# Increasing Quality of Conceptual Models: Is Object-Oriented Analysis That Simple?

[Position Paper]

Davor Svetinovic
School of Computer Science
University of Waterloo
Waterloo, Ontario, Canada
dsvetino@uwaterloo.ca

Daniel M. Berry
School of Computer Science
University of Waterloo
Waterloo, Ontario, Canada
dberry@uwaterloo.ca

Michael W. Godfrey
School of Computer Science
University of Waterloo
Waterloo, Ontario, Canada
migod@uwaterloo.ca

## ABSTRACT

Several researchers have recently indicated an urgent need for re-evaluation and validation of the various software engineering abstraction techniques, and object orientation in particular. This paper presents three questionable practices and one promising direction with respect to achieving high quality analysis models. This work is based on five years of observation of more than 700 students working on software requirements specifications of a small telephone exchange and a related accounts management system.

## Categories and Subject Descriptors

D.2.1 [**Requirements/Specifications**]: Methodologies

## General Terms

Measurement, Documentation, Design, Experimentation, Human Factors

## Keywords

abstraction, abstraction techniques, analysis models, conceptual models, empirical observations, object-oriented analysis, quality of models, software requirements specification

## 1. INTRODUCTION

In 1967, Dahl and Nygaard unveiled Simula 67 [4], the first object-oriented (OO) programming language. In 1982, Booch published his paper on object-oriented design (OOD) [1]. In 1988, Shlaer and Mellor published their book on object-oriented analysis (OOA) [14].

These three events were major milestones in the development of the OO paradigm. Today, object orientation is not only one of the oldest development paradigms, but also one of the most widespread in practice. From OO programming languages to OO modeling standards and frameworks, object orientation shapes the way we think about business and software systems, organize our development processes, and so on. Why did this happen?

We believe that the eventual widespread adoption of object orientation was fueled partially by the impact of Booch's paper [1], and in particular due to the two of his claims in that paper. His first claim, in Section 4.3.1, "Identify Objects and Their Attributes", is:

> *This step is a simple task; we will repeat our informal strategy in the abstract world, underlying* [sic, should be "underlining"] *the applicable nouns and adjectives....*

His second claim, in Section 4.3.2, "Identify Operations on the Objects", is:

> *This too is a simple task; we will repeat the above process, this time underlining the applicable verbs and adverbs....*

These two sections explain the main steps of an OOD method that Booch was advocating. Today, we know that these two steps form the foundation of OOA. What we also know is that these tasks are not that *simple* and they are not sufficient for the production of high-quality OOA models.

Hatton [7], Kaindl [9], and Kramer [10] have indicated an urgent need for experimentation aimed at validating the effectiveness of not just object orientation but of all software engineering abstraction techniques and methods.

## 2. A DISCOURAGING CASE STUDY

Over the last five years we have been observing and evaluating students' work on the requirements analysis and specification of a system [16] composed of:

1. a small telephone exchange or a voice-over-IP telephone network specified using formal finite-state modeling with SDL [2], and
2. the related accounts management subsystem specified using the notations of UML [13].

The specification of this system is the term-long project carried out in the first course of a three-course sequence of software engineering courses that spans the last year and a half of the Software Engineering undergraduate degree program


*ROA'06* May 21, 2006, Shanghai, China

at the University of Waterloo [3]. The average size of the resulting Software Requirements Specification (SRS) document for the whole system is about 120 pages, with actual sizes ranging anywhere from 80 to 250 pages.

We have personally reviewed over 135 different software requirements specifications, out of over 195 that were developed in this time interval by over 740 software engineering, computer science, and electrical and computer engineering students.

We have observed the same problems in all specifications:

- under-specified analysis models
- that differ widely depending on which group produced them, and
- with many concepts at inconsistent abstraction levels.

In particular, the following problems are present in nearly all of the students' OOA models:

- assigning a large business activity as a responsibility of a single object,
- missing responsibilities,
- omitting other objects that participate in a business activity, and
- missing objects that are operated upon or produced through the interactions of several other objects.

We have found that we simply have no guarantee that any OOA model will achieve any level of completeness and consistency. We cannot predict what the differences will be, or even understand why these differences exist in the first place. Contrary to what was requested more than 14 years ago [8], not only do we lack a formal basis for verifying that the analysis model is complete and consistent, but we cannot achieve even an informal understanding. The wide variation among the OOA models produced by different analysts for the same system calls to question the benefit of applying OOA. We must confront this issue and find the ways to measure and manage this variation.

## 3. THREE QUESTIONABLE PRACTICES

We have observed in our students work that three recommended practices of object orientation appear not to work well in practice:

1. iterative analysis,
2. identifying concepts from system-level use cases, and
3. choosing a unified system perspective.

The first practice is the iterative approach to the analysis. Some students used a typical iterative use-case driven approach for discovery of the domain knowledge and then the discovery of the concepts [e.g., 11]. while others used a waterfall-like approach [e.g., 12]. We did not observe any correlation between the number of iterations and the quality of the conceptual models. One possible explanation is that even when students use an iterative approach, and they discover new artifacts that do not fit into the extant conceptual model, they tend to be hesitant to change the conceptual model. Instead, they tend to try to adapt the new artifacts so that they conform to the existing conceptual model, leading to even more bloated and inconsistent models. Thus, rather than expecting the students to incrementally improve their model over the course of several distinct iterations, we have learned that it is more effective to encourage them to try to get their model right immediately. Our impression is that when the students know that they have multiple iterations, they do not put as much effort into getting the model right the first time, figuring that they will always be able to fix any problems later. However, the problems compound before they get fixed, making them even harder to fix.

The second practice is identifying concepts from system-level use cases [e.g., 11]. The purpose of the system-level use cases is to show the interactions between actors and the system, while treating system as a black box, i.e., not showing internals of the system. The use cases in this form capture some of the *interchange data* between system and actors. After writing use cases, one has to try to extract domain concepts. The source of the majority of the concepts is the *interchange data* passed between actors and system. The problem is that often these concepts are used as if they represent the domain and are used to model the internals of the software system. The *interchange data* do not reflect all of the possible domain concepts; they present only a subset of information that exists within only the business system. We have observed that analysts cannot rely on only system-level use cases for the discovery of a complete conceptual model. Rather, it is necessary to search for additional resources, and they vary widely from project to project. For the students' SRSs, the additional sources include project-description documents [3] and interviews with the customers. Each of these additional resources is usually less structured and less consistent than use cases, making it even harder to verify the consistency and completeness of the conceptual model. Thus, we have learned that system-level use cases are quite limited in scope, and it is better that finding them should be only one step in the generation of many types of artifacts for the conceptual models that need to be built.

The third practice is choosing the perspective from which we define the boundaries of the business and software systems. We have observed that the students typically take one of two evident perspectives. The first perspective is treating the business system and the supporting software system as one system together. A follower of this perspective tends to assume that a domain concept is a software concept and vice versa. Through observation of the students' work, we have discovered that this perspective usually results in a reduced ability to distinguish among business concepts and software concepts. A follower of this perspective tends to capture a relatively smaller subset of the domain concepts than do others. The second perspective is treating the business system and the supporting system as two different systems. A follower of this perspective tends to consider concepts from the problem domain and the software system to be different entities. A follower of the second perspective tends to capture more domain concepts than a follower of the first perspective.

The improvement that results from following the second perspective suggests that we should define clear boundaries between the domain system and the software system and keep a clear separation between the domain and software

systems. Thinking of a software system as a direct simulation of the domain system, i.e., the software and domain system are the same systems, results in a lower quality model with fewer discovered concepts. We need to realize that domain system and software systems are *different* systems. We need to realize also that domain concepts, design concepts, and code concepts are three *different* kinds of concepts.

## 4. A PROMISING DIRECTION

The first author, when working with some of his groups, suggested an approach that helped them increase the consistency and completeness of their models:

1. Clearly separate the domain system from the software system;
2. describe and explain domain processes and tasks in addition to the use cases;
3. describe a possible software solution in terms of software processes and tasks based on domain processes and tasks;
4. discover mappings between domain tasks and software tasks; and then
5. break each system into concepts and structural components.

That is, process decomposition and functional decomposition are performed separately before performing the conceptual analysis. This change of procedure helps an analyst move away from the discrete superficial breakdown of the domain into actors and their activities captured through use cases towards what one actually has to analyze. One should analyze two distinct systems, (1) the business system, a.k.a., the domain system, and (2) the software system, which support each other and have to work in full synergy.

Completing process-based decompositions of both business and software systems before attempting the conceptual decomposition of the domain system is probably too radical a change. However, the core idea of attempting to derive some intermediate analysis artifacts that are more constrained than conceptual models might be worth exploring.

The main problem is finding exactly what paradigm and what artifacts should be produced before pursuing with traditional OOA. We believe that the paradigm

1. should minimize the amount of the intermediate work,
2. should ideally be a subparadigm of the OO paradigm, and
3. should provide a mechanism for a smooth transition from domain analysis artifacts to OOA artifacts.

We believe also that properties 1 and 3 should hold also for the artifacts produced by the paradigm.

We believe there are several kinds of modeling that might satisfy these requirements:

1. architecture-driven modeling,
2. procedural modeling,
3. state-based modeling, and
4. goal-driven modeling.

A subparadigm of each of these modeling paradigms might be used as an intermediate tool for constraining the OOA. Perhaps this radical approach of using different paradigms rather than the same paradigm for moving from domain requirements to analysis models will result in higher quality OOA models. We believe also that the same way of thinking might be beneficial for production of other types of development artifacts. For example, using three different paradigms for analysis, design, and implementation might result in more consistent artifacts at each abstraction level than what can be achieved using only the OO paradigm for all three.

## 5. CONCLUSION AND FUTURE WORK

OOA techniques have not changed fundamentally from what Booch described in 1982. What has changed is the realization that these *simple* OOA tasks are not sufficient for producing high-quality analysis models for today's complex business systems. This paper has described three OO practices that do not appear to improve the consistency and quality of the resulting OOA models. How well these practices work and what other practices work better needs further empirical study. The paper has described also another promising direction that might help produce more consistent conceptual models.

We have already finished one part of the research begged by this paper. By working with our students as each group unified its descriptions of its use cases for its VoIP system into a single statechart-expressed domain model of its VoIP system, we have determined that a manual equivalent of the formal unification methods proposed by Glinz [5]; Whittle and Schumann [17]; and Harel, Kugler, and Pnueli [6] helps students create measurably higher quality domain models and ultimately to write measurably higher quality SRSs than in the past, but at a cost of 25% more work by the students and the teaching staff [15].

## 6. REFERENCES


[1] G. Booch. Object-oriented design. *Ada Lett.*, I(3):64–76, 1982.

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

[3] SE463/CS445 course project. `http://www.student.cs.uwaterloo.ca/~cs445/`; accessed February 28, 2006.

[4] O.-J. Dahl, B. Myhrhaug, and K. Nygaard. *SIMULA 67 Common Base Language.* Norwegian Computing Centre, Oslo, Norway, 1968.

[5] M. Glinz. An integrated formal model of scenarios based on statecharts. In *Proceedings of the 5th European Software Engineering Conference*, pages 254–271, London, UK, 1995. Springer-Verlag.

[6] D. Harel, H. Kugler, and A. Pnueli. Synthesis revisited: Generating statechart models from scenario-based requirements. *Lecture Notes in Computer Science*, 3393:309–324, Jan 2005.

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

[8] R. Holibaugh. Object oriented modelling. In *OOPSLA '91: Addendum to the proceedings on Object-oriented*

*programming systems, languages, and applications*, pages 73–78. ACM Press, 1991.

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

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

[11] 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.

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

[13] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, Reading, MA, second edition, 2004.

[14] S. Shlaer and S. J. Mellor. *Object-oriented systems analysis: modeling the world in data*. Yourdon Press, 1988.

[15] D. Svetinovic, D. M. Berry, N. A. Day, and M. W. Godfrey. Domain system statecharts: The good, the bad, and the ugly. Technical report, Cheriton School of Computer Science, University of Waterloo, Waterloo, ON, Canada, 2006. `http://se.uwaterloo.ca/~dberry/FTP_SITE/tech.reports/svetinovicBerryDayGodfreyTR2006.pdf`; accessed February 28, 2006.

[16] D. Svetinovic, D. M. Berry, and M. Godfrey. Concept identification in object-oriented domain analysis: Why some students just don't get it. In *International Conference on Requirements Engineering RE'05*, 2005.

[17] J. Whittle and J. Schumann. Generating statechart designs from scenarios. In *ICSE '00: Proceedings of the 22nd international conference on Software engineering*, pages 314–323, New York, NY, USA, 2000. ACM Press.