Concept Identification in Object-Oriented Domain Analysis: Why Some Students Just Don't Get It

Davor Svetinovic, Daniel M. Berry, Michael Godfrey
School of Computer Science
University of Waterloo
Waterloo, Ontario, Canada
{dsvetino, dberry, migod}@uwaterloo.ca

Abstract

Anyone who has taught object-oriented domain analysis or any other software process requiring concept identification has undoubtedly observed that some students just don't get it. Our evaluation of the work of over 740 University of Waterloo students on over 135 Software Requirements Specifications during the last four years supports this same observation. The students' task was to specify a telephone exchange or a voice-over-IP telephone system and the related accounts management subsystem, based on models they developed using object-oriented analysis. A detailed comparative study of three much smaller specifications, all of an elevator system, suggests that object orientation is poorly suited to domain analysis, even of small-sized domains, and that the difficulties we have observed are independent both of the size of the system under specification and of the overall abilities of the students.

1 Introduction

Anyone who has taught object-oriented domain analysis or any other software process requiring concept identification has undoubtedly observed that some students just don't get it. Recently and not so recently, several authors, including Hatton [11], Kaindl [13], and Kramer [15], have indicated an urgent need for experimentation aimed at validating the effectiveness of software engineering abstraction techniques and methods, in particular of object orientation. Experimentation is required, since many of these methods have proliferated into day-to-day use without significant validation. As observed by Sackman, Erickson, and Grant [24], individual differences among practitioners tend to dominate the differences in methods. The variation among the models produced by different practitioners for the same system calls to question the benefit of applying these methods and causes us to wonder how we can reduce this variation.

The motivation for this paper comes from our observation of students' work on the requirements analysis and specification of a system composed of (1) a small telephone exchange or a voice-over-IP telephone network and (2) the related accounts management subsystem. Production of the specification, in the form of a Software Requirements Specification (SRS) document, is the term-long project carried out in the first course of a three-course sequence of software engineering courses that span the last year and a half of the software engineering undergraduate degree programs at the University of Waterloo [4].

Over the last five years, Svetinovic had a wide variety of roles, being customer, group coordinator, UML and SDL instructor and lecturer, and project evaluator. Each of Berry and Godfrey has been the lead instructor, i.e., the professor, teaching this course. We have personally reviewed over 135 different SRSs, out of over 195 that were developed in this time interval by over 740 software engineering, computer science, and electrical and computer engineering students. Our teaching experience has given us the opportunity to observe various software analysis and specification issues from different perspectives.

The projects in the three-course sequence involve using (1) various techniques for developing software for embedded real-time systems and (2) object-oriented techniques for developing information systems. The real-time components of the telephone systems are specified using formal finite-state modeling with SDL [2]. The information-system components of the telephone systems are specified using the notations of UML [23]. Use cases [e.g., 16] are used for capturing domain-level requirements, and object-oriented analysis is used as a bridge towards the later object-oriented design. In addition, students are responsible for modeling user interfaces of the information system and for the overall management of the specification process. The average

size of the resulting SRS document for the whole system is about 120 pages, with actual sizes ranging anywhere from 80 to 250 pages.

Through specification reviews, interactions with students, and grading preliminary partial and the final full SRSs, we have observed many difficulties that arise during the entire specification process. The most frequently observed difficulty is that of *performing object-oriented domain analysis (OODA)*, i.e., of

1. identifying concepts of the system's domain and
2. ascribing the system's functionality to these concepts.

For the purposes of this paper, we call this difficulty the *fundamental difficulty (FD)*. In addition, we have observed several other difficulties, which are deemed beyond the scope of this paper:

- difficulties in writing high-level use cases as opposed to writing simple scenarios,
- difficulties in avoiding specifying *how* and staying at the *what* level,
- difficulties in working with only system-level use cases and finding appropriate levels of abstraction,
- difficulties arising from the use of different modeling notations, UML and SDL, and
- difficulties in writing and managing many different types of UML and SDL diagrams.

We have found that the FD was occurring more frequently than any of the other difficulties, and it tended to stay unresolved. The effects of the FD were felt also throughout the specification of the system. The FD seemed to occur during the analysis of both the more complex and the less complex parts of the system.

The rest of the paper is organized as follows: Section 2 builds the background and discusses related work relevant to the students' task and our case study. Section 3 enunciates the case study's hypothesis. Section 4 describes our research method. Section 5 presents the case study results and the analysis. Section 6 discusses the results. Section 7 compares the case study results to our observations about the students' projects. Section 8 presents our conclusions and suggests future work.

2 Background and Related Work

In the early days of the software engineering, most of the methodology research and technology transfer efforts focused on improving programming. Programming was perceived to be the main difficulty in the development of software systems. The business systems supported by early software tended to be relatively small, well understood, and stable. As programming methods and technology matured and stabilized, the focus has shifted to automation of other larger, less understood, and more volatile business systems. Today, a typical business system is simply so large, is so complex, and changes so frequently that it is very difficult, if not impossible, to understand it completely. The inability to understand a business system and to precisely capture all of its goals and requirements results in software that does not fully satisfy all the business needs. This nonsatisfaction of needs leads, in turn, to dissatisfied customers and users, frequent changes, and other maintenance difficulties.

The size, complexity, and instability of modern business systems engendered a need for techniques for effectively capturing and understanding business needs and requirements. Often, the effort to understand a business system exceeds the effort to build the supporting software system. This situation is described often in the literature as the *what vs. how problem* [e.g., 12]. Learning *what* to build is often more difficult than learning *how* to build the system, i.e., correctly understanding a business is more difficult than building software to support it [e.g., 10].

To help people understand business domains, researchers have devised several approaches that can be divided into two groups. The first group consists of the methods inspired by different programming paradigms. The second group consists of methods that have roots in traditional business analysis.

Programming paradigms have influenced all development stages, even the early ones, such as requirements analysis and design. For example, structured and object-oriented programming resulted in structured [e.g., 6] and object-oriented [e.g., 3] analysis and design. This tradition continues even with the emergence of newer paradigms such as aspect orientation [e.g., 21] and agent orientation [e.g., 20].

Other researchers have applied traditional business analysis techniques during software requirements engineering. Goal-driven [e.g., 5] requirements engineering has proved to be promising for dealing with domain-level requirements for large systems. Goal-driven requirements engineering has focused on ensuring that software actually fulfills business needs and requirements.

Software requirements analysis techniques that originated from programming paradigms generally do not conflict with those that originated from business analysis; rather, they complement each other. The main difference between the techniques shows up in the requirement abstraction levels for which they are well suited. For example, goal-driven requirements engineering techniques are considered well suited to capture domain-level requirements [17], while object-oriented requirements engineering tech-

niques are considered well suited to capture product-level requirements [17].

Each of these techniques has its strengths and weaknesses. However, the main difficulties arise during the integration of the domain and product requirements, for example, in moving from domain-level requirements, such as goals, use cases, and features, to object-oriented analysis artifacts, such as objects, relationships, object features, and attributes. The need to integrate all these artifacts leads to the FD that is the subject of this paper. It is difficult but essential to establish meaningful and unambiguous relationships among these artifacts.

3 Case Study Hypothesis

The current trend of software development is towards iterative and use-case driven [22, 16] processes. In such processes, most domain objects are discovered iteratively, and the main source for their discovery are use cases and the domain knowledge acquired during development of the use cases. The students' projects were conducted in this manner. Depending on the abstraction level of the use cases, the degree to which sub use cases are separated out, and on how many scenarios are abstracted into a use case, a typical SRS had anywhere between 10 and 30 use cases.

The difficulty of discovering domain concepts does not appear to be greatly affected by the overall size of the system, because the conceptual decomposition was done at the level of the use cases. Since the conceptual decomposition was use-case driven, i.e., concepts were discovered as they came up during the generation of use cases, we have come to believe that the *fundamental difficulty of OODA is mostly independent of the size of the system under consideration.* In order to validate the correctness of this assumption, we have decided to perform a comparative case study of three specifications of a much smaller domain: an elevator system. The discussion about these specifications serves also to illustrate concretely the difficulties of object-oriented concept decomposition.

The hypothesis explored in this case study is that

the FD is present in both small and large systems,

i.e., the difficulties that we have observed in students' work are due not to the size of the system they were specifying but rather to something else, perhaps directly related to the object-oriented analysis paradigm.

4 Research Method

In order to test our hypothesis, we decided to perform a comparative study of several independently produced specifications of elevator systems. We have settled down to three different specifications found using Internet search engines.

Each of the specifications deals with the basic functionality of the elevator as seen from a user's perspective. This view of the elevator system means that there are two basic high-level use cases considered:

UC1: request an elevator cab to move to a particular floor, from *outside* the elevator cab, and

UC2: request an elevator cab to move to a particular floor, from *inside* the elevator cab.

The number of use cases in an elevator system is approximately one tenth of that of the telephone systems with which the students were dealing. At the same time, the elevator system is of a non-trivial size, as it consists of a non-trivial number of concepts in its domain, about forty.

4.1 Choice of the Case Study

To choose the case-study subject systems, we were guided by following requirements, constraints, and opportunities:

1. The hypothesis that the FD is independent of the size of a system required us to look for a domain which is considerably smaller than that of the telephone system used in the students' projects. The elevator system domain seems to fit the bill perfectly, especially since many specifications of it, in a variety of sizes and degrees of completeness, are readily available on the Internet. The elevator system domain has been used as an exemplar for years to demonstrate specification languages and techniques.

2. The chosen specifications were published on the Internet with no restrictions on their use for research purposes.

3. Each specification was authored by people with formal computer science education.

4. If we were to choose elevator system specifications that were composed as pedagogical examples to show the strengths of object-oriented analysis and design, we would expect fewer instances of the FD in the chosen specifications.

5. The elevator system domain is familiar enough to most readers, allowing the discussion here to focus on modeling difficulties rather than on the details of the domain.

6. The focus of each specification we found is different; some are general domain modeling exercises, some are specifications of elevator management systems, and some are for simulation purposes. We decided to use

three with different foci for a more robust test of the hypothesis. However, we expected that, nevertheless, their complete analysis models would be similar.

7. An elevator system should be easier to analyze than most business systems, as the services, i.e., functionality, that an elevator offers are quite simple, and the system itself consists mostly of tangible, physical objects. In contrast, the typical business system provides many complex interrelated services, and consists of many abstract, conceptual entities.

5 Analysis

This section first introduces all three case studies and then presents the results of our analysis.

The viewpoint we took in this analysis is that of an *ignoramus* [1]; we intentionally did not attempt to learn the domain or specify an elevator system ourselves before attempting this analysis. Also, we assumed each specification to be correct until it was proved otherwise. Finally, we assumed that object-oriented analysis is ideal for elevator systems, and we did not attempt any other kind of analysis. This viewpoint and these assumptions were required in order to preserve our objectivity in the case study. We pass the flavor of this objectivity on to the reader by purposely not naming the authors of the case studies, even though the authors can easily be determined simply by going to the case studies' web sites!

5.1 First Case Study

The first case study [7] has the smallest specification of the three. Its main purpose is to teach the basics of UML. Its author focused on analyzing the basic elevator functionality from a passenger's perspective.

The published analysis consists of

1. a problem description,
2. a use-case diagram,
3. a description of each use case's basic scenario,
4. a collection of sequence and collaboration diagrams, one of each for each use case's basic scenario, and
5. a conceptual diagram.

The author does not provide full use-case descriptions. Since a problem description was provided, and it was used as the main source for the concept decomposition, the lack of full use-case descriptions is not a concern.

The author does not make any attempt to distinguish among the types of concepts in the conceptual diagram, and he does not clearly demarcate the system boundary. We suspect that not distinguishing among the types of concepts and not defining the boundary impeded his efforts to discover domain concepts. Nevertheless, we believe that this impediment had less of an impact in this case study than in the course projects due to the smaller size of the case-study domain.

5.2 Second Case Study

The second case study [8] specifies an elevator control system for a three-story building. The size of this specification is close to that of the specification of the first case study.

The published analysis consists of

1. a problem description,
2. a use-case diagram,
3. a conceptual diagram,
4. a collection of state machine diagrams, one for each object in the conceptual diagram.

As in the first case study, the author provides no complete use case description. The specification is based on concept extraction from the problem description.

A very helpful feature in this system specification is the names of discovered domain concepts are bold faced in the problem description. What is bold faced is a good indication that the author used a noun-extraction technique to identify the domain concepts.

As in the first case study, the author does not make any attempt to distinguish among different types of concepts in the conceptual diagram, and he does not clearly demarcate the system boundary.

5.3 Third Case Study

The third case study [9] specifies a system for control of multiple elevators in a high-rise building. The system is supposed to be able to support from one to eight elevators, the exact number being a parameter of the specification. Each of the buildings has its own number of floors. Each elevator serves a possibly different set of possibly non-adjacent floors; the set of floors served by an elevator is called a *part* of the building. Each part of the building has no more than four elevators installed to serve it.

The published analysis consists of

1. a problem description,
2. a use-case diagram,
3. use-case descriptions, one for each use case,

4. a specification in the form of a collection of state machine diagrams, one for each object mentioned in the problem description and the use case descriptions.

Unlike in the first two case studies, this case study has fully developed use-case descriptions in addition to a problem description that is about the same size as the problem descriptions of the first two case studies. This case study's problem description is focused on the system's structure rather than on the system's functionality and constraints.

The main component of interest for us, the conceptual diagram, is not provided. Instead, the specification is divided into different sections, one for each concept; and for each concept, an extensive set of state diagrams is given.

5.4 Comparison

We performed an OODA of each of the case studies to find all concepts present in any case study. Table 1 shows the union of all concepts found in the case studies; the uniting was performed on the bases of (1) the names assigned by the case studies to the concepts, i.e., the same name appearing in two case studies is assumed to name the same concept in both studies and (2) the meanings of concepts, i.e., two concepts in different studies that mean the same thing are considered the same concept. Each concept has a row in the table. For each concept and each case study, the intersection of the concept's row and the case study's column has an entry indicating the origin of the concept within the case study:

- "D" indicates that the concept was discovered by the study's author.
- "I" indicates that the concept was not discovered or was ignored by the study's author, even though noun extraction shows that the concept is clearly in the domain, and
- an empty slot indicates that it was not possible to discover the concept from any domain artifact mentioned in the case study.

A concept labeled "D" is called a "D concept", and a concept labeled "I" is called an "I concept".

The table contains also a *type* column indicating the types for its intersecting concepts. Concept types help in classifying and understanding concepts. The concept types used in the study are:

- Physical Structural Element (PSE): A PSE is an entity that has a responsibility to act as a physical boundary, container, or structural element in a physical system.
- Conceptual Structural Element (CSE): A CSE is an abstract entity that has a responsibility to act as a concept boundary, container, or structural element primarily in an abstract business system, e.g., a department in a company is a CSE.
- Physical Processor (PP): A PP is a physical device that performs computations within the system.
- Conceptual Processor (CP): A CP is an abstract entity that performs computations within the system.
- Actor (A): An A is an external entity that directly communicates with the system.
- Intangible Concept (IC): An IC[1] is an abstract entity that exists within the system.

The concept rows of Table 1 are sorted by the concepts' types.

Physical structural elements and *actors* play important roles in the definition of the system boundary and interface. In our experience, these two types of concepts are the easiest ones to discover in the domain. *Physical processors* are important for the system's interface definition. They are relatively easy to discover, but often difficult to decompose into components. *Conceptual structural elements*, *conceptual processors*, and *intangible concepts* are important for the internal design of the software system. These concepts are the most difficult to discover primarily due to their abstract nature and their only implicit existence within the system.

Finally, the table contains a *purpose* column indicating the purposes that concepts take on in the specifications in which they were indicated.

All told, 44 concepts were discovered in the three specifications. Table 2 shows the numbers of D and I concepts in the three case study columns of Table 1. It shows also for each number of D or I concepts, its percentage out of all concepts found in any case study.

In the first case study, the ratio of discovered to ignored concepts is 3:2, in the second, the ratio is 2.71:1, and in the third, the ratio is 1:4.17. Clearly the ratios vary widely over the case studies with no particular pattern. This observation is consistent with our experiences with the course projects, for which we could never predict how many concepts a particular team would manage to capture.

The concept type with the lowest D-to-I ratio is IC; probably because people generally have difficulty identifying intangible, abstract concepts. The type with the second lowest D-to-I ratio is PSE. We surmise that the authors perceived physical structural entities as being less important

[1]We use the "*Intangible* Concept" instead of just "Concept" as the name of the type of an abstract entity in order to avoid confusion with general term "concept" used to describe arbitrary concepts that appear in the model.

Concept	*CS1 Status*	*CS2 Status*	*CS3 Status*	*Type*	*Purpose*
elevator system	I	D	I	CSE	to define the conceptual boundaries of the system to control and move the elevator cab
passenger	I	I	I	A	to use the elevator
elevator cab	D	D	D	PSE	transport passengers
building			I	PSE	physical system boundary
floor	I	D	I	PSE	to provide building's structure to define elevator travel destinations
top floor		I	I	PSE	special floor - different user interface to provide direction reference
bottom floor		I	I	PSE	special floor - different user interface to provide direction reference
button panel			I	PSE	container for buttons
elevator shaft			I	PSE	pathway for the elevator cab provide elevator access to the floors
button	D		I	PP	to unify all buttons
elevator button	D	D		PP	unify buttons inside the elevator cab
floor request button		D	D	PP	user interface for requesting floors
door button			D	PP	user interface for door opening
open door button		D	I	PP	request to open the doors when the elevator is not moving
close door button		D	I	PP	request to close the doors when the doors are open
floor button	D	D	D	PP	user interface for requesting elevator
stop button			I	PP	request immediate elevator stop at the next floor
door	D	D	D	PP	close the elevator cab
inner door		D		PP	close the elevator cab
outer door		D	I	PP	close the floor access to the elevator shaft
door opening device		I		PP	open the doors
floor number display			I	PP	user interface for indicating travel progress
floor sensor		D		PP	detect elevator position with respect to the floors
elevator engine		D		PP	move the elevator cab
elevator controller	D			CP	to delegate interface requests within the system to delegate internal responsibilities within the system
door timer		D	I	CP	constrain door opening time periods
current floor	I			IC	to define current location of the elevator
designated floor			I	IC	final travel destination
request		I	I	IC	unify all the request types
requested direction		D		IC	track user's traveling preference used for the elevator stopping scheduling purposes
elevator request		D		IC	track user's request for elevator services used for the elevator stopping scheduling purposes
elevator-up request		D		IC	*same as for elevator request*
elevator-down request		D		IC	*same as for elevator request*
pending queue		D		IC	keep track of unprocessed *elevator request button* and *floor request button* requests
summon			D	IC	to capture elevator request
stop request			I	IC	capture users request for stopping at a particular floor
time period		I	I	IC	constraint time allowed for various operations
stop			I	IC	unify different elevator stopping situations
immediate stop			I	IC	unplanned stop initiated by the passenger
planned stop			I	IC	stop at final travel destination
stop notification			I	IC	user interface for indicating elevator stops
button refusal notification			I	IC	user interface for invalid request notification
direction			I	IC	capture current traveling direction of the elevator
light		I		IC	user interface to indicate button status

Table 1. All Discovered Concepts

Case Study	*Discovered*	*% Discovered*	*Ignored*	*% Ignored*
CS1	6	13.64	4	9.09
CS2	19	43.18	7	15.09
CS3	6	13.64	25	56.82

Table 2. Numbers of Concepts

than other concepts of other types, because physical structural entities are perceived as being outside the scope of the system. Nevertheless, these concepts should be captured because they often constrain the behavior of the internal system.

6 Evaluation

This section's subsections contain evaluations of one case study relative to three specific manifestations of the FD:

1. *Misplaced Responsibilities*: determining which D concepts were assigned the responsibilities that really belong to the missing, I concepts,
2. *Omitted Responsibilities*: determining which responsibilities mentioned in a problem description were entirely missed in the corresponding models, and
3. *Omitted Passive Concepts*: determining which concepts, either D or I, are consumed or produced through interactions of other concepts.

Due to space limitations, mostly only the evaluation of the second case study is presented. The full evaluation is found in Svetinovic's Ph.D. dissertation proposal [25]. Because of the focus on one case study, unless otherwise explicitly stated, from here until Section 7, each "author" means the second case study's author, and each published analysis artifact, e.g., the conceptual diagram, is that of the second case study.

6.1 Misplaced Responsibilities

It appears that emphasizing structure over function in decomposing a system leads to difficulties assigning responsibilities to concepts. Moreover, when responsibilities are not clearly observable in a domain description, many activities remain unidentified.

Table 3 shows for each D concept that appears in the conceptual diagram the activities assigned by the author to the concept. The table shows also the purposes of these concepts as derived by us from all three case studies. Five of the eight D concepts in the conceptual diagram do not even have clear definitions of the activities for which they are responsible.

The main symptom of misplaced responsibilities is the existence of many I concepts in a conceptual diagram. When concepts are missing, an activity that is needed to fulfill the system's functionality gets assigned to one of the D concepts, often to a not fully appropriate concept; the overloaded concept gets this additional activity in addition to the activities for which it should be responsible. This misallocation of responsibilities means that each D concept has to fulfill a number of responsibilities that really should be fulfilled by other concepts, often not present in the conceptual diagram.

Even for the three D concepts that have their activities clearly indicated, (1) floor button, (2) open door button, and (3) close door button, we can observe misplaced responsibilities. For each concept, the purpose field indicates responsibility for only a subset of the activities that have been assigned to the concept. According to the author, each of these concepts is responsible for *requesting the elevator to perform a particular activity*. However, the purpose of each of these concepts is to serve as a user interface for the corresponding request. Capturing a user request is only a partial responsibility of the overall activity of requesting an elevator to perform an activity.

The reader may wonder why these three concepts cannot themselves completely fulfill the responsibility of requesting the elevator to do a particular activity. It is sufficient to identify the I concept that *should* collaborate with these three concepts to fulfill the responsibility. That one I concept would be request. This concept's purpose is to capture any request and all of its parameters and to carry out the actual request by distributing parameters to the concepts that participate in doing the request. This mode of thinking is important, because just discovering the request concept leads to discovering a request's parameters. This analysis propagation is necessary to achieve a complete model.

6.2 Omitted Responsibilities

We assume that the author *was* able to identify responsibilities that were mentioned in the problem description but were omitted from the conceptual diagram; after all, the author wrote the problem description! Therefore, this subsection focuses on only I concepts that were neither indicated in the domain description (by the author's having used bold face in the problem description) nor included in the conceptual diagram.

The first I concept to consider is time period. The concept time period is used in the activity of constraining the amount of the time the elevator door is open. The concept that directly depends on and uses time period is door timer. Since time period is not explicitly captured as a concept, and since the door timer concept does not capture the notion of having to keep track of the amount of time for

Concept	*Activity*	*Purpose*
elevator (cab)	*none*	transport passengers
elevator engine	*none*	move the elevator cab
floor button	request the elevator to come to the floor	user interface for requesting elevator
elevator button	*none*	unify buttons inside the elevator cab
open door button	request to open the doors when the elevator is not moving	user interface for opening door
close door button	request to close the doors immediately	user interface for closing door
door	*none*	close elevator cab and shaft access for the safety purposes
door timer	*none*	constrain door opening time periods

Table 3. Second Case Study: Discovered Concept-Activity-Purpose Relationships

which the door can stay open, the responsibility of keeping track of time is omitted.

The second omitted responsibility is that of opening doors. This responsibility should be be assigned to door opening device. It is possible that the author assumed that this activity is part of a door's functionality. However, because this activity is captured neither in the domain description nor in the diagrams, we assume that it was missed entirely or purposely omitted. In addition, that this responsibility has to exist is clear from the existence of the open door buttons and the close door buttons. Obviously, the author had discovered two out of three concepts that participate in the activity of managing door movement but omitted the concept that would have been responsible for the actual action of moving the doors.

The light concept's responsibility to indicate a button's status is missing. This responsibility might have been identified but purposely omitted if the author assumed that responsibility is handled by the button's hardware and thus does not need to be in the software. However, even when hardware does discharge a responsibility, the responsibility needs to be specified so that the responsibility is not lost if hardware that behaves differently is ever used in the future.

The author used elevator request button in the conceptual diagram to unify the buttons and button requests. However, we believe that request should have been a concept in its own right in order to unify all elevator requests. Thus, request is considered to be a partially omitted responsibility.

Another group of obvious, but omitted, responsibilities is those of the passenger, as the user of the elevator. Since some argue that actors should not be included in conceptual diagrams, it is possible that the author made an explicit decision to omit the passenger's responsibilities.

Finally, the unique responsibilities of the top floor and the bottom floor are to deal with the different user interfaces that these floors require. Also, the responsibility of these two floors to provide a direction reference for the elevator cab has been omitted.

6.3 Omitted Passive Concepts

The specification has a rich set of passive concepts. The passive concepts that the author has identified in the domain descriptions are requested direction, elevator request, elevator-up request, and elevator-down request. Although these passive concepts were clearly indicated in the domain descriptions, the author did not include any of them in the conceptual diagram, probably because of their passive nature. None of them is performing active work. Instead they are produced or consumed by other concepts in achieving the other concept's responsibilities. This omission is consistent with what we have seen in students' projects.

Note that the request concept is not really a passive concept but rather an abstract concept since its purpose is to unify many concrete passive concepts. When considering the students' projects, we observed that abstract concepts need to be discovered because their discovery often facilitates the discovery of other passive concepts. This facilitation could be regarded as one purpose of identifying inheritance during analysis. In the case study, however, the author did manage to discover several related concrete passive concepts without discovering this abstract concept.

The third case study is quite similar to the second with respect to the discovery of passive concepts. The third case study author discovered and included only one passive concept: summon. We discovered several additional ones: notification, direction, stop, time period, request, stop request, and stop notification. Overall, in all three case studies, passive concepts were largely omitted, whether from ignorance or inability to discover them.

7 Discussion of Results

These case studies suggest that the presence of the FD in specifications arising from OODA is independent of system size. Even in a small problem such as an elevator system, there are many symptoms of the FD, just as there are in the larger student projects. While our students were inexperienced in OODA, since the case study authors were writ-

ing scientific or pedagogic exemplars of OODA, we assume that these authors were skilled and experienced in OODA. Therefore, we believe that the *cause* of this FD is neither the size of the specified system nor the specifier's lack of OODA experience. Rather, we believe that the FD arises from two inherent properties of complex business systems:

1. Each of most concepts fulfills only a sub-activity of a larger activity by interacting with other concepts, and
2. each of most concepts participates in many different activities, each for different purposes.

Of course, this belief will have to be confirmed in future experiments.

These two properties of complex business systems contradict what most students learn in their study of object-oriented *design and programming*, which tends to emphasize what we call for the purposes of this discussion "crispness", i.e, well-defined classes, each with a single purpose and a focused set of related responsibilities. Trying to analyze a complex problem domain with OODA leads many a student to discover a lot of the concepts, responsibilities, and activities that are there but to bend them so that they fit the first decomposition arrived at in an attempt to make a crisp decomposition. In doing so, students, working in teams, tend to perform the following actions:

1. They assign responsibilities fulfilled through the collaboration of multiple concepts to only one concept. Assigning the responsibilities of multiple concepts to one concept leads in turn to the following difficulties:
   - assigning to a concept indirect responsibilities—those achieved by collaboration with other concepts—that are larger in scope than the responsibilities for which the concept is directly responsible,
   - missing true responsibilities of that concept; the missing responsibilities are hidden within the larger responsibilities, and
   - missing other concepts that participate in the overall responsibilities.
2. They do not capture passive concepts, concepts that are produced or consumed by interaction of other concepts.

We believe that these tendencies are the main cause for the manifestations the FD that we have observed with the students' projects:

- under-specified analysis models: students tend to capture only a subset of the available concepts even though many are visible in the domain-level requirement artifacts and can be extracted using even the relatively simple noun-extraction technique [16];
- drastically different models of ostensibly the same system from different groups, probably from different groups' having focused on very different subsets of concepts; and
- large number of software concepts at different abstraction levels, made visible through the existence of relationships in the conceptual models among concepts that do not have such relationships in the problem domain.

Notwithstanding that the creators and proponents of object-oriented methods have touted object orientation as an excellent way of capturing domain concepts and bridging the conceptual gap between business systems and software [14, 18, 19], our experiences with over 135 projects has shown that capturing domain concepts and bridging the conceptual gap remains difficult even when the practitioner ostensibly knows object orientation.

Please recall that this discussion is about only OODA. We have not made any claims about object-oriented design, object-oriented programming, or any other object-oriented paradigm or technology. Even if our claims about the FD of OODA are eventually validated, it would still be impossible to generalize them to other disciplines that happen to share object orientation.

Our analysis of these results arise from the observation of only the specification artifacts produced as a result of the students' OODA of a business system and of the students' behavior. We have not have the opportunity to analyze the designs later produced by the these students. Nevertheless, we have never noticed that any of them had any problems understanding object-oriented programming concepts. However, whenever it came to discovery and analysis of concepts in the problem domain, it seemed that this object-oriented programming background knowledge did not help much. Therefore, we have come to the conclusion that the FD lies neither in the students nor in their so-called inability to understand object orientation, but rather in limitations of the current OODA techniques, at least as applied in requirements engineering.

8 Conclusion

This paper begins with the description of several OODA difficulties we have observed in a large number of students' software requirements specification projects over the last five years. This paper classifies these difficulties into six categories and focused on one of them—the one that showed up in nearly all the specifications—the difficulty of discovering concepts in a problem domain. The paper terms this difficulty the *fundamental difficulty* of OODA.

This paper has explored one particular hypothesis about this FD, namely that it is independent of the size of the

system under consideration, and that it is prevalent also in published—and presumably polished—exemplar specifications derived from OODAs of small systems. It reports one part of a thorough examination of three particular specifications of elevator systems, and offers evidence from the three specifications to support the hypothesis.

9 Future Work

Our future work will involve completing the analysis described in this paper, carrying out similar analyses of the difficulties declared out of the scope of this paper, and eventually to carrying out controlled experiments of OODA carried out on a variety of problems, using a variety of OO paradigms, expressed in a variety of OO modeling languages.

In particular, we have already gotten the support of 140 students who have allowed the use of their 30 different complete specifications of a voice-over-IP-telephone system for research purposes and for eventual publication of the results. These specifications will be compared with the specifications produced under constrained conditions. These empirical results should provide the research community with the ability to objectively judge the effectiveness of OO methods for requirements engineering.

References

[1] D. M. Berry. The importance of ignorance in requirements engineering. *Journal of Systems and Software*, 28(2):179–184, 1995.

[2] R. Bræk and O. Haugen. *Engineering real time systems: an object-oriented methodology using SDL*. Prentice Hall International, 1993.

[3] P. Coad and E. Yourdon. *Object Oriented Analysis*. Prentice Hall, Englewood Cliffs, NJ, second edition, 1990.

[4] CS445 course project description. `http://www.student.cs.uwaterloo.ca/~cs445/project/`; accessed January 10, 2005.

[5] A. Dardenne, A. van Lamsweerde, and S. Fickas. Goal-directed requirements acquisition. In *IWSSD6: Selected Papers of the Sixth International Workshop on Software Specification and Design*, pp. 3–50. Elsevier Science, 1993.

[6] T. DeMarco. *Structured Analysis and System Specification*. Yourdon Press, New York, 1978.

[7] Elevator system specification. `http://www.geocities.com/SiliconValley/Network/1582/uml-example.htm`; accessed January 10, 2005.

[8] Elevator system specification. `http://se.uwaterloo.ca/~mctanuan/thesis/elevreqt.ps`; accessed January 10, 2005.

[9] Elevator system specification. `http://www.umot.net/examples/elevator.php`; accessed January 10, 2005.

[10] J. F. P. Brooks. No silver bullet - essence and accidents of software engineering. *IEEE Computer*, 20(4):10–19, 1987.

[11] L. Hatton. Does oo really match the way we think? *IEEE Software*, 15(3):46–54, 1998.

[12] M. A. Jackson. The role of architecture in requirements engineering. In *Proceedings of the IEEE International Conference on Requirements Engineering*, p. 241. IEEE Computer Society, 1994.

[13] H. Kaindl. Is object-oriented requirements engineering of interest? *Requirements Engineering*, 10(1):81–84, 2005.

[14] A. Korthaus. Using UML for business object based systems modeling. In M. Schader and A. Korthaus, editors, *The Unified Modeling Language – Technical Aspects and Applications*, pp. 220–237, Heidelberg, Germany, 1998. Physica.

[15] J. Kramer. Abstraction: The key to software engineering? *Keynote: JSSST Japan Society for Software Science and Technology Conference*, 2004.

[16] C. Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process*. Prentice Hall, Englewood Cliffs, NJ, second edition, 2001.

[17] S. Lauesen. *Software Requirements: Styles and Techniques*. Addison-Wesley, Reading, MA, 2002.

[18] P. Loos and P. Fettke. Towards an integration of business process modeling and object-oriented software development. Technical report, Chemnitz University of Technology, Chemnitz, Germany, date unknown.

[19] G. Mentzas. Coupling object-oriented and workflow modelling in business and information reengineering. *Information Knowledge and Systems Management*, 1(1):63–87, 1999.

[20] J. Mylopoulos, M. Kolp, and P. Giorgini. Agent-oriented software development. In *Methods and Applications of Artificial Intelligence: Second Hellenic Conference on AI, SETN*, LNCS 2308, pp. 3–17. Springer, 2002.

[21] A. Rashid, B. Tekinerdoğan, A. Moreira, J. Araújo, J. Gray, J. G. Wijnstra, and P. Clements. Early aspects: Aspect-oriented requirements engineering and architecture design. In *Workshop at AOSD-2002*, 2002. `http://trese.cs.utwente.nl/AOSD-EarlyAspectsWS/`; accessed January 31, 2005.

[22] W. Royce. *Software Project Management - A Unified Framework*. Addison-Wesley, Reading, MA, 1998.

[23] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, Reading, MA, 1998.

[24] H. Sackman, W. Erickson, and E. Grant. Exploratory experimental studies comparing online and offline programming performance. *Communications of the ACM*, 11(1):3–11, 1968.

[25] D. Svetinovic. Ph.d. dissertation proposal. Technical report, University of Waterloo, Waterloo, ON, Canada, 2005.