From Use Cases to Objects: An Industrial Information Systems Case Study Analysis

João M. Fernandes[1], Ricardo J. Machado[2]
[1]Dep. Informática, [2]Dep. Sistemas de Informação
Escola de Engenharia, Universidade do Minho
4700-320 Braga, Portugal

Abstract

Identifying the objects that constitute a software system is a critical task for any object-oriented system development and several techniques have already been introduced to tackle this problem. This paper introduces a new approach to partially solve that problem, based on the use cases that define the system's functionality. The application of the approach is exemplified with a real industrial case study, which highlights some of the approach's main characteristics for complex embedded systems development.

1 Introduction

Although some authors claim that “objects are there just for picking” [1], usually identifying the objects that do constitute a real system is not a trivial task. Generally, the objects' identification introduces some divergences among the project members [2]. Several techniques are available for this task, and they give rise to different object models, so selecting the proper one is a difficult choice for any project team.

A well-known approach for identifying objects, and also attributes and methods, is based on linguistic analysis of the requirements document, written in a natural language [3]. The objects and attributes correspond to the nouns, and the operations (also known as methods or services) are related to verbs [4]. This strategy must be used with some prudence and requires strong linguistic knowledge, since it is possible to transform a noun into a verb and vice-versa.

For embedded and real-time systems, [5] presents 11 useful strategies to find/identify the objects of a system, but they can not be directly applied to obtain the objects based on the use cases, because none of them utilizes the use cases that were identified in previous tasks. These strategies do not justify the appearance of control objects, which are critical for this kind of systems.

Finding methods for bridging the gap between user's requirements and system design elements is an important problem during the development of complex

systems. This paper presents a novel approach for solving a real problem faced by designers, which is the definition of the system's objects based on the use cases and their respective textual descriptions. The authors' approach is based on OOSE's approach [6], trying to make the transition as smooth as possible, which allows the designers to relate each object with the use cases that gave origin to it. The approach also supports the existence of control objects as the crucial components in any embedded and real-time systems.

2 Objects in Systems Modeling

An object model can be used for 2 different purposes. One possibility is that the object model enriches the user's requirements, i.e., it provides a conceptual description of the entities in the application domain. In such a case, it complements use cases in describing requirements. Another possibility is that the object model is a design level notation, i.e., it provides an ideal architecture that first captures the system's requirements. This last alternative is the one addressed by this paper.

Our approach for identifying the system's objects requires the analyst to start the development by defining the functional model that reflects the system's services offered to its users from their perspectives. According to the authors' experience in capturing the user's requirements, use cases are one of the most suitable technique for that purpose. Use case diagrams are simple diagrams, since only three concepts are needed (use cases, actors and relations). This is a crucial property for involving non-technical customers and users in the requirements capture process.

Although use cases are used in object-oriented projects, they do not hold any intrinsic property that can be classified as “pure” object-oriented. Within the research community there is a consensus that use cases are a proper technique for object-oriented projects [7], namely for discovering and specifying the behavior of the system, during the analysis phase. This is also highlighted by the fact that use cases are part of the UML (Unified Modeling Language) notation [8]. Thus, adopting use cases for user's requirements is a valid technique, but poses the problem related to the transformation of use cases into objects.

After the use case diagram is built, the system's analysts must identify the objects and classes (and their relationships) that describe the system under development. As a starting point for this identification use cases should be used to guarantee the semantical continuity with respect to the previously captured user's requirements.

Many software engineers still do not clearly distinguish between objects and classes [1, 9]. This confusion is greatly related to the intangible nature of software. Building models that simultaneously have objects and classes should not be endorsed for two main reasons. Firstly, a class may be viewed as a pattern that allows the creation of objects for the application under development, but also for future applications. Secondly, objects represent the elements that do constitute the application. Therefore, taken into account this perspective, the authors strongly recommend, as a guideline for system's development, that it is preferable to not incorporate simultaneously classes and objects in the same model or diagram.

2.1 Object's categories

The analysis space can be divided in three orthogonal dimensions: information, behavior and presentation [6]. This technique is useful to control the system's specification complexity, creating a multidimensional modeling space to allow a multiple-view analysis of the requirements with adequate semantic references.

The object model is constructed by specifying objects in this tri-dimensional space. Our approach associates, to each object found during the analysis phase, a given category. Thus, we obtain a structure that can more easily adapt itself to the changes and evolutions imposed to the system. Each one of these categories is intrinsically related to one of the dimensions, but can also have minor components at the other two dimensions. Objects can thus be classified in three main categories:

1. An *interface-object* models behavior and information that depend on the system's interface, i.e. the dialogue of the system with the actors that interact with it. An interface-object must be encapsulated in a way such that if a change is made to the communication between the system and its actors, only the interface-object has to be modified, leaving unchanged the other objects.
2. An *entity-object* predominantly models information, whose existence must be lengthy (temporary storage should not be modeled as entity-objects). Apart from the attributes that characterize the entity-object, the entire behavior associated to the manipulation of that information must also be included in the entity-object.
3. A *function-object* models behavior that can not be naturally associated to any other object. For instance, the functionality that operates on several objects and that returns a result to an interface-object is an example of a function-object.

This categorization of objects can be interpreted as natural, since an object unifies in a unique entity (i.e. itself) the models (information and behavior) that are traditionally divided in the structured approach. It should be understood as normal that, in some situations, a given object has higher predominance on one of the analysis dimensions. Please notice the use of the expression "higher predominance", which indicates that, for example, an entity-object has its attributes as expected, but also operations, and that a function-object has its operations, but also attributes, since both are objects in the way object-orientation perceives them. This was the principal motivation that made the authors adopt, for the objects categories, names that are different from those used for the dimensions. This way, it may become more evident that an object is rarely located at one specific axis.

The authors believe that the categories of objects, adopted by them, make the models more stable, in the sense that modifications made during the system development are easier to locate and affect a smaller number of objects (preferably just one). The most frequent changes that a system is subject to are its presentation and its behavior, being the information the most stable component of the systems (this is the main idea around the object-oriented paradigm). Changes to the presentation affect only interface-objects, whilst modifications to the behavior may perturb any object category: (1) If the functionality is associated to a system's

information, then the entity-object that represents it is affected. (2) When the functionality to be altered is related to the way presentation is done, then the respective interface-object has to be changed. (3) Modifications to functionalities that involve various objects are generally local to a function-object.

2.2 Objects vs. Classes

Usually, object-oriented methodologies do not pay too much attention to the object diagram, i.e., they are class-driven. When they do, the class diagram is built firstly and only later the object diagram is specified. The approach presented here reverses this order, in an object-driven perspective. To develop software for embedded and real-time systems, the authors believe that it is more important to have a good object model than a good class diagram, because the system is composed of objects and not their classes. This is the main reason to first identify the objects and to later classify them, that is, to select the classes to which those objects belong. We are not advocating to not work out the class diagram or even to ignore it. The best situation is having good object diagrams and good class diagrams. What the authors promote is that the focus should be directed towards the construction of the object diagram.

Some specialists may classify the object-driven perspective (that puts classes in an apparently secondary role and that firstly defines the objects and later the classes) as object-based rather than object-oriented. Nonetheless, the object-driven approach is somehow similar to the bottom-up discovery of inheritance, as defined in [10], to hierarchically organize the classes.

In general terms, the process for building the object diagram can be described as follows. The first step is identifying the objects the system under development is composed of. Then, the objects are interrelated by establishing connections amongst them. Later, when the objects diagram is finished, the developers must classify every object. By classifying an object, the class diagram must be taken into consideration. If the class of an object already exists, it can be immediately used, which shortens the development time (and hence, the final cost of the system). Otherwise, a new class must be defined and put in the most proper place of the class diagram. In the next section, the authors' approach to construct the object diagram is presented.

The emphasis on objects (instances) is justified by the fact that we are dealing with real-time embedded systems. This object-driven approach is typically followed for developing control-oriented systems, where the final architecture and the concepts of abstraction, modularity and information are key topics to guarantee that the non-functional requirements (heterogeneity, ubiquity, fault-tolerance, security, dependability) are met. In contrast, class-driven approaches are used for data-driven systems, where the relations among classes (types) and its hierarchical categorization are the most important issues to taken into account. The objects' classification and the class diagram usage are not within the strict scope of this paper.

3 The 4-Step Rule Set Approach

The need to include both functional models and object-based models in system specification is nowadays widely accepted, as a cross-strategy approach to complex systems modeling [11].

Object diagrams are used to show the components that constitute the system. Transforming the use cases that divide the system in a functional way into objects is a critical project task, since usually there is no direct one-to-one mapping from use cases to objects, and the system's architecture is starting to possess a shape, near the final one. The authors present some novel guidelines and techniques for this transition, which consists in distributing the behavior specified by the use cases to the objects.

The approach followed by the authors (designated *4-step rule set*) to transform use cases into objects consists on the following steps:

1. Transform each use case into three objects (one interface, one data, and one control). Each object receives the reference of its respective use case appended with the suffix (i, d, c) that indicates the object's category.
2. Based on the textual description for each use case, it must be decided which of the three objects must be maintained to fully represent, in computational terms, the use case, taking into account the whole system and not considering each use case *per si*, as happens in a non-holistic approach.
3. The survival objects (those that were maintained after the execution of step 2), for which there is an advantage in being treated in an unified way, should give origin to aggregations or packages of semantically consistent objects.
4. The obtained aggregates must be linked to specify the associations among the existing objects.

This approach aims to obtain an holistic set of objects, so that the inter-relations amongst the objectified use cases can be successfully simplified in order to obtain the reduced number of relevant and pertinent system-level objects. The proposed approach is a partial solution to the problem of obtaining the objects from the use cases, because there still exist some subjective interpretation in the execution of the steps. This is natural, since, although analysis and design can be systematically supported, it is important to leave some freedom for design space exploration.

Actors do not need to be directly considered in the 4-step rule set strategy, since they are only important as a means to reach the use cases. They are not an aim *per si*. One of the objectives of using the 4-step rule set strategy is to assure the models continuity along the whole analysis phase (user's requirements) and during the models transformation to the design phase (system's requirements), as the first “synthesis” step in the system's design cycle that introduces new design decisions and that imposes modifications in the abstraction level.

4 Case Study Analysis

In this paper, the example is an embedded real-time computer-based system that supports the implementation of an Industrial Control-based Information System (ICIS) for a textile factory. The ICIS (which is dedicated to the supervision and monitoring of the shop-floor) and the traditional Management Information System (MIS) constitute the Global Enterprise Information System (IS). The case study used for analysis is responsible for monitoring the state of a set of machines that are used to produce textile goods, namely socks. According to the authors' methodology [12, 13], the first diagram to be built is the context diagram of the system that shows which actors interact with the system (not presented here).

The next task consists on the definition of the use cases of the system. A use case diagram is a powerful and useful technique for capturing the user's requirements. It is an easy-to-read diagram that divides the system in its functional points. A use case can be understood as a functionality or service that the system offers to its users.

Each use case at the system-level is assigned a tagged value (example: ref=2), and if this use case is eventually refined by other sub-use cases, each one of these will have a reference that uses the super-use case as a prefix (example: ref=2.3). This numbering scheme can be repeated to any depth and is used to ease the mapping from use cases to objects.

Fig. 1 shows the system-level (or top-level) use case diagram, where it is possible to visualize which actors perform which functionalities.

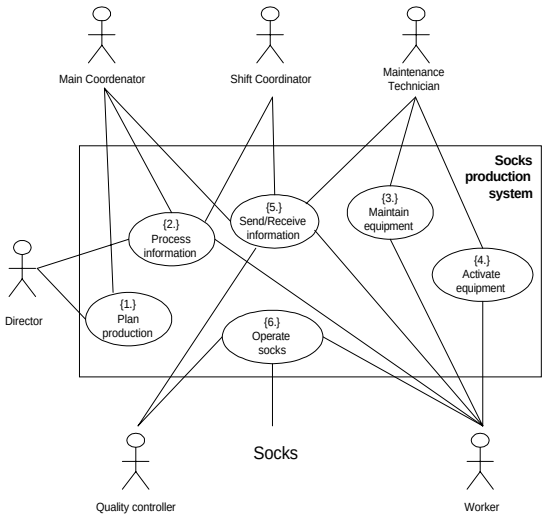

Figure 1. The use case diagram.

Some use cases have a very high abstract nature, and so the project members can refine them in order to have a better knowledge of the system. Although almost all of the top-level use cases were refined during the project, in this paper only two

refinements are presented in fig. 2. The two use case diagrams of this figure are related to the top-level use case 6: they are both orthogonal refinements of the top-level use case, according to different criteria. The left use case diagram (6.a) refines the use case by specialization (the use case is clarified according to its specialties), whilst the right one (6.b) refines it by decomposition (the use case is divided in its sub-tasks). This modeling technique based on orthogonal refinements is a way of coping with the specification complexity; instead of having 20 (5x4) use cases, only 9 (5+4) have to be considered.

Since use cases have different impact on the final system, they must be ranked according to their importance to the main functionality of the system. This allows the project to follow a risk-driven process, where the most important functionalities of the system are first tackled, leaving the less important ones to be treated later.

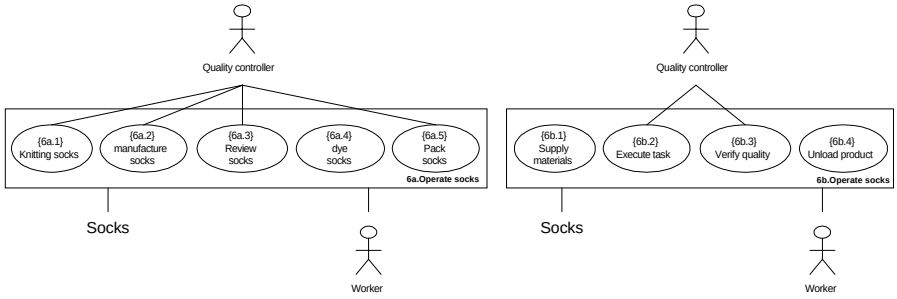

Figure 2. Refined use case diagrams (use case 6: operate socks).

After identifying all the use cases of the system, the next step is to describe their behavior. There are some forms for doing it (informal text, numbered steps with pre- and post-conditions, pseudo-code and activity diagrams [14] and the descriptions for the top-level use cases are next presented following the first proposal.

1. *Plan production*: Define production orders, based on information from the manufacturing process and customers' order.
2. *Process information*: Handle the necessary information, in order to acquire knowledge about the process.
3. *Maintain equipment*: Keep the system operational at the expected level, by replacing, repairing, changing and configuring the electrical, electronic, mechanical and software components of the production system.
4. *Activate equipment*: Switch on and off the machines.
5. *Send/Receive information*: Transfer and collect information to and from the process under supervision.
6. *Operate socks*: Accomplish a set of operations in order to produce socks.

4.1 Applying the 4-Step Rule Set

In the 1st step of the rule set, the designer has to take all the use cases and transform each of them in 3 objects (fig. 3). At this stage, the leaf use cases must be used,

instead of the most hierarchical ones. This means that use case 6 in fig. 1 must not be considered, since it is decomposed in other sub-use cases (fig. 2). Only the non-decomposed use cases must be considered for this step, even if later the corresponding objects are again aggregated to obtain an object that could be obtained directly from the top use case. This guideline is based on the fact that the resulting diagram contains more potential relevant links among objects and also on a practical observation that it is easier to aggregate objects rather than to divide them.

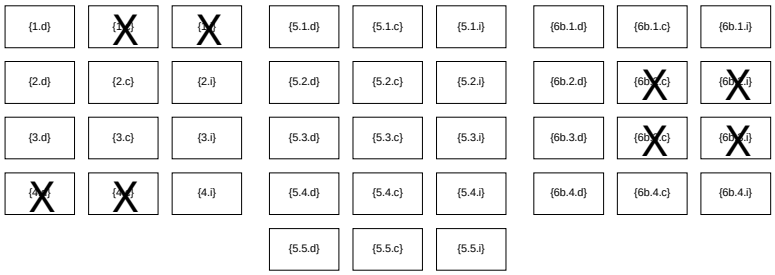

Figure 3. The object diagram after the 2nd step.

During the 2nd step and taking into account the textual descriptions of all the leaf use cases, one can decide to eliminate (by drawing a cross over the object) the control and entity objects of the use case 4, since this constitutes a typical interface functionality. For use case 1, the fundamental perspective is the information that results from the planning process, so the only survival object is the data one. For use case 2, all 3 objects are important and none of them is covered by another use case (holistic perspective), so all of them remain (objects not crossed) in the diagram. Similar considerations were made to the other use cases.

In the 3rd step, the system is divided into two main functional UML packages, both defining the limits between the MIS and the ICIS components of the global corporate IS solution. The MIS package essentially aggregates data objects, since the MIS is a data-oriented system. On the other hand, the ICIS is fundamentally composed of control and interface objects, because control-oriented systems must interact with the environment and supervise its running processes. Inside both packages, the aggregations follow analogous justifications as fig. 4 depicts. The associations that are defined amongst the objects in the 4th step soften the partition defined on step 3 to delimit the boundaries between MIS and ICIS. Within the ICIS package there is a link between each control object and the respective interface object that connects it to the environment.

After applying the last step of the rule set, the object diagram presented in fig. 4 is obtained. This is the final object diagram and it illustrates the usage of active and passive objects. An active object is continuously executing (it has its own thread) and is autonomous, which means that it exhibits some behavior without the command of another object/system. Active objects are represented by a rectangle with wider borders. On the other hand, passive objects do not initiate computation by their own initiative and so they just perform some action after the command of other

objects. Typically, the active objects are the basis for the control of the system under consideration, while the passive objects offer services to the active objects when asked for.

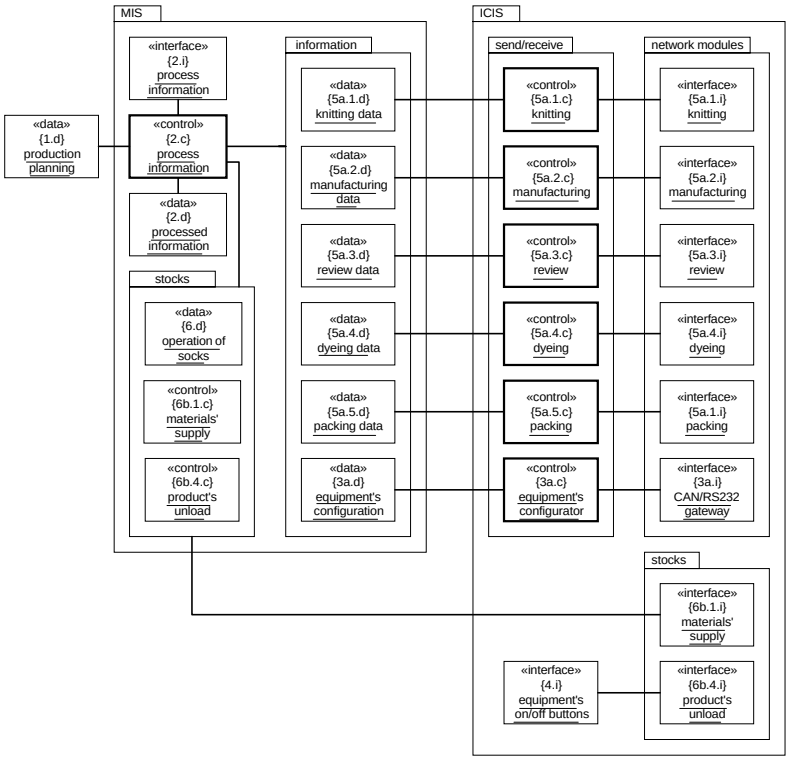

Figure 4. The final object diagram.

Notice that use cases specify the functionality of the system, while the object diagram is related to the structure of the system, which is used as the foundation for the design and implementation phases. The last object diagram represents an ideal architecture for the system, because its construction is supposedly independent of any implementation constraint [15].

5 Conclusions

The definition of the proposed approach was made with the purpose of coping with the complexity of deriving system's requirements from user's requirements and assuring that the models maintain an explicit and rigorous continuity.

Although the authors believe that the approach is helpful, it must always be used with some care and criticism, because it is not a panacea for solving all the problems that analysts have to face while specifying a system. It should, with no doubt, be complemented by other techniques the designers found useful or that they are already

familiar with. The main weaknesses of the approach are the explosion on the number of objects in step 1 and its non-formal nature.

The analysis phase is still very diverse and unpredictable in nature, so creativity still plays a major role. Experience can not be totally replaced by theoretical knowledge, so only with it can an analyst become a good one. This approach means to be a contribution, as an heuristic regulator mechanism, to the engineering of complex systems, where rigorous techniques should be consciously mixed with the necessary design freedom in the exploration of the best possible system solutions.

References

1. B. Meyer. Object-Oriented Software Construction. Prentice-Hall, 1988.
2. I. Graham. Object-Oriented Methods. Addison-Wesley, 1991.
3. N. Juristo, A.M. Moreno, and M. López. How to Use Linguistic Instruments for Object-Oriented Analysis. IEEE Software, 17(3):80-9, May/June 2000.
4. R.J. Abbott. Program Design by Informal Descriptions. Communications of the ACM, 26(11):882-94, Nov. 1983.
5. B.P. Douglass. Real-Time UML: Developing Efficient Objects for Embedded Systems. Object Technology. Addison-Wesley, 1998.
6. I. Jacobson, M. Christerson, P. Jonsson, and G. Overgaard. Object-Oriented Software Engineering: A Use Case Driven Approach. Addison-Wesley, 1992.
7. G. Booch. Best of Booch. SIGS Books, 1996.
8. G. Booch, J. Rumbaugh, and I. Jacobson. The Unified Modeling Language User Guide. Object Technology. Addison-Wesley, 1999.
9. C. Szyperski. Component Software: Beyond Object-Oriented Programming. Addison-Wesley, 1998.
10. J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. Object-Oriented Modeling and Design. Prentice-Hall International, 1991.
11. R.J. Machado and J.M. Fernandes. A Petri Net Meta-Model to Develop Software Components for Embedded Systems. In 2nd IEEE Int. Conf. on Application of Concurrency to System Design, Newcastle, U.K., June, 2001. IEEE CS Press.
12. J.M. Fernandes, R.J. Machado, and H.D. Santos. Modeling Industrial Embedded Systems with UML. In 8th ACM/IEEE/IFIP Int. Workshop on Hardware/Software Codesign (CODES'2000), pp. 18-22, San Diego, CA, USA, May 2000. ACM Press.
13. R.J. Machado, J.M. Fernandes, and H.D. Santos. A Methodology for Complex Embedded Systems Design: Petri Nets within a UML Approach. In Architecture and Design of Distributed Embedded Systems, B. Kleinjohann (ed.), chapter 1, pp. 1-10. 2001. Kluwer Academic Publishers.
14. G. Schneider, and J.P. Winters. Applying Use Cases: A Practical Guide. Addison-Wesley, 1998.
15. R.J. Machado, J.M. Fernandes, A.J. Esteves and H.D. Santos. An Evolutionary Approach to the Use of Petri Net based Models: From Parallel Controllers to HW/SW Co-Design. In Hardware Design and Petri Nets, A. Yakovlev, L. Gomes, L. Lavagno (eds.), chapter 11, pp. 205-222, 2000. Kluwer Academic Publishers.