IGI PUBLISHING ITJ 4656

701 E. Chocolate Avenue, Suite 200, Hershey PA 17033-1240, USA
Tel: 717/533-8845; Fax 717/533-8661; URL-http://www.igi-global.com

This paper appears in the publication, Journal of Database Management, Volume 20, Issue 1
edited by **Keng Siau © 2009, IGI Global**

Applying Learner-Centered Design Principles to UML Sequence Diagrams

Debra VanderMeer, Florida International University, USA
Kaushik Dutta, Florida International University, USA

ABSTRACT

The Unified Modeling Language has been shown to be complex and difficult to learn. The difficulty of learning to build the individual diagrams in the UML, however, has received scant attention. In this article, we consider the case of the UML sequence diagram. Despite the fact that these diagrams are among the most frequently used in practice, they are difficult to learn to build. In this article, we consider the question of why these diagrams remain so difficult to learn to build. Specifically, we analyze the process of learning to build sequence diagrams in the context of cognitive complexity theory. Based on this analysis, and drawing on the theory of learner-centered design, we develop a set of recommendations for presenting the sequence diagram building task to the student analyst to reduce the complexity of learning how to build them.

Keywords: cognitive complexity; learner-centered design; sequence diagrams; systems analysis and design, unified modeling language (UML)

INTRODUCTION

Object-oriented analysis and design (OOAD) is the dominant software design method among practitioners. The Unified Modeling Language (UML) is an ISO standard graphical modeling language used in OOAD (International Organization for Standardization, 2005). The UML consists of a set of diagrams and associated notations, where each diagram represents a different view of a software analysis and design model—structure, interaction, or state (Blaha &

Copyright © 2009, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

Rumbaugh, 2005; Dori, 2002). In the OOAD process, analysts develop a set of *objects*, where each object has a set of *attributes* (data that the object knows about) and *behaviors* (operations that the object can perform). Objects interact by invoking one another's operations to fulfill the high-level behaviors required of the system, representing the business logic of the to-be software system. The analyst captures this logic in a set of UML *sequence diagrams* (SDs). Figure 1 shows an example of such a diagram (all diagrams in this article were developed using Visual Paradigm, a software tool from Visual Paradigm International).

SDs are crucial building blocks in system design. Analyst errors in building them lead to significant rework efforts or serious software defects at implementation time, which in turn leads to increased costs. Thus, it is critical that analysts have the skills to produce high-quality SDs. *While sequence diagrams are among the most widely used diagrams in the UML in practice* (Dobing & Parsons, 2006, 2008; Fowler, 1997) *and are critical to the design process, they are difficult to learn to build* (Bolloju & Leung, 2006; Siau & Loo, 2006).

The OOAD process consists of five basic steps: (1) developing high-level requirements, (2) use case analysis, (3) domain data modeling, and (4) building sequence diagrams (George, Batra, Valacich, & Hoffer, 2006), and (5) building a class diagram. In steps (1) and (2), the analyst develops a description of *what the system should do*. In step (3), the analyst develops the *structure* of the system from a data perspective and, in step (4), defines *how the system should achieve the behaviors required*. In step (5), the analyst refines the structure of the system to include both data and behavior. Good, clear techniques and heuristics are available for steps (1), (2), (3), and (5) through many textbooks and other sources. Textbooks covering OOAD (George et al., 2006), as well as other sources (Fowler, 1997), offer a sizeable set of advice on step (4). Even with so many useful guidelines and heuristics, however, learning to build these diagrams remains difficult.

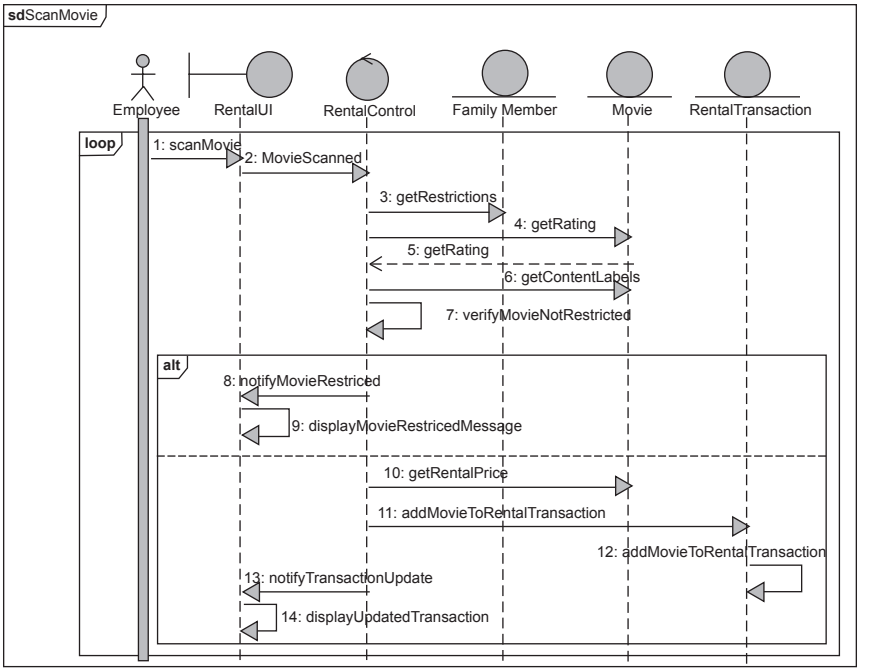
In this article, we consider the task of learning to build SDs. Although much thinking has been done on the problem of developing guidelines and heuristics for building a good SD, and much work has been done on the difficulties of learning OOAD, there has not been much discussion of the process of learning to build SDs in particular. This is our focus in this article. Specifically, we concentrate on two questions:

- What are the complexity factors inherent in the process of learning to build SDs?
- How can these factors be mitigated to minimize the cognitive load of learning to build SDs?

In this article, we apply Reeves' Learner-Centered Design (LCD) framework (Reeves, 1999) to the task of learning to build SDs. We chose

Copyright © 2009, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

Figure 1. Example UML sequence diagram



this framework for two major reasons. First, it provides both a way to describe the *cognitive complexity of learning to perform a task*, in this case learning to build SDs, as well as several *learner models* and *recommendations* for reducing cognitive loads for each learner type—making it directly applicable to our research question. Second, it is grounded in more than thirty years of research in cognitive science, information systems, and design theory, and thus provides a strong theoretical foundation for our work.

The two main contributions of this article are as follows: (1) *we present a qualitative analysis of cognitive complexity of learning to build SDs to identify the characteristics of the task that increase its difficulty; and (2) we develop a set of recommendations for mitigating the complexity of the task of learning to build SDs, based on the theory of LCD.* These recommendations focus primarily on how to present the SD-building task to beginners, rather than on redesigning the actual SD-building task itself.

EXAMPLE

We present an example based on a video rental store scenario. This serves as the basis for examples throughout the article. Suppose a portion of the requirements reads as follows:

Copyright © 2009, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

The rental chain allows families to give separate membership cards to all family members, and to restrict a given family member's access to movies by movie rating, e.g., a child in the family cannot rent an 'R' rated movie on his own, or content, e.g., a child in the family cannot rent a movie that is labeled with an 'L' for potentially objectionable language. At rental time, the software must enforce any restrictions associated with the family member presenting his membership card.

Based on these requirements, one might write a use case narrative similar to the one shown in Figure 2, and develop a partial class diagram similar to the one shown in Figure 3. Based on these diagrams, Figure 1 shows one possible solution SD for this scenario.

The remainder of this article is organized as follows. We first describe related work, and then provide an overview of cognitive complexity and learner-centered design. We then present an analysis of the cognitive complexity of learning to build sequence diagrams, and propose a set provide a set of recommendations aimed at reducing the complexity of learning to build SDs based on LCD principles. Finally, we conclude the article and describe future work.

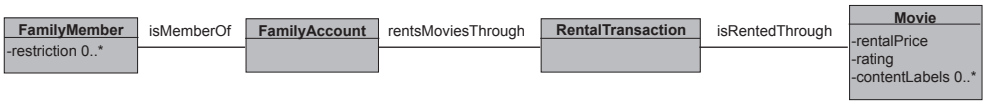
RELATED WORK

Compared to structural techniques (developed prior to OO methods), OOAD is cited as a more natural method of design (Rosson & Alpert, 1990). Booch (1986) proposed one of the earliest generalized OO design methods, providing a starting point for the development of what we know today as OOAD. Others refined and expanded these notions, defining key OO concepts, for example, reusability (Johnson & Fotte, 1988; Micallef, 1988). To support the OOAD process, several early modeling languages were proposed, for example, (Rumbaugh, Blaha, Premerlani, & Lorensen, 1991); over time, these languages were assimilated together and extended by others to produce the current UML specification (International Organization for Standardization, 2005).

Since we are interested in the complexity of UML SDs in this work, it is useful to consider other work describing UML complexity and related difficulties. As a language, UML is known to be complex (Siau & Cao, 2001; Siau, Erickson & Lee, 2005), as well as difficult to learn (Bolloju & Leung, 2006; Siau & Loo, 2006) and use (Agarwal & Sinha, 2003).

Siau & Cao (2001) present a quantitative analysis of the theoretical

Figure 2. Example use case narrative



Copyright © 2009, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

Figure 3. Example UML class diagram

User	System
1. For each movie the cardholder wishes to rent, the Employee scans the movie's barcode	2. For each movie, the System checks the cardholder's restriction set for any holds based on rating or content
	3. For each movie, the System retrieves the cost of rental for the movie, adds the movie to the rental transaction, and updates the total transaction cost
Exceptions: Step 2. If the desired movie violates the restrictions associated with the customer's membership card, the system displays a message "unable to rent this movie", along with the reason (which restrictions the movie violates). A movie violating the customer's restrictions is not added to the rental transaction list in step 3.	

mathematical complexity of UML. This study considered both individual diagrams as well as the UML in aggregate, comparing both the diagrams and the aggregate UML to other OO methods. This analysis concluded that, while individual diagrams are approximately as complex as other OO methods, the UML as a whole is 2 to 11 times more complex than other OO methods. In a follow-on study (Siau, et al., 2005), the authors begin to consider the comparative complexity among pairs of diagrams, looking specifically at class diagrams and use case diagrams, and finding that class diagrams are more complex than use case diagrams. Our work differs from this work in two ways. First, we are interested in the cognitive complexity of the task of learning to build SDS, rather than the structural complexity (defined mathematically) of the diagrams themselves. Second, we are interested in the complexity of SDs, rather than class or use case diagrams, or the UML as a whole.

Siau & Loo (2006) present the results of an empirical study aimed at identifying the factors that make learning UML difficult. This study surveyed students in OOAD courses to identify what they found difficult, using concept mapping to categorize the results. This study identifies several specific areas students found difficult in building SDs as well as other UML diagrams. In (Bolloju & Leung, 2006), the authors study a set of projects produced by OOAD students in a university setting to identify the typical mistakes that novices make in building the four most commonly-used diagrams in UML.

In Sheetz (2002), the author presents the results of an empirical study examining the difficulties professional developers experience when using OO techniques. This study surveyed several groups of developers, from novices to experts, to determine what aspects of OO development they found difficult. The subject matter scope of this study is much larger than our scope—this study considers not only technical issues (e.g., analysis, design, implementation), but also managerial and organizational issues (e.g., project estimation, managing user expectations, stakeholder buy-in), whereas our focus

Copyright © 2009, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

is on novice-level analysts learning to build SDs. The results detail specific aspects of OO development that specific experience groups found difficult (i.e., what novices found difficult, what intermediates found difficult, etc.), and what aggregated groups found difficult (i.e., what novices and intermediates found difficult, what intermediates and experts found difficult, etc.).

In (Agarwal & Sinha, 2003), the authors present a survey-based study of developers, focused on questions of UML usability and ease-of-use. Of particular interest here are the results for novice developers, which indicate that this class of developers found the UML diagrams difficult to use. This study advocates simplifying the UML to make it more user-friendly.

Work on teaching OOAD and UML also is related. In (Beck & Cunningham, 1989), the authors introduce one of the earliest studies of teaching OO techniques to students, focusing on OO thinking. More recent research (Brewer & Lorenz, 2003; Burton & Bruhn, 2004) considers how teaching UML along with OOAD can improve learning in OOAD overall. Tabrizi, Collins, Ozan & Li (2004) extend this work by proposing that OO concepts and UML should be integrated into beginning programming courses. In a similar vein, Wei, Moritz, Parvez & Blank (2005) advocate an automated tutoring method to provide immediate feedback to students while learning OOAD and programming.

Our work differs from these efforts in its focus and method. These studies focus on identifying *what is difficult within OOAD and UML* and *how OOAD education in general can be improved*, while we focus on *why it is difficult to learn, with a specific focus on SDs*. These studies are complementary to our work in that they identify the difficulties we will consider in our analysis of the cognitive complexity of learning to build sequence diagrams.

COGNITIVE COMPLEXITY AND LEARNER-CENTERED DESIGN

We base our analysis of the cognitive complexity of learning to build sequence diagrams on the Reeves model (Reeves, 1999) and develop our recommendations for reducing the complexity of learning to build SDs on LCD theory. We provide overviews of cognitive complexity and LCD in this section.

Cognitive Complexity

Reeves' model of cognitive complexity considers different sources of complexity in learning to perform a task with the aim of identifying the concrete factors that make it difficult to understand. By exposing these factors, we can then modify the task to reduce the complexity of learning to perform it using the principles of learner-centered design.

Reeves' model considers multiple different potential sources of cognitive complexity, some of which, for example metasocial forces, are not relevant to the topic at hand, and are omitted from this

Copyright © 2009, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

discussion. We consider each of the relevant potential sources of complexity.

Information about the Task

The information available regarding a task falls into two categories: (1) *process-related information* about how to perform the task in general; and (2) *domain-specific information* about the task at hand. In the context of learning to build SDs, process-related information refers to the specific steps to follow in building an SD, while domain-specific information refers to the actual problem scenario to be modeled, as described in a high-level requirements document.

The level of cognitive complexity arising from information sources is dependent on the *quantity* of information as compared to the *utility* of the information presented. Not all information-based complexity is bad—considering additional complexifying information can lead to a higher-quality design. However, complexifying information that the task performer cannot use to improve the design is simply noise, where increasing noise leads to increasing difficulty in selecting the useful information, and hence additional cognitive complexity.

The Design of the Task

We consider the difficulties introduced by the task itself, in terms of the process steps and guidelines provided to lead the analyst from the problem formulation to the goal solution. Generally, the design of a task introduces cognitive complexity if it exhibits one or more of the following characteristics.

- **The task process does not provide enough information for task performer to build a complete mental model.** A mental model of a task is a framework that allows the task performer to deduce new information about the task or predict the future effects of a choice made during the task. With an incomplete mental model, the task performer may make incorrect inferences about the task.
- **The next step in the process is not always evident.** If a task process is not sufficiently detailed, and the task performer must guess at the next step in the process, the probability of error is high.
- **A lack of constraints among choices forces the task performer to choose from too many options.** As the number of possible choices increases, the task performer must attempt to search through the option space for the optimal choice. As the size of the option space increases, the difficulty for the task performer in finding the optimal choice increases significantly.

In each of these cases, the task performer is forced to make decisions without an adequate basis for the choice, making each decision more difficult and thus introducing cognitive complexity.

Copyright © 2009, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

Problem Solving Within the Task

Problem-solving refers to the process of transforming a problem formulation into a goal solution. In the context of this article, the problem faced by the novice analyst is to produce an SD that is both correct and of a high quality. The complexity of solving a problem can be defined along multiple dimensions, where a problem formulation can vary from simple to complex along each dimension. A problem becomes less complex as:

- **Sufficient information is provided** about the problem. A lack of crucial information increases the cognitive complexity of solving a problem. The simplifying aspect of additional information applies only so far as the added information is useful (as described above).
- **The goal is defined more precisely.** The less clearly the expected solution is defined, the more difficult it is for the learner to work toward the goal solution.
- **There are fewer variables, and fewer interrelationships and dependencies among them.** Larger numbers of variables and increasing numbers of relationships among those variables leads to increased numbers of decisions the problem-solver must make in working toward a goal solution, increasing the potential for errors.
- **Significant expertise is not required to solve the problem.** The more experience a problem-solver requires to reach a quality solution to a problem formulation, the more complex it is for the novice to approach. Essentially, the novice must develop expertise before being able to confidently produce quality solutions.
- **There are fewer possible solutions.** A larger number of potential solutions makes a problem more difficult to solve, particularly when some solutions are of higher quality than others, or when there is a trade-off between competing priorities.
- **Logic and known patterns/expertise from other domains can be applied to solve the problem.** The applicability of existing expertise from other domains tends to simplify the problem-solving task.

Learner-Centered Design

Learner-Centered Design theory presents a set of design principles aimed at reducing the cognitive complexity of learning how to perform a task through the redesign of the task itself. Once we have determined the root causes of complexity, we can apply the relevant LCD principles to ensure that the learner experiences as little confusion as possible.

In his LCD theory, Reeves (1999) proposes several models of learners, classified by the goal of the learning process at hand. For example, the *learner as categorizer* model considers ways to help learners filter and categorize

Copyright © 2009, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

large amounts of information, while the *learner as searcher* model suggests ways to help learners search through an information space to identify high-quality, useful and relevant information. For each model, he then proposes a set of techniques for reducing cognitive complexity for tasks that fit the model.

The learner model most useful to the problem of helping beginner analysts learn to build SDs is the *learner as expert problem solver* model. Effectively, we can think of the analyst's progress from novice to intermediate to expert SD-builder as one of *building expertise*.

Reeves (1999) defines the characteristic difference between an expert problem solver and a novice problem solver can be summarized as follows: novices have small amounts of information of varying quality in a loose organization, while experts have a high quantity of highly relevant information in an intricate web of interconnections. Empirical studies (Wiedenbeck, Fix, & Scholtz, 1993; Wiedenbeck, Ramalingam, Sarasamma, & Corritore, 1999) show that the mental models for novice and expert programmers differ significantly, where an expert typically has built an intricate, pattern-oriented mental model that the novice lacks.

The question that arises in terms of LCD for the task of building SDs is this: *How can we help the novice analyst (a) build a strong knowledge base; (b) filter out low-quality information; and (c) build a mental model of the task with high-quality relationships among information elements?*

To help reduce the complexity of developing expertise in problem solving, Reeves suggests several recommendations for organizing the relevant material:

Provide Scaffolding

We can think of the "scaffolding" here as providing a knowledge framework upon which a learner can learn while gaining expertise, as a way to help the user climb the learning curve. This suggests organizing content to build on the learner's accumulated knowledge.

Decompose and Recompose

Smaller problems are easier for the learner to solve than larger ones. Solving a large problem becomes easier if it can be broken down into smaller problems, where the solutions to the smaller problems can be combined into a complete solution.

Use Examples and Exercises Extensively

A novice develops expertise by reasoning from the specific to the abstract. Exercises and examples, provided each exposes some new information or variation, add new knowledge to the learner's mental model of the task. The larger the number of exercises and examples the novice encounters, the more opportuni-

Copyright © 2009, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

ties there will be to build a better abstract task model, and the greater the resulting expertise level will be.

Engage the Learner Actively

Engaging learners actively, through spoken questions and interactive exercises, forces them to "think on their feet," which helps the learner actively construct a mental model of the task. This type of interaction not only motivates the learner to think in the desired direction, but it also allows the educator/mentor to determine the learner's current level of expertise and provide feedback in real time.

ANALYSIS OF THE COGNITIVE COMPLEXITY OF LEARNING TO BUILD UML SEQUENCE DIAGRAMS

In this section, we enumerate several characteristics of the SD-building task that make learning to build SDs difficult for the beginner analyst. These characteristics were drawn primarily from work in identifying typical errors and difficulties in OOAD and programming, as described in our earlier discussion of related work.

For the purpose of this analysis, we consider the beginner to be a student with no prior knowledge of OOAD or programming experience (a worst-case scenario). We also assume that the student has progressed in the course to the point where SDs arise—after use case analysis and domain data modeling discussions. For students with some prior experience, some parts of this discussion may not apply. We also assume that the learner has the basic building blocks for an SD (i.e., requirements, use case narratives, domain data model) in place at the start of the SD building task.

For each characteristic, we: (1) describe why it makes the task difficult; (2) identify the type of cognitive complexity associated with the difficulty, and (3) discuss whether the complexity stems from the design process in general, or from SDs in particular. We then apply the LCD principles to recommend ways to mitigate the complexity associated with these characteristics.

Unfamiliar Metaphors

A number significant metaphor in SD syntax and semantics refers to OO programming, including concepts such as message passing, parameters, and returns. However, not every student learning OOAD techniques has been exposed to programming.

Students without prior programming experience increased complexity in learning SDs, primarily coming from problem-solving complexity—knowledge from domains other than programming cannot be applied; rather some level of expertise is required. This complexity is not related to the design process, but is introduced by the SDs themselves.

Copyright © 2009, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

Working with SD Syntax

SDs have a detailed and very specific syntax. At every step in the SD-building process, the student's choice of syntactic element makes a specific statement about the design of the to-be software. Message arrows are a good example of this—should the arrow line be solid or dashed? Each has its own specific meaning, respectively, message call or return. A solid-line arrow placed on an angle has yet another meaning. Modeling alternative flows is another example. SDs are inherently linear diagrams, but if-then-else branching is a very common software structure in practice. One method of handling this within the SD syntax is to use alt boxes. If the logic within each branch is complex, these alt boxes quickly become large and very cumbersome.

Working with the SD syntax creates a combination of task design complexity and information complexity: the student must apply a complex set of syntax rules from the very outset of learning to build SDs. This complexity stems from SD syntax, not the overall design process.

Omitting Explicit Returns

The notion of flow of control through the logic is not evident in an experienced analyst's SDs, since explicit return notations are often omitted to reduce clutter and fit more logic into a given space. If returns are added to the diagram, one can trace the flow of logic through the SD with a pencil as the focus of control moves from one object to another.

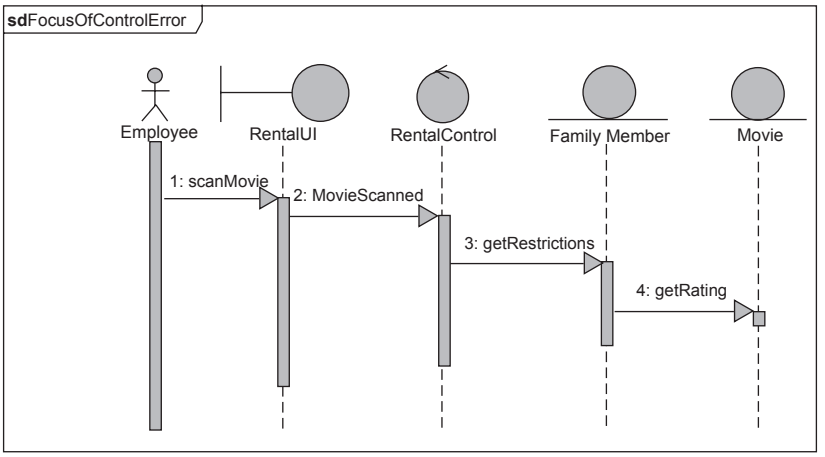
A student who omits these returns may mistakenly leave the focus of control at the recipient object, rather than returning it to the calling object, once processing is complete. As a result, the novice will often mistakenly assign responsibility for initiating some processing (i.e., sending a message to another object), to an inappropriate object.

For example, Figure 4 shows a fragment of an SD representing the first few interactions of the main video store checkout SD (shown in Figure 3). Here, after the *RentalControl* object asks the *FamilyMember* object for its restriction set, rather than implicitly returning its restriction set and the focus of control back to the caller object, the *FamilyMember* object instead sends a message to the *Movie* object asking for its rating. While this is not necessarily incorrect, this places the responsibility for obtaining the movie's rating on the *FamilyMember* object. This is likely not the analyst's intention, given that the *FamilyMember* object was only asked to return its restriction set, and no more.

The requirement that the focus of control should return to the calling object comes directly from programming, where an explicit return call at the end of each method returns the focus of control to the caller. A student, however, may have little or no exposure to coding. When building the first few SDs, then,

Copyright © 2009, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

Figure 4. Focus of control remains on the called object



a novice who is not advised to explicitly include returns may apply a more familiar "connect-the-dots" metaphor.

Here, we can see problem-solving cognitive complexity issues arising: the novice analyst cannot apply familiar metaphors; rather, some domain-specific knowledge of programming is required. Task design complexity is also evident: without this domain knowledge, the novice cannot build an appropriate mental model of the task, so the next step the novice should take is not evident. This complexity is due to SD-building conventions in practice, and not to OO design in general.

Decomposing High-Level Requirements into Logic Steps

For a given step in the use case narrative, the analyst must decompose the system-side requirements into more detailed responsibilities to define the step-by-step business logic. For example, consider the use case narrative fragment for step 2 as presented in Figure 2. Here, some object in the system must be responsible for (1) obtaining the movie's rating and content labels, (2) obtaining the restriction set for the cardholder, and (3) comparing the movie rating and content labels to the cardholder's restriction set to determine if the cardholder should be permitted to rent the movie.

This is a logical jump in abstraction from a lower granularity of detail to a higher granularity of detail, where the student analyst must add detail not explicitly provided in the requirements document or use case narrative. This can seem like making up information. A more experienced analyst, however,

Copyright © 2009, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

will recognize that the design process inherently involves adding new detail at each stage, and that not all detail can be mapped directly back to explicit statements from documents developed earlier in the OOAD process.

Here, we can see cognitive complexity arising from the design of the SD-building task in two ways. First, the next step in assigning responsibilities is not always obvious, since the high-level requirements often do not explicitly enumerate the low-level steps of the business logic. Second, this lack of explicit step-by-step logic leaves the analyst free to choose among all available options with very few constraints. This complexity is part of the design process, as has been noted in previous work (Sheetz, 2002), but arises noticeably in SD-building process.

Choosing to Centralize or Decentralize Responsibility Assignments

While the responsibilities for knowing information are relatively easy to assign (e.g., the *Movie* object should be responsible for answering queries about its own rating), it is less obvious which object should be responsible for initiating processing.

We consider step 2 in Figure 2, where we can decompose the required behavior of the system into three responsibilities: (1) obtaining the movie's rating and content labels, (2) obtaining the restriction set for the cardholder, and (3) comparing the movie rating and content labels to the cardholder's restriction set to determine if the cardholder should be permitted to rent the movie.

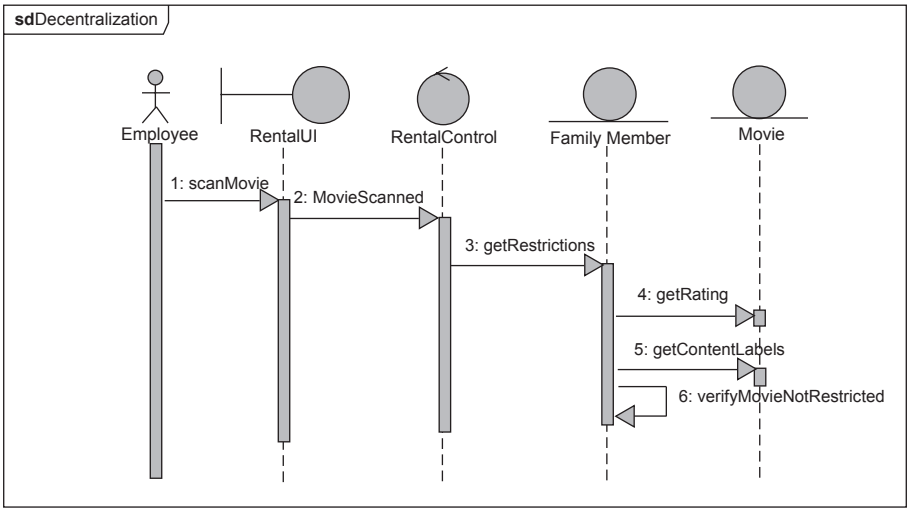
Which object should be responsible for initiating this processing? Many introductory texts suggest assigning this responsibility to a centralized controller object, which contains the main flow of logic for responding to a specific external event, and make all other objects responsible only for providing information about their attributes or doing simple calculations. Figure 1 shows an example of centralization of responsibility for initiating processing, where the *RentalControl* object is assigned these responsibilities.

Centralization doesn't necessarily represent good OO design because it places virtually all the responsibility for initiating processing on a single object (the controller object). When implemented, the code for the centralized controller class would likely be significantly more complicated than that of other objects. Assigning entity objects responsibility for initiating messages is not necessarily incorrect. In fact, such messaging assignments can lead to a more decentralized assignment of responsibilities.

Consider, for example, Figure 5, which shows a decentralized logic flow. Here, the *RentalControl* object asks the *FamilyMember* object to determine whether or not the movie is acceptable to rent based on the movie's ID code. This places the responsibility for obtaining the movie's rating and content information, as well as checking this information

Copyright © 2009, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

Figure 5. Decentralization example



against the cardholder's restriction set, on the *FamilyMember* object, rather than the controller object.

This freedom presents the beginner with a wide spectrum of choices from which to choose, increasing the problem-solving complexity in terms of the design of the task. This complexity is a part of OO design in general, but arises mainly in the specific context of SDs.

Choosing among Multiple Candidate Solutions

The large number of choices available when assigning responsibilities results in a large number of candidate solutions. While many of these solutions might be incorrect in terms of syntax or semantics, many will also be acceptable as solution SDs. The fact that multiple solutions are possible introduces problem-solving complexity into the SD-building task in that the goal is not precisely defined—there are no specific guidelines in the literature that will guarantee that the analyst will be able to narrow the candidate solution pool to a single SD, particularly when there is more than one acceptable solution. Here, the novice analyst may recognize that there are multiple possible acceptable solutions, but may lack the expertise to recognize that one solution is better than another, or that two solutions are equally good. The lack of a precise goal definition introduces uncertainty into the process. This issue is due to both the characteristics of SDs as well as the overall design process. Design, by its very nature, does not presuppose a goal. This is further complicated by the fact that minor differences in SD syntax choice can lead to major quality and correctness differences between candidate solutions, where a novice may have

Copyright © 2009, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

difficulty identifying the most correct or highest quality design.

Rate of Information Presentation

Most OOAD texts provide a wealth of information about SDs. As noted in the previous sections, many characteristics of the SD-building process are sources of cognitive complexity. The beginner analyst must digest a large amount of SD-building information in a short time. Here, information is a significant source of cognitive complexity, in the sense that the novice analyst is unlikely to be able to internalize and make use of all the information on building SDs at the same time.

For example, an analyst building a first SD might be able to model only a very simple scenario using a few guidelines. At this level of expertise, providing sophisticated SD-building heuristics introduces significant cognitive complexity because the novice is not prepared to incorporate them, but feels as though it is required.

With more experience in building SDs, the student becomes better prepared to apply more complicated techniques and model more detailed scenarios. While the new information does increase the complexity of the task, this complexity is useful because it will generally result in more detailed and higher quality SDs, and better overall designs. This complexity is due primarily to the overall complexity of SD building, rather than design in general.

Designing for Quality

The quality of an SD is difficult for a novice to evaluate. For example, the notions of coupling and cohesion, which refer to the quality of an overall design (encompassing not only SDs, but also class and other diagrams), are often introduced in texts, for example, (Stumpf & Teague, 2005) aimed at the beginner analyst in the course of the discussion of building SDs. These concepts are difficult for the novice to apply, especially early in the analyst's introduction to OOAD. These concepts are subject to the "too much information too quickly" cognitive complexity issue.

Once the analyst is prepared to approach coupling and cohesion, there is additional complexity introduced by the relationship between the two concepts. Designing for low coupling tends to increase responsibility assignments to an object, while designing for high cohesion tends to reduce them. Thus, there is a clear trade-off between coupling and cohesion—reducing coupling tends to increase cohesion and vice versa. This introduces interrelationship and dependency-based problem-solving complexity in that the novice analyst must find a good balance between the two properties. Generally, quality is an issue for the overall design process, but the issue is exacerbated by the complexities associated with building SDs.

Copyright © 2009, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

The Cumulative Effect

To see the overall effect of these complexity factors, let us consider what happens when a student analyst is actually building an SD. At each step, the student must be able to answer several questions:

- What is the next bit of behavior/processing required?
- Have I broken down the use case narrative step into sufficient detail, or have I glossed over something important? Have I missed any dependencies?
- Where is the focus of control right now? Is it in the correct place? Have I incorporated all returns required?
- Does the current focus of control match the next caller object? If not, how do I get the focus to the correct object?
- When I add the message for the next bit of behavior, am I using the correct syntax and semantics? Dashed or solid line arrow? Many syntax-related questions are possible here.
- Have I given the message an appropriate name, one that fully describes and limits the scope of responsibility? Does the called object fully perform that scope of responsibility, and no more?
- Does the next bit of behavior represent the start of if-then-else processing (needing an alt box)?
- If the current focus of control is inside an alternative within an alt box, does the next bit of behavior belong inside or outside the alt box?

At this point, the student is rapidly approaching Miller's (Miller, 1956) "seven plus or minus two" limit on information processing capacity, but hasn't yet considered the full scope of possible syntax- and semantics-related questions, or considered design quality.

OVERCOMING THE COGNITIVE COMPLEXITY OF BUILDING SEQUENCE DIAGRAMS

We introduce a set of recommendations aimed at reducing the cognitive complexity of learning to build SDs based on the LCD principles described earlier.

Build on the learner's knowledge level. As described earlier, information presented to the novice analyst before it can be processed and applied is not useful; however, as the student gains experience, additional information can be helpful and increase the quality of SDs.

In the context of learning to build SDs, we suggest focusing on different dimensions of SD-quality separately, from simpler to more complicated measures of SD-quality. Here, we suggest thinking about the quality of an SD using a (partially) language-based model similar to that described in Lindland, Sindre & Solvberg (1994). Specifically, we can

Copyright © 2009, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

think of the quality of an SD as varying along three general dimensions:

1. **Syntax quality:** how well an SD conforms to the rules for using SD notation.
2. **Semantic quality:** how well an SD conforms to the scenario described in the high-level requirements, use case narratives, and any domain knowledge available.
3. **Design quality:** how well an SD can be executed in a high-quality software implementation (e.g., using concepts like coupling and cohesion).

In order to reduce the complexity associated with giving the beginner too much information too quickly, we suggest that SD-related information be presented first focusing on syntax, then on semantics, and finally on quality. Here, syntactic quality provides a foundation for the other quality dimensions—if the SD's syntax is not correct, it will never have high semantic or design quality. Similarly, once the novice has mastered SD syntax, if the SD does not match the requirements document (semantic quality), there is no point in thinking about design quality.

This recommendation makes use of the LCD principle of *scaffolding* to directly address the cognitive complexity associated with the rate of information presentation and evaluating design quality, and help to address the other issues by assuring that information is not presented before the learner is ready to use it.

Provide a Wide Variety of Exercises for Each Stage

To support the development of varying dimensions of quality (syntax, semantic, and design quality), we can reduce the complexity of developing skills in each area by providing focused exercises specific to each dimension of quality.

For example, an exercise focusing on syntax may provide both an ordered set of logical steps, as well as a mapping that describes which object should perform each step, and ask the novice to simply draw the diagram. Similarly, an exercise focusing on semantics may provide a flowchart of the logic and a set of objects participating in the SD, and ask the student to assign responsibilities to objects. A design quality-focused exercise might ask a student to compare and contrast two SDs in terms of coupling and cohesion, or to modify an SD to improve its design quality.

This recommendation makes use of the LCD principles of *scaffolding* and *extensive use of exercises* to address the cognitive complexity associated with many of the SD-building complexity issues, including unfamiliar metaphors, working with SD syntax, omitting returns, decomposing high-level requirements, choosing between centralization and decentralization, choosing among candidate solutions, and quality evaluation.

Copyright © 2009, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

Consider Intermingling the OOAD and OO Programming Learning Experiences

The programming roots of some SD syntax and semantic elements introduce complexity for some students, particularly those with little or no previous exposure to programming. This is a bit of a chicken-and-egg problem—the student needs to become familiar with these concepts, but the question arises: teach programming or design first? One proposal (Tabrizi, et al., 2004) suggests incorporating design and programming into single cohesive learning experience. This allows the student to see the goal of the design process—finished software (and all that it takes to turn design into implementation)—while concurrently building expertise in design.

This uses a form of scaffolding, making the end goal clear, and potentially enabling the student to foresee the consequences of design decisions in the implemented software, and helps to address the cognitive complexity associated with unfamiliar metaphors, working with SD syntax, omitting returns, decomposing high-level requirements, and choosing between centralization and decentralization.

Separate Logical Flow Design from Responsibility Assignment

Many SD-building guidelines suggest working through the logic required for a use case and assigning responsibility for these logical steps at the same time. This requires the analyst to have a clear view of both the data structure of the to-be software as well as a strong sense of the detailed logic required to satisfy the use case scenario, and to be able to map these logical steps to the objects that will be responsible for them—all at once.

One way of reducing the complexity associated with this is for the beginner analyst to separate out the task of detailed logical flow development from that of responsibility assignment. This is useful when the use case steps require further decomposition into system-internal logic steps, because it allows the novice analyst decompose a less-detailed logic description in the use case into more detailed logical steps without worrying immediately about which object will be responsible for a given step.

Many texts (Fowler, 2004; George, et al., 2006) suggest taking advantage of any available useful tools in building analysis and design models, even if those tools are not part of the UML. In separating logic from responsibility assignment, a useful tool for the novice analyst is the familiar *flowchart*. This allows the analyst to map out individual logical steps, including conditional and repeating logic, without thinking about specific objects. Once this is done, the student can then consider responsibility assignment for each step.

This recommendation makes use of the LCD principles of *scaffolding* and *decomposition and recomposition*, by suggesting the separation of logic

Copyright © 2009, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

and structure, to address the cognitive complexity associated with decomposing high-level requirements into logic steps.

Actively Think the Problem Through

Another potential mechanism for reducing the complexity of the responsibility-assignment task, one that would work best primarily in an educational setting, is to have a group of students role-play to work through which object should be responsible for which logical steps. Here, each student is assigned a role as an object, and given a list of the object's attributes. A token of some sort represents the focus of control, and students must ask one another to perform tasks, passing around the token, to fulfill the requirements of a particular use case. (The authors have applied this technique in the classroom with good results.)

This approach allows multiple student analysts to contribute to the group's common model of the logic flow and responsibility assignment set, reducing the cognitive load on any given individual, while simultaneously ensuring that all students are engaged in the learning process.

For a large group of students, it is possible to ask the larger group to break the problem into smaller pieces. Smaller groups can then attack each sub-problem, and the group as a whole can then integrate the solutions to the sub-problems into a larger solution to the whole problem.

This recommendation makes use of the LCD principles of *engaging the learner actively*, *using exercises extensively*, and *decomposition and recomposition* to address the cognitive complexity associated with several of our SD complexity characteristics: decomposing high-level requirements into logic steps, choosing among multiple candidate solutions, assigning responsibilities, and choosing between centralization and decentralization.

Document and Use Patterns

Applying a pattern is a well-known way of reusing previous work and reducing the overall workload of a task. Patterns can reduce the cognitive complexity of the SD-building process for an analyst who has mastered SD syntax and semantics by providing a starting point of a known quality design for a standard organizational scenario (e.g., the workflow for processing paychecks). While a pattern might not apply completely to a given scenario, it can still provide a modifiable starting point to the analyst that would otherwise have required significant thought to produce from scratch.

Design patterns have been discussed in the literature (France, Kim, Ghosh & Song, 2004), and are cited in some textbooks, for example Stumpf and Teague (2005). These abstract patterns provide a basis for solving particular design patterns (e.g., using a singleton class to represent an interface), and are meant to provide the experienced designer

Copyright © 2009, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

with generally applicable solutions. For a novice, however, applying these patterns presents a major challenge: the patterns are described at a level of abstraction that makes it difficult for the novice to recognize where they can be applied, partially due to the fact that these patterns have their origins in coding design patterns.

Fowler (1997) introduced the idea of detailed domain-specific analysis patterns. This book contains a set of highly-detailed analysis patterns describing common situations from accounting and health-care domains, and provides data and behavior models for each situation. This book was written prior to the development of UML, so the models in this book are not built using the familiar UML syntax. However, the idea of reuse of detailed analysis-stage models holds. The availability of a set of tried-and-true analysis patterns, covering a large number of domains, could significantly reduce the complexity of designing logic for a frequently-encountered domain.

This recommendation makes use of the LCD principle of the *use of examples* to address the cognitive complexity associated with several SD-building complexity issues: omitting returns, decomposing high-level requirements, choosing objects when assigning responsibilities, choosing between centralization and decentralization, and quality evaluation.

CONCLUSION

In this article, we have considered the question of why SDs are difficult to learn to build. We have described a set of characteristics of the SD-building task that add complexity to the task, and identified why each of these adds difficulty using concepts from cognitive complexity theory. Further, we developed a set of recommendations aimed at reducing the difficulty associated with the task of learning to build SDs by applying concepts from the theory of LCD. We believe these recommendations will be of interest to educators, experienced analysts mentoring novices, and others.

Future work includes empirical testing to determine the efficacy of our recommendations in practice, as well as developing a set of reusable foundation SDs that provide novice analysts with a strong starting point for learning to build their own SDs.

REFERENCES

Agarwal, R., & Sinha, A. (2003). Object-oriented modeling with UML: A study of developers' perceptions. *Communications of the ACM, 49*(9), 248-256.

Beck, K., & Cunningham, W. (1989). A laboratory for teaching object oriented thinking. *Proceedings of the 1989 ACM OOPSLA Conference on Object-Oriented Programming*, (pp. 1-6).

Copyright © 2009, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

Blaha, M., & Rumbaugh, J. (2005). *Object-oriented modeling and design with UML, Second Edition.* Pearson Prentice Hall.

Bolloju, N., & Leung, F. S. (2006). Assisting the novice analyst in developing quality conceptual models with UML. *Communications of the ACM, 49*(7), 108-112.

Booch, G. (1986). Object-oriented development. *IEEE Transactions on Software Engineering, 12*(2), 211-221.

Brewer, J., & Lorenz, L. (2003). Using UML and agile development methodologies to teach object-oriented analysis and design tools and techniques. *Proceedings of the 4th Conference on Information Technology Education*, (pp. 54-57).

Burton, P., & Bruhn, R. (2004). Using UML to facilitate the teaching of object-oriented systems analysis and design. *Journal of Computing Sciences in Colleges, 19*(3), 278-290.

Dobing, B., & Parsons, J. (2006). How UML is used. *Communications of the ACM, 49*(5), 109-113.

Dobing, B., & Parsons, J. (2008). Dimensions of UML diagram use: A survey of practitioners. *Journal of Database Management, 19*(1), 1-18.

Dori, D. (2002). Why significant UML change is unlikely. *Communications of the ACM, 45*(11), 82-85.

Fowler, M. (1997). *Analysis patterns: Reusable object models.* Addison-Wesley Pearson Education.

Fowler, M. (2004). *UML distilled third edition: A brief guide to the standard object modeling language.* Addison-Wesley Pearson Education.

France, R., Kim, D.-K., Ghosh, S., & Song, E. (2004). A UML-based Pattern Specification Technique. *IEEE Transactions on Software Engineering, 30*(3), 193-206.

George, J. F., Batra, D., Valacich, J. S., & Hoffer, J. A. (2006). *Object-oriented systems analysis and design, second edition.* Prentice Hall.

International Organization for Standardization. (2005, April). *ISO/IEC 19501:2004 -- Unified Modeling Language (UML).* Retrieved from http://www.iso.org.

Johnson, R. E., & Fotte, B. (1988). Designing reusable classes. *Journal of Object-Oriented Programming, 1*(2), 22-35.

Lindland, O., Sindre, G., & Solvberg, A. (1994). Understanding quality in conceptual modeling. *IEEE Software, 11*(2), 42-49.

Micallef, J. (1988). Encapsulation, reusability, and extensibility in object-oriented programming languages. *Journal of Object-Oriented Programming, 1*(1), 12-36.

Miller, G. (1956). The magical number seven, plus or minus two: Some limits

Copyright © 2009, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

on our capacity for processing information. *The Psychological Review, 63*(2), 209-227.

Reeves, W. (1999). *Learner-centered design: A cognitive view of managing complexity in product, information, and enviromental design.* Sage Publications.

Rosson, M. B., & Alpert, S. R. (1990). The cognitive consequences of object-oriented design. *Human-Computer Interaction, 5*(4), 345-379.

Rumbaugh, J., Blaha, M., Premerlani, W., & Lorensen, W. (1991). *Object-oriented modeling and design.* Englewood Cliffs, NJ: Prentice Hall.

Sheetz, S. (2002). Identifying the difficulties of object-oriented development. *Journal of Systems and Software, 64*(1), 23-36.

Siau, K., & Cao, Q. (2001). Unified Modeling Language (UML) -- A complextiy analysis. *Journal of Database Management, 12*(1), 26-34.

Siau, K., & Loo, P.-P. (2006). Identifying difficulties in learning UML. *Information Systems Management, 23*(3), 43-51.

Siau, K., Erickson, J., & Lee, L. (2005). Theoretical vs. practical complexity: The case of UML. *Journal of Database Management, 16*(3), 40-57.

Stumpf, R., & Teague, L. (2005). *Object-oriented systems analysis and design with UML.* Pearson Prentice Hall.

Tabrizi, M., Collins, C., Ozan, E., & Li, K. (2004). Implementation of object-orientation using UML in entry level software development courses. *Proceedings of the 5th Conference on Information Technology Education*, (pp. 128-131).

Visual Paradigm International. (2006). Retrieved from Visual Paradigm for UML: http://www.visual-paradigm.com

Wei, F., Moritz, S., Parvez, S., & Blank, G. (2005). A student model for object-oriented design and programming. *Journal of Computing Sciences in Colleges, 20*(5), 260-273.

Wiedenbeck, S., Fix, V., & Scholtz, J. (1993). Characteristics of the mental representations of novice and expert programmers: An empirical study. *International Journal of Man-Machine Studies, 39*(5), 793-812.

Wiedenbeck, S., Ramalingam, V., Sarasamma, S., & Corritore, C. (1999). A comparison of the comprehension of object-oriented and procedural programs by novice programmers. *Interacting with Computers, 11*(3), 255-282.

Copyright © 2009, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

Debra VanderMeer is an assistant professor at in the College of Business at Florida International University. Her research interests focus on applying concepts developed in computer science and information systems to solve real-world problems. She is widely published in well-known journals, such as Management Science, ACM Transactions on Database Systems and IEEE Transactions on Knowledge and Data Engineering*, as well as prestigious conference proceedings, including the International Conference on Data Engineering, International Conference on Distributed Computing Systems and the Very Large Database Conference. She also has significant professional experience in the software industry. She has served in software engineering and managerial roles in large companies, as well as early-stage venture-funded software enterprises.*

Kaushik Dutta is an assistant professor at in the College of Business at Florida International University. He has several years of professional and research experience in the fields of electronic commerce and enterprise IT infrastructure. Dr. Dutta has published articles in INFORMS Management Science, INFORMS Journal of Computing, ACM Transactions on Database Systems, European Journal of Operations Research, INFORMS Journal of Group Decision and Negotiation, IEEE Transactions on Systems, Man, and Cybernetics, VLDB Journal, IEEE Internet Computing and IEEE Transactions in Mobile Computing*. Dr. Dutta also has several publications in various IEEE and ACM conference proceedings. He has received awards for conference papers and academic performance, and was a finalist in Pennsylvania State University's e-BRC Doctoral Proposal Award (2002). Dr. Dutta has received several college-wide research award at FIU. Prior to joining FIU, Dr. Dutta was Director of Engineering for Chutney Technologies, a software company that developed solutions to improve the scalability and performance of enterprise Web applications. Dr. Dutta has almost a decade of experience in software product development in India, Europe and the U.S.*

Copyright © 2009, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.