externally observable properties of a system are realized by the properties of its components and their interactions.

Figure 3-4. Decomposition of a system into components

To summarize the Requirements Model presented, we have a number of kinds of system properties that we can specify. The following list relates these to the Function Decomposition Table of the system.

- **System functions.** These are the useful pieces of behavior ordered to the environment of the system. This corresponds to items in the top row of the table.

- **System behavior.** This is the behavior of the system over time. This concerns the way in which the functions in the top row are, can, or must be ordered in time. It thus concerns the entire top row.

- **System communication.** This concerns the communication of the system with external entities in its environment. It does not correspond to an aspect of the function decomposition table of the system, but it would correspond to a column of the function decomposition table of the environment, of which the system is one component.

- **Essential decomposition.** These are the essential components of the system, defined in terms of their meaning for the external environment. This corresponds to the items in the left-most column of the table.

- **Component functions.** These are the useful pieces of behavior ordered by the components to their environment. Component functions correspond to entries in the table.

- **Component behavior.** This is the way in which component functions are, can, or must be ordered in time. This corresponds with a row in the function decomposition table.

- **Component communication.** This is the way in which the components interact in order to realize the external functions. It corresponds to the columns of the function decomposition table. In each column, the component functions that interact to realize the external function are listed.

There are of course other properties, often called “non-functional” such as security, reliability, or user-friendliness, but these are not considered directly in this thesis and will be taken into account in future work.
In the next section, we introduce the different techniques and guidelines proposed for the Requirements Model.

### 3.2. Requirements Model Architecture

The architecture of the Requirements Model proposed in this thesis provides the means to represent the external interactions of the system. As was mentioned in the previous section, we can describe external interactions in three relevant ways. First, interactions can always be partitioned into functions. Second, interactions are by nature always communications with one or more entities in the environment of the system. And, third, the way in which functions are ordered in time is called the behavior of the system. In order to represent and organize these external functions, some behavior and communication techniques must be introduced.

The proposed techniques for this method are:

- **Mission Statement.** In a general sense, it describes the purpose of the system in one or two sentences and the system’s major responsibilities that agreed to perform.

- **Function Refinement Tree.** To deal with external interaction partitioning and organization (a key element in identifying the functions of the system).

- **Use Case Model.** It includes the Use Case description to specify the behavior of the Use Case at a very high-level. The Use Case diagram shows the communication between the environment (actors) and the system.

#### 3.2.1. Mission Statement

An extremely important activity is to define the Mission Statement (MS) when a system has to be developed. The mission of the system is the most general service (the main goal) that the system under development provides to its environment. Any external property specification should contain a specification of its mission and relate this to the objectives of the business.

The Mission Statement is a high-level description of the nature and purpose of the system. In the Yourdon System Method (YSM) it was called the *statement of purpose* [96].

A Mission Statement consists of three elements:

- **Definition.** This is a short description of the kind of system to be designed.

- **Purpose.** This is an overall indication of the purpose of the system. Its description should preferably contain only one sentence.

- **Responsibilities.** Each responsibility is a general functionality of the system. It is described by at most two or three sentences.
Although a Mission Statement is brief and should be easy to read, it is often very difficult to find and specify.

The reason for writing down the mission statement is that customers, users, designers, programmers, testers, and all other stakeholders should keep the system mission in mind during the entire software development process. If this is not done, then there are bound to arise differences of opinion about what the purpose of the system actually is.

### 3.2.2. Function Refinement Tree

External interactions can always be partitioned in functions. It is very useful to organize these goals or functions in a refinement hierarchy so that the root of the hierarchy is the overall system function (the Mission Statement), and the leaves are the elementary functions.

The intermediate nodes are groups of elementary functions and usually represent a kind of activity or a business area where the system is under development. It is not a trivial task to distinguish between intermediate and leaf nodes for the function refinement tree. In this context, the definition of Wieringa [7] is followed: “a function is regarded as elementary if it is triggered by an event sent by a user of the system (actor) or by the occurrence of a temporal event”.

In this way, the Function Refinement Tree (FRT) can be used to represent a hierarchical decomposition of business functions of a system independently from the actual system structure. The resultant tree is merely an organization of external functions and does not say anything about the internal decomposition of the system. This leads to the idea of a refinement hierarchy of behavior descriptions in which the behavior of a system is described at increasing levels of detail. Moreover, it gives the entry point for building the Use Case Model instead of starting from scratch and it avoids the potential problem of mixing the abstraction level of Use Cases.

The mixing of abstraction levels is an important and recognized problem in other approaches based on Use Cases [67], [97], [98] where there are usually no techniques used for the identification of Use Cases. It is not an easy task (without the appropriate methodological guidance) to identify in the problem domain when a behavior must be considered as an external function or just a sub-function. We provide this methodological guidance, which is to take into account only elementary functions as leaves of the FRT.

FRT can also be used to represent and reason about goals, specifically user goals. A user goal is the goal of the primary actor trying to get work done. This corresponds to what might be called “user task”, “elementary business process”, or “elementary function”. We found that a user goal often corresponds to a transaction in transaction systems (completing a transaction to the database). “Register a new customer” is likely to be a meaningful user goal, and in the other hand, “Log on” is not considered a meaningful user goal (maybe it is one of the steps to fulfill a user goal).

The FRT is a key technique in our Requirements Model because it relates the overall mission of the system (the goal of the system) to its functions down to its atomic
steps. This representation shows why the system must have each function in order to realize its mission.

An example of a FRT for a typical Car Rental system is shown in Figure 3-5. In this example, three main business areas are shown: Car Management, Contract Management, and Customer Management. In the Car Management functional group, we have three external interactions: Sell, Purchase, and Deliver. In the Contract Management functional group, we have two external interactions, Rent and Return, and the Extras Management functional group. And finally, in the Customer Management functional group we have two external interactions: Create and Delete.

![Figure 3-5. Excerpt of a FRT for a Car Rental system](image)

### 3.2.3. Use Cases

Use Cases (UC) have been introduced in OOSE [67] to represent external system functionality and since then have been adopted by several other methods and notations. A Use Case is an interaction between the system and an external entity that has a use for that external functionality. This interaction need not be atomic; it is usually decomposed into steps using natural language (Use Case description), or more formally by a specification of pre-post conditions of the Use Case. An overview of the Use Cases can be given by a Use Case diagram showing which external entities have a communication relationship with each Use Case.

In our approach, Use Cases are commonly used as a synonym of scenarios [99]. A scenario describes a situation that happens in the Universe of Discourse (UofD) [100] in the same way the Use Case does². Scenarios and Use Cases have received considerable attention in requirements for the following reasons:

- They describe the externally visible behavior of a system only, thus avoiding solution bias and premature design.

---

² In order to manage complexity, it is possible to define Use Cases that specify partial behavior. These Use Cases can be reused in other Use Cases or they can simple package a well-defined but partial behavior. In those cases, the definition of scenario doesn’t match that of the Use Case.
• They describe how users will work with a system, hence they are much easier to validate with users than for example class models or dataflow diagrams.

• They can be used for both elicitation and description of requirements.

• They inherit the comfort and ease of natural language specifications, but avoid many of the problems of a purely narrative specification.

The particular strength of scenarios lies in the fact that they provide a decomposition of a system into functions from a user's perspective and that each such function can be treated separately – a classical application of the principle of separation of concern. However, the lack of precision of when and how Use Cases (or scenarios) should be used has spread to the software engineers who are using the technique in the field [100], [74], [73]. To solve this problem, we propose using Use Cases together with the Function Refinement Tree, in such a way that each primary Use Case will correspond to a leaf node of the FRT. Also, it should be possible to have secondary Use Cases. Secondary Use Cases should be those scenarios that have no direct correspondence with the FRT (and, therefore, they are not external interactions) but are important for organizing and managing complexity (secondary Use Cases are explained in subsection 3.2.3.2).

Some strategies are found in the literature to identify and specify Use Cases, although the most commonly used are the actor and event-based strategies.

The actor-based strategy [67], [101] first identifies actors related to the system under development or organization, and then identifies the relevant external events that are initiated or participated in by each actor.

The event-based strategy [101] first identifies external events that the system under development must respond to, and then relates these events to the corresponding actors.

The approach presented in this thesis is similar to event-based Use Case model construction but with two important differences: first, with the Function Refinement Tree external interactions are partitioned and organized in a refinement hierarchy, and second, we abstract the system's actors until a second step of the iterative process is reached (the Use Case modeling). At this moment (in our point of view), software engineers have a clearer view of the system external functionality and it would be easier for them to model the communication flow between actors and Use Cases.

According to the classification presented in section 2.2.2.2 and developed by Rolland in [74], scenarios can be used for different purposes with different contents, expressed in different levels of abstraction, with different notations, and with different life cycles.

In the approach presented in this thesis, insofar as their purpose is concerned, descriptive scenarios are used [61]. They capture requirements by enabling software engineers and users to walk through a process and understand its operations, actors, and events triggering the process. They aid in the clarification of how a process performs,
who the involved parties are and how the process is activated as well as the conditions under which it is activated.

Also, scenarios have different contents. In this approach, they have behavioral information identifying the actions, actors involved, activities, and events carried out in the usage world.

Scenarios have been expressed at different levels of abstraction. We use the type scenarios \[68, 67\] where no individual entities but entity types are used. Thus, they do not refer to Smith but to customers. Each execution of a type scenario is an instance scenario (or Use-Case instance).

Scenarios have been expressed in different notations. We have semi-formal scenarios using a structured notation with tables \[61\].

From the life cycle point of view, our scenarios are persistent scenarios because they exist as long as the project exists. The scenarios are part of the requirements specification and the base for the following steps in the software development process.

### 3.2.3.1. Use Case Description Structure

The purpose of the Use Case description is to specify the internal composition of a scenario in detail, including how the Use Case starts, ends, modifies the system, and interacts with actors (for a Use Case description template see Figure 3-6). A three-section structure \[100, 89, 102\] is usual for the Use Case description so that it can be understandable to customers and be accurate enough to capture the purpose of the Use Case in terms of its compound steps.

In this three-section structure, the first section is a summary of what the Use Case is about. The second section describes the basic course, which is the most important course of events giving the best understanding of the Use Case. Variants of the basic course of events that can occur are described in a third section called the alternative section. A discriminant (a condition) must decide which alternative to execute, and the flow of control is transferred to this alternative.

An overview of this structure is the following:

1. **Use Case summary** section. This section contains summary information of the scenario. The provided information is:
   a. **Name.** This is the Use Case name.
   b. **Actors.** These are external agents that communicate with the Use Case, indicating who initiates the Use Case and the type of communication involved: input, output or input/output,
   c. **Cross-reference.** This is a link to a leaf node of the Function Refinement Tree (it is a very important information for traceability purposes),
d. **Pre-Condition.** This is a condition that should be satisfied in order to execute the Use Case.

e. **Purpose.** This is an explanation in natural language of what the Use Case is for.

Additionally, this section shows the *included* and *extended* Use Cases (these Use Case relationships are explained in 3.2.3.2 debajo de).

f. **Includes to.** This is a list of all the Use Cases that are included in this Use Case.

g. **Extend to.** This is a list of all the Use Cases from which the current Use Case is an extension. For each Use Case the corresponding *condition* and *extension point* in the referred Use Case is indicated.

2. **Basic course** section. This section contains the steps that occur during the Use Case. This specification should include all the steps of the Use Case from the triggering event through to the accomplishment of the goal. They are numbered paragraphs (steps) and usually they are written in a conversational style between actors and the system [101]. The structure of a step is presented in the next subsection.

3. **Alternative** section. This section contains steps that complements the specification of a scenario described in the basic course. The steps included in this section are not a course of steps on their own; they are only used in those cases where a given condition holds and some step should be accomplished to complete the scenario. After the execution of this step, the original scenario continues its course of steps (e.g. in a Car Rental system, when a customer returns a car, if the return hour is before the return hour of the contract, a discount is created for his or her personal account).

An overview of this structure can be seen in Figure 3-6.
3.2.3.1.1. **Steps Structure**

The step is the building block for the scenario. A step is a description of an activity carried out by the actor or the system during the scenario execution. Clear and simple sentences should be used for this purpose so that more information can be obtained from the step. A good rule of thumb is to write sentences with only one subject and predicate as shown in Figure 3-7. An example can be seen in Figure 3-8.

![Figure 3-7. Step structure](image-url)

```markdown
[<condition>,] <actor> | <system> <activity>

where:
- **Condition**: An optional boolean expression. Depending on its value (`true`/`false`) the step must be executed or canceled.
- **Actor | System**: The responsible to accomplish or request the activity.
- **Activity**: Description of the activity accomplished.
```
3.2.3.1.2. **Step Classification**

The scenario specification is the process used to describe its internal composition, including how the Use Case starts, ends, modifies the system, and interacts with actors. To accomplish this goal, a structure for the steps specification was proposed. This specification is called *episode specification* in [100]. For many approaches with these rules it is enough although other general guidelines and more sophisticated structures are provided in literature [67], [101].

We believe that we have more information about the different steps we are specifying and that we must capture that information. For example, we know a) if a step is an activity related to the communication between the system and an actor, or b) if the activity will cause a change in the information stored in the system, or c) if the step is specified only to get more understanding from the scenario but is not going to be part of the automated system.

We present a step classification with all this kind of information. Basically, three kinds of steps have been identified: general, actor/system communication and system response steps.

- **General**: these steps represent general activities performed by actors in the problem domain that are outside the scope of the software requirements model. They are important because they help the software engineer to better understand under what conditions the Use Case communicates with its environment; for example, when a customer phones the Car Rental company to change the number of days of his/her contract. Of course this event in the system environment doesn’t directly affect the system itself (unless there is an automatic system with voice recognition or tone processing). In the Use Case description, it would be useful to annotate that the change of number of days in a contract occurs when the customer phones the company.

- **Actor/System communication**: these steps represent specific actions accomplished by actors to interact with the system, or performed by the system to query or offer information to actors. They are related to user interface requirements; for example, when the employee selects an option from an options menu, or when he or she introduces or receives information to/from the system.

- **System response**: these steps specify concrete reactions (changes of state) in the system because of the occurrence of the Use Case; for example, when there is a contract change request in the Car Rental system, the corresponding modification should be recorded.
This classification is also a contribution of this approach in order to make the specification of the Use Case easier for the software engineer and also to get more useful and precise information from the users in order to ensure traceability with further steps in the software development process.

3.2.3.2. Managing Complexity

Complexity is inherent to medium-large size projects. Brooks, in [103], says that complexity is the essence of software systems. Use Cases (graphically represented as a Use Case Diagram or textually in a description) are flat artifacts (they do not have nested Use Cases). For dealing with complexity, there may be refinement relationships between Use Cases, as explained in [67] and [11], or we can create subsections inside the textual description.

A Use Case description may contain decision points and extension points [11]. An extension point is a part of the scenario specification (a step) where the scenario can be extended (using an extend relationship). A decision point is a condition in a step of the specification where there is one or more possible paths.

If one of these decision paths represents the overwhelmingly typical case, and the other alternatives are rare, unusual, or exceptional, then that typical case should be the only one written about in the basic course section of the Use Case description, and the alternatives should be written in the alternative section. There can be zero or more alternatives in a Use Case description.

However, sometimes the decision points represent alternatives that are all relatively equal and normal in their likelihood. In this case, only the branches are indicated in the basic course section and all the alternative paths are written in the alternative section.

When the alternatives get complex, they may be expanded into their own Use Cases, which are called secondary Use Cases. In this case, we have to relate the primary Use Case to these secondary Use Cases. But the question is: when should a major step or branching activity in a Use Case be written as a subsection branch or as a secondary Use Case?

To answer this question two guidelines are proposed:

- When they are duplicated in other Use Cases, and
- When they are complex and long, and separating them helps factor the Use Case into manageable comprehensible units.

UML, in its current version 1.5 [11], proposes three kinds of relationships between Use Cases: include, extend and generalization.

- An include relationship between two Use Cases means that the behavior defined in the target Use Case is included at one location in the sequence of behavior performed by an instance of the base Use Case. When a Use-Case instance reaches the location where the behavior of another Use Case is to be
included, it performs all the behavior described by the included Use Case and then continues according to its original Use Case. This means that although there may be several paths through the included Use Case due to conditional statements, all of them must end in such a way that the Use-Case instance can continue according to the original Use Case. One Use Case may be included in several other Use Cases, and one Use Case may include several other Use Cases. The included Use Case must not be dependent on the base Use Case. In that sense, the included Use Case represents encapsulated behavior, which may easily be reused in several Use Cases. The notation for the include relationship is a normal Use-Case dependency with the stereotype of «include>>. Figure 3-9 shows an example where Use Case A includes Use Case B.

![Figure 3-9. An include relationship](image)

- An extend relationship defines that a Use Case may be augmented with some additional behavior defined in another Use Case. One Use Case may extend several Use Cases, and several Use Cases may extend one Use Case. The base Use Case must not be dependent on the extending Use Case. The extend relationship contains a condition, and references one (or more) extension points in the target Use Case. The condition must be satisfied if the extension is to take place. The references to the extension points define the locations in the base Use Case where the additions are to be made. Once an instance of a Use Case is to perform some behavior referenced by an extension point, the condition of the relationship is evaluated. If the condition is fulfilled, the Use Case instance is extended to include the sequence of the extending Use Case. The notation for the extend relationship is a normal Use-Case dependency with the stereotype of «extend>>. Figure 3-10 shows an example where Use Case B extend Use Case A when condition X holds in step n (in Use Case A).
A generalization relationship between Use Cases implies that the child Use Case contains all the attributes, sequences of behavior, and extension points defined in the parent Use Case, and participates in all relationships of the parent Use Case. The child Use Case may also define new behavior sequences, as well as add additional behavior into a specialized existing behavior. This implies that the parent Use Case can be replaced by the child Use Case without breaking the business flow. One Use Case may have several parent Use Cases, and one Use Case may be a parent to several other Use Cases.

Both generalization and extend appear to be quite similar; however, there is a subtle difference. When you establish a generalization relationship between Use Cases, this implies that the parent Use Case can be replaced by the child Use Case without breaking the business flow. On the other hand, an extend relationship between Use Cases implies that the extending Use Case enhances the functionality of the parent Use Case into a specialized functionality. The parent Use Case in an extend relationship cannot be replaced by the extending Use Case. Another difference is that the base Use Case defines extension points, which are the only places where its behavior may be extended. The extending Use Case must define at which of these extension points it adds behavior.

The main difference between include and extend is that the include relationship makes an explicit invocation of the included Use Case. In the case of the extend relationship, the extension is implicit (whenever the condition holds in the extension points, the parent Use Case enhances its functionality with the extending Use Case). The use of the extend relationship allows us to augment the behavior of an already defined Use Case without changing it.

In our Requirements Model, we do not use the generalization relationship. In our point of view, the extend and include relationships are expressive enough to deal with the structuring of Use Cases, thereby avoiding the often confuse difference between generalization and extend.
Chapter 4


OO-Method [104], [105] is an object-oriented method for Conceptual Modeling and automatic code generation. It was created on the formal basis of OASIS [106], [107], [108], an object-oriented, formal, specification language for Information Systems. Basically, we can distinguish two modeling components in OO-Method: the Conceptual Model and the Execution Model.

4.1. Conceptual Model

In the OO-Method proposal, the combination of formal specification techniques with conventional, widely used, object-oriented modeling techniques has been a basic feature since the very beginning. Another important objective was to avoid the complexity that is traditionally associated with the use of formal methods, by making software engineers view the method as being compliant to industrial standards. In order to achieve these objectives, two main aspects must be clearly stated:

- Which conceptual modeling patterns are provided by the method
- Which notation is provided to properly capture those conceptual modeling patterns

This is why OO-Method adopted the well-known OMT strategy of dividing the conceptual modeling process into three complementary views: the object view, the dynamic view, and the functional model view (adding a fourth view to specify presentation patterns). However, there is a big difference: in OO-Method, when the software engineer is specifying the system, what he or she is really doing is capturing a formal specification of the system “on the fly”, according to the OASIS formal specification language. This feature allows us to introduce a well-defined
expressiveness in the specification, which is often lacking in the conventional methodologies.

The use of such a formal specification provides the context to validate and verify the system in the solution space, obtaining a software product that is functionally equivalent to the specification.

This equivalence is achieved by creating a model compiler that implements all the mappings specified between the conceptual patterns (problem space level) and their software representations (at the solution space level).

Naturally, we have had to introduce relevant information to address specific features of OASIS in these diagrams (Object Model, Dynamic Model, Functional Model, and Presentation Model). Nevertheless, this is always done preserving the external view that is compliant with the most extended modeling notation, which is the UML.

For this reason, the subset of UML used in OO-Method is the one necessary to complete the information that is relevant for filling a class definition in OASIS. In this way, the arid formalism is hidden from the modeler when he or she is describing the system by making him/her feel comfortable using a conventional modeling notation.

According to the previous arguments, a main interest in the design of OO-Method was to keep modelers from having to learn another graphical notation in order to model an information system. Having a formal basis allows us to provide a modeling environment where the set of needed diagrams is clearly established.

We now introduce briefly the corresponding OO-Method set of models. Basically, OO-Method collects the system requirements using four complementary models that are explained below.

## 4.1.1. Object Model

The Object Model (OM) is a graphical model where system classes and relationships (association, aggregation, and inheritance) are defined. Additionally, agent relationships are specified to state the services that objects of a class are allowed to activate. The corresponding UML base diagram is the Class Diagram, where the additional expressiveness is introduced by defining the corresponding stereotypes.

Specifically, for every class, the OM captures the information about attributes, services, derivations, constraints, and relationships (aggregation and inheritance). More precisely, the additional information associated with association, aggregation, and inheritance is the following:

- For associated classes, to specify whether we have an aggregation or a composition (following the UML characterization) and whether the association is static or dynamic;
- For inheritance hierarchies, to specify whether a specialization is permanent or temporal. In the former case, the corresponding condition on constant
attributes must characterize the specialization relationship; in the latter, a condition on variable attributes or the carrier service that activates the child role must be specified.

Figure 4-1 shows an example of an OM instance, a view of a Car Rental system with Contracts, Cars, and Customers. It can be seen that classes and their relationships are depicted using an UML-compliant class diagram notation. For instance, the arrow between Customer and AgencyCustomer denotes that AgencyCustomer is a specialization of Customer. Next to the arrow the specialization condition (type = “Agency”) is specified. The solid line between Contract and Customer and Car represent an association between these classes.

Agent relationships are represented by using the stereotype “agent”. This stereotype connects the associated client class and server class services. In the example, the objects of the Administrator class can activate the services Purchase, Enable, and Disable of the Car class.

Figure 4-1. Excerpt of the Class Diagram for the Car Rental system

Shared services are those services that are included in the specification of more than one class, representing a synchronous communication mechanism between the objects involved in their occurrence. They are represented linking the participating services with a straight line labeled with the stereotype “shared”.

63
We must specify additional information in the class diagram to complete the formal description. Specifically, for each class the OM captures:

- **attributes**: to indicate if the attribute is constant, variable or derived;
- **services**: name of the services with their corresponding arguments, distinguishing the new and destroy class services;
- **derivations**: derivation expressions for the derived attributes (those whose value is dependent on other attributes);
- **constraints**: well-formed formulas representing conditions that objects of a class must satisfy

### 4.1.2. Dynamic Model

The system class architecture has been specified using the OM. Additionally, basic features, such as which object life cycles can be considered valid and which interobject communication can be established, have to be introduced in the system specification. To do this, OO-Method proposes a dynamic model. It uses two kinds of diagrams:

- **State Transition Diagrams** The state transition diagram (STD) is used to describe correct behavior by establishing valid object life cycles for every class. By valid life, we mean an appropriate sequence of service occurrences that characterizes the correct behavior of the objects that belong to a specific class. The corresponding UML base diagram is the State Diagram. Figure 4-2 shows a simple STD for the *Customer* class. Every service occurrence (i.e. *create_customer*) is labeled by the agent *Administrator* (Admin) that is allowed to activate it. In this example, the “*” denotes that any valid agent class can activate the transition. As the only valid agents for the *create_customer* service are objects of class *Administrator*, both representations are equivalent.

The syntax for transitions is the following:

```
[list_of_agents | *]:service_name [WHEN control_condition]
```

Control conditions are conditions defined on object attributes to avoid the possible non-determinism for a given service activation. In the example, once a *Customer* is in the state labeled *Without Cars*, if a *rent* service occurs the object will move to the *With Cars* state. Here, if a *return* service occurs the selected transition will be the one satisfying the corresponding condition control (*number_cars = 1 or number_cars > 1*).
Interaction Diagram

The interaction diagram (ID) specifies the interobject communication. We define two basic interactions: triggers, which are object services that are activated in an automated way when a condition is satisfied, and global interactions, which are transactions involving services of different objects. The corresponding UML base diagram is the Collaboration Diagram where the context of the interaction is not shown. Figure 4-3 shows a simple interaction diagram for our example. We have one trigger indicating that a triggered action (Admin:disable()) must be activated when the kms attribute of a Car is greater than 100,000.

Trigger specifications follow the syntax:

destination :: (trigger_condition) agent:service

The first part of the formula is the destination (the object(s) to which the triggered service is addressed). The trigger destination can be the same object where the condition is satisfied (self), or a specific object (indicating the oid involved), or the entire class population if we are broadcasting the service (class). The last part of the formula is the triggered service with the corresponding agent.
Global interactions are graphically specified by connecting the services involved in the declared interaction. These services are represented as solid lines linking the objects (boxes) that provide them.

There is one STD for every class, but only one ID for the whole system, where all the inter-object interactions will be graphically specified.

### 4.1.3. Functional Model

A correct functional specification is a shortcut of many of the most extended OO methods. Sometimes, the model used breaks the homogeneity of the OO models, as happened with the initial versions of OMT, which proposed using the structured DFDs as a functional model. The use of DFD techniques in an object-modeling context has been criticized for being imprecise, mainly because it offers a perspective of the system (the functional perspective), which differs from the other models (the object perspective). Other methods leave the free-specification of the system operations in the hands of the designer. The OO-Method functional model (FM) is quite different from these conventional approaches. In this model, the semantics associated to any change of an object state is captured as a consequence of a service occurrence. To do this, it is declaratively specified how every service changes the object state depending on the arguments of the service involved and the object's current state.

A clear and simple strategy is given for dealing with the introduction of the necessary information. The relevant contribution of this functional model is the concept of categorized attributes.

Three types are defined: *push-pop*, *state-independent*, and *discrete-domain based*. Each type will define the pattern of information required to define its functionality.

- **Push-pop attributes** are those whose relevant events increase or decrease their value by a given quantity. Events that reset the attribute to a given value can also exist.

- **State-independent attributes** have a value that depends only on the latest action that has occurred. Once a relevant action is activated, the new attribute value of the object involved is independent of the previous one. In such a case, we consider that the attribute remains in a given state, having a certain value for the corresponding attribute.

- **Discrete-domain valued attributes** take their values from a limited domain. The different values of this domain model the valid situations that are possible for objects of the class. Through the activation of carrier actions (which assign a given domain value to the attribute) the object reaches a specific situation. The object abandons this situation when another event occurs (a “liberator” event).

To illustrate, Table 1 shows the Functional Model for the attribute *number_cars* of the *Car* class. This attribute is categorized as a *push-pop* because its relevant services
increase or decrease its value by a given quantity. In the example, \textit{Admin:rent()} has the increasing action and \textit{Admin:return()} has the decreasing action.

Table 1. Functional Model for the attribute \textit{number_cars}

<table>
<thead>
<tr>
<th>Class:</th>
<th>Attribute: number_cars</th>
<th>Category: push-pop</th>
</tr>
</thead>
<tbody>
<tr>
<td>Action</td>
<td>Effect</td>
<td>Action Type</td>
</tr>
<tr>
<td>Increase</td>
<td>Admin:rent()</td>
<td>+1</td>
</tr>
<tr>
<td>Decrease</td>
<td>Admin:return()</td>
<td>-1</td>
</tr>
</tbody>
</table>

This categorization of attributes allows us to generate a complete OASIS specification in an automated way, where service functionality is completely captured.

4.1.4. \textit{Presentation Model}

The object society structure, behavior, and functionality are specified above using the three previous models. The last step is to specify how users will interact with the system. This is done by the Presentation Model through the definition of a set of Presentation Patterns. The Presentation Patterns capture the information required to characterize what appearance the application will have, and how the user will interact with the quoted application.

Those Presentation Patterns can be organized in three levels:

- Level 1: The \textit{first level} contains the Hierarchical Action Tree pattern, providing the access to the application.

- Level 2: The \textit{second level} contains the Interaction Units. The user interface is then decomposed in several scenarios to support user tasks.

- Level 3: Eventually, the \textit{third level} is composed of patterns that add additional semantics to the interaction units.

The precise description of the patterns can be found in [109]. Figure 4-4 shows globally the levels and the corresponding interdependencies among patterns.
4.2. **OASIS Formal Specification**

OASIS [106], [107] is an OO formal specification language developed at the Information Systems and Computation Department at the Valencia Polytechnic University. In this section, we give a general explanation of the foundation and structure of the language. A detailed description of the language can be found in [106] and [107].

A class in OASIS is made up of a class name, an identification function for instances (objects) of the class and a type or template that all the instances share.

The identification function characterizes the naming mechanism used by objects. It gives a set of surrogates belonging to a predefined sort or to a sort defined by the user (the so-called domains in OASIS). They are imported in the class definition. The most usual are predefined as *Int, Nat, Real, Bool, Char, String* and *Date*. They
represent numbers, boolean values, characters, strings, and dates in a particular format. New domains can be introduced in a specification by defining the corresponding abstract data type.

A type is the template that collects all the properties (structure and behavior) shared by all the potential objects of the class being considered.

Syntactically, it can be formalized as:

- A signature, which contains sorts, functions, attributes and events to be used
- A set of axioms, which are formulas in a dynamic logic
- A process query, as a set of equations with variables of sort process that are solved in a given process algebra

When these variables are instantiated, we have the ground terms that represent possible lives of instances (objects)\(^3\). Its semantics is given in terms of the \((W, \tau, \rho)\) structure (KRIPKE semantics).

We now consider these three class definition components:

A **class signature** contains:

- A set of sorts with a partial order relation. Among this set of sorts, there is, in particular, the sort of interest (the class name) associated with the class being defined
- A set of functions including:
  - Those functions included in the definition of the (predefined) sorts
  - The identification function whose sort is the ADT for identities implicitly provided with a class specification. It provides us with values of a given sort to identify objects, to assure that any object of a given class has a unique identity\(^4\)
  - A set of (constant, variable and derived) attributes, (see `constant_attributes` and `variable_attributes` sections in Figure 4-5). They all have the sort of the class as domain, and the given sort associated to the attribute being considered as codomain
  - A set of events (see `private events` and `shared events` in Figure 4-5), with the sort of the class as domain (plus any additional sort representing event information) and with the sort of the class (sort of

---

\(^3\) It is then a presentation of a theory in the modal logic outlined.

\(^4\) For specification purposes, an identification mechanism is introduced consisting of the declaration of one or more key maps used as aliases for identifying objects. They are similar to the candidate key notion of the relational model. From a given key value these maps return its associated object identity. Key maps will be declared as (tuples of) constant attributes.
interest) as codomain. It is important to remark that this so-called sort of interest can be seen as a subsort of a general sort process when objects are viewed as processes

CONCEPTUAL SCHEMA rentcar
domains Nat,Bool,Int,Date,String
class car
identification
  by_car_number: (car_number);
constant_attributes
  car_number : String ;
  model : String ;
  year : String ;
variable_attributes
  rent_count : Int init 0 ;
  kilometers: Int;
private_events
  new_car() new;
  destroy_car() destroy;
  disable();
  unable();
shared_events
  rent() with client;
  return(km) with client;
constraints
  static kilometers < 100000;
valuation
  [rent()] rent_count= rent_count + 1;
  [return(km)] kilometers = kilometers + km;
  rent_count mod 10 = 0 [return(km)] rent_count= 0
  rent_count mod 10 <> 0 [return(km)] rent_count= rent_count - 1;
preconditions
  usr:destroy_car () if rent_count = 0;
triggers
  Self :: disable() if rent_count = 0;
process
  car = usr:new_car() car0;
  car0::usr:destroy_car() + disable() car2 +
  rent() car1;
  car1= return() car0
  car2 = unable() car1
end_class
END CONCEPTUAL SCHEMA

Figure 4-5. Excerpt of an OASIS specification

The agent that is allowed to activate it will label every event occurrence. Following the approach taken by [110] when dealing with this actor notion, we will write $i:a$ if the agent $i$ initiates event $a$ and call $i:a$ an action. Note that $i$ could be the environment or any object of a system class.

When defining an event, the designer is forced to state who will be able to activate it. In consequence, we will assume that a set $A$ of actions is defined. This set of actions is obtained from and attached to the initial set of events. In this way, we can represent the notion of the set of object services as the interface that allows other objects to access the state. They can be events (server view) or actions (client view) depending on whether these services are offered or requested. Actions become services requested by an object, by which it can consult or modify another object’s state (or its own state).
In OASIS, we have the following kinds of dynamic formulas (set of class axioms):

- **Evaluations.** They are formulas of the form $\Phi \ [a] \Phi'$ whose semantics is given by defining a $\rho$ function that, from a ground action $a$ returns a function between possible worlds.

  $$\rho(a) \in W \rightarrow W$$

  In other words, with any valid state being a possible world for an object, the $\rho$ function determines which transitions between object states are valid after the execution of an action $a$. In the example of the Figure 4-5, we have the following evaluations:

  $$[\text{rent()}] \ rent\_count=\text{rent\_count}+1;$$

  $$\text{rent\_count mod 10<>0} \ [\text{return()}] \ rent\_count=\text{rent\_count}-1;$$

  It is important to realize that, within this dynamic logic environment, the formula $\Phi$ is evaluated in $s \in W$, and $\Phi'$ is evaluated in $\rho(a)$, with $\rho(a)$ being the world represented by the object state after the execution in $s$ of the action considered.

- **Derivations.** They are formulas of the type $\Phi \rightarrow \Phi'$. They allow us to define derived attributes ($\Phi'$) in terms of the given derivation condition (stated in $\Phi$).

  They basically differ from the evaluation formulas in the fact that these derived evaluations are done in a unique state. In our example, there are no derivations.

- **Integrity constraints.** They are formulas that must be satisfied in every world. We distinguish between static and dynamic.

  Static integrity constraints are those defined for every possible world. They must always hold. In the example:

  $$\text{static kilometers < 100000;}$$

  On the other hand, dynamic integrity constraints are those relating different worlds. They require the use of a temporal logic, with the corresponding temporal logic operators.

- **Preconditions.** They are formulas with the template $\neg \Phi \ [a] \ false$, where $\Phi$ is a formula that must hold in the world previous to the execution of action $a$. Only in the worlds where $\Phi$ holds, is $a$ allowed to occur. If $\neg \Phi$ holds, the occurrence of $a$ gives no state as successor. We have the following precondition in the OASIS specification:

  $$\neg(\text{rent\_count=0}) \ [\text{usr:destroy\_car()}] \ false$$

  or in a more convenient way for specification purposes we can write

  $$\text{usr:destroy\_car()} \ if \ \text{rent\_count=0};$$
• **Triggers.** They are formulas of the form $\Phi \ [\neg a \ false]$, where $\neg a$ is the action negation. If $\Phi$ holds and an action other than $a$ occurs, then there is no successor state. This forces $a$ to occur or the system remains in a blocked state. For instance, using the appropriate dynamic formula, where we include information about the destination in the triggered service (according to the trigger expressiveness presented when the OO-Method Interaction Diagram was introduced), we will declare:

\[
Rent\_count=0 \ [\neg (Self::disable()) \ false];
\]

Once more, for specification purposes, we will write in an equivalent but more conventional way:

\[
Self::disable() \text{ if } rent\_count=0;
\]

Thus, triggers are actions which are activated when the condition stated in $\Phi$ holds. The main difference between preconditions and triggers comes from the fact that, in triggers, there is an obligation to activate an action as soon as the given condition is satisfied. In this way, triggers allow us to introduce internal activity in the object society that is being modeled.

In any of these dynamic formulas, $\Phi$ and $\Phi'$ are well-formed formulas in a first order logic that usually refer to a given system state characterized by the set of values attached to attributes of objects in the state or world considered.

In OASIS, an object is defined as an observable process [111]. The *process specification* in a class allows us to specify object dynamics and determines the access relationship between the states of instances. Processes are constructed by using *events* as atomic actions. But the designer also has the choice of grouping events in execution units, so-called *transactions*.

The molecular units that are the *transactions* have two main properties:

- They follow an *all-or-nothing* policy with respect to the execution of the involved events: when a failure happens during a transaction execution, the resultant state will be the initial one.

- The *non-observability* of intermediate states.

We finish this section by introducing the process specification of the Car class from Figure 4-5.

\[
car=usr:new\_car()\text{car0};
\]

\[
car0=usr:destroy\_car()+disable() \text{ car2+rent() car1};
\]

\[
car1=return() \text{car0};
\]

\[
car2=unable() \text{car1};
\]

---

5 It means that $a$ does not occur.
The execution of processes are represented by terms in a well-defined algebra of processes, which is based on the approach presented in [112]. It allows us to declare possible object lives as terms whose elements are transactions and events. Every process can be rewritten to a term in a basic process algebra BPAδε with the • (sequence) and + (alternative) process operations. This provides an implementation of concurrency based on arbitrary interleaving.

This is a short introduction of the formal features of OASIS. A more detailed description of OASIS and the complete description of its semantics and formal foundations can be found in [113].

### 4.2.1. The Formal Specification as the System Repository

After having presented the OO-Method architecture, and the OASIS formal concepts attached to it, the basic mappings to translate the graphical representation to the textual one are presented. This process takes as input the graphical information introduced in any OO-Method conceptual schema and generates a textual system representation that is a specification in OASIS. This formal specification has in fact been obtained using conventional techniques, and constitutes a solid system documentation to obtain a final software product which is compliant with the initial requirements, as represented in the starting conceptual schema.

According to the class template introduced in the previous section, the set of OO-Method conceptual patterns and their corresponding OASIS representation are identified.

The system classes are all obtained from the Object Model. For each class, we have:

- its set of (constant, variable or derived) attributes
- its set of services, including (private and shared) events and local transactions
- the integrity constraints specified for the class
- the derivation expressions corresponding to the derived attributes

If we are dealing with a complex class (those defined by using the provided aggregation and inheritance class operators), the Object Model also provides the particular characteristics specified for the corresponding complex, aggregated, or specialized class.

With the information given by the Object Model, we basically have the system class framework, where the class signature is precisely declared for every class of the system. The Dynamic Model used two kinds of diagrams: State Transition Diagrams (STD) and Interaction Diagrams (ID).

For the STD, we obtain:
• Event preconditions (those formulas labeling the event transitions)

• The process definition of a class, where the template for valid object lives is fixed.

And from the Interaction Diagram, we complete two other features of an OASIS class specification:

• Trigger relationships

• Global transactions (those involving services of different class objects).

Finally, the Functional Model gives the dynamic formulas related to evaluations, where the effect of events on attributes is specified.

This is how having clearly defined the set of relevant information that can be introduced in an OO-Method conceptual schema, the formal specification corresponding to it provides a precise system repository where the system description is completely captured, according to the OASIS object-oriented model. This allows us to undertake the implementation process (Execution Model) from a well-defined starting point, where the involved pieces of information are meaningful because they come from a finite catalog of conceptual modeling patterns, that also have a formal counterpart in the OASIS language.

### 4.3. Execution Model

The OASIS specification is the source for an execution model that must accurately state the implementation-dependent features associated to the selected object society machine representation. In order to easily implement and animate the specified system, we predefine a way in which users interact with system objects. The template used to achieve this behavior is:

1. Identify the user
2. Obtain the object system view
3. Service activation
   3.1 Identify the object server
   3.2 Introduce service arguments
   3.3 Send the message to object server
   3.4 Check state transition
   3.5 Check preconditions
   3.6 Valuations fulfillment
   3.7 Integrity constraint checking in the new state
   3.8 Trigger relationships test

The process starts by logging the user into the system (step 1) and providing an object system view (step 2) determining the set of object attributes and services that it can see or activate. After the user is connected and has a clear object system view, he or she can then activate any available service in his worldview. Among these services, we will have observations (object queries) or local services or transactions served by other objects. Any service activation (step 3) has two steps: build the message and execute it (if possible). In order to build the message, the user has to provide information to
identify the object server (step 3.1). Subsequently, he or she must introduce service arguments (step 3.2) of the service being activated (if necessary). Once the message is sent (step 3.3), the service execution is characterized by the occurrence of the following sequence of actions in the server object:

- Check state transition (step 3.4), which is the process of verifying that a valid transition exists in the OASIS specification for the selected service in the current object state.

- The precondition satisfaction (step 3.5) indicates that the precondition associated to the service must hold.

If any of these actions do not hold, an exception will arise and the message is ignored. Otherwise, the process continues with:

- The valuation fulfillment (step 3.6), where the induced service modifications take place in the involved object state.

- To assure that the service execution leads the object to a valid state, the integrity constraints (step 3.7) are verified in the final state. If the constraint does not hold, an exception will arise and the previous change of state is ignored.

- After a valid change of state, the set of condition-action rules that represents the internal system activity is verified. If any of them hold, the specified service will be triggered (step 3.8).

The previous steps guide the implementation of any program to assure the functional equivalence between the object system specification collected in the conceptual schema and its reification in an imperative programming environment.
Chapter 5

5. Requirements Analysis Process

The first step in computer system development is gathering, understanding and representing user requirements. This process is recognized as a critical task since a great number of software failures are known to originate from poor requirements definition, and these errors are the most difficult to detect and correct during the later phases of system development. This added to the lack of means to ensure consistency between requirements specifications and subsequent phases of the software development process, makes it difficult to verify in early stages whether a specification is right or if there are any missing requirements.

In this approach, we propose a Requirements Analysis Process (RAP) as a bridge between the requirements specification and the conceptual modeling phase. The Conceptual Model is an essential model of the system that is defined in terms of the external functionality of the system and is independent from possible implementations. The purpose of the RAP is that given a specification of external interactions (Requirements Model), we can obtain a decomposition of this model in terms of classes (Conceptual Model) following a predefined set of traceability rules. These classes will capture the behavior and responsibilities specified in the Requirements Model ignoring the properties of underlying implementation layers.

5.1. Identification and Allocation of Responsibilities

A responsibility is a specific task an object is in charge of accomplishing. As Jacobson points out in [93], any single usage of the system (a Use Case) can result in a set of responsibilities that may extend completely throughout the system. There cannot be a one-to-one correspondence between a Use Case and a single class inside the system. There must be several classes that will collaborate in the response to each Use Case.
The behavior in a system should be exactly that which is required to provide the Use Case to the users of the system. In this proposal, the general behavior of the system is divided into meaningful pieces of information, which are Use Cases. In order to evaluate whether that Use Case is provided, the documentation of the task of identifying and allocating responsibilities should show what responsibility is allocated to which class for each Use Case.

This is a “divide-and-conquer” technique because we have to focus on only one Use Case description at a time. We must think in terms of classes and the interactions between them that are necessary to realize the scenario. The technique consists of walking through the Use Case description. For each step described or implied in the Use Case description, one or more responsibilities appear and the software engineer has to allocate it into a class. This class is searched in the subject domain. There are always two classes involved in each responsibility: one to recognize the need and invoke the responsibility (client), and another to carry out the responsibility (server). Each step described in the Use Case description will require one or several responsibilities in the system.

We emphasize that two separate and very important engineering decisions are being made for each responsibility:

- **Identify the responsibility.** This is a decomposition of the larger responsibility described in each Use Case description step. The granularity of the decomposition (how big each of the responsibilities are) is a key element of the decision. The semantic level of the name of the responsibility can serve as a guide, but experience is an important decision factor.

- **Allocate the identified responsibility.** This involves deciding which object component (class) should ask for the behavior (the actor) and which component (class) should provide the behavior (the server). The quality of these decisions will determine the richness and flexibility of the conceptual schema.

## 5.2. Interaction Diagram

Using Interaction Diagrams to show the internal view of a system for a scenario is not new. In the telecommunications industry, these diagrams have long been used to show the complex interaction patterns between various hardware or software components. When building an object-oriented system, Interaction Diagrams are a natural way of showing how the different objects work together to provide the required system services. Moreover, a documentation of the internal components of the object-oriented system, which includes Interaction Diagrams, allows software engineers to find out quickly and easily what should happen within the system under certain circumstances in response to a given external event. Even if not all special cases are modeled in this way, the scenarios are an invaluable help for all those who are not very familiar with the architecture of the system.

The description of required behavior involves two aspects: 1) the structural description of the participants and 2) the description of the communication patterns. The
structure of classes in a behavior and their relationships is called **Collaboration**. The communication pattern performed by instances of classes to accomplish a specific purpose is called **Interaction**.

In UML, the description of behavior is mainly made through interaction diagrams. The interaction diagrams come in two forms based on the same underlying information, specified by a **Collaboration** and possibly by an **Interaction**, but each form emphasizes a particular aspect of it. The two forms are **Sequence Diagrams** and **Collaboration Diagrams**. A **Sequence Diagram** can show a sequence pattern of communications between classes or objects required to realize a Use Case (scenario). A **Collaboration Diagram** can show an interaction organized around the classes or objects in the interaction and their relationships. It does not show time as a separate dimension, so the sequence of communications and the concurrent threads must be determined using sequence numbers.

To deal with the activities of identifying and allocating responsibilities into classes we use the Sequence Diagram. This Sequence Diagram will show the communication pattern between classes that are necessary to realize the Use Case. The structural description of the involved classes will be defined later as a result of merging all the different Sequence Diagrams obtained from the Use Cases.

UML indicates that the Sequence Diagrams are a means to model an aspect of the dynamic behavior of the system [11]. They can be used in the context of the whole system, of a subsystem, or they can be attached to a Use Case. Some authors indicate that a Sequence Diagram should be drawn (at least one per Use Case [67], [114]). When a Sequence Diagram is developed for a Use Case, the Use Case description can be used to develop at least the initial draft of the Sequence Diagram. Throughout the design process the Use Case diagram can be revised based on the results of the Sequence Diagram, and vice versa, until both models are appropriately tuned [101].

In the approach presented in this thesis, there is at least one Sequence Diagram per Use Case. One Sequence Diagram for the basic course of actions, and another for each alternative course of action (if any).

All the Sequence Diagrams, insofar as their purpose is concerned, are **descriptive** (they capture requirements by enabling software engineers and users to walk through the scenario and understand its operations, actors, and interactions). With respect to their content, they have **behavioral information** (help to identify actions, actors involved and operations carried out) using entity types rather than instance entities (level of abstraction: **type**). The **notation** used is semiformal (the one provided by the UML with some extensions) and, from the **life cycle** point of view, they have a **static** perspective (they exist as long as the documentation of the application exists).

### 5.2.1. Notation

A Sequence Diagram represents a **pattern interaction**, which is a set of messages among classes to realize a behavior. It has two dimensions: the vertical dimension represents time, and the horizontal dimension represents different classes. Time proceeds down the page and there is no significance to the horizontal ordering of the classes.
The emphasis in these diagrams is to graphically represent the pattern interaction between object classes by sending and receiving messages as time advances. The focus should not be on specifying detailed behavior with complex iterations and conditional messages.

Named boxes together with a vertical dashed line called the “lifeline” represent classes. Actors (outside of the system) are represented by a “stick man” figure with the name of the actor below the figure. The different kinds of messages used in Sequence Diagrams are described in section 5.2.3 debajo de. The schematic structure and notation of a Sequence Diagram is shown in Figure 5-1.

![Sequence Diagram](image)

**Figure 5-1. Structure and Notation of the Sequence Diagram**

There are situations in the Sequence Diagram where it is necessary to show that many different actors can interact with the system and have the same behavior. If these actors are in an inheritance relationship between actors, all the behaviors specified for the super class are inherited by the subclass. Therefore, only the super class is specified in the Sequence Diagram.

When the actors involved are not in an inheritance relationship, we use the UML extension mechanism of stereotyping by defining a *multi-actor* icon. This new icon includes the list of actors that share the same interactions in the current Sequence Diagram.

If the *multi-actor* is used, it is also possible to use an actor, which appears in the list of shared interactions, as an independent actor for particular interactions. An example is shown in Figure 5-2.
5.2.1.1. Message Labels

The message may also be labeled with a sequence number to show the order in the overall interaction. However, sequence numbers are usually omitted because the physical location of the message shows the relative sequences (they are sometimes useful for referencing).

Messages can also be labeled with:

- **Condition.** This is a Boolean condition (that if satisfied) allows the interaction to occur.

  The syntax is the following:

  \[ \text{Boolean-expression} \ \text{message-name} \]

- **Iteration expression.** Expression that specifies iteration.

  The syntax is the following:

  \[ \text{lower-bound}..\text{upper-bound} \ \text{DO} \ \text{message-name} \]

  \[ \text{FOR} \ \text{expression} \ \text{DO} \ \text{message-name} \]

  \[ \text{WHILE} \ \text{expression} \ \text{DO} \ \text{message-name} \]

  \[ \text{query-expression} \ \text{DO} \ \text{message-name} \]

  \[ \ast \ \text{DO} \ \text{message-name} \]
5.2.1.2. Blocks

When a set of messages (a block) is conditional or it represents iteration, it can be graphically represented by a box labeled with the condition or iteration expression. An example is shown in Figure 5-3. The syntax for a condition and iteration expression is:

\[
\begin{align*}
&\text{[Boolean-expression]} \\
&\text{[lower-bound..upper-bound]} \text{ DO} \\
&\text{FOR [expression] DO} \\
&\text{WHILE [expression] DO} \\
&\text{[query-expression] DO}
\end{align*}
\]

Figure 5-3. Sequence Diagram Notation with Blocks

5.2.2. Class Definition

The first step in creating a Sequence Diagram is to identify and define the participating classes in this pattern interaction. It was Jacobson [67] who first introduced the classification of entity, control and boundary classes for developing a robust system architecture. The idea was to distribute the behaviors specified in a Use Case description into those three classes.

- Entity classes represent real-life domain objects or concepts that are internal to the system. Examples are Customer, Rental and Car classes. External
actors usually have no direct contact with entity objects. Instead, they are accessed through boundary objects.

- **Boundary classes** handle the communication with external actors, and they encapsulate environmental-dependent behavior, thereby protecting the integrity of the entity object. Examples are windows, screens, and menus that are used for input and output.

- **Control classes** are transaction classes that capture a sequence of operations. These classes capture business rules and policies. Jacobson [67] and Larman [101] use control classes to handle a Use Case. A control class is also frequently called a Handler or a Controller [101], [115].

The UML refers to these class categories as stereotypes [11]. In our approach, there is no distinction between entity and control classes. Boundary classes are used in the Sequence Diagram to explicitly show the border between the system and the environment. In this way, actors communicate only with boundary classes, boundary classes communicate with (internal) classes, and (internal) classes communicate with each other. The relationship among actors and classes is summarized in Figure 5-4.

![Figure 5-4. Relationships between actors and classes](image)

### 5.2.2.1. Class Properties

**Note:** It may be considered too early to specify properties of this kind at this abstraction level and at this software development process phase. However, if the user provides this information, it could be recorded in the class definition without detriment to the other properties already specified.

The definition of class properties is for the whole system, that is, they are not related to a Sequence Diagram specifically. The properties that can be (optionally) defined during this phase are:
• **Attributes.** List of attributes that describe the state of objects of the class. It may include the name, a description, and the type of the attribute (integer, real, etc.).

• **Constraints.** List of integrity conditions that objects of the class must satisfy independently of the actual service or situation. The condition is a description in natural language.

• **Extension.** This is not a usual property when defining a class (at this abstraction level) but a useful one. This is a condition expression defined in a class (in natural language) that when satisfied invokes a Use Case.

The syntax is the following:

```
WHEN <condition-natural-language> CALL <Use Case name>
```

This is similar to the concept of extension in Use Case relationships with the difference that we only know explicitly the extension Use Case and not (all) the extended Use Cases.

The extended Use Case will be all the Use Cases in whose Sequence Diagrams appear the class, where the condition is defined, as a receiver class of a message with the stereotype <<service>> (see Message Classification in section 5.2.3) e.g. all the Sequence Diagrams that can change the state of the object making the extension condition to be true. As a result of this extension definition, all the induced relationships should be represented in the Use Case diagram.

In Figure 2-1, we have an example of an extension definition. Defining an extension property “when discount = true CALL Use Case Get Bonus” in the class Contract makes the relationship extend appear between the Use Cases Rent, ModifyContract and Get Bonus (the Use Cases Rent, ModifyContract modify the state of the class Contract).
5.2.3. Message Classification

When the Use Case receives a stimulus (an External Interaction), the system produces a set of interactions between its internal components (objects) as a response. These interactions are represented as messages in the Sequence Diagram and they can be classified according to their nature using the following UML stereotypes:

- \texttt{<<signal>>} message between an actor and the system.
- \texttt{<<service>>} message that updates the state of an object. The properties new/destroy/update can be used if the object will be created, destroyed or modified.
- \texttt{<<query>>} message to query the state of some object or set of objects.
- \texttt{<<connect>>} message to indicate that an object of the sender class needs to be connected (or related) to an object of the receiver class (this message can have other properties to specify the maximum and minimum number of objects to be selected and to indicate the purpose of the selection insertion or deletion).
5.2.3.1. Signal Messages

These messages are labeled with the stereotype <<Signal>>. They represent interactions between an actor (an external entity type) and the system. Actors communicate with the system by sending stimuli to boundary objects in the system, and the boundary objects send responses to actors. This kind of interaction of external entities with the boundary objects can be used to build the user interface as shown in [115].

A boundary class, called System, is used in Sequence Diagrams to explicitly show the border between the system and the environment. In this way, all the classes to the left of the class System represent actors and classes to the right of the class System represent (internal) classes. Actor classes are not allowed to communicate directly with classes (they do it through the class system) and vice-versa.

It is not necessary to show these stereotypes in Sequence Diagrams because it is easy to recognize them (all the messages to or from actors are interactions from this category).

An example of this kind of message (see Figure 5-6) occurs when a User introduces information to the system or receives a response from the system.

![Figure 5-6. Signal Message](image)

Additional properties are defined for this stereotype type as shown in Table 2.

<table>
<thead>
<tr>
<th>Stereotype</th>
<th>Properties</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;&lt;Signal&gt;&gt;</td>
<td>Direction</td>
<td>Input</td>
<td>Related to messages where the origin is an actor class and the receiver the boundary class system.</td>
</tr>
</tbody>
</table>
5.2.3.2. Service Messages

These messages are labeled with the stereotype <<service>>. They are interactions where the receiver class represents an object that changes its state when the interaction occurs.

We recognize two kinds of change of state: weak and strong.

- **Weak change of state** refers to the modification of the value of attributes (update).

- **Strong change of state** refers to the creation (new) or destruction (destroy) of objects of the receiver class (all the values of attributes in an instance are initialized or destroyed).

An example of this kind of message (see Figure 5-7) occurs when after the *User* introduces information in the system to *rent* a *Car*, the system reacts (internally) creating a new rental contract. This internal interaction produces a strong change of state inside of the system (a new contract is created).

![Figure 5-7. Service Message](image)

Additional properties are defined for this stereotype type as shown in Table 3.
### Table 3. Service Message Properties

<table>
<thead>
<tr>
<th>Stereotype</th>
<th>Properties</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;&lt;Service&gt;&gt;</td>
<td>Kind of Change</td>
<td>New</td>
<td>An interaction that represents the creation of a new object instance of the receiver class. It is a strong change of state.</td>
</tr>
<tr>
<td></td>
<td>Destroy</td>
<td>Destroy</td>
<td>An interaction that represents the destruction of an existing object of the receiver class. It is a strong change of state.</td>
</tr>
<tr>
<td></td>
<td>Update</td>
<td>Update</td>
<td>An interaction that represents the state change of an object instance of the receiver class. It is a weak change of state.</td>
</tr>
</tbody>
</table>

#### 5.2.3.3. Query Messages

These messages are labeled with the stereotype <<query>>. They represent queries on related objects or on a class population. In the context of Sequence Diagrams, the mechanism used to know the state of another object(s) is by means of this kind of interaction.

An example of this kind of message (see Figure 5-8) occurs when a Car is returned. The system should calculate how much to pay. This is done by an interaction between the class Contract and the class Rate. The Contract has to query the corresponding Rate to get an appropriate price for calculation.
Additional properties are defined for this stereotype type as shown in Table 4.

### Table 4. Query Message Properties

<table>
<thead>
<tr>
<th>Stereotype</th>
<th>Properties</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;&lt;Query&gt;&gt;</td>
<td>Type</td>
<td></td>
<td>Value Population: the query is performed on the population of the receiver class, and the result is a list of objects. Value State: the query is performed on a related object or objects. The return is a calculated (or observed) value.</td>
</tr>
<tr>
<td></td>
<td>Population State</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;&lt;name&gt;&gt;</td>
<td>Description</td>
<td>--</td>
<td>A natural language description of the query.</td>
</tr>
</tbody>
</table>

Figure 5-8. Query Message
### 5.2.3.4. Connect Messages

These messages are labeled with the stereotype `<<connect>>`. It captures a very important kind of interaction between participating objects (sometimes it is difficult to identify messages of this kind at first glance).

In the object-oriented model, an object can be composed of other objects or an object can be associated to other objects. Due to encapsulation, these cases are the only way there is for an object to know the state of another object. For this reason, when the object is created, you should indicate all its related objects or, later, you should relate or unrelate them. Because we are using Interaction Diagrams, and specifically Sequence Diagrams, interactions of this kind should be taken into account. We call them connect.

An example of this kind of message occurs when a Car is rented, a Contract is created, and a Customer and a Car should be connected to this contract. A Customer could (later) rent another Car but a Car cannot be rented again (at least while this contract is open – exclusive property). This is an interaction between the Contract, the Customer, and a Car.

![Diagram](image.png)

**Figure 5-9. Connect Message**

Additional properties are defined for this stereotype type, as shown in Table 5.
### Table 5. Connect Message Properties

<table>
<thead>
<tr>
<th>Stereotype</th>
<th>Properties</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Minimum Multiplicity</td>
<td></td>
<td>Minimum number of objects in the receiver class that may be (dis)connected to/from the sender of the interaction.</td>
</tr>
<tr>
<td></td>
<td>Maximum Multiplicity</td>
<td></td>
<td>Maximum number of objects in the receiver class that may be (dis)connected to/from the sender of the interaction.</td>
</tr>
<tr>
<td>&lt;&lt;Connect&gt;&gt;</td>
<td>Activity</td>
<td>Insert</td>
<td>The interaction is to connect the receiver to the sender.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Deleted</td>
<td>The interaction is to disconnect the receiver from the sender.</td>
</tr>
<tr>
<td></td>
<td>Exclusivity</td>
<td>Exclusive</td>
<td>This property is used only with the Insert Activity. It means that the receiver object/s will be connected for exclusive use of the object sender.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Not Exclusive</td>
<td>Not Exclusive means that the receiver object could be connected to as many sender objects as desired.</td>
</tr>
</tbody>
</table>

### 5.3. From Requirements Model to Sequence Diagrams

The Requirements Model is composed of three techniques at different abstraction levels: the Mission Statement, the Function Refinement Tree, and the Use Case Model. The Use Case Model, which refines the Function Refinement Tree representing each function (a leaf in the FRT) as a Use Case, is at the lowest level.

An individual Use Case describes the aspects related to communication in the Use Case diagram, and the behavior in the Use Case description (natural language organized in steps). The Use Case description has a basic course section and zero or
many alternate paths in the alternative section. There may be one Sequence Diagram for the basic course and one Sequence Diagram for each path in the alternative section.

- A Use Case will be described by one or many Sequence Diagrams, one for the basic course section and another for each alternative section (if any).

A Sequence Diagram is an Interaction Diagram, which is a graphical representation to show how a set of classes describes a pattern interaction between objects in the real world.

At this step, our mission is to discover the sequence of messages as observed in the inter-object space for each scenario, revealing the object types (classes) and characteristics of these messages.

The process starts building Sequence Diagrams for each Use Case by analyzing it at two levels:

- **Use Case diagram level**, the actors that communicate with the Use Case.
- **Use Case description level**, the set of steps or responsibilities to be performed in order to accomplish the Use Case.

At the **Use Case diagram level**, an actor can send and/or receive information to/from the Use Case. Every Use Case (coming from the Function Refinement Tree) has at least one actor, which will be the Use Case initiator (there can be more than one initiator) and other collaborator actors.

Actors (that represent external entity types) are potentially classes to be taking into account for the Sequence Diagram. However, this is not always true. Use Case actors often represent external entities (people, organizations, systems...) that exist in the real world (entities in the subject-domain) but not in the conceptual model.

- The software engineer should decide which actors (from the set of actors related to the Use Case) would participate as classes in the Sequence Diagram.

At this level, we are only interested in classes whose instances (active objects) could activate (call) services offered by classes in the Conceptual Model (this relationship between active objects and services is called the **Agent Relationship**, as explained in section 4.1 por encima de). Classes whose objects are passive (those that only offer services) are considered at the **Use Case description level**.

In Figure 5-10, there is an example of a Use Case representing a sale in a typical Point-of-Sale system. In the subject domain, the **Customer** initiates the Use Case, however inside the computational system the **Cashier** is responsible for initiating this process (so, the **Cashier** is considered as a class placed to the left of the **System** class in the corresponding Sequence Diagram). The **Customer** comes into the store, participates anonymously, and leaves again. Another problem domain could maintain information on **Customers** in order to offer a better service. For example, the store could issue cards with machine-readable codes. The **Cashier** (or even the **Customer**) could run the card through a scanner and start the sale operation.
At the Use Case description level, the Use Case contains a textual description of the functionality offered in terms of its compound steps. The basic course and the alternative sections of the Use Case description classify these steps according to their nature (General, Actor/System communication and System response) as was explained in 3.2.3.1.2.

*General steps* (first column in Use Case description) represent general activities performed by actors in the context of use of the Use Case. These activities are outside the scope of the system but are recorded in the Use Case description only to help the software engineer to better understand under what conditions the Use Case communicates with its environment.

- *General steps* are not represented in Sequence Diagrams.

*Actor/System communication steps* (second column in Use Case description) represent specific actions accomplished by actors to interact with the system, or performed by the system to query or offer information to actors. They are related to user interface requirements.

- Each *Actor/System communication step* in the Use Case description will be represented in the Sequence Diagram by one or more messages between the boundary class (usually called *system*) and an actor class (class to the left of the boundary class).

*System response steps* (third column in Use Case description) represent the internal reaction of the system when receiving an external stimulus (or when some condition holds for active systems). Essentially, when the Use Case receives an external stimulus the system produces a response that may consist of:

- A state update
- An object creation
- Object deletion
- A query

This response is specified in natural language in the *System response* column in one or many steps. Each step may correspond to one or more responsibilities that the software engineer has to name and allocate into a class (the server) in the Sequence Diagram. Also, the software engineer has to identify the class that recognizes the need and invoke the involved responsibility (the client).
Each System response step in the Use Case description will be represented in the Sequence Diagram by one or more messages between the boundary class (usually called system) and entity classes (class to the right of the boundary class), or by one or more messages between entity classes.

Finally, the Sequence Diagram will show all the classes necessary to realize this Use Case. Each participating class gets a named lifeline (vertical dotted line). The communications between participating classes in each individual responsibility (as client and as server) are shown as arrows between the time lines. The sequence of messages down the time line represents the actual sequence of messages in that particular Use Case.

5.3.1. Completeness of a Sequence Diagram specification

The definition of completeness for the Sequence Diagram specification depends on the goals defined on this kind of specification. A Sequence Diagram specification for the system analysis has goals other than a Sequence Diagram specification for a detailed design (and thus other completeness criteria).

A user interface specification with Sequence Diagrams has to show the internal view of the user interface subsystem down to the level of single menu events or text entries. The specification should have two goals: to define the external design of the user interface and to validate it with users, and to make the internal design of the user interface subsystem ready for implementation. In this case, the targeted audience could be the users and the software engineer.

In our approach, the Sequence Diagram specification has to show all the internal interactions needed to realize the Use Case (scenario). After receiving a stimulus at the system boundary (maybe after a complex dialog at the interface level for data entry) the system should appropriately react by sending and receiving messages among their compound objects.

The specification has three goals:

- To define the sequence of atomic interactions among classes needed to realize the scenario,
- To check the validity of the scenario specification with users in a more precise language than the natural one,
- To classify the identified interactions according to their nature.

The specification is considered finished when it is deemed understandable, correct (validated by user), complete (i.e. it describes all the interactions among object classes), and consistent. In this context, it is important to emphasize that two separate and very important engineering decisions are made for each responsibility specified in the Use Case description: identify the responsibility (the granularity of the
corresponding interaction, or set of interactions, is a key element) and allocate the identified responsibility.

The quality of these decisions will determine the richness and flexibility of the resultant conceptual schema.

5.4. Requirements Traceability

Requirements traceability, which is the ability to relate requirements specifications with other artifacts created in the development life cycle of a software system, has wide and long been recognized as a significant factor for efficient software project management and software systems quality. This is because traceability relations can be used to:

- Verify that a system meets its requirements
- Establish the impact of changes in the requirements specification of a system on other artifacts in its documentation (e.g. design, testing and implementation artifacts) and vice versa, or
- Understand the evolution of software artifacts starting from the requirements document until the code.

The definition, which is most commonly cited in the literature, is:

"A software requirements specification is traceable if (i) the origin of each of its requirements is clear and if (ii) it facilitates the referencing of each requirement in future development or enhancement documentation" [116].

The standard that this definition comes from ANSI/IEEE Std. 830-1998 that further recommends:

- Backward traceability to previous development stages, which "depends upon each requirement explicitly referencing its source in previous documents"
- Forward traceability to all documents spawned from the software requirements specification, which "depends upon each requirement in the software requirements specification having a unique name or reference number".

This definition requires requirements traceability to be established (bi-directional) between all the documents generated and used throughout the system development.

Based on this definition, Gotel defines requirements traceability [117], [118] in the following way:

"Requirements traceability refers to the ability to describe and follow the life of a requirement, in both a forward and backward direction (i.e., from its origins, through
its development and specification, to its subsequent deployment and use, and through periods of on-going refinement and iteration in any of these phases”.

5.4.1. Classification

The most referenced traceability classification in the literature is the one proposed by Gotel [117], [118]. She suggests that requirements traceability can be divided into two basic types: Pre-requirements specification (pre-RS) and Post-requirements specification (post-RS) traceability. This classification is focused on the written specification of requirements (the RS):

- **Pre-requirements specification (pre-RS) traceability** is concerned with those aspects of a requirement's life prior to its inclusion in the RS (requirement production).

- **Post-requirements specification (post-RS) traceability** is concerned with those aspects of a requirement's life that result from its inclusion in the RS (requirement deployment).

The former is concerned, not only with the ability to record and access the origin of a requirement, but also any additional information which can help describe what has existed or happened prior to its inclusion in the RS. The latter is concerned with any such information related to a requirement's use.

Figure 5-11 shows the typical setting of requirements traceability to illustrate this distinction. Note the way in which requirements knowledge is distributed and merged in successive representations. Also note the added complication of iteration and change propagation.

The primary differences between these two types of Requirements Traceability involve the information they deal with and the problems they can resolve. Post-RS traceability depends on the ability to trace requirements from, and back to, a baseline document (the RS), through a succession of documents and products in which they are
distributed. When changes are made to this baseline, they need to be re-propagated through this chain of distribution. Pre-RS traceability depends on the ability to trace requirements from, and back to, their originating statement(s), through the process of requirements production and refinement, in which statements from diverse (often conflicting and overlapping) sources are eventually integrated into a single requirement in the RS. Any changes in the process need to be re-worked into the RS. When changes are made to the RS, they need to be carried out with reference to this production and refinement process.

In this thesis, we focus on *Post-RS traceability*, and more specifically going from the Requirements Model (where requirements are identified and specified) to the Conceptual Model (where requirements are partitioned into class components).

The Requirements Model is organized in different abstraction levels, which gets less abstract as the system development process advances. For this reason, traceability links are also necessary inside this model. In Figure 5-12, (1) and (2) represent what we call the *Internal-RS traceability*.

- **Internal-RS (In-RS) traceability** is concerned with those aspects of a requirement's life after its identification and during its full specification inside of the Requirements Model.

In Figure 5-12, (3) and (4) represent the *Post-RS traceability* (that is going from Requirements Model to the Conceptual Model in two phases, through the intermediate level represented by the Requirements Analysis Process - RAP).

In step (3), the first step of this *Post-RS traceability*, the software engineer still works with the Scenario or Use Case concept in mind and he or she has to represent it in terms of communicating objects in the Sequence Diagram (this is the real change of abstraction level). Later, in step (4) of the *Post-RS traceability*, the software engineer builds the conceptual schema helped by a Traceability Rules Catalog (see section 5.6 debajo de). This transition should be precise and powerful enough to be automated and flexible at the same time.

Martin Fowler in [118] says that “Conceptual Models are not right or wrong: they are more or less useful”. The result of this is that we can have two or more equivalent conceptual schemas and one of them could be considered more useful than the other (depending on the domain to be used, the technology in which it will be implemented, in priorities of users, etc.).

For this reason, if a conceptual schema is built from a Requirements Model (what we argue in this thesis) some structures identified at requirements level will have a unique representation at the conceptual model level, and other structures could have many representations. In order to deal with this kind of relationships between requirements and conceptual schemas, we need another sub classification for *Post-RS traceability*, that is, of *weak* and *strong* traceability.
• **Weak traceability** is concerned with those aspects of the Requirements Analysis Process (Sequence Diagram) that will be translated to elements in the conceptual schema but that can be changed or even deleted on destination. This means that aspects with weak traceability will only be proposed as elements in the conceptual schema.

• **Strong traceability** is concerned with those aspects of the Requirements Analysis Process (Sequence Diagram) that will be translated to elements into the conceptual schema that cannot be changed or deleted on destination. This means that aspects with strong traceability will be mandatory elements in the conceptual schema.

![Diagram](image)

Figure 5-12. Traceability between the Requirements and Conceptual Models

### 5.5. From Sequence Diagrams to Conceptual Schemas

Most of the existing approaches to Post-RS traceability assume that software developers create and maintain traceability relations manually [119], [120]. This clearly makes the establishment of traceability a time-consuming and error-prone task. As a consequence traceability is rarely established in industrial settings where systems are documented by large collections of artifacts, which are often expressed in natural language (e.g. IEEE Std 830-1998 requirements specification [116]). In [121], a
traceability metamodel is presented as an extension of the UML metamodel and it is used in a prototype traceability environment called SmarTTrace [122].

In our approach, we propose to solve the Post-RS traceability problem (or at least minimize it) in three phases:

- Gathering and representing requirements by means of a precise and well-defined but flexible Requirements Model
- Using a precise Conceptual Model
- Defining appropriate mechanisms to derive the Conceptual Model (at least its main architecture) from the Requirements Model

In Figure 5-13, we show a simplified signed map to go from the set of specified requirements to specific elements in the Conceptual Schema. This step is also called synthesis following the definition in Webster’s dictionary [123]: 1) “The composition or combination of parts or elements so as to form a whole” 2) “The combination of often diverse conceptions into a coherent whole”

![Figure 5-13. Traceability from Requirements to Conceptual Model](image)

The Conceptual Model in OO-Method is composed of an Object Model, a Dynamic Model, a Functional Model, and a Presentation Model as was explained in Chapter 4.
Following, an overview of the information obtained from the Requirements Model and the Requirements Analysis Process is presented. A detailed Traceability Rules Catalog is presented in Chapter 6.

- For the Object Model, we have the Class Diagram.

  - **Class Diagram.** The process performs an analysis of Sequence Diagrams looking for classes, services, attributes, integrity constraints, associations, aggregations, inheritances, and agent relationships.

  - **Classes** appear in the horizontal dimension of Sequence Diagrams. This information is directly available. All the classes must be considered except the class System.

  - **Services** are all the messages received by a class with the stereotype <<service>>.

    Moreover, the set of Sequence Diagrams that represents a scenario⁶ (the one that realize the basic course section, and each one of those that represents the alternative sections) are also considered a service. In particular, it is a transaction. If this transaction is allocated to a specific class, it is considered a local transaction (and represented as a service in the Object Model). Otherwise, it is a global transaction (and represented in the Interaction Diagram of the Dynamic Model).

    Preconditions for services can also be obtained from the Use Case description (not for all services, only for those that represent a scenario. See Figure 3-6 for Use Case description structure)

  - **Attributes** and integrity constraints can also be obtained from the Sequence Diagram (class properties) to fulfill the Object Model. These properties can be defined (optionally) when the class is identified during its use in a scenario specification (as explained in section 5.2.2.1). This information is directly available. Other attributes can be obtained from arguments of messages in the Sequence Diagram.

  - **Association and aggregation relationships** can be obtained looking for messages with the stereotype <<connect>> and some patterns of service sequences.

---

⁶ Scenarios and Use Cases are generally used as synonyms. In this context, the term scenario is used to describe the complete specification of an External Interaction in contrast with the term Use Case. It is possible to define secondary Use Cases that specify partial behavior in order to manage complexity. These Use Cases can be reused in other Use Cases or they can simple package a well-defined but partial behavior.
Inheritance relationships can be obtained from the Use Case diagram when specializations between actors are defined (only if they are considered as classes in the Sequence Diagram).

Agent relationships represent activation authorization relationships for objects of a class on services offered by classes. This information is obtained from classes (initiators) that appear to the left of Sequence Diagrams (only if Sequence Diagrams represent scenarios).

- For the Dynamic Model we have two diagrams: the Interaction Diagram and the State Transition Diagram.

  - Interaction Diagram. The process performs an analysis of Sequence Diagrams looking for Triggers\(^7\) and Global Transactions.

  - Triggers can be obtained from the Sequence Diagram (class properties) to fulfill the Interaction Diagram. This property can be defined (optionally) when the class is identified during its use in a scenario specification (as explained in section 5.2.2.1). This information is directly available.

  - Global Transactions are groups of services involving different non-structural related objects. Every Sequence Diagram that represents a scenario is a candidate to be a Global Transaction. If so, the software engineer identifies it and gives it a name (by default the same name of the corresponding Use Case with the prefix “TR_”).

  - State Transition Diagram. This diagram represents behavioral patterns that are valid object life cycles for every class. By valid life, we mean an appropriate sequence of services that characterizes the correct behavior of the objects that belong to a specific class.

A Sequence Diagram describes a partial behavior of the system, and State Transition Diagrams describe the complete behavior of objects in terms of valid sequences of services. The proposal intends to generate a (basic) State Transitions Diagram for each identified class that will be “hand modified” by the designer.

The technique extracts all the classes and services from Sequence Diagrams and proposes a (basic) State Transition Diagram per identified class where creation, deletion, and update services are organized.

- For the Presentation Model we have Presentation Patterns.

\(^7\) Triggers are services that automatically are activated when a condition is satisfied.
- **Presentation Patterns.** The process (by now) performs an analysis of the *Function Refinement Tree* in order to capture the information required to characterize what aspect the application will have, and how the user will see the organization of the External Interactions identified [124]. This is specified by the Hierarchical Action Tree (HAT) pattern. The complete specification of the UI interaction required to execute a scenario, such as the one presented in [115], will be performed in a future work.

- Finally, for the *Functional Model*, there are descriptions of the change of an object state as a consequence of a service occurrence.

- In the Requirements Model, we do not capture information related to the state of objects. For this reason, all the information related to change on the object state should be introduced later on during the Conceptual Modeling phase.

### 5.5.1. Synthesizing Object Behavior from Scenarios

Sequence Diagrams describe system behavior in terms of the interaction scenarios between multiple objects (inter-object behavior). State Transition Diagrams (or Statecharts) on the other hand, describe the behavior of a single object (intra-object behavior). Ultimately, if behavioral models are going to be used for simulation, detailed design, or implementation, an appropriate relationship between inter and intra-object behavior should be proposed.

The importance of generating behavioral descriptions directly from expected scenarios of system behavior was noted as early as 1987 in Harel’s original paper on Statecharts [125]:

> [...] many of the people that were involved in the avionics project [...] were able to state many desirable scenarios, such as firing a missile or updating the aircraft’s location, in precise detail [...] it would seem beneficial to be able to derive a *reasonable* statechart description from a large set of scenarios [...].

Despite recent work on this synthesis problem, only limited consideration has been given to what kind of synthesis algorithm could produce a *reasonable* statechart. Since, in general, scenarios are partial descriptions, the generated statechart will most likely be modified by the system designer. Hence, it must be concise and readable. Whittle and Schumann in [126] propose some characteristics that a synthesis algorithm should have:

- Scenarios must be consistent which each other – any synthesis engine should detect (at least partially) these inconsistencies;

- Scenarios will likely duplicate behaviors – these should be merged during synthesis, thus leading to an optimized, concise statechart representation;
• Generated statecharts should be readable – they should be understandable.

• Generated statecharts will be modified manually – the synthesis engine should support iterative refinements by checking the original scenarios against the modified statechart.

The algorithm proposed by Whittle and Schumann in [126] uses messages from Sequence Diagrams that are interpreted and specified with their pre-conditions and post-conditions in order to check that event sequences are valid. Each event is annotated using partial information from these pre/post conditions. This result in an annotated sequence diagram which is then translated into a set of finite state machine, one for each scenario object. Once all the sequence diagrams have been considered, the state machines for each object are merged using the annotations as a guide in order to merge states where possible.

Another related work in this area is the one proposed by Somé in [127], where different alternatives for the same scenario are combined to obtain a single State Transition Diagram skeleton. It is an incremental process where scenarios are obtained one by one and the system’s entire behavior is constructed as one goes along. A scenario is a possible chain of events and associated temporal constraints. The process analyzes new scenarios and merges their partial behavior with that obtained from scenarios previously acquired. This activity produces a specification skeleton that accomplishes all the requirements included in the scenarios. Composing scenarios may also unveil contradictions between them, causing a return to requirements acquisition.

In the work of Koskimies et al. [128], state diagrams are synthesized from Sequence Diagrams as a problem of grammatical inference. This is an algorithm where the analyst has to respond to questions in order to propose a solution.

Harel et al. [129] have developed an inter-object, scenario-based approach to reactive system specification. It is based on the graphical language of live sequence charts, LSCs [130]. This language is used because the Sequence Charts (MSCs or their UML variant, sequence diagrams) possesses a rather weak-partial order semantics that make it impossible to capture many kinds of behavioral requirements of a system. This is mostly due to the fact that they do not distinguish between possible and mandatory behavior and are thus weaker than temporal logic or other formal languages for requirements and constraints.

Despite the visuality, LSCs constitute a formal language, which may not always be appropriate for the people involved in the early stages of requirements capture, and tool support becomes necessary. The tool supporting the play-in/play-out approach is called the Play-Engine. The Play-Engine environment provides features that allow playing in the behavior directly from a GUI that builds LSCs automatically. It then allows us to play out just as if it were an intra-object model.

In our approach, the technique only extracts all the classes and services from Sequence Diagrams and proposes one (basic) State Transition Diagram per class where creation, deletion, and update services are organized. Although we made an initial effort to generate more complete State Transitions Diagrams, the result was that human decisions were always needed. Sequence Diagrams (as pointed out by Harel) do not
have enough expressiveness to reason about relationships and behavioral requirements among different scenarios.

The decision at that point was to define or use a more expressive representation for scenarios (e.g. MSCs or LSCs) or to keep it simple by trying to specify only participating objects and annotated (stereotyped) main interactions. The former would increase the specification complexity, making our approach more difficult to use. The latter would decrease behavioral expressiveness but, in our point of view, it would still be expressive enough to derive the structure, the communication, and part of the behavior.

Further work will address the problem of precedence, pre-condition, and post-condition among scenarios as well as Whittle and Schumann’s characteristics for behavioral synthesis algorithms.

### 5.6. Traceability Matrix

To assure that each requirement specification is realized by some part of the Conceptual Schema, a *traceability matrix* is built. The matrix traces the correspondence between the requirements and elements from the Conceptual Schema. Assuring the traceability of requirements to a Conceptual Schema is achieved by verifying that: all requirements are listed in the matrix; and that a class or operation is assigned to fulfill each requirement.

In this context, the verification consists of reviewing each one of the Use Cases and its corresponding Sequence Diagrams with the user to see that they provide the behavior implied in the Use Case specification. That is, we look for completeness and feasibility. The Conceptual Schema in OO-Method is composed of an Object Model, a Dynamic Model, a Functional Model and a Presentation Model as was explained in chapter 4.

For the *Object Model*, the verification we carry out involves looking at all the identified classes and their services in the Sequence Diagrams. The services are all the messages received by the class with the stereotype *service*.

This information is directly and immediately available from the Sequence Diagrams. Incomplete structure or behavior in a class or Class Diagram either indicates that some of the Sequence Diagrams missed some behavior, or that there are missing Use Cases (e.g. a class without a service to create objects). The solution is to reexamine the Use Cases and the corresponding Sequence Diagrams to find that missing behavior.

For traceability purposes, we can represent the allocation and flowdown of Use Cases to classes by means of a Traceability Matrix. The top row lists all the Use Cases of the system and the leftmost column represents all the identified classes extracted from the Sequence Diagrams. The flowdown is done by assigning a set of classes (the ones that have the responsibility to accomplish that functionality) to each Use Case.

A cross in an entry means that the class plays a role in the function, without yet indicating what this role is; absence of a cross indicates that this class plays no role in
the realization of this Use Case. If we add more information and indicate in each entry how the class will participate in order to play its role in the system function, then it is called function decomposition table.

In the realization of the Use Case, there are one or many Sequence Diagrams with stereotyped messages. These stereotyped messages give us information which is related to instances that must be created (C - stereotype service/new), read (R - stereotype query), updated (U - stereotype service/update) and deleted (D - stereotype service/destroy). Also, we include the information related to connections between objects, this is when an object is “associated” with another object (the <<connect>> stereotype signed in the table with “A”). This Function Decomposition Table (also called CRUD+A table) is shown in Figure 5-14.

<table>
<thead>
<tr>
<th>Use Case 1</th>
<th>Use Case 2</th>
<th>...</th>
<th>Use Case n</th>
</tr>
</thead>
<tbody>
<tr>
<td>class 1</td>
<td>C</td>
<td></td>
<td></td>
</tr>
<tr>
<td>class 2</td>
<td>R</td>
<td>C</td>
<td>RD</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>class n</td>
<td>U</td>
<td>U</td>
<td>C</td>
</tr>
</tbody>
</table>

Figure 5-14. Traceability Matrix (CRUD+A Table)

If one of these operations is missing, this is not wrong in itself, but it should be justified. Objects that are never created during normal system operation should be created at system initialization time. Objects that are never deleted should be relevant during the entire life of the system. Objects that are never read or updated are probably not useful and could be omitted. Those objects that are not updated apparently represent an unchanging part of the system.

Figure 5-15. Traceability Matrix explained
Chapter 6

6. **Traceability Rules Catalog**

These rules specify structured mechanisms to analyze Sequence Diagram to generate the main components of the Conceptual Schema.

Although it is not possible to generate a complete Conceptual Schema from Requirements, the resultant Conceptual Schema *skeleton* represents the main framework that the software engineer should refine and complete to have a precise representation of the software to be developed. In this way, automatic traceability structures are established between the main components of both models.

Our experience using this Traceability Rules Catalog indicates that, it represents a step forward in Requirements Engineering and Conceptual Modeling, whereas most approaches focus mainly on the Requirements specification or on the Conceptual Model design giving fuzzy relationships between these worlds.

The hierarchical relationships between elements described in this catalog are shown in Figure 6-1.
The Traceability Rules Catalog is organized by grouping rules according to the kind of result they produce. The catalog organization is the following:

- Class
- Attribute
- Service
- Association Relationship
- Cardinality
- Aggregation Relationship
- Agent Relationship
- Inheritance Relationship
- Preconditions
- Integrity Constraints
- Triggers
- Static and Dynamic Association-Aggregation Relationship
6.1. **Class Rules**

The generation of classes for the Class Diagram is a process which is based on the analysis of participating actors and classes in all the Sequence Diagrams.

- **Rule 1.** For every distinct actor class participating in any Sequence Diagram a class will be generated in the Class Diagram.
  
  - **Traceability:** Strong

- **Rule 2.** For every distinct class participating in any Sequence Diagram a class will be generated in the Class Diagram.
  
  - **Traceability:** Strong

  In OO-Method conceptual schemas, the class *system* is not represented explicitly because it implicitly exists as the parallel composition of all the identified classes of the system.

- **Rule 3.** The boundary classes (usually called System) in Sequence Diagrams will not have an explicit representation in the Class Diagram.

  - **Traceability:** Strong

  A Sequence Diagram for renting a car is shown in Figure 6-2.
Applying rules 1, 2 and 3 to this Sequence Diagram, the classes than can be obtained are: **Contract**, **User**, **Customer** and **Car**.

Following, an excerpt of the corresponding Class Diagram is shown in Figure 6-3. These classes will have strong traceability. This means that classes cannot be deleted or modified (with respect to these rules to modify its name) in the Class Diagram without updating the requirements. It is also not possible to add new classes that do not trace to any class in a Sequence Diagram.

![Figure 6-2. SD for Renting a Car](image)

**6.2. Attribute Rules**

In the Sequence Diagram, we focus on specifying behavior in terms of a sequence of message exchanges. To perform this activity, the software engineer has to understand the corresponding scenario and decompose the system into meaningful pieces (in the Object-Oriented approach, these pieces are classes).
Although it is not the purpose of the Sequence Diagram specification to think in terms of attributes that constitute each one of the classes identified, the software engineer can register meaningful information related to these classes if desired (see Class Properties in section 5.2.2.1 por encima de).

- **Rule 4.** All the attributes recorded in a class (when it is created in a Sequence Diagram or modified later) will be considered as attributes for the class in the Class Diagram.

- **Traceability:** weak

Moreover, there are some messages in the Sequence Diagram that can give us information related to the attributes of a class. They are the services that are labeled with the stereotype <<service/new>>. These messages create a new instance in the receiver class. Usually, arguments of these messages are used to give values to the attributes of the new instance, but not always. Moreover, it is a good rule of thumb to propose them as attributes of the receiver class with *weak* traceability. This means that these attributes will be *proposed information* in the class model and the software engineer could modify, delete or add new attributes without having to update the referring requirements.

- **Rule 5.** For every message labeled with the stereotype <<service/new>>, all its arguments will be translated into attributes of the receiver class.

- **Traceability:** weak

A Sequence Diagram for creating (Purchase) a car is shown in Figure 6-4. Applying **rule 5** to this Sequence Diagram we have the class *Car* with the attributes: *platenumber, state, color, class* and *group*.

An excerpt of the resulting Class Diagram is shown in Figure 6-5.
6.3. Service Rules

In section 5.2.3 por encima de, we have presented a message classification. In that classification, messages can be stereotyped as one of the following: <<signal>>, <<query>>, <<service>> or <<connect>>.
All the messages with the stereotype <<service>> will be translated into services of the receiver class. The purpose of messages of this kind is to change the state of an object that belongs to the receiver class of the message.

- **Rule 6.** For every message with the stereotype <<service>>, a service will be generated in the corresponding receiver class.

- **Traceability:** strong

### 6.3.1. New and Destroy Service Rules

In the Object-Oriented model, we recognize two kinds of change of state: *weak* and *strong*.

- *Weak change of state* refers to the modification of the value of attributes on instances of a class (update).

- *Strong change of state* refers to the creation (new) or destruction (destroy) of instances of the class (all the values of attributes in an instance are initialized or destroyed).

These two types of change of state are used to set the property *Kind of Change* of the stereotype <<service>> that is defined in the message classification of section 5.2.3.

If the change of state refers to a *weak change of state* the property is set to the value “update” and if the change of state refers to a *strong change of state* the property is set to the value “new” or “destroy” if and object is created or destroyed, respectively. In OO-Method, services that produce strong change of state are known as *new* and *destroy* services.

A *weak change of state* is assumed by default if the property *Kind of Change* is not explicitly set.

- **Rule 7.** For every message with the stereotype <<service/new>>, a service of type “new” will be generated in the corresponding receiver class.

- **Traceability:** strong

- **Rule 8.** For every message with the stereotype <<service/destroy>>, a service of type “destroy” will be generated in the corresponding receiver class.

- **Traceability:** strong
6.3.2. Transaction Rules

A Use Case is an interaction between the system and an external entity that has a use for that external functionality. This interaction need not be atomic. It is usually decomposed into steps that are specified first, in natural language in the Use Case description, and later, specified more precisely using Sequence Diagrams in the Requirements Analysis Process.

If a Use Case stands for an interaction between the system and an actor (external entity), this means that a Use Case will be represented (reified) in some way in the system so that the actor could interact with it.

In our approach, this representation is a service and more specifically, a transaction, which is a molecular interaction unit.

A Use Case has a basic course section and zero or many alternate paths in the alternative section. The detailed specification of a Use Case using Sequence Diagrams stated that we should have one Sequence Diagram for the basic course and one Sequence Diagram for each path in the alternative section.

- **Rule 9.** Every Sequence Diagram will be translated into a transaction. The name of the transaction is “implicit and partially defined” with the same name as the Sequence Diagram prefixed by “TR_”.

- **Traceability:** strong

  It is very important to consider transactions as implicit and partially defined from the very beginning of Sequence Diagram specifications because we can then avoid the problem of precedence of specifications.

  The problem with precedence of specifications is that every Use Case must be realized or it should at least be named before using it. This way, Sequence Diagrams will have a fixed name and can be referenced as soon as the scenario is defined as a Use Case (implicit). We say that the transaction is partially defined because it is not internally specified (we only know its name and not all the service components).

  Another problem related to transactions is that we do not know which class provides a transaction.

  These problems are dealt with rules 10 and 11.

- **Rule 10.** The transaction definition will be composed of all the services derived from the messages with the <<service>> stereotype found inside of the current Sequence Diagram.

- **Traceability:** strong.

  The application of this rule can give us an important help in the system design and, in particular, in transaction design. In spite of the fact that the information provided is only the list of component services of the transaction, this facilitates traceability while flexibility is maintained. The transaction designer can freely add conditions, control
paths, iterations, etc. in the transaction specification without worrying about backward traceability. This detailed specification should be done in the Conceptual Modeling phase (specifically in the Class Diagram) following the syntax of OASIS for well-formed transaction formulas.

In order to define which class will offer the transaction (service) to the environment, we have to consider two types of transactions: local and global transactions.

- **Local** transactions belong to one class of the system,
- **Global** transactions belong to the whole system (that is the implicit class that represents the parallel composition of all the classes of the system).

There is no reliable information in the scenario specification that could tell us which class should be the owner of the transaction. However, as a guideline, some heuristics can be helpful.

In the Sequence Diagram, the initial dialog between the actor and the system boundary take place to introduce the relevant data (left part of the Sequence Diagram). After this, we have to analyze the internal interaction of the system (the right part of the Sequence Diagram). If the class system interacts with only one class inside of the system with a message with the stereotype <<service>>, and if there is no recurrence in the message (iteration), this is a typical case of a local transaction.

- **Rule 11.** The transaction will be considered as a local transaction if, in the scenario, there is only one message with the <<service>> stereotype leaving from the class system. In other cases, it will be considered a global transaction.

- **Traceability:** weak.

Note that there can be other messages with the stereotype <<service>>, but not leaving from the class system. This is because, if one object is responsible for (actively) propagating the change of state into another object inside of the system, it should “know” the other object. We know from the object-oriented model that this is only possible if the object is related to the other(s). If they are related, the transaction is local.

The Sequence Diagram related to the Use Case DeleteInsurance that performs the deleting of the insurance policy of car is shown In Figure 6-6.
Applying rule 9, 10 and 11 to this Sequence Diagram, we have the local transaction $TR_{DeleteInsurance}$ in the $Insurance$ class. It has the following composition in OASIS notation:

$$TR_{DeleteInsurance}() =$$
$$InsuranceCompany.modify\_number\_of\_insurances(integer) .$$
$$delete\_insurance(string)$$

An excerpt of the resulting Class Diagram is shown in Figure 6-7.

In Figure 6-8, we have an example of a Sequence Diagram producing a global transaction. This is a scenario (Use Case Change Currency) with the goal of changing the currency values of $Invoices$ and $Products$ (but not just those products related to the changed invoices). Note how, after introducing the relevant information in the Sequence Diagram, there are two messages from the class system that modify the entire population of $Invoices$ and $Products$, respectively.
Applying rule 9, 10 and 11 to this Sequence Diagram, we have the global transaction \( TR_{\text{ChangeCurrency}} \). It has the following composition in OASIS notation:

\[
TR_{\text{ChangeCurrency}}() = (\text{For all}) \text{Invoice.ChangeCurrency} . \\
(\text{For all}) \text{ChangeCurrency}()
\]

### 6.3.2.1. Include Relationship

In the Use Case diagram notation, we have the Include relationship. This is a binary relationship between Use Cases, which allows a base Use Case to perform an explicit inclusion of an included Use Case. This relationship is graphically shown in Figure 6-9.

These Uses Cases are represented as Sequence Diagrams, where the \( \text{CreateOperation} \) Sequence Diagram performs an explicit inclusion of the \( \text{Disable} \) Sequence Diagram. This explicit inclusion of the included Sequence Diagram is represented in OASIS notation as a service invocation in the base Sequence Diagram.

- **Rule 12.** For INCLUDE relationships between Sequence Diagrams, the transaction generated from the base Sequence Diagram will perform an invocation of the transaction generated from the included Sequence Diagram (the included transaction is not changed).

- **Traceability:** strong.
The CreateOperation and Disable Sequence Diagrams from the Use Case diagram of Figure 6-9 are shown in Figure 6-10 and Figure 6-11, respectively.

Figure 6-10. SD Create Operation

Figure 6-11. SD Disable
Applying rule 9, 10, 11 and 12 to these Sequence Diagrams, we have the following transactions in OASIS notation:

\[ TR_{\text{CreateOperation}} = \text{create\_operation}(\text{string}, \text{string}). \]

\[ \text{type} = \text{“Repair”} \]

\[ TR_{\text{Disable}} = \text{modify\_state}(\text{string}) \]

An excerpt of the resulting Class Diagram is shown in Figure 6-12.

![Class Diagram](image)

**Figure 6-12. Partial Class Diagram for Create Operation and Disable**

### 6.3.2.2. Extend Relationship

In the Use Case diagram notation, we have the *Extend relationship*. This is a binary relationship between Use Cases, which allows a *base Use Case* to be extended by an *extension Use Case*. This relationship is graphically shown in Figure 6-13.

![Extend Relationship](image)

**Figure 6-13. Extend Relationship**

These Uses Cases are represented as Sequence Diagrams, where the *Rent with Driver* Sequence Diagram performs an *implicit extension* of the *Rent* Sequence Diagram when the *extend condition* “Customer.Type = VIP” holds (the *extend condition* is specified in natural language).

This implicit extension of the *extended* Sequence Diagram (*Rent*) should be made *explicit* when represented as a formula in OASIS.

- **Rule 13.** For *EXTEND* relationships between Sequence Diagrams, the transaction generated from the base Sequence Diagram will perform an explicit invocation (with the *extend condition* as control condition) of the transaction generated for the extended Sequence Diagram.

- **Traceability:** strong.

The structure of the resultant OASIS formulas will be:
\[ TR_{RentWithDriver}() = service_x() \cdot service_y() \]
\[ TR_{Rent}() = service_m() \cdot service_n() . \]
\[ [\text{Customer.Type} = \text{"VIP"}] \ TR_{RentWithDriver}() \]

### 6.4. Association Relationship Rules

In order to derive association relationships from interactions among objects, we have to analyze the nature of these interactions. Roughly speaking, if two objects are associated, that means that they know each other and can share structure and/or behavior. To establish this kind of relationship, we specially have to take into account those interactions that create objects and those that put together objects. Those interactions are the ones labeled with the stereotype <<service/new>> and <<connect>>.

- **Rule 14.** For every message between two classes labeled with the stereotype <<service/new>> where both classes are distinct from the system class, an association relationship between these classes will be generated.

- **Traceability:** strong

A Sequence Diagram for creating an Invoice and Invoice Lines is shown in Figure 6-14. Note that there are two messages with the stereotype <<service/new>>. However, in the first one, the message sender is the class system. In the second one, Invoice is the class in charge of creating Invoice Lines, which means that Invoices will need one or many Invoice Lines.

![Figure 6-14. SD for creating an Invoice and Invoice Lines](image)

Applying rule 14 to this Sequence Diagram, the resulting association relationship in the Class Diagram is shown in Figure 6-15.
Another example to be considered for Association Relationships is <<connect>>. This kind of message is used when an already existing object needs to be related to another object (maybe to share information or behavior). In the Object-oriented model, this is only possible if there is a structural relationship between them: association or aggregation.

- **Rule 15.** For every message between two classes labeled with the stereotype <<connect>> an association relationship between these classes will be generated.

- **Traceability:** strong

A Sequence Diagram for creating an Insurance Policy is shown in Figure 6-16. The creation of a new Insurance Policy necessarily implies that one and only one Insurance Company (which already exists) should be related to the new Insurance Policy and also to the corresponding insured Car.

![Sequence Diagram for creating an Insurance Policy](image)

**Figure 6-16. SD for creating an Insurance Policy**

Applying rule 15 to this Sequence Diagram, the resulting association relationships in the Class Diagram are shown in Figure 6-17.
Another important situation to be considered when dealing with association relationships is that of multiple relationships between classes, that is when there is more than one relationship between two classes. In this case, we should classify the interactions in order to detect whether a given interaction corresponds to a relationship and, if so, to which relationship. This can be done by means of the role names in Sequence Diagrams.

- **Rule 16.** For every message with the stereotype `<service/new>` or `<connect>` where classes using role names appears, an association relationship between these classes will be generated using these role names on the ends of the relationship.

- **Traceability:** weak

To illustrate this situation, in Figure 6-18 we show an example of a (hypothetical) scenario for renting a car. In this scenario, when a *Contract* is created, a *Customer* and a *Car* are assigned to the *Contract*. Also, a *Guarantor* is needed (another customer who cosigns for the customer who rents the car).

Applying **rule 16** to this Sequence Diagram, the resulting association relationships in the Class Diagram are shown in Figure 6-19.
6.5. **Cardinality Rules**

Cardinality refers to constraints on the number of participating objects on association or aggregation relationships. This information can be obtained from the properties of stereotypes assigned to the messages that establish these relationships.

Cardinality information has *weak* traceability to the Class Diagram because it can be modified during the Conceptual Modeling phase. This modification is justified because the source information (properties of the message stereotypes and number of occurrence of messages – conditions and iterations) does not need to be completely specified at this abstraction level. A complete specification for all scenarios would be very tedious.

It is very important to note that, when deriving cardinalities, all the scenarios that contain the participating classes must be analyzed. This is in contrast to the association relationships which can be obtained from analyzing only one scenario.

Because cardinality information is related to the association and aggregation relationships, we have to analyze the context where these relationships are established. These are the messages with <<service/new>> and <<connect>>.

### 6.5.1. Message <<service/new>>

The occurrence of a message with this stereotype implies the existence of a relationship (association or aggregation) between the involved classes. The cardinalities related to this relationship (maximum and minimum) will depend on the issues that we analyze below.

- **Iteration.** The first issue to consider is whether the message is enclosed in an iteration block or it is an iteration message. In this case, the maximum cardinality on the side of the receiver class is “**n**”. If the message is not in an iteration block and it is not an iteration message, the maximum cardinality on the side of the receiver class is “**1**”.

![Partial Class Diagram from Association Rule with role names](image)
- **Creation.** The second issue is related to the creation of objects in the receiver class. If all the sender classes in the set of all Sequence Diagrams where objects from the receiver class are created are the same, this means that the only creators of objects from the receiver class are objects of the sender class. In this case, the minimum cardinality on the side of the sender class is “1”. Otherwise, it is “0”.

- **Object creation followed by creation or connection on receiver.** The third issue is related to the message <<service/new>>. If an object from class A receives this message, then it has to create an object from class B by sending a <<service/new>> message or it has to be related to an object from class B by sending a <<connect>> message. If this situation occurs, it means that an object from class A needs an object from class B in order to exist. As a result, the minimum cardinality on the side of the receiver class is “1”. Otherwise, it is “0”.

  - **Rule 17.** If there is a message with the stereotype <<service/new>> between two classes A and B, both distinct from class system, and it is enclosed in an iteration block or it is a iteration message, the maximum cardinality in the receiver class will be “n” and otherwise it will be “1”.

  - **Traceability:** weak

  - **Rule 18.** If there is a message with the stereotype <<service/new>> between two classes A and B, both distinct from class system, and in the set of ALL Sequence Diagrams where objects from the class B (receiver) are created all the sender classes A refer to the same class, the minimum cardinality in the side of the class A (sender) will be “1” and otherwise it will be “0”.

  - **Traceability:** weak

  - **Rule 19.** If there is a message with the stereotype <<service/new>> in a class A and in the set of ALL Sequence Diagram where A is created, always the object from the class A has to create or be related to an object from class B (sending a <<service/new>> or <<connect>> message) the minimum cardinality for the class A with respect to class B will be “1” and otherwise it will be “0”. In addition, if this message is inside of an iteration block or it is an iteration message, the minimum cardinality will be the lower value of the iteration. If the message is in a conditional block or it is a conditional message the minimum cardinality will be zero. And finally, if more than one of these situations holds, the minimum cardinality will be the less restrictive.

  - **Traceability:** weak

A schema of possible combinations for stereotype <<service/new>> is shown in Table 6.
Table 6. Possible combinations for stereotype <<service/new>>

<table>
<thead>
<tr>
<th>Sender</th>
<th>Receiver</th>
<th>Message</th>
<th>Iteration</th>
<th>Condition*</th>
<th>Min</th>
<th>Max</th>
<th>Min</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>Service/</td>
<td>No</td>
<td>C1 ^C2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>New</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>B</td>
<td>Service/</td>
<td>No</td>
<td>C1^Not(C2)</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>New</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>B</td>
<td>Service/</td>
<td>No</td>
<td>Not(C1) ^C2</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>New</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>B</td>
<td>Service/</td>
<td>No</td>
<td>Not(C1) ^Not(C2)</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>New</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>B</td>
<td>Service/</td>
<td>Yes</td>
<td>C1^C2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>n</td>
</tr>
<tr>
<td></td>
<td></td>
<td>New</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>B</td>
<td>Service/</td>
<td>Yes</td>
<td>C1^Not(C2)</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>n</td>
</tr>
<tr>
<td></td>
<td></td>
<td>New</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>B</td>
<td>Service/</td>
<td>Yes</td>
<td>Not(C1) ^C2</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>n</td>
</tr>
<tr>
<td></td>
<td></td>
<td>New</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>B</td>
<td>Service/</td>
<td>Yes</td>
<td>Not(C1) ^Not(C2)</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>N</td>
</tr>
<tr>
<td></td>
<td></td>
<td>New</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

C1: There is no other Sequence Diagram where objects from class B are created from a class distinct from class A.

C2: There is no other Sequence Diagram where objects from class A are created and objects from class B are not created from A or related to class A.

The last issue related to the message with the stereotype <<service/new>> is the one concerned with conditional messages.

- **Condition.** The last issue related to the message <<service/new>> occurs if the corresponding message is preceded by a condition. In this case, the minimum cardinality of the receiver class will be affected.

- **Rule 20.** If there is a message with the stereotype <<service/new>> between two classes distinct from class system, and the message is a conditional message or it is in a conditional block or it is in a iteration that can occur 0 times, the minimum cardinality in the receiver class will be “0”.

- **Traceability**: weak

Finally, the modified combinations for cardinalities derived from the <<service/new>> message with condition are shown in Table 7.
Table 7. Possible combinations for stereotype <<service/new>> with condition

<table>
<thead>
<tr>
<th>Sender</th>
<th>Receiver</th>
<th>Message</th>
<th>Iteration</th>
<th>Condition*</th>
<th>A</th>
<th>B</th>
<th>Min</th>
<th>Max</th>
<th>Min</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>[cond] Service/ New</td>
<td>No</td>
<td>C1 ^C2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>B</td>
<td>[cond] Service/ New</td>
<td>No</td>
<td>C1^Not(C2)</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>B</td>
<td>[cond] Service/ New</td>
<td>No</td>
<td>Not(C1) ^C2</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>B</td>
<td>[cond] Service/ New</td>
<td>No</td>
<td>Not(C1) ^No(C2)</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>B</td>
<td>[cond] Service/ New</td>
<td>Yes</td>
<td>C1^C2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>n</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>B</td>
<td>[cond] Service/ New</td>
<td>Yes</td>
<td>C1 ^Not(C2)</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>n</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>B</td>
<td>[cond] Service/ New</td>
<td>Yes</td>
<td>Not(C1) ^C2</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>n</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>B</td>
<td>[cond] Service/ New</td>
<td>Yes</td>
<td>Not(C1) ^No(C2)</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>n</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

C1: There is no other Sequence Diagram where objects from class B are created from a class distinct from class A.

C2: There is no other Sequence Diagram where objects from class A are created and objects from class B are not created from A or related to class A.

6.5.2. Message <<connect>>

The occurrence of a message with this stereotype implies the existence of a relationship (association or aggregation) between the involved classes. The cardinalities related to this relationship (maximum and minimum) will depend on many issues that are analogous to the ones introduced for the stereotype <<service/new>> above.

- **Rule 21.** If there is a message with the stereotype <<connect>> between two classes A and B, both distinct from the class system, and if it is enclosed in a iteration block or it is a iteration message, the maximum cardinality in the class B (receiver) will be “n” and otherwise it will be “1”.

- **Traceability:** weak
Rule 22. If there is a message with the stereotype <<connect>> between two classes A and B, both distinct from the class system, and if objects from the receiver class B cannot be created from another class different of A, the minimum cardinality in the sender class A will be “1”, otherwise it will be “0”.

Traceability: weak

Rule 23. If there is a message with the stereotype <<connect>> between two classes A and B, both distinct from the class system, and if in ALL the Sequence Diagrams when an object from the class A is created, an object from the class B is created or related, the minimum cardinality in the class B will be “1”, otherwise it will be “0”.

Traceability: weak

A schema of possible combinations for stereotype <<connect>> is shown in Table 8.

<table>
<thead>
<tr>
<th>Sender</th>
<th>Receiver</th>
<th>Message</th>
<th>Iteration</th>
<th>Condition*</th>
<th>Min</th>
<th>Max</th>
<th>Min</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>Connect 1,1</td>
<td>No</td>
<td>C2 ^C3</td>
<td>1</td>
<td>?</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>A</td>
<td>B</td>
<td>Connect 1,1</td>
<td>No</td>
<td>C2^Not(C3)</td>
<td>0</td>
<td>?</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>A</td>
<td>B</td>
<td>Connect 1,1</td>
<td>No</td>
<td>Not(C2) ^C3</td>
<td>1</td>
<td>?</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>A</td>
<td>B</td>
<td>Connect 1,1</td>
<td>No</td>
<td>Not(C2) ^Not(C3)</td>
<td>0</td>
<td>?</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>A</td>
<td>B</td>
<td>Connect 1,1</td>
<td>Yes</td>
<td>C2^C3</td>
<td>1</td>
<td>?</td>
<td>1</td>
<td>n</td>
</tr>
<tr>
<td>A</td>
<td>B</td>
<td>Connect 1,1</td>
<td>Yes</td>
<td>C2^Not(C3)</td>
<td>0</td>
<td>?</td>
<td>1</td>
<td>n</td>
</tr>
<tr>
<td>A</td>
<td>B</td>
<td>Connect 1,1</td>
<td>Yes</td>
<td>Not(C2) ^C3</td>
<td>1</td>
<td>?</td>
<td>0</td>
<td>n</td>
</tr>
<tr>
<td>A</td>
<td>B</td>
<td>Connect 1,1</td>
<td>Yes</td>
<td>Not(C2) ^No(C3)</td>
<td>0</td>
<td>?</td>
<td>0</td>
<td>n</td>
</tr>
</tbody>
</table>

C2: There is no other Sequence Diagram where objects from class A are created and objects from class B are not created from A or related to class A.

C3: There is no other Sequence Diagram where objects from class B are created from other classes (different from A).
Similar to the case with the << service/new>>, there is the situation with the message with the stereotype <<connect>> that is preceded by a condition. In those cases, only the minimum cardinality will be affected.

- **Rule 24.** If there is a message with the stereotype <<connect>> between two classes A and B, both distinct from the class system, and the message is a conditional message or it is in a conditional block or it is in an iteration that can occur 0 times, the minimum cardinality in the class B (receiver) will be “0”.

- **Traceability:** weak

The resultant table, considering the <<connect>> message with a conditional is shown in Table 9.

Table 9. Possible combinations for stereotype <<connect>> with condition

<table>
<thead>
<tr>
<th>Sender</th>
<th>Receiver</th>
<th>Message</th>
<th>Iteration</th>
<th>Condition*</th>
<th>A Min</th>
<th>A Max</th>
<th>B Min</th>
<th>B Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>Connect 1,1</td>
<td>No</td>
<td>C2 ^C3</td>
<td>1</td>
<td>?</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>A</td>
<td>B</td>
<td>Connect 1,1</td>
<td>No</td>
<td>C2^Not(C3)</td>
<td>0</td>
<td>?</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>A</td>
<td>B</td>
<td>Connect 1,1</td>
<td>No</td>
<td>Not(C2) ^C3</td>
<td>1</td>
<td>?</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>A</td>
<td>B</td>
<td>Connect 1,1</td>
<td>No</td>
<td>Not(C2) ^Not(C3)</td>
<td>0</td>
<td>?</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>A</td>
<td>B</td>
<td>Connect 1,1</td>
<td>Yes</td>
<td>C2^C3</td>
<td>1</td>
<td>?</td>
<td>0</td>
<td>n</td>
</tr>
<tr>
<td>A</td>
<td>B</td>
<td>Connect 1,1</td>
<td>Yes</td>
<td>C2^Not(C3)</td>
<td>0</td>
<td>?</td>
<td>0</td>
<td>n</td>
</tr>
<tr>
<td>A</td>
<td>B</td>
<td>Connect 1,1</td>
<td>Yes</td>
<td>Not(C2) ^C3</td>
<td>1</td>
<td>?</td>
<td>0</td>
<td>n</td>
</tr>
<tr>
<td>A</td>
<td>B</td>
<td>Connect 1,1</td>
<td>Yes</td>
<td>Not(C2) ^Not(C3)</td>
<td>0</td>
<td>?</td>
<td>0</td>
<td>n</td>
</tr>
</tbody>
</table>

**C2:** There is no other Sequence Diagram where objects from class A are created and objects from class B are not created from A or related to class A.

**C3:** There is no other Sequence Diagram where objects from class B are created from other classes (different from A).

The last issued related to the <<connect>> message is the analysis of the “Exclusive” property.

The exclusive property refers to the situation where an object of the receiver class that is already participating in a <<connect>> message, cannot participate in a <<connect>> message with the same sender class unless it has been disconnected. It can be disconnected by sending the <<connect>> message with the Activity property Delete.
When an object participates in a relationship it participates exclusively. For example, in the scenario of renting a car, when a Car is assigned to a Contract, it is done in an exclusive way to prevent the same Car for being rented again. Before creating the Contract the desired Car is free. However, when the Contract is registered, the Car is related to it and cannot be rented by another Contract.

The analysis of this property will affect the generation of cardinalities, in particular the values related to the sender class that will be always “0..1” (“0” if the object in the receiver class is free and “1” if the object in the receiver class is already in use).

- **Rule 25.** If there is a message with the stereotype <<connect/exclusive>> between two classes A and B, both distinct from the class system, the cardinality in the class A (sender) will be “min = 0” and “max = 1”.

- **Traceability:** weak

In order to conclude this subsection related to cardinalities, we have to mention that in different scenarios we will have many rules to apply and many “intermediate” values. In some cases there may also be “incomplete” values. In order to decide which values should be proposed for cardinalities in the Class Diagram, the following rules are stated.

- **Rule 26.** For those cases that the same association or aggregation relationship has different proposed values for cardinalities (more than one rule can be applied), those values less restrictive will be used.

- **Traceability:** weak

- **Rule 27.** In those cases where the values for cardinalities cannot be computed, the values “Min = 0” and “Max = n” will be used.

- **Traceability:** weak

### 6.6. Aggregation Relationship Rules

Some kinds of interactions among objects in the Sequence Diagram can give us important information related to aggregation relationships. Aggregation relationships are a special kind of association where the objects involved have a stronger dependency between them (at least in one direction). In practice, the aggregation relationship is a subjective relationship and depends on the software engineer’s view of reality. For example, one software engineer can see an Invoice as an aggregation of Invoice Lines, and another software engineer can consider it as a simple association (because of visibility, navigability or other considerations).

The aggregation relationship can be seen as a special kind of association and could be obtained by making a more exhaustive interpretation of the messages that create association relationships. These are messages labeled with the stereotypes <<service/new>> and <<connect>> between different classes of the system class. Of the two types of interactions, the <<service/new>> message represents a stronger kind
of interaction, that is, the object from the sender class needs to create an object from the receiver class (normally to have access to its structure or behavior). When this situation occurs, we propose generating an aggregation relationship.

- **Rule 28.** *For every message with the stereotype <<service/new>> between two classes A and B, which are distinct from the class system, then an aggregation relationship between these classes will be generated.*

- **Traceability:** weak

Note that the traceability for this rule is weak. This means that the resulting aggregation relationship can be modified later but only to be considered as an association relationship (see rules in section 6.4 por encima de) that has strong traceability (the relationship cannot be deleted).

A Sequence Diagram for creating an Invoice and Invoice Lines is shown in Figure 6-20. When an Invoice is created, it is possible to create as many Invoice Lines as desired. This is done by the iterative message create_invoice_line() with the stereotype <<service/new>>. All the Invoice Lines are created from the Invoice and related to it.

![Sequence Diagram](image)

**Figure 6-20. SD for Creating Invoice and Invoice Lines**

Applying rule 28 to this Sequence Diagram, the resulting aggregation relationship in the Class Diagram is shown in Figure 6-21.

![Class Diagram](image)

**Figure 6-21. Partial Class Diagram from Aggregation Rule**
6.7. Agent Relationship Rules

An agent relationship, in an OO-Method Conceptual Schema, is a relationship between a class and a service of a class in the Class Diagram. This information can also be obtained from interactions among classes captured on the Sequence Diagrams. Every external interaction for the system is identified in the Function Refinement Tree (leaf nodes) that is represented by a Use Case. Each Use Case is represented by a set of Sequence Diagrams that specifies the necessary object interactions for the basic course and other Sequence Diagrams for the alternative sections.

The initiator actor is always identified and represented as an actor class to the left of the system class in the Sequence Diagram. The relationship between this actor class and the corresponding Sequence Diagram (that is the realization of the Use Case) establishes the corresponding agent relationship. Class actors to the left of the system class of Sequence Diagrams that represent alternative sections are not agent relationships because they do not represent an external interaction.

- **Rule 29.** For every actor class (representing an initiator actor) of a Sequence Diagram for the basic course of a Use Case, an agent relationship will be generated between the actor class and the service (transaction) that is derived from the quoted Sequence Diagram.

- **Traceability: strong**

6.8. Inheritance Relationship Rules

The inheritance relationships that can be obtained from the Requirements Model are the relationships between actors that are specified in the Use Case diagram. In this Diagram, the only relationship allowed between actors is that of inheritance.

For all the actors that fulfill all of the conditions below, an inheritance relationship is generated in the Class Diagram:

- have inheritance relationship in the Use Case diagram,
- are initiators for Use Cases,
- are considered as actor classes in the corresponding Sequence Diagram

- **Rule 30.** All the inheritance relationships between actors in the Use Case diagram are represented as an inheritance relationship between the corresponding actor classes in the Class Diagram.

- **Traceability: strong**

The relationship between two actors from the Use Case diagram is shown in Figure 6-22. Both actors are initiators of Use Cases and are used to the left of the system class in different Sequence Diagrams.
Applying rule 30 to this excerpt of Use Case diagram, the resulting inheritance relationship in the Class Diagram is shown in Figure 6-23.

6.9. **Precondition Rules**

*Preconditions* are conditions that must be satisfied in order to execute the corresponding service. Every external interaction is identified in the Function Refinement Tree (leaf nodes) that is represented by a Use Case. In the Use Case description, it is possible to write down the associated precondition to the Use Case. As each Use Case is represented by a set of Sequence Diagrams (at least one), the precondition in the Use Case description is the precondition to the Sequence Diagram that realizes the Use Case.

- **Rule 31.** The precondition from the Use Case description is the precondition of the corresponding Sequence Diagram and, transitively, of the corresponding transaction derived from it.

- Traceability: strong

6.10. **Integrity Constraint Rules**

*Integrity Constraints* are conditions specified in a class that must be satisfied by objects of the class in any state. These *integrity constraints* can also be obtained from
the Sequence Diagram (class properties) to fulfill the Object Model. These properties can optionally be defined when the class is identified during its use in a scenario specification.

- **Rule 32.** Any integrity constraint specified in the class during the Sequence Diagram specification will be represented as an integrity constraint for the class in the Class Diagram.

- **Traceability:** weak

### 6.11. Trigger Rules

Triggers are services that are automatically executed when a condition is satisfied. They must be defined in the class specification, specifically, in the Interaction Diagram.

These triggers can also be obtained from the Sequence Diagram or, more specifically, from the class properties specification.

Triggers are not explicitly specified in a Sequence Diagram because they depend on a condition satisfaction in order to execute the indicated service and not in a sequence of messages. However, they can be defined in the class properties specification. These properties are optionally defined when the class is identified during its use in a scenario specification.

- **Rule 33.** All the triggers specified in a class during the Sequence Diagram specification will be represented as a trigger specification in the Interaction Diagram.

- **Traceability:** weak

### 6.12. Static and Dynamic Association-Aggregation Relationship

Structural relationships between objects (association or aggregation) can be static or dynamic depending on the time this relationship is established, that is, when the objects are created or at some moment during their life. This information can be obtained by analyzing the set of scenarios where these relationships are established (basically messages with the stereotype <<service/new>> or <<connect>>).

#### 6.12.1. Static Relationship

A relationship between two classes A and B is static when an object from class B is created and the corresponding links with other objects are also created and do not
changed during the object’s life. There can be zero or many links depending on the cardinalities.

- **Rule 34.** Given two classes, A and B, both of which are distinct from the class system, and if there is a message with the stereotype <<service/new>> from A to B and there are no messages with the stereotype <<connect>> from A to B or B to A (in another Sequence Diagram), then the relationship between A and B will be static from the point of view of B.

- **Traceability:** weak

### 6.12.2. Dynamic Relationship

A relationship between two classes A and B is **dynamic** if the links between these objects can be changed after their creation. That is, objects are created in some scenarios and links between these objects are created or deleted in other scenarios.

- **Rule 35.** Given two classes, A and B, both of which are distinct from the class system, and if there is a message with the stereotype <<service/new>> from A to B and there are some messages with the stereotype <<connect>> from A to B or B to A (in another Sequence Diagram), then the relationship between A and B will be dynamic from the point of view of B.

- **Traceability:** weak

It is important to remark that the static and dynamic property of the association and aggregation relationship depends on the interaction between the objects involved in all the specified scenarios. Because a scenario is a partial description of behavior in an object’s life, the default value of the static/dynamic property is dynamic. This helps to deal with misspecification.

- **Rule 36.** In case of misspecifications or contradictions for the static/dynamic property of the association-aggregation relationship it will be dynamic.

- **Traceability:** weak

### 6.13. State Transition Diagram Rule

The State Transition Diagram is used to describe the correct behavior by establishing valid object life cycles for every class. By valid life, we mean an appropriate sequence of service occurrences that characterizes the correct behavior of the objects that belong to a specific class.

In the Requirements Model specification and in the Requirements Analysis Process, we are concerned with scenarios and object classes that are necessary to realize these scenarios, respectively. The complete behavior for each class should be specified in the conceptual modeling phase. Nevertheless, the information that can be directly obtained from requirements is that of basic State Transition Diagrams characterized by
the grouping of creation, modification, destruction, and association (connection) services. This information is explicitly shown in the Traceability Matrix, in the CRUD+A (Create, Read, Update, Destroy, and Association) Table, following the appropriate row for each class as shown in Figure 5-15.

- **Rule 37.** A basic State Transition Diagram (STD) will be generated for each class. This basic STD is composed of three states: “pre-creation”, “created” and “destroyed”. Each service identified in the class will be represented as a transition between states.

All the creation services (local events of type “new” or local transactions where its first service is a local event of type “new”) will be transitions from the ”pre-creation” to the “created” state.

All the destruction services (local events of type “destroy” or local transactions where its last service is a local event of type “destroy”) will be transitions from the “created” to the “destroyed” state.

The rest of services will be transitions departing from and going to the “created” state.

- **Traceability:** weak

## 6.14. Presentation Model Rules

The Presentation Model allows the specification of user interface properties at the problem space level using conceptual user interface patterns. This set of conceptual patterns provides a language pattern to specify the application user interface in an implementation-independent manner.

At this moment, we are only dealing with the Hierarchical Action Tree (HAT). This pattern organizes and structures the functionality of the system for each user type, that is, which services will be available for each user type for interaction.

The other patterns of the Presentation Model complete the user interface specification by capturing the properties of human-application interaction (interaction units) showing how the information will be presented, how the information will be introduced into the system, and which navigations among interaction units will be allowed.

The Function Refinement Tree (FRT) provides a refinement mechanism that starts from the Mission Statement and then organizes and structures the desired functionality of the system until it obtains the external interactions (services offered by the system). These services are independent of the actors, whose interactions are identified and specified in the Use Case model (UCm).

A Use Case view represents a subset of Use Cases (UCv ⊆ UCm) with their respective actors. We define an actor Use Case view by restricting the Use Case model
to those Use Cases where a specific actor is the initiator (ai) of the Use Cases (UCm ↓ ai = UCvai).

If we project this UCvai to the FRT restricting it to the external interactions of the actor ai, there is an external interaction hierarchy grouped by actor ai (FRT ↓ UCvai = FRTai). This takes into account only those branches of the tree whose leaves represent some external interaction of ai.

This FRTai is taken as the first approximation to the Hierarchical Action Tree (HAT) of the Presentation Model. This HAT can later be refined and enriched using the language pattern provided.

- **Rule 38** The Use Case model is restricted to each initiator actor (UCvai), and this view is applied to the FRT obtaining a Hierarchical Action Tree (HAT) for each initiator actor (ai).

- **Traceability**: weak

An excerpt of the FRT for the Car Rental system is shown in Figure 6-24, and an excerpt of the Use Case view for the actor initiator Administrator is shown in Figure 6-25.

![Diagram](attachment:image.png)

*Figure 6-24. Excerpt of a FRT*
Applying rule 38, we project the excerpt of the Administrator Use Case view to the excerpt of the FRT. The result is the FRT for the initiator actor Administrator shown in Figure 6-26, which graphically represents the initial Hierarchical Action Tree (HAT) for the Administrator.

Figure 6-25. Excerpt of the Administrator Use Case view

The set of rules presented in this chapter shows a working catalog that summarizes all the traceability structures that we create when moving from the Requirements Model to a Conceptual Schema. This catalog is under continuous study and revision in accordance with the experience we acquire when applying it to different development projects.

Although the resulting conceptual schema is only a skeleton of the final conceptual schema that will be used as input for the code generation process, this is a very important starting point for conceptual modeling.

The software engineer should maintain all the elements in the conceptual schema with strong traceability, and add, delete, or modify those with weak traceability. In addition, those elements with no relationship to the Requirements Model (i.e. Functional Model) must be fulfilled.

The elements in the conceptual schema with strong traceability are:
• Classes
• Services
• Association and Aggregation relationships
• Inheritance between actor classes
• Preconditions
• Agent relationships
7. The Requirements Engineering Tool

The current implementation of our prototypical requirements engineering environment is RETO (Requirements Engineering Tool). This tool captures user requirements following the Requirements Model presented in this thesis, and implements the Requirements Analysis Process (RAP) to generate elements of an OO-Method conceptual schema. Figure 7-1 shows a rough architecture of the RETO prototype environment.

![Figure 7-1. A view of RETO architecture](image)

The XML RETO is the data repository of RETO and XML ONME is the data repository of OlivaNova Model Execution Software. The transformation process between these documents is performed using XSL.

OlivaNova Model Execution Software® (ONME)\(^8\) supports the conceptual modeling and implementation phase of the software development process. This tool family provides true, fully executable application source code starting from the conceptual schema specification [131].

\(^8\) OlivaNova Model Execution Software® (ONME) is a trademark of CARE Technologies S.A.
In order to allow communications between RETO and the OlivaNova Model Execution Software tools, the following tasks were necessary:

- Defining a DTD (with a structure based on the RETO Metammodel and similar to the DTD of ONME tools) allows us to send and receive elements to and from the conceptual schema.

- Implementing a loader that allows the RETO tool to load the XML RETO documents.

- Implementing a writer that allows the RETO tool to produce an XML RETO document based on the Requirements information captured on RETO.

- Defining and implementing translation process between XML documents (XML RETO ↔ XML ONME) based on XSL.

A more detailed view of the communication structure is shown in Figure 7-2.

![Diagram showing communication structure between RETO and ONME Software](image)

**Figure 7-2. RETO and ONME Software communication schema**

At this time, the transformation process between RETO and ONME XML documents is only one-way, that is, to transform a requirements specification to a conceptual schema (the solid arrow line between these documents in Figure 7-2).

**RETO DTD Definition**

The RETO DTD definition is based on the design of the metamodel of the Requirements Model. In this way, the relationship between the metamodel and RETO DTD is defined as follows:

- Class → Element
- Attribute → Attribute
- Aggregation Relationship → Element
- Inheritance Relationship → Element

All classes of the metamodel are represented as elements.

An attribute of this element will be named as _Id using the ID type. This attribute will be used to identify the element in the XML document. If the class is
inherited and it does not have an identification function, then the \_Id attribute will not be declared.

The constant and variable attributes will be represented as attributes of the element that represents the class if they do not allow XML valid characters (\<, \>, &, ', "\). If the attributes allow invalid characters, they will be represented as elements of type CDATA.

Following, an example of the Actor class with three attributes: RCode, RName and RComments is shown:

```xml
<!ATTLIST Actor
   _id ID #REQUIRED
   RCode CDATA #REQUIRED
   RName CDATA #REQUIRED
   RComments CDATA #IMPLIED
>
```

The relationships defined between two classes will be represented by roles that are defined as elements in XML.

- The inheritance relationship will be defined as an element with the name of the child class in the element created for the parent class.
- All aggregation relationships of a class are included as elements in the element of the DTD that is created from this class.

It is possible to include the aggregation relationship in two ways:

- The element that represents the aggregation relationships contains the related class as an element.
- The element that represents the aggregation relationships contains a reference to the element that was created from the related class.

**Naming conventions**

The name of an XML element, which represents a class of the metamodel, is the name of the corresponding class.

The names of the attributes of an XML element, which represent attributes of a class of the metamodel, are the names of the attributes of the class.

To represent an aggregation relationship, an element labeled with the name `Role.RoleName` and a subelement labeled with the related class name `ClassName` are created.

When the aggregation is by reference, an element labeled with the name `Ref.Role.RoleName` and a subelement labeled with `Ref.ClassName` are created. The subelement will be an empty element with a attribute `_Ref` of type IDREFS.
In order to represent cardinalities greater than one, the prefix `ListOf` is used 
(`ListOf.Role.RoleName` or `ListOf.Ref.Role.RoleName`). For example:

```
Listof.Role.Assignment?)>
```

For inheritance relationships, the prefix `Inher` is used in the name of the element 
that represents the child class.

As an example, the specification of the class Service is shown.

```
<!ELEMENT ListOf.Role.Argument (Argument+)>
<!ELEMENT Argument (Ref.Role.Attribute?)>
<!ATTLIST Argument
   _id ID #REQUIRED
   RCode CDATA #REQUIRED
   RType CDATA "D"
   RDataType CDATA #IMPLIED
   RI_O CDATA "I"
   RName CDATA #REQUIRED
   RCollectionType CDATA "0"
   RNull (True | False) "False"
   RHelpMensaje CDATA #IMPLIED
   RAlias CDATA #IMPLIED
   RIsMultivaluate (True | False) "False"
   RComments CDATA #IMPLIED
>}
```

Once the RETO XML document is built and validated, it is possible to generate 
a ON ME XML document using XSLT and applying the rules of the Traceability Rules 
Catalog of Chapter 6 por encima de. The obtained document is validated using the ON 
ME DTD in order to check whether it is correct.

The XML Metadata Interchange (XMI) is the standard from the Object 
Management Group (OMG) for saving the meta-data that make up any particular UML 
model (or metamodel). In principle, this will allow us to take a model created in RETO 
or ON ME and import it into another tool. This clearly has advantages that allows UML 
to meet its goal of being a standard for communication between designers.

The reality is not quite this good. XMI is a recent standard and there are few 
tools to implement it. Furthermore, the XMI itself doesn't support all that is needed, for
example, it says nothing about the graphical representation of the models and the different views that are necessary in CASE tools, and so the diagram layout is lost. For this reason, every vendor implements it differently.

Therefore, we use XML instead of XMI. The DTDs that have been defined for our XML documents (Requirements and Conceptual) are as simple and similar as is possible to allow efficient and simple symbol interchange between them.

The language that has been used to translate a Requirements XML document to a Conceptual XML document is XSL Transformation (XSLT). XSLT is designed for use as part of XSL (eXtensible Stylesheet Language). In addition to XSLT, XSL includes an XML vocabulary to specify formatting. XSL specifies the styling of an XML document by using XSLT to describe how the document is transformed into another XML document that uses the formatting vocabulary.

An easy example of how the elements of a Requirements XML document are translated to a Conceptual XML document using XSLT is in Figure 7-3.

Note that there are two kinds of specifications for classes, one in the XML document for the Requirements tool and another in the XML document for the ON Software. Although the information that is stored is almost the same, the structure that stores this information is different.

Figure 7-3. Example of Class translation in XSLT

As future work, we are planning the development of the reverse process to obtain a Requirements Specification from a Conceptual Schema. It will then be possible to work in a Spiral Life Cycle development and it will also be possible to obtain a documentation of an existing model.
7.1. **OlivaNova Model Execution Software**

The *OlivaNova Model Execution Software* is a family of products that provides true, fully executable source code starting from conceptual model specifications. The development of the RETO prototype environment is intended to provide the requirements specification level to this product family.

Models can be easily specified and maintained by an analyst with the *OlivaNova Modeling Software*, and transformed into an executable application using the *OlivaNova STAR*. Moreover, *OlivaNova Model Execution* family supplies another tools in order to make the software process development easier:

- **OlivaNova Model Comparer**, which allows to find differences between two conceptual schemas,
- **OlivaNova Schema Evolution tool**, which allows to migrate data from an old conceptual schema to a new one when modifications (evolution) are made,
- **OlivaNova Tester** which tests the communication between client and server programs and the server functionality, and finally
- **OlivaNova Function Points Measurer**, which allows the functional size measurement of projects based on the conceptual schema specification. Results are provided in HTML format.

Detailed information on these tools can be obtained in the CARE Technologies Web Site (http://www.care-t.com).

7.1.1. **OlivaNova Modeling Software**

The OlivaNova Modeling Software is an UML-compliant graphical environment that gives support to the conceptual modeling phase of OO-Method (see Chapter 4 por encima de). In this phase, four complementary models has to be specified in order to capture the structure, behavior, functionality, and presentation of the desired system:

- **Object Model**, to capture static properties of the system. It uses the Class Diagram.
- **Dynamic Model**, to capture dynamic and inter-object communication properties. It uses the State Transition Diagram and the Object Interaction Diagram
- **Functional Model**, to capture functionality related to changes in the object state.
- **Presentation Model**, to capture how objects interact with users by means of interaction units.
Once the conceptual schema is finished and validated, the OlivaNova Modeling Software can generate a XML file that represents the conceptual schema. This XML file can be used by many other OlivaNova Model Execution tools to produce source code, to evaluate its complexity, to view the model evolution and so on.

A general view of OlivaNova Modeling tool is shown in Figure 7-4. To the left there is a classical navigation tree to explore the classes and main properties, and to the right the Class Diagram showing classes and relationships.

Furthermore, OlivaNova Modeling Software will produce all the documentation for users and developers like the user’s manual, the complete OASIS specification or the documentation of the model.

### 7.1.2. OlivaNova STAR

This tool provides the main feature of the OlivaNova Model Execution Software. It receives as input an XML model generated with OlivaNova Modeling Software and produces one or more of these services:

- Transformation services: code generation
- Business logic (Server part, including server code and database scripts)
  - JAVA / EJB
  - Visual Basic / COM+
  - C# (under development)
- User Interface (Client part)
  - Visual Basic
  - ColdFusion
  - JSP
- Function Points Counter service

The tool has a Client/Server architecture. The OlivaNova STAR *Client* is the application used to request any of the services provided by the OlivaNova STAR *Transformation Engine*, the server part (the transformation engine is installed in the main offices of CARE Technologies). The OlivaNova STAR *Client* sends by e-mail a request to the OlivaNova STAR *Transformation engine*. Later, the *Client* receives another e-mail with the result of the request.

One of the best qualities of the source code provided by the OlivaNova STAR is that almost any client application will work the same with any server application. Developers could have a Visual Basic client and JAVA server, or ColdFusion client and C# server... almost any possible combination will work fine, only JSP clients have to run against an specific server component, which is JAVA Server.

### 7.2. *RETO Metamodel*

A metamodel is a precise definition of the modeling elements and their relationships for any semantic model. In particular, we use the ON ME Software to represent and precisely define the Requirements Model metamodel. The metamodel also plays a fundamental role in the CASE tool construction, being the source for the automatic code generation process.

An overview of the RETO metamodel is shown in Figure 7-5.
This figure is a snapshot of the ON ME Software showing the Use Case diagram elements: classes and relationships for the Use Case, Actor, relationships between Use Cases (Include and Extension) and relationships between Use Cases and Actors (Assigned) appear.

Other packages and the dependencies among them also appear. The packages and their content descriptions are the following:

**UCSpec.** The Use Case Specification structure is defined in this package: the basic and alternate trajectories and their corresponding steps.

**FRT.** The Function Refinement Tree is defined in this package: the Mission Statement, Intermediate Nodes, and Leaf Nodes.

**RAP.** The Sequence Diagram is defined in this package: the different kind of messages and relationships with actors and classes.

**Conceptual Model.** The different elements that can be generated for the Conceptual Schema are defined in this package.

**User.** The different users for the metamodel are defined in this package.

**Extras.** References to external elements (e.g. addresses for image files, forms, etc.) are defined in this package.

Once the metamodel is completed and validated, a full formal specification in OASIS is obtained and the corresponding XML can be generated in order to start the code generation process.
7.3. **RETO Architecture**

The RETO architecture is based on the common three-tiered architecture where responsibilities are clearly separated and assigned to software components (see Figure 7-6).

![Three-tier Architecture Diagram](image)

The main layers and responsibilities are:

- **User interface**, which is composed of components that handle the presentation of information between the user and the application.

- **Application logic**, which itself is composed of problem domain and service support components.
  - *Problem domain* components represent the implementation of classes from the conceptual schema.
  - *Service support* components are components that provide supporting services such as interfacing with the database (commonly called mediator or database brokers), writing/reading XML documents, etc.

- **Persistence manager**, which represents the persistent object system, in particular, we use relational database systems (RDBMS).

Figure 7-7 shows a general view of the RETO environment. Note that the ON ME Software appears twice. This is because we use the ON ME Technology to build RETO (to the left of the figure), and later on, we use RETO as a part of the ON ME Software family tools (to the right of the figure).

The communication between ON Modeling Software and RETO is by means of transformations of XML documents using XSLT. We think that in a future, RETO could be integrated with the OlivaNova Modeling Software. Now, OlivaNova Modeling Software organizes the specification information in four models of the Conceptual
Model: Object, Dynamic, Functional, and Presentation Models. Later on, we could add a new model, the Requirements Model.

Figure 7-7. The RETO environment

7.4. RETO User Interfaces

RETO has two user interfaces: the one we call form-based user interface, which was automatically generated by ON ME Software, and the icon-based user interface that is being developed “manually”.

7.4.1. Form-based user interface

The RETO form-based user interface is the presentation layer of the automatically generated application of the Requirements Model metamodel using the ON ME Software. Figure 7-8 shows the Function Refinement Tree for the Car Rental system. Although this application has all the required functionality to build requirement specifications, following the approach presented in this thesis, it is not user-friendly and not suitable for this kind of applications.

For example, to see the information related to the Use Case of the function Purchase (highlighted in Figure 7-8) we have to open to the corresponding window using the Use Case button (this button is below the list of nodes of the FRT).
All the functionality to work with the FRT is supported. Figure 7-9 shows the FRT displaying the functions available:

- **Create Node**, to create a new intermediate or leaf node. The node is created below the selected node.

- **Modify Node**, to modify the properties of an intermediate or leaf node: name, description, priority, or lock it to prevent changes.

- **Change of Parent**, to change the position of the node in the FRT. The new parent cannot be a leaf node.

- **Change to Leaf**, to change an intermediate node in a leaf node (a leaf node can not have children nodes).

- **Change to Node**, to change a leaf node in an intermediate node.

- **Unlocked Node**, to give the possibility to change the properties of a node. When the node is created or modified, it can be locked.

- **Delete Node**, to delete the node.
In order to have a better overview of the functionality of the RETO form-based user interface, the management of the Use Case specification is also explained. For example, to create a step inside of a Use Case description the following information should be provided:

- **type of step**, if the step is of type General, Actor/System communication, or System response, according to the step classification presented in 3.2.3.1 por encima de.
- **a condition**, that must be satisfied for this step (if any).
- **a description** of the step.
- **a name** for the step, it is used to define extension points.

Figure 7-10 shows the step creation window.
Finally, Figure 7-11 shows part of the Use Case description of the *Purchase* Use Case. This window offers also facilities to search Use Cases by Project name, or by Use Case name, and to navigate to related information of a selected Use Case (extensions,
inclusions, external files references (Word, Excel, etc), actors or the corresponding node in the FRT).

7.4.2. Icon-based user interface

This RETO icon-based interface is being developed as an extension of the form-based interface, that is, the icon-based interface reuses all the structure of the form-based interface and adds the corresponding graphical representation for the needed diagrams.

In Figure 7-12, we have the FRT implemented in the icon-based version. In this view, all the typical operations for a tree management are provided: creation of a new node, modification of the selected node properties, change of an intermediate node in a leaf node or vice versa, and deletion of the selected node. All these operations correspond to behavior that is specified in the RETO metamodel and can be directly accessed by the menu, by the icons on top of the FRT, or by clicking with the right button on a node.

Figure 7-12. FRT in RETO icon-based version
In addition to the graphical representation of the FRT, we have implemented the Use Case diagram, as is shown in Figure 7-13. The diagram shows the Car Management functional group.

This diagram is semi-automatically created, that is, when the Car Management functional group is created, the tool knows that it is composed of the Garage Management, Insurance Company Management, Rate Management, Insurance Management, and Operational Management functional subgroups, and they are displayed as packages. In addition, the Use Cases Deliver, Purchase, and Sell are displayed. Finally, the analyst has to complete this diagram adding the corresponding actors and establishing the corresponding relationships among actors and Use Cases.

In order to specify the Use Case description, making a double-click in the Use Case icon on the diagram, the tool will display the window presented in Figure 7-11. This window is part of the functionality that was automatically created using the ON ME Software.

The FRT and Use Case diagram implementations complete the Requirements Model in the RETO icon-based version. With respect to the Requirements Analysis Process, it comprises the Sequence Diagram specification, the implementation of the traceability rules catalog, and the communication with the ON Modeling Software by transforming the XML RETO document into the XML ONME document. These are under development and it is planned to release the first full version of RETO in the first quarter of 2004. Figure 7-14 gives an overview of the complete RETO environment including part of the Purchase Sequence Diagram acceded from the FRT.
Figure 7-14. RETO general overview
Chapter 8

8. Conclusions

In the introduction, we have stated that the goal of this thesis is to provide a Requirements Engineering Environment with a sound, flexible, and precise transition to an OO-Method Conceptual Schema. This environment comprises a Requirements Model that includes mechanisms to identify and specify user requirements, and a Requirements Analysis Process that establishes the traceability bridge between requirements and the OO-Method Conceptual Schema.

We now discuss whether this goal has been met.

The Requirements Model proposed in this thesis provides the means to represent the external interactions of the system. These external interactions are specified in three ways: functions, communication, and behavior. Three techniques were proposed to deal with the organization, reasoning, and specification of these external interactions: Mission Statement, Function Refinement Tree, and Use Case Model. In our point of view, the use of these techniques in an integrated way allows analysts to effectively deal with the separation of concerns and functional abstraction refinement (two major problems found in most of the Requirements Specification methods based on Use Cases).

We have defined a Requirements Analysis Process between the Requirements Specifications and the Conceptual Schemas. This process uses the well-known UML Sequence Diagrams extended with a message classification. This extension is one of the important contributions of this thesis. In our approach, these Sequence Diagrams are detailed specification of Use Case descriptions. One cannot expect to automatically synthesize Sequence Diagrams from Use Cases, since Use Cases are informal and highly abstract. This leaves us with having to construct Sequence Diagrams manually. However, this is limited to a single scenario. We strongly believe that if Use Cases (or scenarios) are appropriately identified and specified (at a rational and manageable abstraction level) this would be extremely useful to the analysts.

Another important component of the Requirements Analysis Process is the Traceability Rules Catalog. After a precise definition of the Requirements Model structure, and with an already defined Conceptual Model structure (the OO-Method
Conceptual Model) the challenge was to define a bridge between them. The result is this set of thirty-eight traceability rules that give us the possibility of obtaining an important part Conceptual Schema, and even more important than this, is the traceability that it establishes between the requirements specification and the conceptual schema. We do not believe that this catalog is finished. The more experience we get using our proposal the more precise this catalog will become.

RETO (our REquirements Engineering Tool), which has been jointly developed by the Valencia Polytechnic University and CARE Technologies S.A., has given us the context to test the ideas presented in this thesis. At this time, the tool is under development at CARE technologies S.A. by a team led by the author of this thesis (two full-time people and two part-time people).

The current version of the tool implements all the facilities in order to work with the Requirements Model (Mission Statement, Function Refinement Tree, and Use Case diagrams). This part of the tool is being tested in three medium-size projects at CARE (a Document Management system, a Title Deed System, and a Project Management System) giving us important feedback, especially in aspects related to user interface and performance of the tool.

The Requirements Analysis Process has been partially implemented.

- Sequence Diagrams have been partially specified (blocks definitions, labeling messages with stereotypes, and hyperlinks with other related Sequence Diagrams are not yet finished).

- Traceability rules related to classes, attributes, and services have been implemented (those that are applied separately to each scenario). With respect to the other rules that relate many scenarios to be applied, we are defining the strategy to process efficiently (with few iterations) the (potentially) great number of scenarios.

- Transformations between XML RETO and XML ONME documents are currently being tested.

We hope to release the first full version of RETO in the first quarter of 2004.

Students at the UPV have also developed several case studies as a part of their Final Course Project this year. These experiences have helped us to understand that good training identifying external interactions is needed. Mixing the level of abstraction during the Function Refinement Tree specification was a recurring problem. Another recurring problem was the level of detail specifying Sequence Diagrams. The students were tend to give design level specifications (including error messages, detailed descriptions of iterations with exceptions and so on). Our teaching efforts were oriented to clearly making them understand that the goal when specifying Sequence Diagrams is to identify participating classes and to describe the principal interactions among them. The students did not have major problems labeling messages with the defined stereotypes in Sequence Diagrams.

Traceability between Requirements and the Conceptual Schema was done by hand (there was no tool support for the approach). This was an important handicap for
two main reasons: a) the analyst had to learn the complete traceability rules catalog, and 
b) an iterative refinement approach was difficult to test (changes in requirements require 
checking many traceability rules). In spite of this, the results were satisfactory.

Following the approach presented in this thesis, we have found it helpful to have 
an organized scenario specification (at the level of Use Case and Sequence Diagrams) 
and to have elements in the Conceptual Schema with an identifiable source. When 
alysts do not agree with the resultant Conceptual Schema, they know which scenarios 
should be checked (or whether they are missing).

Main efforts are now oriented to finishing the implementation of the traceability 
rules catalog, improving the usability of the tool, and to continue doing the empirical 
validation of results. In order to help in this validation of results new “real-life” 
applications will be developed following our approach.

We understand that the presented results have merely indicated feasibility. However, they are encouraging in this respect.

To complete this chapter, the most relevant contributions and original proposal 
of the thesis are presented. Then, the scientific publications generated during the 
development of this thesis are listed. Finally, future and ongoing research work and 
open issues are commented upon.

8.1. **Contributions**

The most relevant contributions of this thesis are the following:

- The integration of three well-known techniques (Mission Statement, 
  Function Refinement Tree, and Use Case Model) into a three-layer 
  Requirements Model.
  - It provides a mechanism to detect and organize the desired system 
    functionality starting from the Mission Statement and refining it until 
    the elementary functions.
  - It provides a strategy to solve the classical Use Case problem of 
    abstraction level definition relating the elementary functions of the 
    desired function and the Use Cases.

- A new Use Case template is proposed to specify Use Case descriptions. This 
  template is based on the classification of steps in accordance with their 
  nature (General, Actor/System Communication, and System Response). This 
  template facilitates the traceability with Sequence Diagrams.

- Guidelines to move from textual Use Case descriptions to Sequence 
  Diagrams.
• A new message classification (with the stereotype definitions) for different kinds of interactions in Sequence Diagrams, in accordance with their nature (signal, service, query, and connect).

• A traceability rules catalog that provides concrete mechanisms to translate elements from a detailed requirements specification to elements of an OO-Method Conceptual Schema. In most of cases, these rules can be generalized to “classic” UML diagrams.

• A Requirements Engineering environment prototype (RETO). This tool gives support to the Requirements Model presented in this thesis, and implements the corresponding communication between this tool and the OlivaNova Model Execution Software (using XML documents).
  
  o The communication (and future integration) between RETO and the OlivaNova Model Execution Software allows analysts to accelerate the development time. The identification and specification of scenarios before building the Conceptual Schema gives the analysts a better understanding and provides documentation of the problem domain.

• The development of case studies following the proposed approach. As a result, many rules from the traceability rules catalog presented were modified, added, and deleted.

• The resultant requirements documentation (in a scenario basis) was found to be useful for verifying the detailed specification of complex transactions when many classes must interact with each other.

8.2. Related Publications

The work related to this thesis was published in 6 International Journals (4 of them in the JCR), 1 book chapter, 21 conferences and workshops (total of 28).

• International Journals


- **Book chapters**


- **Conferences and Workshops**


8.3. **Future and Current Research**

This thesis is not the end of the research efforts in this area. Many research activities are currently underway, and further research is ongoing in different and complementary directions.

Current research and development activities are being carried out jointly with the CARE Technologies Research Group. The main issues currently underway are:

- The development of a Hyper-Dictionary to capture and manage user-frequent-terms in the context of a requirements specification. The aim of this hyper-dictionary is to set a common terminology with an agreed understanding of the different terms used throughout the software development process.

- The refinement and testing of the Traceability Rules Catalog. The different rules proposed in the catalog are being used and refined as case studies are applied. New patterns are being identified and proposed to enrich the traceability structures between the requirements and the conceptual schemas.

- Non-Functional Requirements. These requirements are also called quality requirements. We are developing an integrated framework to deal with structured functional and non-functional requirements from the very beginning of the requirements specification.

- The requirements specification environment (RETO) in its industrial version is currently under development. The main characteristic of this tool is that it was automatically generated from the Requirements Model metamodel (which includes all the static and dynamic properties) using the OlivaNova Model Execution Software® (ONME). At the time of the writing of this thesis, it is in a beta-version in the process of testing, and a new the graphical user interface is being developed.

Immediate future research activities include:

- Requirements for the Web. Properties related to the Web environment are being analyzed to extend our Requirements Model in order to capture aspects such as navigation and presentation that are main issues for Web applications.

- Requirements for user interface specification. Following the scenario approach for requirements specification, the different interactions between users and the system are naturally captured. For now, efforts are centered on deriving the internal decomposition of the system from an external interaction specification. A forthcoming work will focus on the characteristics of the external interactions in order to capture more specific interface properties.

- Requirements quality. The study and proposal of a quality model based on common standards like the ISO 9126.
• Generation of test cases from the scenario specification. As projects move to the testing phase, the use of scenarios aids integration testing. The scenario can act as a basis for determining test cases. Also, users can be involved in testing based on these scenarios.

• Software estimation from the Requirements Model. In this area, works will be related to estimation models based on requirement specifications that could be quantified using techniques based on Function Points.
List of Figures

Figure 2-1. The semiotic tetrahedron of FRISCO .......................................................... 26
Figure 2-2. Example of the use of the semiotic tetrahedron of FRISCO .................. 27
Figure 2-3. Models and model denotations ................................................................. 30
Figure 2-4. Models, model denotations and meta-levels ........................................... 30
Figure 2-5. System and system denotation ................................................................. 31
Figure 2-6. Organizational and information systems and their denotations ............. 33
Figure 2-7. The relationships between the UofD and the system world ................ 34
Figure 2-8. The submodels of the F3 approach ......................................................... 37
Figure 2-9. Four views on scenarios: form, purpose, content, and life cycle ......... 40
Figure 3-1. A Framework for Systems ........................................................................ 46
Figure 3-2. Function Decomposition Table (Traceability Matrix) ......................... 47
Figure 3-3. The Magic Square .................................................................................... 48
Figure 3-4. Decomposition of a system into components ........................................ 49
Figure 3-5. Excerpt of a FRT for a Car Rental system ............................................ 52
Figure 3-6. Use Case description structure ............................................................... 56
Figure 3-7. Step structure ......................................................................................... 56
Figure 3-8. Step example ......................................................................................... 57
Figure 3-9. An include relationship .......................................................................... 59
Figure 3-10. An extend relationship .......................................................................... 60
Figure 4-1. Excerpt of the Class Diagram for the Car Rental system ................... 63
Figure 4-2. State Transition Diagram for the Customer class .................................. 65
Figure 4-3. Interaction Diagram .............................................................................. 65
Figure 4-4. Pattern Language and use relationship .......................................................... 68
Figure 4-5. Excerpt of an OASIS specification ................................................................. 70
Figure 5-1. Structure and Notation of the Sequence Diagram ........................................ 80
Figure 5-2. A Sequence Diagram with multi-actor ......................................................... 81
Figure 5-3. Sequence Diagram Notation with Blocks .................................................... 82
Figure 5-4. Relationships between actors and classes ................................................... 83
Figure 5-5. Extension Property in class definition ........................................................ 85
Figure 5-6. Signal Message .............................................................................................. 86
Figure 5-7. Service Message .......................................................................................... 87
Figure 5-8. Query Message ............................................................................................. 89
Figure 5-9. Connect Message ........................................................................................ 90
Figure 5-10. Use Case for selling products ...................................................................... 93
Figure 5-11. Requirements Traceability in Gotel's classification ..................................... 96
Figure 5-12. Traceability between the Requirements and Conceptual Models ............. 98
Figure 5-13. Traceability from Requirements to Conceptual Model ............................ 99
Figure 5-14. Traceability Matrix (CRUD+A Table) ......................................................... 105
Figure 5-15. Traceability Matrix explained ..................................................................... 105
Figure 6-1. Traceability pyramid ................................................................................... 108
Figure 6-2. SD for Renting a Car .................................................................................... 110
Figure 6-3. Partial Class Diagram from class rules ....................................................... 110
Figure 6-4. SD for Creating (Purchase) a Car .............................................................. 112
Figure 6-5. Partial Class Diagram from Attribute Rules .............................................. 112
Figure 6-6. SD for deleting an insurance policy ............................................................ 116
Figure 6-7. Partial Class Diagram from Transaction Rules ........................................... 116
Figure 6-8. SD for a Global Transaction ....................................................................... 117
Figure 6-9. Include Relationship .................................................................................. 117
Figure 6-10. SD Create Operation ................................................................................. 118
Figure 6-11. SD Disable ................................................................. 118
Figure 6-12. Partial Class Diagram for Create Operation and Disable ............. 119
Figure 6-13. Extend Relationship.................................................................. 119
Figure 6-14. SD for creating an Invoice and Invoice Lines .................................. 120
Figure 6-15. Partial Class Diagram from Association Rule <<service/new>> .......... 121
Figure 6-16. SD for creating an Insurance Policy ............................................. 121
Figure 6-17. Partial Class Diagram from Association Rule <<connect>> ............. 122
Figure 6-18. SD for Renting a Car with Guarantor ........................................... 122
Figure 6-19. Partial Class Diagram from Association Rule with role names........ 123
Figure 6-20. SD for Creating Invoice and Invoice Lines .................................... 130
Figure 6-21. Partial Class Diagram from Aggregation Rule ............................... 130
Figure 6-22. An Actor Relationship from the Use Case Diagram....................... 132
Figure 6-23. Partial Class Diagram from Actor Relationship Rule..................... 132
Figure 6-24. Excerpt of a FRT ........................................................................ 136
Figure 6-25. Excerpt of the Administrator Use Case view ............................... 137
Figure 6-26. FRT for user Administrator .......................................................... 137
Figure 7-1. A view of RETO architecture ....................................................... 139
Figure 7-2. RETO and ONME Software communication schema .................... 140
Figure 7-3. Example of Class translation in XSLT ............................................. 143
Figure 7-4. OlivaNova Modeling Software ....................................................... 145
Figure 7-5. An overview of the RETO metamodel ........................................... 147
Figure 7-6. Three-tier Architecture ................................................................ 148
Figure 7-7. The RETO environment .................................................................. 149
Figure 7-8. FRT in the RETO form-based version ............................................ 150
Figure 7-9. FRT with functionality ................................................................. 151
Figure 7-10. Creation of a step in the Use Case description .............................. 152
Figure 7-11. Use Case description .................................................................... 152
Figure 7-12. FRT in RETO icon-based version ................................................................. 153
Figure 7-13. RETO Use Case diagram ........................................................................... 154
Figure 7-14. RETO general overview .............................................................................. 155
List of Tables

Table 1. Functional Model for the attribute number_cars........................................... 67
Table 2. Signal Message Properties................................................................. 86
Table 3. Service Message Properties............................................................. 88
Table 4. Query Message Properties................................................................. 89
Table 5. Connect Message Properties............................................................. 91
Table 6. Possible combinations for stereotype <<service/new>> ....................... 125
Table 7. Possible combinations for stereotype <<service/new>> with condition........ 126
Table 8. Possible combinations for stereotype <<connect>>................................. 127
Table 9. Possible combinations for stereotype <<connect>> with condition .......... 128
Bibliography


31. Musen M., Protege-II: An Environment for Reusable Problem-Solving Methods and Domain Ontologies, in Int'l Joint Conf. Artificial Intelligence IJCAI '93. 1993, Morgan Kaufmann.

32. Maiden N.A.M. and Sutcliffe A.G., Computational mechanisms for reuse of domain knowledge during requirements engineering, in ICSE-Workshop on research issues in the intersection between software engineering and artificial intelligence. May, 1994: Sorrento, Italy.


44. Bubenko J., Rolland C., Loucopoulos P. and DeAntonnellis V., *Facilitating Fuzzy to Formal Requirements Modelling*, in *Int. Conf. on Requirements Engineering (ICRE)*. 1994: Colorado Springs, USA.


Appendix A

Traceability Rules Catalog

In this appendix, a summary of the Traceability Rules Catalog is presented. The rules in the catalog are organized according to the kind of Conceptual Schema elements they can generate.

These elements are:

- Class
- Attribute
- Service
- Association Relationship
- Cardinality
- Aggregation Relationship
- Agent Relationship
- Inheritance Relationship
- Preconditions
- Integrity Constraints
- Triggers
- Static and Dynamic Association-Aggregation Relationship
- State Transition Diagram
- Presentation Model
Class Rules

- **Rule 1.** For every distinct actor class participating in any Sequence Diagram a class will be generated in the Class Diagram.
  - *Traceability:* Strong

- **Rule 2.** For every distinct class participating in any Sequence Diagram a class will be generated in the Class Diagram.
  - *Traceability:* Strong

- **Rule 3.** The boundary classes (usually called System) in Sequence Diagrams will not have an explicit representation in the Class Diagram.
  - *Traceability:* Strong

Attribute Rules

- **Rule 4.** All the attributes recorded in a class (when it is created in a Sequence Diagram or modified later) will be considered as attributes for the class in the Class Diagram.
  - *Traceability:* weak

- **Rule 5.** For every message labeled with the stereotype <<service/new>>, all its arguments will be translated into attributes of the receiver class.
  - *Traceability:* weak

Service Rules

- **Rule 6.** For every message with the stereotype <<service>>, a service will be generated in the corresponding receiver class.
  - *Traceability:* strong
• **Rule 7.** For every message with the stereotype `<service/new>`, a service of type “new” will be generated in the corresponding receiver class.
  - **Traceability:** strong

• **Rule 8.** For every message with the stereotype `<service/destroy>`, a service of type “destroy” will be generated in the corresponding receiver class.
  - **Traceability:** strong

• **Rule 9.** Every Sequence Diagram will be translated into a transaction. The name of the transaction is “implicit and partially defined” with the same name as the Sequence Diagram prefixed by “TR_”.
  - **Traceability:** strong

• **Rule 10.** The transaction definition will be composed of all the services derived from the messages with the `<service>` stereotype found inside of the current Sequence Diagram.
  - **Traceability:** strong.

• **Rule 11.** The transaction will be considered as a local transaction if, in the scenario, there is only one message with the `<service>` stereotype leaving from the class system. In other cases, it will be considered a global transaction.
  - **Traceability:** weak.

• **Rule 12.** For INCLUDE relationships between Sequence Diagrams, the transaction generated from the base Sequence Diagram will perform an invocation of the transaction generated from the included Sequence Diagram (the included transaction is not changed).
  - **Traceability:** strong.

• **Rule 13.** For EXTEND relationships between Sequence Diagrams, the transaction generated from the base Sequence Diagram will perform an explicit invocation (with the extend condition as control condition) of the transaction generated for the extended Sequence Diagram.
  - **Traceability:** strong.
Association Relationship Rules

- **Rule 14.** For every message between two classes labeled with the stereotype `<service/new>>` where both classes are distinct from the system class, an association relationship between these classes will be generated.
  - **Traceability:** strong

- **Rule 15.** For every message between two classes labeled with the stereotype `<connect>>` an association relationship between these classes will be generated.
  - **Traceability:** strong

- **Rule 16.** For every message with the stereotype `<service/new>> or `<connect>>` where classes using role names appears, an association relationship between these classes will be generated using these role names on the ends of the relationship.
  - **Traceability:** weak

Cardinality Rules

- **Rule 17.** If there is a message with the stereotype `<service/new>>` between two classes A and B, both distinct from class system, and it is enclosed in a iteration block or it is a iteration message, the maximum cardinality in the receiver class will be “n” and otherwise it will be “1”.
  - **Traceability:** weak

- **Rule 18.** If there is a message with the stereotype `<service/new>>` between two classes A and B, both distinct from class system, and in the set of ALL Sequence Diagrams where objects from the class B (receiver) are created all the sender classes A refer to the same class, the minimum cardinality in the side of the class A (sender) will be “1” and otherwise it will be “0”.
  - **Traceability:** weak

- **Rule 19.** If there is a message with the stereotype `<service/new>>` in a class A and in the set of ALL Sequence Diagram where A is created, always the object from the class A has to create or be related to an object from class B (sending a


<<service/new>> or <<connect>> message) the minimum cardinality for the class A with respect to class B will be “1” and otherwise it will be “0”. In addition, if this message is inside of an iteration block or it is an iteration message, the minimum cardinality will be the lower value of the iteration. If the message is in a conditional block or it is a conditional message the minimum cardinality will be zero. And finally, if more than one of these situations holds, the minimum cardinality will be the less restrictive.

- **Traceability**: weak

- **Rule 20.** If there is a message with the stereotype <<service/new>> between two classes distinct from class system, and the message is a conditional message or it is in a conditional block or it is in an iteration that can occur 0 times, the minimum cardinality in the receiver class will be “0”.
  - **Traceability**: weak

- **Rule 21.** If there is a message with the stereotype <<connect>> between two classes A and B, both distinct from the class system, and if it is enclosed in a iteration block or it is a iteration message, the maximum cardinality in the class B (receiver) will be “n” and otherwise it will be “1”.
  - **Traceability**: weak

- **Rule 22.** If there is a message with the stereotype <<connect>> between two classes A and B, both distinct from the class system, and if objects from the receiver class B cannot be created from another class different of A, the minimum cardinality in the sender class A will be “1”, otherwise it will be “0”.
  - **Traceability**: weak

- **Rule 23.** If there is a message with the stereotype <<connect>> between two classes A and B, both distinct from the class system, and if in ALL the Sequence Diagrams when an object from the class A is created, an object from the class B is created or related, the minimum cardinality in the class B will be “1”, otherwise it will be “0”.
  - **Traceability**: weak

- **Rule 24.** If there is a message with the stereotype <<connect>> between two classes A and B, both distinct from the class system, and the message is a conditional message or it is in a conditional block or it is in a iteration that can occur 0 times, the minimum cardinality in the class B (receiver) will be “0”.
  - **Traceability**: weak
Rule 25. If there is a message with the stereotype <<connect/exclusive>> between two classes A and B, both distinct from the class system, the cardinality in the class A (sender) will be “min = 0” and “max = 1”.

- Traceability: weak

Rule 26. For those cases that the same association or aggregation relationship has different proposed values for cardinalities (more than one rule can be applied), those values less restrictive will be used.

- Traceability: weak

Rule 27. In those cases where the values for cardinalities cannot be computed, the values “Min = 0” and “Max = n” will be used.

- Traceability: weak

Aggregation Relationship Rules

Rule 28. For every message with the stereotype <<service/new>> between two classes A and B, which are distinct from the class system, then an aggregation relationship between these classes will be generated.

- Traceability: weak

Agent Relationship Rules

Rule 29. For every actor class (representing an initiator actor) of a Sequence Diagram for the basic course of a Use Case, an agent relationship will be generated between the actor class and the service (transaction) that is derived from the quoted Sequence Diagram.

- Traceability: strong
Inheritance Relationship Rules

- Rule 30. All the inheritance relationships between actors in the Use Case diagram are represented as an inheritance relationship between the corresponding actor classes in the Class Diagram.
  - Traceability: strong

Precondition Rules

- Rule 31. The precondition from the Use Case description is the precondition of the corresponding Sequence Diagram and, transitively, of the corresponding transaction derived from it.
  - Traceability: strong

Integrity Constraint Rules

- Rule 32. Any integrity constraint specified in the class during the Sequence Diagram specification will be represented as an integrity constraint for the class in the Class Diagram.
  - Traceability: weak

Trigger Rules

- Rule 33. All the triggers specified in a class during the Sequence Diagram specification will be represented as a trigger specification in the Interaction Diagram.
  - Traceability: weak
Static and Dynamic Association-Aggregation Rules

- **Rule 34.** Given two classes, A and B, both of which are distinct from the class system, and if there is a message with the stereotype <<service/new>> from A to B and there are no messages with the stereotype <<connect>> from A to B or B to A (in another Sequence Diagram), then the relationship between A and B will be static from the point of view of B.
  - *Traceability:* weak

- **Rule 35.** Given two classes, A and B, both of which are distinct from the class system, and if there is a message with the stereotype <<service/new>> from A to B and there are some messages with the stereotype <<connect>> from A to B or B to A (in another Sequence Diagram), then the relationship between A and B will be dynamic from the point of view of B.
  - *Traceability:* weak

- **Rule 36.** In case of misspecifications or contradictions for the static/dynamic property of the association-aggregation relationship it will be dynamic.
  - *Traceability:* weak

State Transition Diagram Rules

- **Rule 37.** A basic State Transition Diagram (STD) will be generated for each class. This basic STD is composed of three states: “pre-creation”, “created” and “destroyed”. Each service identified in the class will be represented as a transition between states.

  *All the creation services (local events of type “new” or local transactions where its first service is a local event of type “new”) will be transitions from the ”pre-creation” to the “created” state.*

  *All the destruction services (local events of type “destroy” or local transactions where its last service is a local event of type “destroy”) will be transitions from the “created” to the “destroyed” state.*
The rest of services will be transitions departing from and going to the “created” state.

- Traceability: weak

**Presentation Model Rules**

- **Rule 38** The Use Case model is restricted to each initiator actor (UCv_{ai}), and this view is applied to the FRT obtaining a Hierarchical Action Tree (HAT) for each initiator actor (ai).

  - Traceability: weak
Appendix B

Case Study: Car Rental

1. Introduction

A Car Rental company needs to automate the management of car rentals to customers. This company is located in a tourist area, and the set of cars to be rented varies greatly between the winter and the summer season.

The cars are usually bought at the beginning of the season and sold, at the end of it. The purchase operations of the cars usually take into account their sale after a certain period of time (six months).

The main activity, the rental, involves other kinds of derived activities such as the car maintenance and repair, and extra rentals (telephone, driver, etc.) that also need to be automated.

2. Description of the System

This system will perform the management of the purchase, the rental, maintenance, and sale of cars, as well as the management of the different types of customers that work with the company. These functionalities are described following:

Cars

The fleet of cars of the company varies approximately from 250 units at high season (months of April until September) to 50 units at low season (the rest of the year). We describe a car record of the company below:
The cars are classified in accordance with certain criteria such as:

- **Class**: refers to the purpose that the car is destined for (i.e., Tourism, Van, and Industrials).

- **Group**: depends on the size or options of the cars. It is defined for each class (i.e., for a class of vehicle). For example, the cars in Group A are the simplest and smallest cars in a class, and those that correspond to a lower rental price.

The state of a car represents its current situation:

- **Rented**: when a car is assigned to an open contract.

- **To prepare**: when a customer has just returned the car. The car needs to be prepared (cleaning, maintenance, and so on) before renting again. In addition, when the car is new, the state should be to prepare in order to perform a general check before being used for the first time.

- **Ready**: the car is ready to be rented.

Another aspect to consider is the availability of the car. Whenever the car is not in the rented state, it can be disabled for some special activity (i.e., to be reserved by some manager of the company) or for the accomplishment of specific tasks. These activities are called Operations.

The different kinds of operations are: preparation, maintenance, and repair. Following, we give more information about the operations:

**Preparation**: the car goes to cleaning, verification of fuel, water, oil, etc.

**Maintenance**: the car goes for maintenance (periodic revisions). The maintenance data must be registered (date, hour, description, etc.). We give an example of a maintenance record below:
**Maintenance Record**

<table>
<thead>
<tr>
<th>Maintenance number:</th>
<th>124111</th>
</tr>
</thead>
<tbody>
<tr>
<td>Plate number:</td>
<td>V-4512-EC</td>
</tr>
<tr>
<td>Date:</td>
<td>05/01/2002</td>
</tr>
<tr>
<td>Ordered:</td>
<td>Carl Sagan</td>
</tr>
<tr>
<td>Description:</td>
<td>Oil change</td>
</tr>
<tr>
<td>Price:</td>
<td>24.00 €</td>
</tr>
<tr>
<td>Return date:</td>
<td>05/02/2002</td>
</tr>
</tbody>
</table>

**Repair:** the car must be repaired because of some failure. In this case, the repair can be carried out in an external garage or in a garage of the company. In both cases, the data related to the repair must be registered (date, hour, price, authorized by, etc.) and the car must be disabled. We give an example of a reparation record below:

**Repair Record**

<table>
<thead>
<tr>
<th>Reparation number:</th>
<th>124000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Plate number:</td>
<td>V-4512-EC</td>
</tr>
<tr>
<td>Date:</td>
<td>05/01/2002</td>
</tr>
<tr>
<td>Ordered:</td>
<td>Steve Stuart</td>
</tr>
<tr>
<td>Description:</td>
<td>Repair of the ignition system.</td>
</tr>
<tr>
<td>Price:</td>
<td>32,00€</td>
</tr>
<tr>
<td>Return date:</td>
<td>05/02/1999</td>
</tr>
</tbody>
</table>

An important point to be considered is that, when a car is disabled it cannot be rented but it can be sold.

*Observation:* in accordance with the company rules, those cars that exceed 150,000 km must be automatically disabled and cannot be rented again.

Other aspects to consider with respect to the cars are:

**Purchase:** during the purchase of the car, its data is registered (the plate number, the make, the model, etc.). Every new car should be covered by an insurance policy. Sometimes, it is possible to agree to the later sale of the car, registering a date of delivery (normally when the season is ended) and the price. Finally, the car is automatically disabled in order to be prepared by the personnel of the company (clean, check levels, and so on) before it’s first renting.

**Sale:** the car can be sold agreeing a price and a date of delivery. In the date of delivery, if the car is not rented it must be automatically disabled to be delivered. We give an example of a sale record of a car below:
**Sale Record**

<table>
<thead>
<tr>
<th>Sale number:</th>
<th>114030</th>
</tr>
</thead>
<tbody>
<tr>
<td>Plate number:</td>
<td>4512-EBC</td>
</tr>
<tr>
<td>Delivery Date:</td>
<td>10/11/2003</td>
</tr>
<tr>
<td>Ordered:</td>
<td>Jack Rosenfield</td>
</tr>
<tr>
<td>Sale Price:</td>
<td>4.000,00 €</td>
</tr>
<tr>
<td>Operation Date:</td>
<td>05/02/2003</td>
</tr>
<tr>
<td>Observations:</td>
<td>--</td>
</tr>
</tbody>
</table>

**Insurance:** All the cars must have covered by insurance to be rented. The information related to the insurance policy must be provided (i.e., the name of the insurance company, type and period of coverage). If the period of coverage of the insurance ends, and the car is not rented, it will be automatically disabled and it will not be possible to enable it unless the period of coverage is renewed.

**Customers**

The customers can be of the following types:

**Direct:** regular or occasional customer to whom the predefined rate is applied according to the contract. All the personal data of the customer is gathered for the contract: name, SSN, address, telephone, credit card type, credit card number, license number, etc. We give an example of a record for this kind of customer:

**Direct Customer**

<table>
<thead>
<tr>
<th>Code:</th>
<th>124111</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name:</td>
<td>John Stevenson</td>
</tr>
<tr>
<td>Title:</td>
<td>Sr.</td>
</tr>
<tr>
<td>Address:</td>
<td>Poeta Artola, 27</td>
</tr>
<tr>
<td>City:</td>
<td>Valencia</td>
</tr>
<tr>
<td>SSN:</td>
<td>54 452 452R</td>
</tr>
<tr>
<td>License Number:</td>
<td>6546665</td>
</tr>
<tr>
<td>Credit Card Number:</td>
<td>1224 2152 2152</td>
</tr>
<tr>
<td>Type of Card:</td>
<td>VISA</td>
</tr>
<tr>
<td>Phone:</td>
<td>+34 96-5455211</td>
</tr>
<tr>
<td>Fax:</td>
<td>+3496-4656552</td>
</tr>
</tbody>
</table>

**Agency or intermediary:** In this case, the customer is a branch agency or another intermediary company; therefore, the personal data of the driver is not collected.
Additionally, a commission can be agreed to. We give an example of a customer record of an agency below:

<table>
<thead>
<tr>
<th>Agency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Agency Code:</td>
</tr>
<tr>
<td>Name:</td>
</tr>
<tr>
<td>Address:</td>
</tr>
<tr>
<td>City:</td>
</tr>
<tr>
<td>Contact:</td>
</tr>
<tr>
<td>Commission:</td>
</tr>
</tbody>
</table>

**Rates**

Rates represent the price of the car and the insurance per time unit (day). In fact, this price depends on the characteristics of the contract and the class and group of the chosen car. We give an example of this kind of record below:

<table>
<thead>
<tr>
<th>Rent Rates / insurance (per day)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Tourism</td>
</tr>
<tr>
<td>Group A</td>
</tr>
<tr>
<td>Group B</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>

**Contracts**

Once the customer agrees upon the terms, a contract is made. This contract must be unique for each car rental.

When a contract is confirmed and the client signs it, the deposit is received (if it exists) and a copy of the contract and the car keys are given to the client.

The contract is open until the client returns the car. At that moment, we have to calculate the total amount to be paid by the client or to return him the difference between the initial deposit and the total amount.

The contracts should have information about the rent date, return date, rental location, return location, number of days, cost of insurance, amount of the deposit, the discount and, finally, the total cost.

An example of a Rental Contract record is shown next:
**Other** important aspects to consider with respect to the contract are:

**Insurance:** When a car is rented, it must be covered by an insurance policy. This should be paid by the customer in time units (day). The insurance rates vary according to the car selected and are specified in the car rental fee. Additionally, an amount can be registered in order to express the level of responsibility of the customer in case of accident.

**Extras:** sometimes, other additional services that are called extras can be rented, such as Airport services (pick up or delivery), fuel, and car’s accessories (roof rack, baby chair, telephone). For each extra, a code and a price per day per unit are assigned independently of the characteristics of the car.

Whenever the contract is open, it is possible to add these extras to it for the desired time and quantity. For example, a car can be rented for one week and include the rental of two mobile phones for three days.

An example of an Extra Rate Record is presented below:
Employees

In the company, there are two kinds of employees: Users and Administrators.

The Users can perform all the habitual tasks of management (rent or return cars, change rentals, maintain customer data, maintain operations, etc.).

The Administrators can perform the same operations as users. However, they can also handle higher-level management tasks:

- purchase, sale, and eliminate cars
- maintain rates (rental, insurances, and extras)
- control insurance policies
- manage employees

We give an example of an Employee Record below:

<table>
<thead>
<tr>
<th>Employee Record</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Number:</strong></td>
</tr>
<tr>
<td><strong>Employee Type:</strong></td>
</tr>
<tr>
<td><strong>Name:</strong></td>
</tr>
<tr>
<td><strong>Start Date:</strong></td>
</tr>
<tr>
<td><strong>SSN</strong></td>
</tr>
<tr>
<td><strong>Observations:</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Extra Rate Record</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Code:</strong></td>
</tr>
<tr>
<td><strong>Name:</strong></td>
</tr>
<tr>
<td><strong>Price:</strong></td>
</tr>
<tr>
<td><strong>Description:</strong></td>
</tr>
<tr>
<td><strong>Units:</strong></td>
</tr>
</tbody>
</table>
3. **Requirements Model**

In this section, we specify the Requirements Model for the case study previously presented. This model is obtained by the specification of a Mission Statement, a Function Refinement Tree, Use Case Diagrams, and Sequence Diagrams. In the following subsections, we describe each one of them in detail.

3.1. **Mission Statement**

The mission of the Car Rental system is to automate the management of cars, rentals, and customers of a Car Rental company. The main activity, the rental, involves another series of derived activities like the maintaining and repairing of cars, additional accessories to be rented (extras), and customer management. These derived activities must also be automated.

3.2. **Function Refinement Tree (FRT)**

As a first step, we have elaborated a list of functionalities that the system would have to fulfill. With this list of functionalities, we have elaborated a Function Refinement Tree (FRT), which is shown in Table 1.

The following step consists of representing each one of the functions contained in the FRT in Use Cases (UC) and grouping them according to the identified functional groups.

The Use Cases are graphically represented using Use Case diagrams (UCD).
Table 1. Function Refinement Tree

<table>
<thead>
<tr>
<th>Functional Group</th>
<th>Sub-Functional Group</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. CarManagement</td>
<td></td>
<td>1.0.1. Purchase</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1.0.2. Sell</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1.0.3. Deliver</td>
</tr>
<tr>
<td>1.1 RateManagement</td>
<td></td>
<td>1.1.1. CreateRate</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1.1.2. DeleteRate</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1.1.3. ModifyRate</td>
</tr>
<tr>
<td>1.2 GarageManagement</td>
<td></td>
<td>1.2.1. CreateGarage</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1.2.2. DeleteGarage</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1.2.3. ModifyGarage</td>
</tr>
<tr>
<td>1.3. InsuranceManagement</td>
<td></td>
<td>1.3.1. CreateInsurance</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1.3.2. DeleteInsurance</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1.3.3. ModifyInsurance</td>
</tr>
<tr>
<td>1.4. InsuranceCompanyManagement</td>
<td></td>
<td>1.4.1. CreateInsuranceCompany</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1.4.2. DeleteInsuranceCompany</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1.4.3. ModifyInsuranceCompany</td>
</tr>
<tr>
<td>1.5. OperationManagement</td>
<td></td>
<td>1.5.1. CreateOperation</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1.5.2. Disable</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1.5.3. Enable</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1.5.4. DeleteOperation</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1.5.5. FinalizeOperation</td>
</tr>
<tr>
<td>2. CustomerManagement</td>
<td></td>
<td>2.1. CreateCustomer</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2.2. DeleteCustomer</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2.3 ModifyCustomer</td>
</tr>
<tr>
<td>3. ContractManagement</td>
<td></td>
<td>3.0.1. Rent</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3.0.2. ModifyContract</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3.0.3. Return</td>
</tr>
<tr>
<td>3.1. ExtrasManagement</td>
<td></td>
<td>3.1.1. CreateExtraType</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3.1.2. DeleteExtraType</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3.1.3. ModifyExtraType</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3.1.4. ExtrasAssignment</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4.2. DeleteUser</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4.3. ModifyUser</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4.4. Promote</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4.5. Demote</td>
</tr>
</tbody>
</table>
4. Use Case Diagrams

In this section, we describe the Use Case diagrams corresponding to the nodes of the Functional Refinement Tree. The intermediate nodes are Functional Groups that are represented as packages, and the leaf nodes are represented as Use Cases.

The main Use Case Diagram is built using the nodes of the first-level of the Functional Refinement Tree. These nodes represent the main areas of the system. In addition, the actors and their relationships, identified in the problem domain, are shown.

The set of all the Use Case diagrams created from the Functional Refinement Tree are described following:

**Main Use Case diagram.** The Functional Groups *Car Management*, *Customer Management*, *User Management*, and *Contract Management* are nodes at the top-level of the Functional Refinement Tree and are represented in the Use Case diagram as packages. In addition, we identify two actors in the problem domain: *User* and *Administrator*. The *Administrator* actor is specialized from the *User* actor. This means that an *Administrator* inherits all the behavior specified for a *User*. These elements are depicted in the diagram in Figure 1

![Figure 1. Use Case View](Image)

**1. Car Management.** We describe the management of the fleet of cars of our Car Rental company in the diagram presented in Figure 2. The Functional Groups *Rate Management*, *Garage Management*, *Insurance Company Management*, *Insurance Management*, and *Operation Management* are represented as UML packages. The actor *Administrator* is responsible for activating the following Use Cases: *Purchase*, *Sell*, and *Deliver*.

The *Purchase* Use Case includes three Use Cases: *Disable* from the *Operation Management* functional group, *Create Insurance* from the *Insurance Management* functional group, and *Sell* that is defined in this functional group (*Car Management*).

The *Sell* Use Case includes the *Deliver* Use Case, and the *Deliver* Use Case includes the *Delete Insurance* Use Case from the *Insurance Management* functional group.
1.1. Rate Management. We describe the management of the rates used for car renting in the diagram presented in Figure 3. The actor *Administrator* is responsible for activating the Use Cases *CreateRate*, *DeleteRate*, and *ModifyRate*.

1.2. Garage Management. We describe the management of garages that work with the company in the diagram presented in Figure 4. The actor *Administrator* is responsible for activating the Use Cases *CreateGarage*, *DeleteGarage*, and *ModifyGarage*. 

Figure 2. Car Management

Figure 3. Rate Management
1.3. **Insurance Management.** We describe the management of insurance policies in the diagram presented in Figure 5. As can be observed, the actor *Administrator* can only modify the data of a policy and cannot create or destroy it directly. This is done at the moment of the *Purchase* and *Delivery* of the car. (Please refer to the Use Case relationships between *Purchase* and *CreateInsurance*, and *Deliver* and *DeleteInsurance* in the Use Case diagram shown in Figure 2).

1.4. **Insurance Company Management:** We describe the management of Insurance Companies that work with our Car Rental company in the diagram presented in Figure 6. The actor *Administrator* is responsible for activating the Use Cases *CreateInsuranceCompany*, *DeleteInsuranceCompany*, and *ModifyInsuranceCompany*.
1.5. Operations Management: We describe the management of the different kind of operations that can be performed on a car in the diagram presented in Figure 7. The actor User is responsible for activating the following Use Cases: CreateOperation, Disable, DeleteOperation, Enable, and FinalizeOperation.

The CreateOperation Use Case includes the Disable Use Case. In the same way, the FinalizeOperation includes the Enable Use Case.

![Figure 7. Operations Management](image)

2. Customer Management. We describe the management of customers that work with the company in the diagram presented in Figure 8. The actor User is responsible for activating the following Use Cases: CreateCustomer, DeleteCustomer, and ModifyCustomer.

![Figure 8. Customer Management](image)
3. Contract Management: We describe the management of rental contracts in the diagram presented in Figure 9. The actor User can Car Rental, modify a contract or return a car by activating the Use Cases Rent, ModifyContract, and Return, respectively.

The Rent Use Case includes the ExtrasAssignment Use Case from the Extras Management functional group. Note that the Contract Management functional group also includes the Extras Management functional (sub)group.

![Diagram of Contract Management](image)

**Figure 9. Contract Management**

3.1. Extras Management. We describe the management of extras (accessories) that can be rented with the cars in the diagram presented in Figure 10. The actor Administrator is responsible for activating the following Use Cases: CreateExtraType, DeleteExtraType, and ModifyExtraType. The specialization relationship between the actor Administrator and the actor User is also shown.

As can be observed, the actor Administrator cannot perform extras assignments. This is done when the car is rented. (Please refer to the Use Case relationships between Rent and ExtrasAssignment in the Use Case diagram shown in Figure 9).
4. **User Management.** We describe the management of users for the Car Rental system in the diagram presented in Figure 11. The actor *Administrator* is responsible for creating, modifying and eliminating users, as well as, promoting and demoting them. These are done by activating the following Use Cases: *CreateUser, ModifyUser, DeleteUser, Promote* and *Demote*, respectively.
5. Sequence Diagrams

All Use Cases described in the previous section are realized by Sequence Diagrams in accordance with the strategy introduced in section 5.3 of the thesis (from Requirements Model to Sequence Diagrams).

1.0.1. Purchase: This scenario describes the purchase of a new car by the Car Rental company.

In the Sequence Diagram shown in Figure 12, the Administrator initiates this Sequence Diagram with the message *starts_purchase*.

After data introduction, the system creates a new object from the *Car* class using the message *create_car* that has the stereotype *<<service / new>>* assigned to it. The new object needs to select its corresponding *Rate*. This is done using the message *connect_rate* that has the stereotype *<<connect / 1,1>>* assigned to it. In addition, the car needs to create an insurance policy, which is performed by including the Use Case *TR_Create_Insurance*. If desired, the car can be sold (usually to be delivered at the end tourist season) by the conditional inclusion of the Use Case *TR_Sell*. Finally, in the *Purchase* scenario, the car has to be disabled before its first renting by including the Use Case *TR_Disable*.

![Figure 12. SD_Purchase](image-url)

Figure 12. SD_Purchase
1.0.2. Sell: This scenario represents the sale of a car by the Car Rental system. This Use Case is usually performed just after purchasing a new car, when the terms are agreed upon and a delivered date is established (usually at the end of the tourist season).

In the Sequence Diagram shown in Figure 13, the Administrator initiates this Sequence Diagram using the message starts_sell.

After data introduction, the system creates a new object from the class Sale using the message create_sale that has the stereotype <<service / new>> assigned to it. The new object needs to select its corresponding car. This is done by using the message connect_car that has the stereotype <<connect / 1,1>> assigned to it.

The excl indicates that this object cannot participate in another sale. In addition, if desired, the car can be delivered by the conditional inclusion of the Use Case TR_Deliver.

Figure 13. SD_Sell
1.0.3. **Deliver:** This scenario describes the delivery of a car sold to the buyer.

In the Sequence Diagram showed in Figure 14, the *Administrator* initiates this scenario with the message *starts_deliver*.

After data introduction, the system queries the *Car* for information using the message *get_car_info* that has the stereotype *<<query / 1,1>>* assigned to it.

Then, two main operations are performed: deletion of the car and modification of the delivery date (if this is different to the agreed upon delivery date). The first one is done using the message *delete_car* that has the stereotype *<<service / destroy>>* assigned to it. The second operation is done using the message *modify_date_of_delivery* that has the stereotype *<<service / update>>*.

In addition, the object of the *Insurance* class associated with the object of the class *Car* can be eliminated using the Use Case *TR_DeleteInsurance*.

![Figure 14. SD_Deliver](image-url)
### 1.1.1. CreateRate

This scenario describes the creation of a rate according to values provided by the user.

In the Sequence Diagram shown in Figure 15, the *Administrator* initiates this scenario with the message **starts_create_rate**.

After data introduction, the system verifies whether the *Rate* exists using the message **exist_rate** that has the stereotype \(<\text{query}>\) assigned to it. If it does not exist, the system creates a new object from the *Rate* class using the message **create_rate** that has the stereotype \(<\text{service/new}>\) assigned to it.

![Figure 15. SD_CreateRate](image-url)
1.1.2. **DeleteRate**: This scenario describes the elimination of a rate in the Car Rental system.

In the Sequence Diagram shown in Figure 16, the Administrator initiates this scenario with the message `starts_delete_rate`.

After data introduction, the system eliminates a Rate using the message `delete_rate` that has the stereotype `<<service / destroy>>` assigned to it. Note that this message is only executed if the control condition is satisfied.

![Figure 16. SD_DeleteRate](image-url)
1.1.3. ModifyRate: This scenario describes the change of the rate information in the Car Rental system.

In the Sequence Diagram shown in Figure 17, the Administrator initiates this scenario with the message starts_modify_rate.

After data introduction, the system queries the Rate for information using the message get_rate_info that has the stereotype <<query / 1,1>> assigned to it. If desired, the modification of a Rate is done using the message modify_rate that has the stereotype <<service / update>> assigned to it.
1.2.1. **CreateGarage**: This scenario describes the creation of a garage in the Car Rental system. It supposes the creation of a new garage to which the company will take its cars to be repaired.

In the Sequence Diagram shown in Figure 18, the *Administrator* initiates this scenario with the message *starts_create_garage*.

After data introduction, the system verifies whether a *Garage* exists using the message *exist_garage* that has the stereotype *<<query>>* assigned to it. If it does not exist, the system creates a new object from the *Garage* class using the message *create_garage* that has the stereotype *<<service / new>>* assigned to it.

![Figure 18. SD_CreateGarage](image)
1.2.2. **DeleteGarage**: This scenario describes the elimination of a garage in the Car Rental system.

In the Sequence Diagram shown in Figure 19, the *Administrator* initiates this scenario with the message *starts_delete_garage*.

After data introduction, the system eliminates a *Garage* using the message *delete_garage* that has the stereotype `<service / destroy>` assigned to it. Note that this message is only executed if the control condition is satisfied.

![Sequence Diagram](image)

*Figure 19. DS_DeleteGarage*
1.2.3. **ModifyGarage**: This scenario describes the change of the garage information in the Car Rental system.

In the Sequence Diagram shown in Figure 20, the Administrator initiates this scenario with the message `starts_modify_garage`.

After data introduction, the system queries the Rate for information using the message `get_garage_info` that has the stereotype `<<query>>` assigned to it. If desired, the modification of a Garage is done using the message `modify_garage` that has the stereotype `<<service / update>>` assigned to it.

![Sequence Diagram](image.png)

**Figure 20. SD_ModifyGarage**
1.3.1. CreateInsurance: This scenario describes the creation of an insurance policy for a car in the Car Rental system.

In the Sequence Diagram shown in Figure 21, the Administrator initiates this scenario with the message starts_create_insurance.

After data introduction, the system verifies whether an object of the Car class exists by using the message exist_car that has the stereotype <<query>> assigned to it.

As can be observed in this diagram, the creation of an Insurance object supposes its association with an object of the Car class and with an object of the InsuranceCompany class. They represent the car to be covered and the company that will assure it, respectively. Thus, the system creates a new object from the Insurance class using the message create_insurance that has the stereotype <<service / new>> assigned to it. The new object needs to select its corresponding car and insurance company. This is done by means of the messages connect_car and connect_insurance_company that have the stereotype <<connect / 1,1>> assigned to them. In addition, the number of contracts of the InsuranceCompany is increased by means of the message modify_number_of_insurances that has the stereotype <<service / update>> assigned to it.

![Figure 21. SD_CreateInsurance](image-url)
1.3.2. **DeleteInsurance**: This scenario takes place at the moment the car that is associated to the insurance policy is eliminated from the system. It also supposes the elimination of the insurance policy.

In the Sequence Diagram shown in Figure 22, the **Administrator** initiates this scenario with the message `starts_delete_insurance`.

After data introduction, the system verifies whether an insurance exists using the message `exist_insurance` that has the stereotype `<<query / 0,1>>` assigned to it. If it exists, the system increments the number of `InsuranceCompany` using the message `modify_number_of_insurances` that has the stereotype `<<service / update>>` assigned to it.

Following, the system eliminates the insurance policy using the message `delete_insurance` that has the stereotype `<<service / destroy>>` assigned to it.

![Sequence Diagram](image)

**Figure 22. SD_DeleteInsurance**
1.3.3. **ModifyInsurance**: This scenario takes place when the terms of the insurance policy change (i.e., the price or the dates of the policy).

In the Sequence Diagram shown in Figure 23, the Administrator initiates this scenario with the message `starts_modify_insurance`.

After data introduction, the system queries the Insurance for information using the message `get_insurance_info` that has the stereotype `<query / 1,1>` assigned to it. If desired, the modification of an Insurance is done using the message `modify_insurance` that has the stereotype `<service / update>` assigned to it.
1.4.1. **CreateInsuranceCompany**: This scenario describes the creation of an insurance company in the Car Rental system.

In the Sequence Diagram shown in Figure 24, the *Administrator* initiates this scenario with the message *starts_create_insurancecompany*.

After data introduction, the system verifies whether the *InsuranceCompany* exists using the message *exist_insurancecompany* that has the stereotype *<<query>>* assigned to it. If it does not exist, the system creates a new object from the *InsuranceCompany* class using the message *create_insurance* that has the stereotype *<<service / new>>* assigned to it.

![Figure 24. SD_CreateInsuranceCompany](image-url)
1.4.2. **DeleteInsuranceCompany**: This scenario describes the elimination of an insurance company in the Car Rental system.

In the Sequence Diagram shown in Figure 25, the **Administrator** initiates this scenario with the message *starts_delete_insurancecompany*.

After data introduction, the system eliminates an **InsuranceCompany** using the message *delete_insurancecompany* that has the stereotype *<<service / destroy>>* assigned to it. Note that this message is only executed if the control condition is satisfied.

![Figure 25. SD_DeleteInsuranceCompany](image-url)
1.4.3. **ModifyInsuranceCompany**: This scenario describes the changing of an insurance company information in the Car Rental system.

In the Sequence Diagram shown in Figure 26, the Administrator initiates this scenario with the message `starts_modify_insurancecompany`.

After data introduction, the system queries the `InsuranceCompany` for information using the message `get_insurancecompany_info` that has the stereotype `<<query / 1,1>>` assigned to it. If desired, the modification of an `InsuranceCompany` is done using the message `modify_insurance` that has the stereotype `<<service / update>>` assigned to it.

![Sequence Diagram](Figure 26. SD_ModifyInsuranceCompany)
1.5.1. CreateOperation: This scenario describes the creation of an operation that will be performed on a car. There are three types of operations: preparation, reparation and maintenance. After registering an operation the corresponding car must be selected. If the type of operation is repair, the garage that will make the repair must be selected. When the car is in the garage, it cannot be rented (it is disabled).

In the Sequence Diagram shown in Figure 27, the User initiates this scenario with the message starts_create_operation.

After data introduction, the system verifies whether the Car exists using the message exist_car that has the stereotype <<query>> assigned to it. If it does not exist, the system creates a new object from the Operation class using the message create_operation that has the stereotype <<service / new>> assigned to it.

The new object needs to select its corresponding car and garage. This is done using the messages connect_car and connect_garage that have the stereotypes <<connect / 1,1>> assigned to them. Note that the message connect_garage is only activated if the type of the operation is Repair (represented by a control condition). In addition, if the type of the operation is Repair the system also calls the TR_Disable Use Case.

![Figure 27. SD_CreateOperation](image-url)
1.5.2. Disable: This scenario describes the disabling of a car.

In the Sequence Diagram shown in Figure 28, the User initiates this scenario with the message `starts_disable`.

After data introduction, the system verifies whether the Car exists using the message `exist_car` that has the stereotype `<<query>>` assigned to it. Following, the system modifies the state of the Car object using the message `modify_state` that has the stereotype `<<service / update>>` assigned to it.

Figure 28. SD_Disable
1.5.3. Enable: This scenario describes the enabling of a car.

In the Sequence Diagram shown in Figure 29, the User initiates this scenario with the message starts_enable.

After data introduction, the system verifies whether the Car exists using the message exist_car that has the stereotype <<query>> assigned to it. Following, the system modifies the state of the Car object using the message modify_state that has the stereotype <<service / update>> assigned to it.

![Figure 29. SD_Enable](image-url)
1.5.4. DeleteOperation: This scenario describes the elimination of an operation object that was performed on a car. When an operation object is created if the operation type is reparation the car should be disabled. If the referred operation object is deleted, this does not necessarily imply that the car should be enabled. This is specified in the Sequence Diagram FinalizeOperation shown in Figure 31.

In the Sequence Diagram shown in Figure 30, the User initiates this scenario with the message starts_delete_operation.

After data introduction, the system eliminates an Operation object with the message delete_operation that has the stereotype <<service / destroy>> assigned to it.

![Sequence Diagram](image_url)

Figure 30. SD_DeleteOperation
1.5.5. FinalizeOperation: This scenario describes the finalization of an operation. When the operation is finalized the car should be enabled for renting.

In the Sequence Diagram shown in Figure 31, the User initiates this scenario with the message starts_finalize_operation.

After data introduction, the system queries the Operation for information using the message get_operation_info that has the stereotype <<query / 1,1>> assigned to it. Following, the Operation is changed using the message finalize_operation that has the stereotype <<service / update>> assigned to it. In addition, the system calls the Use Case TR_Enable to enable the car.

![Figure 31. SD_FinalizeOperation](image-url)
2.1. CreateCustomer: This scenario describes the creation of a customer in the Car Rental system.

In the Sequence Diagram shown in Figure 32, the User initiates this scenario with the message *starts_create_customer*.

After data introduction, the system verifies whether the Customer exists using the message *exist_customer* that has the stereotype <<query>> assigned to it. If it does not exist, the system creates a new object from the Customer class using the message *create_customer* that has the stereotype <<service / new>> assigned to it.

Figure 32. SD_CreateCustomer
2.2. **DeleteCustomer**: This scenario describes the elimination of a customer in the Car Rental system.

In the Sequence Diagram shown in Figure 33, the *User* initiates this scenario with the message *starts_delete_customer*.

After data introduction, the system eliminates a *Customer* with the message *delete_customer* that has the stereotype `<service/destroy>` assigned to it. Note that this message is only executed if the control condition is satisfied.

![Figure 33. SD_DeleteCustomer](image)

Figure 33. SD_DeleteCustomer
2.3. **ModifyCustomer**: This scenario describes the change of a customer information the Car Rental system.

In the Sequence Diagram shown in Figure 34, the *User* initiates this scenario with the message `starts_modify_customer`.

After data introduction, the system queries the *Customer* for information using the message `get_customer_info` that has the stereotype `<<query>>` assigned to it. If desired, the modification of a *Customer* is done using the message `modify_customer` that has the stereotype `<<service / update>>` assigned to it.

![Figure 34. SD_ModifyCustomer](image)

Figure 34. SD_ModifyCustomer
3.0.1. Rent: This scenario describes the rental of a car in the Car Rental system. This is maybe the most important function of the system.

Renting a car implies that a Customer needs to be registered and a Car should be ready to be rented. This is done in a Contract that collects general information related to the rental (initial dates, prices, extras, etc.) and specific data about the customer and the selected Car.

In the Sequence Diagram shown in Figure 35, the User initiates this scenario with the message starts_rent.

After data introduction, the system creates a new object from the Contract class using the message create_contract that has the stereotype <<service / new>> assigned to it.

The new object needs to select its corresponding Customer and Car. This is done using the messages connect_customer and connect_car that have the stereotypes <<connect / 1,1>> assigned to them. The excl property of the connect stereotype of the Car class indicates that once a Car selected for renting, it cannot be selected for another rental until delivered. In addition, if the Customer desires to include extras like mobile phone, baby’s chair, etc. (represented by the condition control “Yes”), the Use Case TR_ExtrasAssignment should be included.

Finally, the number of contracts of the Customer is increased using the message modify_number_of_contracts that has the stereotype <<service / update>> assigned to it. The state of the Car object is also modified using the message modify_state that has the stereotype <<service / update>> assigned to it.
Figure 35. SD_Rent
3.0.2. ModifyContract: This scenario describes the change in the data of a contract (i.e., date of return, return location, etc.) in the Car Rental system. The customer asks to change the data of a contract (by phone or personally) and provides the corresponding contract number.

In the Sequence Diagram shown in Figure 36, the User initiates this scenario with the message `starts_modify_contract`.

After data introduction, the system queries the Contract for information using the message `get_contract_info` that has the stereotype `<query / 1,1>` assigned to it. If desired, the modification of a Contract is done using the message `modify_contract` that has the stereotype `<service / update>` assigned to it.

![Figure 36. SD_ModifyContract](image-url)
3.0.3. Return: This scenario describes the return of a car in the Car Rental system.

This process implies the payment of the total amount of the contract and the return of the corresponding extras (if any) besides the return of the car.

In the Sequence Diagram shown in Figure 37, the User initiates this scenario with the message starts_return.

After data introduction, the system verifies whether the Contract exists using the message exist_contract that has the stereotype <<query>> assigned to it.

The rate is obtained for each ExtraContract (that is stored in an ExtraType object) using the message get_extratype_info that has the stereotype <<query / 1,1>> assigned to it. Later, the stock is updated in the corresponding ExtraType object using the message modify_stock that has the stereotype <<service / update>>.

Then, the total amount of the set of extras (if any) is calculated for the Contract class using the get_total_extra with the stereotype <<query / 0,n / state>> assigned to it. The Rate is needed to calculate the subtotal of the Contract. This is done using the message get_rate that has the stereotype <<query /1,1 / state>> assigned to it.

Following, the Contract is closed by executing the message finalize_contract that has the stereotype <<service / update>> assigned to it, and the state of the Car and the Customer should be updated. The Car is updated using the message modify_state that has the stereotype <<service / update>> assigned to it, and the Customer is updated using the message modify_number_of_contracts that has the stereotype <<service / update>> assigned to it.

Finally, the total amount of the Contract is shown to the User using the message show_total that has the stereotype <<signal>> assigned to it.
Figure 37. SD_Return
3.1.1. CreateExtraType: This scenario describes the creation of an extra type in the Car Rental system. Each type of extra has information describing the characteristics of the extra and its availability (stock) at any given time. For instance, an extra can be a mobile phone including its data such as model, price per day, and stock.

In the Sequence Diagram shown in Figure 38, the Administrator initiates this scenario with the message \texttt{starts\_create\_extratype}.

After data introduction, the system verifies whether the \texttt{ExtraType} exists using the message \texttt{exist\_extratype} that has the stereotype \texttt{<<query>>} assigned to it. If it does not exist, the system creates a new object from the \texttt{ExtraType} class using the message \texttt{create\_extratype} that has the stereotype \texttt{<<service / new>>} assigned to it.

![Sequence Diagram](image-url)
3.1.2. DeleteExtraType: This scenario describes the elimination of an extra type in the Car Rental system.

In the Sequence Diagram shown in Figure 39, the Administrator initiates this scenario with the message starts_delete_extratype.

After data introduction, the system eliminates an ExtraType using the message delete_extratype that has the stereotype <<service / destroy>> assigned to it.

![Sequence Diagram of DeleteExtraType](image-url)
3.1.3. **ModifyExtraType**: This scenario describes the change in the data of an extra type (i.e., rental price per day, extra type description, etc.) in the Car Rental system.

In the Sequence Diagram shown in Figure 40, the *Administrator* initiates this scenario with the message *starts_modify_extratype*.

After data introduction, the system queries the *ExtraType* for information using the message *get_extratype_info* that has the stereotype `<query / 1,1>` assigned to it. If desired, the modification of an *ExtraType* is done using the message *modify_extratype* with the stereotype `<service / update>`.
3.1.4. ExtrasAssignment: This scenario describes the addition of an extra or a set of extras to a contract.

In the Sequence Diagram shown in Figure 41, the User initiates this scenario with the message *starts_extras_assignment*.

After data introduction, the system creates a new object from the *ExtraContract* class using the message *create_extracontract* that has the stereotype `<service / new>>` assigned to it.

The new object needs to select its corresponding *ExtraType* and *Contract*. This is done using of the messages *connect_extratype* and *connect_contract* that have the stereotype `<connect / 1,1>>` assigned to them. In addition, the stock of an *ExtraType* is modified using the message *modify_stock* that has the stereotype `<service / update>>` assigned to it. This is an iteration that repeats until all the desired extras (*ExtraContract*) are included in the *Contract*.

![Sequence Diagram](image)

**Figure 41. SD_ExtrasAssignment**
4.1. **CreateUser**: This scenario describes the creation of a user in the Car Rental system. This function is done by the Administrator that creates all the users of the system.

In the Sequence Diagram shown in Figure 42, the **Administrator** initiates this scenario with the message *starts_create_user*.

After data introduction, the system verifies whether a *User* exists using the message *exist_user* that has the stereotype `<query>` assigned to it. If it does not exist, the system creates a new object from the *User* class using the message *create_user* that has the stereotype `<service / new>` assigned to it.
4.2. **DeleteUser**: This scenario describes the elimination of a user in the Car Rental system.

In the Sequence Diagram shown in Figure 43, the *Administrator* initiates this scenario with the message *starts_delete_user*.

After data introduction, the system eliminates a *User* using the message *delete_user* that has the stereotype *<<service / destroy>>* assigned to it.

![Sequence Diagram](image)

**Figure 43. SD_DeleteUser**
4.3. ModifyUser: This scenario describes a change in the data of a user in the Car Rental system.

In the Sequence Diagram shown in Figure 44, the Administrator initiates this scenario with the message starts_modify_user.

After data introduction, the system queries the User for information using the message get_user_info that has the stereotype $<query \ / \ 1,1>$ assigned to it. If desired, the modification of a User object is done using the message modify_user that has the stereotype $<service \ / \ update>$ assigned to it.
4.4. **Promote**: This scenario describes the promotion of a user to an administrator in the Car Rental system.

In the Sequence Diagram shown in Figure 45, the *Administrator* initiates this scenario with the message *starts_promote*.

After data introduction, the system promotes a *User* object using the message *promote* that has the stereotype \(<\text{service / update}>\) assigned to it.

![Sequence Diagram SD_Promote](image)

**Figure 45. SD_Promote**
4.5. **Demote**: This scenario describes the demotion of an administrator to a user in the Car Rental system. This function is done by another administrator.

In the Sequence Diagram shown in Figure 46, the **Administrator** initiates this scenario with the message *starts_demote*.

After data introduction, the system demotes a **User** object using the message *promote* that has the stereotype *<<service / update>>* assigned to it.

![Figure 46. SD_Demote](image-url)
6. Conceptual Model

In order to obtain the Conceptual Model the Requirements Model should be analyzed. The first step is to apply the Requirements Analysis Process (RAP) that includes the object interaction specification for each scenario (Sequence Diagram specifications). This is done in Chapter 5 of this appendix. The other important step for building the Conceptual Model is to obtain the Traceability Matrix. This matrix can be shown in various formats. In this appendix, we show two tables: the classical Traceability Table (CRUD+A table) in Table 2, and the Actors Table in Table 3.

Table 2. Traceability Table (CRUD+A Table) for the Car Rental system

<table>
<thead>
<tr>
<th>Car</th>
<th>Rate</th>
<th>Sale</th>
<th>Garage</th>
<th>Insurance</th>
<th>Insurance Company</th>
<th>Operation</th>
<th>Customer</th>
<th>Contract</th>
<th>Extra Type</th>
<th>Extra Contract</th>
<th>User</th>
<th>Administrator</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purchase</td>
<td>RC</td>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sell</td>
<td>A</td>
<td>RC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Deliver</td>
<td>RD</td>
<td>U</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Create Rate</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>RC</td>
</tr>
<tr>
<td>Delete Rate</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>D</td>
</tr>
<tr>
<td>Modify Rate</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>RU</td>
</tr>
<tr>
<td>Create Garage</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>RC</td>
</tr>
<tr>
<td>Delete Garage</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>D</td>
</tr>
<tr>
<td>Modify Garage</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>RU</td>
</tr>
<tr>
<td>Create Insurance Company</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>RA</td>
</tr>
<tr>
<td>Delete Insurance Company</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>RD</td>
</tr>
<tr>
<td>Modify Insurance Company</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>RU</td>
</tr>
<tr>
<td>Create Insurance Company</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>RC</td>
</tr>
<tr>
<td>Delete Insurance Company</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>D</td>
</tr>
<tr>
<td>Modify Insurance Company</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>RU</td>
</tr>
<tr>
<td>Create Operation</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>RA</td>
</tr>
<tr>
<td>Disable</td>
<td>RU</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Enable</td>
<td>RU</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Delete Operation</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>RU</td>
</tr>
<tr>
<td>Finalize Operation</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>RU</td>
</tr>
<tr>
<td>Create Customer</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>RC</td>
</tr>
<tr>
<td>Delete Customer</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>D</td>
</tr>
<tr>
<td>Modify</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>RU</td>
</tr>
<tr>
<td>Car Rate Sale Garage Insurance Insurance Company Operation Customer Contract Extra Contract User Administrator</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-----------------------------------------------</td>
<td>------------</td>
<td>------------</td>
<td>-------------</td>
<td>-----------------</td>
<td>-----------------</td>
<td>-----------------</td>
<td>-----------------</td>
<td>-----------------</td>
<td>----------------</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Customer Rent AU AU C Modify Contract RU</td>
<td>Return U R U RU RU R Create Extra Type RC</td>
<td>Delete Extra Type D Modify Extra Type RU Extras Assignement A AU C Create User Administrator</td>
<td>Delete User D Modify User RU Promote U Demote U</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The Actors Table shows all the actors of the system, each one of them in a column. Entries in this table show the initiator actors related to services obtained from Use Cases.

Table 3. Traceability Matrix (Actors table)

<table>
<thead>
<tr>
<th></th>
<th>Administrator</th>
<th>User</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purchase</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Sell</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Deliver</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Create Rate</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Delete Rate</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Modify Rate</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Create Garage</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Delete Garage</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Modify Garage</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Create Insurance</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Delete Insurance</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Modify Insurance</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Create Insurance Company</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Delete Insurance Company</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Modify Insurance Company</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Create Operation</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Disable</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Enable</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Delete Operation</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Finalize Operation</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Create Customer</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Delete Customer</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Administrator</td>
<td>User</td>
</tr>
<tr>
<td>---------------------</td>
<td>---------------</td>
<td>------</td>
</tr>
<tr>
<td>Modify Customer</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Rent</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Modify Contract</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Return</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Create Extra Type</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Delete Extra Type</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Modify Extra Type</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Extras Assignment</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Create User</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Delete User</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Modify User</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Promote</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Demote</td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>
6.1. **Generated Class Diagram**

Following, in Figure 47, the automatically generated Class Diagram is shown. This is obtained by applying the Traceability Rules Catalog presented in Chapter 6.
6.2. **Refined Class Diagram**

After analyzing the class diagram obtained from the Requirements Analysis Process, the analyst should decide the desired changes to perform on it. Remember that not all kinds of changes are allowed without impact on the traceability structure established. All those elements with *weak traceability* can be modified, deleted, or added to the referred class diagram. Figure 48 shows the new refined class diagram.

![Refined Class Diagram](image)

This diagram is a refinement of the generated Class Diagram presented in Figure 47. The main structural change observed is the specialization relationship of the *Customer* class. In addition, two new classes were added to this diagram to represent particular properties of different kinds of customers. These are *Direct Customer* and *Agency Customer* that are specializations of the class *Customer*. 
List of Figures

Figure 1. Use Case View ................................................................. 240
Figure 2. Car Management ............................................................. 241
Figure 3. Rate Management ............................................................ 241
Figure 4. Garage Management ....................................................... 242
Figure 5. Insurance Management .................................................... 242
Figure 6. Insurance Company Management ..................................... 242
Figure 7. Operations Management .................................................. 243
Figure 8. Customer Management .................................................... 243
Figure 9. Contract Management ...................................................... 244
Figure 10. Extras Management ....................................................... 245
Figure 11. Users Management ......................................................... 245
Figure 12. SD_Purchase ................................................................. 246
Figure 13. SD_Sell .......................................................................... 247
Figure 14. SD_Deliver ................................................................. 248
Figure 15. SD_CreateRate ............................................................. 249
Figure 16. SD_DeleteRate ............................................................. 250
Figure 17. SD_ModifyRate ........................................................... 251
Figure 18. SD_CreateGarage ......................................................... 252
Figure 19. DS_DeleteGarage ........................................................ 253
Figure 20. SD_ModifyGarage ......................................................... 254
Figure 21. SD_CreateInsurance ...................................................... 255
Figure 22. SD_DeleteInsurance ....................................................... 256
Figure 23. SD_ModifyInsurance ..................................................... 257