Requirements Engineering-Based Conceptual Modeling

[¤]E. Insfrán, [¤]O. Pastor, [¥]R. Wieringa

Abstract

The software production process involves a set of phases where a clear relationship and smooth transitions between them should be introduced. In this paper, a requirements engineering-based conceptual modeling approach is introduced as a way to improve the quality of the software production process. The aim of this approach is to provide a set of techniques and methods to capture software requirements and to provide a way to move from requirements to a conceptual schema in a traceable way. The approach combines a framework for requirements engineering (TRADE) and a graphical object-oriented method for conceptual modeling and code generation (OO-Method). The intended improvement of the software production process is accomplished by providing a precise methodological guidance to go from the user requirements (represented through the use of the appropriate TRADE techniques) to the conceptual schema that properly represents them (according to the conceptual constructs provided by the OO-Method). Additionally, as the OO-Method provides full model-based code generation features, this combination minimizes the time dedicated to obtaining the final software product.

Keywords: Requirements Engineering, Conceptual Modeling, Use Cases, traceability, OO methods.

1 Introduction

In this paper, we introduce a *Requirements Engineering-Based Conceptual Modeling* approach as a way to improve the quality of the software production process. The approach introduces techniques to clearly specify functional requirements and a process to systematically decompose these high-level software requirements into a more detailed specification that constitutes the conceptual schema of the desired system. By improvement of the quality of the software production process we mean:

- [2] providing predictability, in the sense of having a conceptual schema that is a precise, well-defined representation of the user requirements

[¤]Department of Information Systems and Computation. Valencia University of Technology. Valencia, Spain. e-mail: [einsfranjopastor]@dsic.upv.es

[¥]Department of Computer Science. University of Twente. Enschede, the Netherlands. e-mail: roelw@cs.utwente.nl

- improving productivity, because the source conceptual schema that guides the generation of the final software product is linked to the user requirements in a precise way. Having such a precise link allowing to trace changes in the user requirements in the conceptual schema and, consequently, in the final software product. This solves a classical problem in requirements engineering practice.

In order to accomplish these ideas we use the Use Case metaphor, which is widely extended in requirements engineering environments, but oriented to collect a specific kind of information required to generate the corresponding Conceptual Schema. In particular, our approach defines a *Requirements Model,* which captures both functional and usage aspects in a comprehensive manner. This is organized through the use of three complementary techniques: Mission Statement, Function Refinement Tree and Use Case Diagrams, which are explained in more detail in the following sections.

When dealing with complex systems, it does not seem feasible to go directly from a requirements description provided by the customer to a formal specification in one step [Reg95]. For this reason, we also introduce a *Requirements Analysis Process* (RAP) to translate this structured Requirements Model into a Conceptual Schema specification in a traceable way.

In short, the original contribution of this paper is twofold:

- the introduction of a *Requirements Model* (RM) which has been created to collect all the functional information needed to face the problem of specifying a Conceptual Schema

- a *Requirements Analysis Process* (RAP) which provides methodological guidance in the process of building such a Conceptual Schema based on the functional requirements identified

To deal with the Requirements Model, a well-founded set of techniques for requirements specification is needed. To introduce a precise RAP, the target Conceptual Modeling constructs must be clearly defined; in consequence, a formal conceptual modeling approach should be used. This is why our approach combines some specific selected techniques from the **T**echniques for **R**equirements and **A**rchitecture **DE**sign (TRADE [Wie99c]) and a graphical **O**bject-**O**riented **Method** for conceptual modeling and automated code generation (OO-Method [Pas97c]). The blend between the TRADE and OO-Method results in a powerful environment for modeling software requirements and in a strategy for translating these requirements into a graphical conceptual schema. In the OO-Method, this conceptual schema is backed-up by a formal specification language (OASIS [Pas95a]), and the final implementation is obtained in an automated way by applying the corresponding mappings between conceptual model constructs and their representation in a particular software development environment.

In our point of view, our approach overcomes two common weaknesses currently existing in software development methods. First, software engineers do not know whether a conceptual model represents the user's requirements. These requirements are usually represented in a non-structured or semi-structured way with fuzzy traceability in the conceptual model constructs. Second, when the development step is reached, the value of conceptual modeling efforts is unclear, mainly because it is not possible

to produce an accurate code that is functionally equivalent to the conceptual schema built earlier. In our approach, a Requirements Analysis Process assures that each element from the Requirements Model (problem space) will have a representation in the Conceptual Model and, transitively, this will have its representation in the target development language (solution space) in the context of a Model-Based Code Generation (MBCG) [Bel98].

This paper focuses only on the Requirements Model and its representation in a Conceptual Schema by means of the Requirements Analysis Process (RAP). A detailed work on translating conceptual models to a specific implementation can be found in [Pas97], [Pas98], [Pas01].

The paper is organized as follows: after the introduction, section 2 presents a short description of the main features of the TRADE and the OO-Method and how we combine the best properties of each one. Section 3 presents the requirements engineering-based conceptual modeling approach, where the building of the Conceptual Schema from the Requirements Model is explained. The last section gives the conclusions and outlines further work.

2 TRADE and OO-Method

The TRADE (**T**echniques for **R**equirements and **A**rchitecture **DE**sign) [Wie99c] is a set of techniques and heuristics which is based upon an analysis of structured and object-oriented specification methods [Wie98]. These techniques have been backed up by a formal semantics (when possible and necessary) and they are defined so that they fit together in a simple and well-defined way. The conceptual framework of TRADE distinguishes *external system interactions* from *internal components* as a structural mechanism for classifying system properties. External interactions are external functions from which we can specify their behavior and communication. Some techniques used in TRADE for specifying external interactions and their properties are: Mission Statement, Function Refinement Tree, Context Diagrams, Use Case Diagrams and Scenario Diagrams.

The OO-Method [Pas97c], [Pas01] is an object-oriented method that provides a set of well-defined and complementary graphical techniques to build a *Conceptual Schema* of the system. These graphical UML-based techniques are backed up by a formal object-oriented specification language (OASIS [Pas95a]) in such a way that every element in the graphical techniques corresponds (one-to-one) to a section in the formal specification language. In this way, the formal specification acts as a high-level system repository for all the static and dynamic properties of the system (structure, behavior and functionality). Furthermore, an application that is functionally equivalent to the OASIS specification can also be generated in an automated way. This is achieved by defining an *Execution Model,* which gives the pattern for obtaining a specific implementation in an imperative software development environment. A CASE Tool[1] provides an operational environment that supports all the methodological aspects of the OO-Method which has been developed in the context of a R&D project carried

[1]More information about the OO-Method and its CASE Tool can be found at http://www.upv.es/oomethod.

out jointly by the Valencia University of Technology, CARE Technologies S.A. and Consoft S.A. in Spain.

The work presented in this paper is based on the union of these two approaches. It is important to justify the good properties obtained by such a combination. It is clear that TRADE provides an interesting set of techniques for dealing with Requirements Engineering. Its main weak point is the absence of a well-defined method to guide the process of requirements specification for design and implementation. On the other hand, the OO-Method gives a rigorous software production method to obtain a final software product starting from a given Conceptual Schema. However, the problem of how to build this source Conceptual Schema is not dealt with in this method. Following the arguments briefly introduced in the previous section, the combination of the TRADE and the OO-Method provides a solution to their corresponding pitfalls by

- defining a specific Requirements Model. A subset of the TRADE techniques with a determined semantic and relationships between them is precisely determined.
- linking this Requirements Model with the OO-Method Conceptual Schema. The absence of a process for obtaining such a Conceptual Schema in the OO-Method is overcome.

In the reminder of the paper, we will focus on giving a detailed description of what we call a *Requirements Engineering-Based Conceptual Modeling* approach by developing these features.

3 A Requirements Engineering-based Conceptual Modeling approach

In this section, we present a software requirements engineering approach for applications development. The use of the term *engineering* implies that systematic and repeatable techniques are used in order to deal with the software requirements and to ensure that these requirements are reified in a conceptual schema specification. Further steps (specified in the Execution Model) will produce accurate code in a programming language that is functionally equivalent to this conceptual model specification. A general view of the approach is shown in Figure 1. This figure shows the whole software production method that goes from requirements to a final software product in a precise way. As the translation of an OO-Method Conceptual Schema into its corresponding software product is a characteristic of the OO-Method, this paper won't deal with it. This work will focus in the definition of the Requirements Model and in the guided translation from it into an OO-Method Conceptual Schema. This is the main paper contribution as stated above.

3.1 Requirements Modeling phase

The purpose of the Requirements Model is to accurately and precisely capture what the customer wants to have built. Furthermore, the purpose is to model requirements

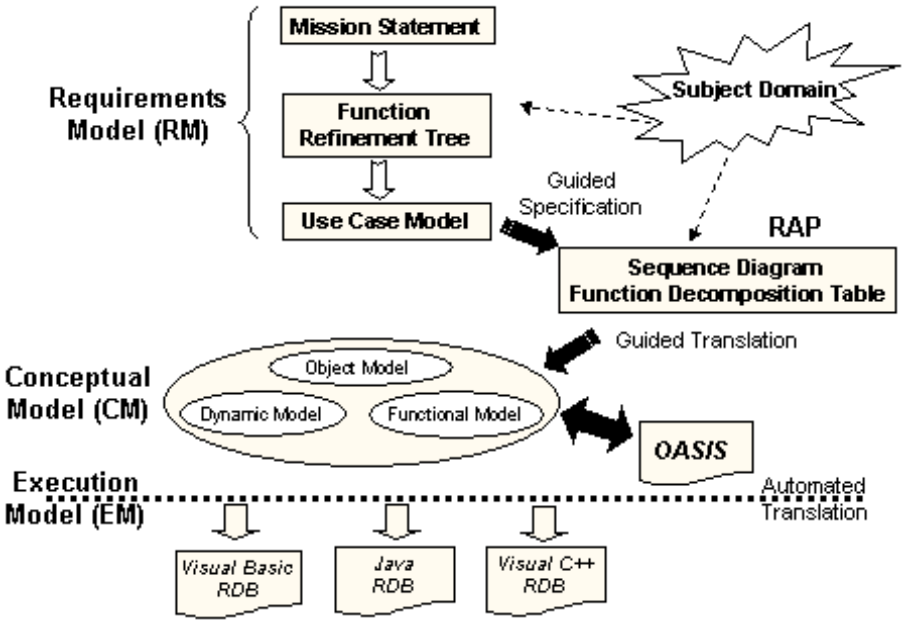

Figure 1: A general view of the fully integrated approach. It shows the techniques used in the Requirements Model together with the Requirements Analysis Process (RAP).

in such a way that people without high-level training in the notation can understand and review them. The notation is nevertheless precise enough so that it can be the basis for the conceptual modeling phase. The Objectory method [Jac92] introduces the Use Case Driven Analysis (UCDA) where the basic concepts are *actors* and *Use Cases*. UCDA helps to deal with the complexity of the requirements analysis process. Since the idea of UCDA is simple, and the Use Case descriptions are based on natural concepts that can be found in the problem space, the customers and the end users can actively participate in requirements modeling.

Two disadvantages of using UCDA are the following:

- the difficulty in finding the correct abstraction level to specify the Use Case (what a Use Case really is)
- finding a process to analyze and translate the Use Case specification into a conceptual model. This process is the called *synthesis* in Usage-Oriented Requirements Engineering (UORE) in [Reg95].

These two problems often make it very difficult to put Requirements Engineering techniques into practice. If we want to provide a sound Requirements Engineering framework, these problems need to be overcome. As *reinventing the wheel* is never a good strategy, we will base our proposal on the basic UCDA concepts. However we

are going to refine them to clarify what must be specified. Specifically, the techniques that are used in our Requirements Model include:

- *Mission Statement*: describes the purpose of the system in one or two sentences. It also describes the major responsibilities as well as a list of things the system is *not* to do.
- *Function Refinement Tree*: deals with external interaction partitioning according to the different business areas or business objectives.
- *Use Case model:* includes the *Use Case Specification* to specify the composition of external interactions and the *Use Case Diagram* to show communication between the environment (actors) and the system.

The use of the Mission Statement and the Function Refinement Tree together with the Use Case model is the key to finding a good abstraction level for Use Cases answering the question of what a Use Case really is which solves the first disadvantage of UCDA. The second disadvantage of UCDA is solved by the Requirements Analysis Process that we introduce in section 3.3.

The three components of the Requirements Model are described in detail below.

3.1.1 Mission Statement and Function Refinement Tree

External interactions can always be partitioned into functions. It is very useful to organize these 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 an area of business where the system is under development. It is not a trivial task to distinguish between intermediate and leaf nodes for the Function Refinement Tree. 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 [Wie98].

This way, the Function Refinement Tree[2] can be used to represent a hierarchical decomposition of the business functions of a system which is independent 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. However, it gives the entry point for building the Use Case Model instead of starting from scratch and avoids the potential problem of mixing the abstraction level of Use Cases.

An example of a Function Refinement Tree for a typical point-of-sale terminal (POST) system for a store is shown in Figure 2. In this case study, we deal with three business areas: Sales, Items to sell and user management. In the *Sales* area we have two internal interactions: Sale and Return items. In the *Items* area, we specify three internal interactions necessary to manage the items in the store. The third business area, *Management*, maintains information related to the two different kinds of users of the system: cashiers and administrators. Even if this example could be considered a simple one, we have selected it on purpose, first due to the obvious space limitations, and second because it is a well-known case that gives us the opportunity to clearly

[2]Function Refinement Trees have been introduced in Information Engineering [Mar89].

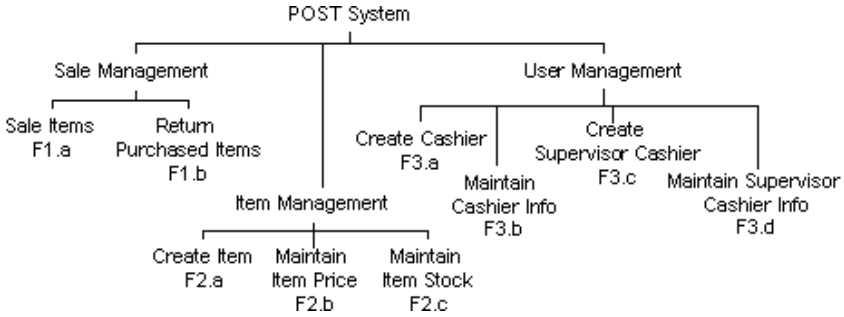

Figure 2: Function Refinement Tree for a Point-Of-Sale Terminal (POST)

explain the relevant features of our approach. An extended, complete point-of-sale terminal Case Study with more complex functionality can be found in [Ins99].

3.1.2 Use Cases

As the well-known Use Case approach is the basis for our Requirements Model, we are going to give a brief description of its main features in order to define the vocabulary used. Use Cases were introduced in OOSE [Jac92] 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 actor. This interaction need not be atomic, it is usually decomposed into *steps* indicated in the Use Case specification. An *actor* is a specific role played by a system user, and it represents a category of users that demonstrate similar behavior when using the system. By users we mean both human beings and other external systems or devices communicating with the system. An actor is regarded as a class, and users as instances of this class. One user may appear as several instances of different actors depending on the context. A *Use Case* is a system usage scenario involving one or more actors. The descriptions of actors and Use Cases form the Use Case Model (UCM).

These concepts are widely accepted and used in Requirements Engineering practice. What is not often so clear is how to properly use Use Cases from a methodological point of view and within a precise software production process. Our approach provides such a precise guideline: as we are combining Use Case specifications with the Function Refinement Tree, each one of the Use Cases will correspond to a *leaf node* of the Function Refinement Tree. This statement provides a solid mechanism to structure the Use Case specification, whose structure is presented below.

Use Case specification structure The purpose of Use Case specification is to describe the flow of events in detail, including how the Use Case starts, ends, modifies the system and interacts with actors. The different sections used to introduce the needed information are shown in Figure 3. We will use our POST example to explain the

contents of the different specification components in detail.

Following our twofold objective of introducing a rigorous Requirements Model, but using standard notations to avoid ”reinventing the wheel”, a conventional three-section structure for the Use Case specification is used. Our method contribution consists of introducing a new structure to deal with the Use Case step specification in a precise way. What we accomplish with this is that it can be understood by customers and at the same time be accurate enough to capture the purpose of the Use Case in terms of its compound steps (it should be of such a size that it can be described as a single thread of activities). In consequence, the top section is a *summary* of what the Use Case is about. The middle 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 and errors that can occur are described in a bottom section called *alternative* section.

An overview of this enriched structure is the following:

1. The top section (*Use Case summary*) of the Use Case specification is summary information. This gives details about:

   (a) the *Use Case name*

   (b) participating *actors* (external agents) indicating who initiates the Use Case

   (c) *cross-reference* with a leaf node of the Function Refinement Tree. For the purpose of traceability this is an important link between these techniques and constitutes a relevant feature of the approach.

   (d) an *overview* of the Use Case purpose

   These four pieces of information can be clearly identified in our example in Figure 3. For documentation purposes, specific information related to Non-Functional Requirements can also be specified in this section although currently we don’t deal with requirements of this kind.

2. The middle section (*basic course*) contains the steps that occur during the Use Case. This middle section is different from other approaches like [Jac92] that use only a narrative flow description or [Wir93], [Lar98] and [Con95] that use a two-column structure to separate *Actor action* and *System response* ignoring or mixing steps that describe the context where the Use Case happens. This specification should cover all the steps of the Use Case from the triggering event through to the accomplishment of the goal of the Use Case. They are numbered paragraphs and are usually done in a *conversational* style between actors and the system [Lar98].

   The middle and bottom sections have three columns to indicate the corresponding nature of the step:

   (a) *General*: these steps represent general activities or things to do in the problem domain that are outside of the scope of the software requirements model. They are important because they give software engineers the context in which the Use Case occurs. For example, in our case study in step

9 the Cashier tells the Customer the total amount of sale so that he/she can make the corresponding payment (step 10). This step is not supposed to be implemented by the system, but it helps the analyst and users to better understand the behavior of this Use Case.

(b) *Actor/System communication*: these steps represent specific actions performed by actors in order to interact with the system. They are related to user interface requirements.

(c) *System response*: these steps represent specific reactions (changes of state) in the system because of the occurrence of the Use Case. This is the most important part of the Use Case specification because it is the basis for building the conceptual schema as is shown in 3.3.

3. The bottom section (*alternative*) describes important alternatives or exceptions that may arise with respect to the basic course of events described in the middle section. They usually refer to a specific number in the middle section where the exception may arise.

The Requirements Model is obtained once this specification is completed. It is important to answer the question of when a Use Case specification can be considered complete. The key to building a good Use Case is to remember that it serves two purposes: it will be used as the basis for the conceptual schema and it will be reviewed by both clients and analysts. Use Case descriptions are finished when they are deemed understable, correct (i.e., they capture the right requirements), complete (e.g., they describe all possible paths), and consistent.

3.2 Conceptual Modeling phase

The purpose of the conceptual modeling phase is to represent user requirements in a specification of *what* the system does as if there were a perfect implementation technology available [McM84]. This is not a model of *how* an implementation works but of what any implementation must accomplish. Here the major activity is to find out which classes the software will need in order to satisfy the requirements based on Use Cases identified in the requirements model.

The OO-Method adopts a well-known strategy of dividing the conceptual modeling process into three complementary views using an UML-based notation:

- *Object Model*: classes are identified and their static properties are specified (attributes, services, integrity constraints, and relationships: aggregation, inheritance and agent[3] relationships).

- *Dynamic Model*: valid object life cycles and object interactions are specified. Two kinds of diagrams are used:

[3] Agent relationships in OO-Method specify which objects are allowed to activate other object services (client/server relationship).

Use Case:	Sale items	
Actors:	Customer (init), Cashier	
Cross reference:	F1.a	
Description	A Customer arrives at a checkout with items to purchase. The Cashier records the purchase items and collects payment	
Steps Specification		
General	Actor/System communication	System response
1. This use case begins when a customer arrives at a POST checkout with items to purchase		
	2. The cashier starts a new sale session	
		3. The cashier creates a new sale
		4. The cashier records the identifier of each item
		5. The system determines the item price and description and adds information to the current sales transaction
	6. On completion of item entry, the Cashier indicates to the POST that item entry is complete	
		7. The system calculates the total and updates the stock
	8. The system shows the total	
9. The cashier tells the customer the total		
10. The customer gives a cash payment (possibly greather than the sale total)		
...	...	...
Alternative Steps Specification		
	4. If there is more than one of the same item, the Cashier can enter the quantity as well. The subtotal is shown	
	4. Invalid identifier entered. Indicate error	
		7. If the item's stock gets below a predefined minimum place a reposition order
10. Customer didn't have enough money. Cancel transaction		

Figure 3: An Use Case specification

- The *State Transition Diagram* describes correct behavior by establishing valid object life cycles for every class. By valid life, we mean a right sequence of states that characterizes the correct behavior of the objects.
- The *Collaboration Diagram* describes object interactions. In this diagram, 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.

- *Functional Model*: it is a textual model used to capture the semantic that is attached to any change of an object state as a consequence of a service occurrence. We specify declaratively how every service changes the object state depending on the service arguments involved (if any) and the object's current state. It allows us to generate a complete OASIS specification in an automated way. More detailed information can be found in [Pas95a].

The most important difference among the OO-Method models and other "conventional" methodologies is that when software engineers specify the conceptual schema, what they are really doing is capturing a formal specification "on the fly" according to the OASIS formal specification language. This feature allows us to introduce an accuracy level that is often not present in other approaches. In the following section, we present the Requirements Analysis Process as a way to identify and represent the involved classes and structure of the desired Conceptual Schema.

3.3 Requirements Analysis Process (RAP)

The behavior in a system should be exactly that which is required to provide the Use Case to the users of the system. In order to evaluate whether that Use Case has been provided, we should show *what* functionality was allocated to *which* class or classes for each Use Case. The technique consists of walking through the Use Case specification. For each step described or implied in the Use Case specification, a responsibility (or a set of responsibilities) should be identified and the designer has to allocate it to a class (a previously identified class or a new one). This is a complex and unavoidable task. However, the contribution of our approach is that the designer has to focus its attention on only one Use Case and in particular on one step of the involved Use Case each time.

There are always two components involved in each responsibility: one to recognize the need and to invoke the responsibility (the client), and another to carry out the responsibility (the server).

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

- Identify the responsibility. This is a decomposition of the larger responsibility described in each Use Case specification step. The granularity of the decomposition (how big each of the responsibilities is) is a key element in 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 class component should know to ask for the behavior (the actor) and which component should provide the behavior (the server). The quality of these decisions determines the richness and flexibility of the conceptual schema.

In the following subsections, we introduce the notation and technique used to accomplish these tasks.

3.3.1 Notation and Technique

To deal with the activity of identifying responsibilities within the Use Case specification and allocating them into class components, we use Sequence Diagrams. These diagrams show how participating classes realize the corresponding Use Case through their interaction.

The Sequence Diagram is usually used to specify detailed behavior and low-level specifications. In our approach this diagram is used at a very high-level as a tool for *identifying* responsibilities and *allocating* them into class components. For this purpose, we use a subset of the corresponding standard UML [OMG01] diagram. We do not provide elements for showing branches, iterations or states. Iterations and branches can be described in the text of the Use Case specification, but they cannot, and should not, be shown in the diagram itself. One reason for this restriction is that the logic and state are inside the class specification. We have to specify these properties inside the conceptual schema only after the responsibility identification and allocation decisions have been made.

There is at least one Sequence Diagram per Use Case, one for the *basic course* of action, and one for each *alternative course* of action[4] (if any).

To build the Sequence Diagram for a Use Case, we analyze each Use Case at two levels:

- *Use Case diagram level*: actors that communicate with the Use Case
- *Use Case specification 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 has at least one actor, which is the Use Case *initiator* and other collaborator actors. Actors are potentially objects to be taken into account for the Sequence Diagram (this could be an actor for services offered by classes, agent relationship[5] in the OO-Method). However, this is not always true. Often Use Case actors represent entities (people, organizations, systems,...) that exist in the real world (entities in the subject-domain) but not in the conceptual schema. If we go back to our POST example, for the Use Case *Sale item* in Figure 3, a Customer initiates the Use

[4] It is necessary to create a Sequence Diagram for each alternative course of action when the corresponding action requires a response from the system.

[5] In the OO-Method, we model objects with two perspectives: *client* or *actor*, when they request services to the society; and *server*, when they offer services to the society.

Case. However, inside the conceptual schema[6] the Cashier is responsible for initiating this functionality. The Cashier is selected as a participating object for the quoted sequence diagram. Other actors (Customer and Accounts Receivable/ToPay) participate anonymously.

At the *Use Case specification level*, the main task is to obtain the set of responsibilities implied by the corresponding Use Case steps. Essentially, 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:

- **<<signal>>** message between an actor and the system
- **<<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.
- **<<query>>** message to query the state of some object
- **<<select>>** message to select instances of the server 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*)

An example of an analysis of a Use Case at the diagram and specification level is shown in Figure 3. It is very important to note that this is a complex, and very important task in the Requirements Analysis Process where the traceability between the Requirements Model and the Conceptual Schema is established.

Each Sequence Diagram shows the classes that participate in the Use Case. Each participating class gets a named time-line (vertical dotted line). The interaction between participating classes are shown as arrows between the time lines. Each message must be shown as a message name with all its arguments. The message must also be labeled with the corresponding stereotype as described above (query, service, etc.).

Following, we present an example showing some of the steps of specification of the Use Case *Sale items* from Figure 3. The result is the Sequence Diagram from Figure 4:

- *Step 2. The cashier starts a sale*. This message represents the starting communication between the *Cashier* and the *System*. The following message *introduce_sale_data* is induced by the same step.
- *Step 3. The cashier creates a new sale*. This message implies the responsibility of creating a new sale and registering its information. We named this responsibility *create_sale* and allocated it to a new class named *Sale*.
- *Step 4. The cashier records the identifier of each item*. This action implies the responsibility of registering information about the items to sell (at this level of

[6] In a typical case, every Customer comes into the store, participates anonymously, and leaves again. Another problem domain could maintain information on Customers for better service. For example, the store could issue cards with machine-readable codes. The cashier could run the card through a scanner and start a Sales operation.

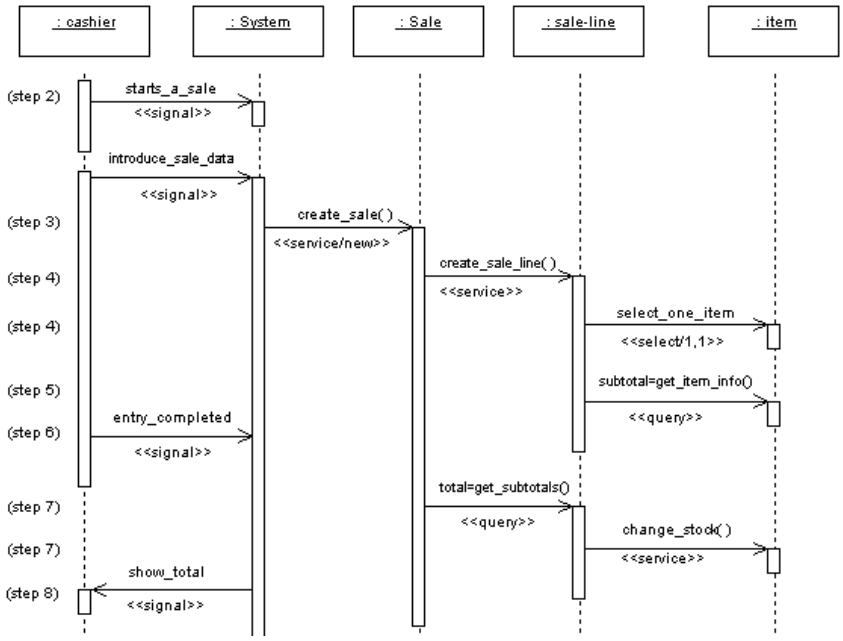

Figure 4: Sequence Diagram for the Use Case *Sale items*.

specification the details of the iteration are irrelevant as we are only discovering classes and services and not describing the complete behavior). We named this responsibility *create_sale_item* and allocated it to a new class component named *Sale-line*.

The next message *select_one_item* (with the stereotype <<*select*>>) is also implied by Step 4 because the designer determines that a complete *Sale-line* must have one and only one *Item* (product) assigned to it.

- *Step 5. The system determines the item price and description and adds information to the current sales transaction*. This action implies the responsibility of querying an object that stores the item price and description. We named this responsibility *get_item_info* with two output arguments: *price* and *description* specified as properties of the service. We allocated this query to a new class component named *Item*.

- *Step 7. The system calculates the total and updates the stock.* This action implies the responsibility of calculating the total amount of the sales transaction. These calculations should later be a derived attribute in terms of the sold items and the corresponding tax calculations.

  Another responsibility is to update the item stock. We named this responsibility *change_stock* and allocated it to the previously detected class component *Item*.

The object that recognizes the need and invokes this responsibility (the client) is an instance of the class *Sale-line*.

In this way, the process continues until all the steps are specified in their corresponding Sequence Diagram. This process should be evaluated together with the users during and after its construction.

3.3.2 Validation, verification and traceability

The purpose of the RAP is to obtain a consistent first version of the Conceptual Schema. In this context, validation, verification and traceability are key issues.

The traceability model we use is the *structural* and *cross-reference* based model according to Gotel's work [Got95]. It is *structural* because there are structural relationships between the elements in the conceptual schema and the different elements in the RAP and CM (e.g. classes in the conceptual schema are classes used in sequence diagrams). It is also *cross-reference* based because there are tags included between elements at different abstraction levels (e.g. Use Cases have tags pointing to leaves in the Function Refinement Tree).

The *validation* 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 and a Functional Model as was explained in 3.2. 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 behavior in a class 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 *Function Decomposition Table*, a technique taken from Structured Analysis and Information Engineering. 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. Instances of classes (objects) must be created (C - stereotype *service/new*), read (R - stereotype *query*), updated (U - stereotype *service/update*) and deleted (D - stereotype *service/destroy*). We show this information in Figure 5 (this is also called CRUD 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 probably are not useful and could be omitted. Those objects that are not updated apparently represent an unchanging part of the system.

Other important information obtained from the Sequence Diagram is provided by

Use Cases / Classes	Sale item	Return purchased item	...
Cashier			
Sale	C		
Sale Line	C		
Item	RU	RU	
Order	C		
Return		C	
Return Line		C	
...			

Figure 5: CRUD Table showing the flowdown of use cases into classes

messages with the stereotype *select*. These messages describe the needed for one object to be related to another. This implies an aggregation relationship in the OO-Method. In Figure 4, step 4 is an example with the message: *select_one_item* <<*select/1,1*>>. An algorithm can take all the messages of the type *select* between two classes to establish this kind of relationships and propose their cardinalities by analyzing their properties. An example is shown in Figure 6.

An important evaluation activity for the Object Model consists of checking the responsibility allocation decision on a per-class basis. The coupling, cohesion, and modularity of the system are determined entirely by how the responsibility identification and allocation is made in the Sequence Diagrams. The point is that unless this activity is made consciously with an eye toward modularity and consistency, maintainability and extensibility of the resulting system will be less than it could be. A Conceptual Schema associated to the POST System is shown in Figure 7.

In the **Dynamic Model** of the Conceptual Schema, we specify valid lives for objects (State Transition Diagrams -one per class) and interaction between objects (Triggers[7] and Global Transactions). *Triggers* are labeled in the message in the Sequence Diagram and attached to a class. *Global Transactions* are groups of services involving different objects. Every Use Case is a candidate to be a Global Transaction and if so, the analyst identifies it and gives it a name (by default the same name of the Use Case). We are now working on generating automatically *State Transitions Diagrams* from the Sequence Diagrams for the identified classes of the system. Sequence Diagrams describe a partial behavior of the system and State Transition Diagrams describe the complete behavior of objects in terms of valid sequences of services. The technique extracts the partial behavior from all the Sequence Diagrams where the intended class participates

[7]Triggers are services that automatically are activated when a condition is satisfied.

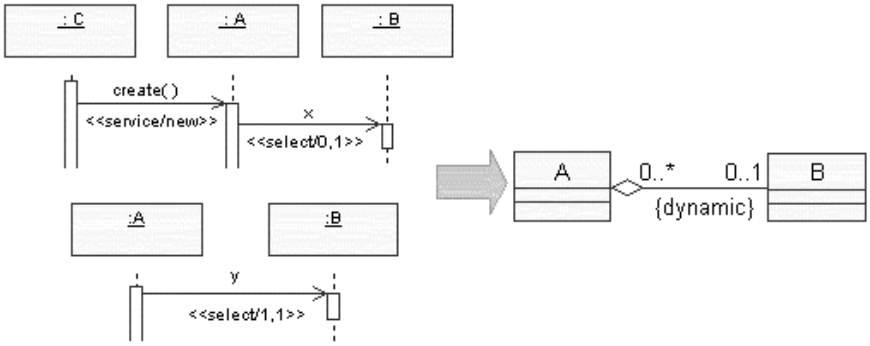

Figure 6: Obtaining aggregation relationships

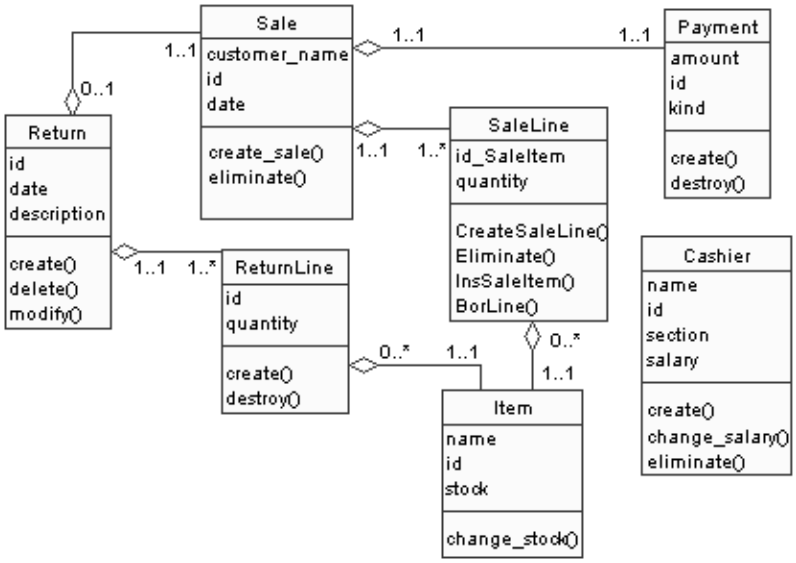

Figure 7: A Class Diagram obtained from the Requirements Model and the Requirements Analysis Process

as a server of a message and proposes a State Transition Diagram (after a dialog with the analyst) that satisfies all of them. Some related works in this area are [Som00] where different alternatives for the same scenario are combined to obtain one STD. It is an incremental process starting from the *normal sequence of events*. [Kos98] synthesizes State Diagrams 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. In [Whi00], they interpret and specify the messages from the Sequence Diagram with a precondition and a postcondition in order to obtain the resultant State Diagram.

Finally, the **Functional Model** in the Conceptual Schema of OO-Method describes the change of an object state as a consequence of a service occurrence. In the Requirements Model we don't capture any information related to the state of objects. This information should be introduced later on during the Conceptual Modeling phase (usually after many iterations between the Requirements and the Conceptual Schema).

4 Conclusions and further work

In this paper, we attempt to give an answer to the classical question of why it is so difficult to introduce Requirements Engineering into practice. People know that Requirement Engineering must be done, but in practice it seems to be still immature. In our opinion, it will remain so until

- There are generally accepted and well-defined notational standards
- There are professional standards which require a rigorous Requirements Analysis Phase with specific outputs

The main consequence of this situation is that the value of performing Requirements Engineering is too often not clear, due to the absence of a guided software production process starting from Requirements Engineering and continuing through the final product in a structured and well-defined way. In this context, we defend a clear idea: research in the Requirement Engineering area should be oriented towards properly embedding Requirements Engineering into the Software Production Process as a whole.

Following these arguments, we have presented such a unified software production process based on the blend of the TRADE and the OO-Method proposals. This integration provides a software production process that

1. starts from generally accepted notational standards and techniques to specify user requirements (based on the techniques of TRADE)
2. gives methodological guidance (following a Requirements Analysis Process) to convert these requirements into a precise Conceptual Schema (provided by the OO-Method conceptual modeling constructs)
3. links this Conceptual Schema with the Model-based Code Generation techniques of the OO-Method

This approach is the result of a long-term project developed in the context of a R&D activity carried out jointly by the Valencia University of Technology and CARE Technologies S.A. During its development, a central aim was to transfer results from academic research to the industrial sector. The adoption of this approach by software engineers is motivated by the simple and well-stated set of techniques and methods proposed and the traceability among them. Understanding what is to be built providing mechanisms to capture requirements and accurately translate them into pieces (classes) of a Conceptual Schema are major issues in the approach presented. The method has been successfully used in two medium-size projects where the main benefits reported were that:

- the ambiguity historically associated in the company to the process of building a Conceptual Schema disappeared with the introduction of the Requirements Model and the Requirements Analysis Process
- changes in user requirements could be traced to audit those that were the cause of misunderstandings between clients and analysts

A consequence of this collaboration, funded by CARE Technologies S.A., is the development of a CASE tool to capture the elements of the Requirements Model and automatically translate the output of the Requirements Analysis Process (class specifications) into a XML schema as a bi-directional communication link with the OO-Method/Case. In this way, our main objective of providing a complete software production process, from requirements to a final software product with a sound methodological basis, is accomplished by the cooperative use of these CASE tools.

References

[Bel98] Bell R. Code generation from object models. *Embedded Systems Programming*, (3), March 1998.

[Con95] Constantine L. Essential modeling: Use cases for user interfaces. *ACM Interactions*, pages 34–46, April 1995.

[Got95] Gotel O. *Contribution Structures for Requirements Traceability*. PhD thesis, Imperial College. Department of Computing. London-England., 1995.

[Ins99] Insfrán E., Wieringa R., and Pastor O. Using TRADE to improve an object-oriented method. Technical report, University of Twente. Computer Science Department, Enschede, the Netherlands, July 1999.

[Jac92] Jacobson I., Christerson M., Jonsson P., and Overgaard G. *Object Oriented Software Engineering, a Use Case Driven Approach*. Addison -Wesley. Reading, Massachusetts, 1992.

[Kos98] Koskimies K., Systä T., Tuomi J., and Männistö T. Automated support for modeling oo software. *IEEE Software*, January-February 1998.

[Lar98] Larman C. *Applying UML and Patterns*. Prentice-Hall, 1998.

[Mar89] Martin J. *Information Engineering, Book I: Introduction*. Prentice-Hall, 1989.

[McM84] McMenamin S.M. and Palmer J.F. *Essential Systems Analysis*. Yourdon Press. Prentice Hall, 1984.

[OMG01] Object Management Group. OMG UML Specification v. 1.4. Technical report, September 2001.

[Pas01] Pastor O., Gomez J., Insfran E., and Pelechano V. The OO-Method approach for information systems modelling: from object-oriented conceptual modeling to automated programming. *Information Systems*, 26(7):507–534, 2001.

[Pas95a] Pastor O. and Ramos I. *OASIS version 2 (2.2): A Class-Definition Language to Model Information Systems*. Servicio de Publicaciones Universidad Politécnica de Valencia, Valencia, España, 1995. SPUPV-95.788.

[Pas97] Pastor O., Insfrán E., Pelechano V., and Ramírez S. Linking object-oriented conceptual modeling with object-oriented implementation in java. In *Database and Expert Systems Applications (DEXA'97)*, pages 132–142, Toulouse, France, September 1997. Springer-Verlag. LNCS (1308). ISBN: 3-540-63478-9.

[Pas97c] Pastor O., Insfrán E., Pelechano V., Romero J., and Merseguer J. OO-Method: An oo software production environment combining conventional and formal methods. In *9th Conference on Advanced Information Systems Engineering (CAiSE'97)*, pages 145–159, Barcelona, Spain, June 1997. Springer-Verlag. LNCS (1250). ISBN: 3-540-63107-0.

[Pas98] Pastor O., Pelechano V., Insfrán E., and Gómez J. From object oriented conceptual modeling to automated programming in java. In *17th International Conference on Conceptual Modeling (ER'98)*, pages 183–196, Singapore, November 1998. Springer-Verlag. LNCS (1507). ISBN 3-540-65189-6.

[Reg95] Regnell B., Kimbler K., and Wesslen A. Improving the Use Case Driven Approach to Requirements Engineering. In *Second IEEE International Symposium on Requirements Engineering*. IEEE, March 1995.

[Som00] Somé S., Dssouli R., and Vaucher J. Toward an automation of requirements engineering using scenarios. Université de Montréal, 2000.

[Whi00] Whittle J. and Schumann J. Generating statecharts designs from scenarios. NASA Ames Research, 2000.

[Wie98] Wieringa R.J. Postmodern software design with NYAM: Not Yet Another Method. *Requirements Targeting Software and Systems Engineering*, pages 69–94, 1998.

[Wie99c] Wieringa R.J. TRADE: Techniques for requirements and design engineering of software systems. Technical report, Computer Science Department. University of Twente, Enschede, the Netherlands, June 1999.

[Wir93] Wirfs-Brock R. Designing scenarios: Making the case for a use case framework. *Smalltalk Report*, pages 9–20, Nov-Dec 1993.