Using Software Engineering Approaches to Model Dynamics in Interactive Software Systems

Mikael B. Skov and Jan Stage
Laboratory of Human-Computer Interaction, Department of Computer Science
Aalborg University
Fredrik Bajers Vej 122
9220 Aalborg Øst, Denmark
{mskov, jans}@intermedia.auc.dk

1. Introduction

Software engineering comprises all aspects of designing and implementing computer-based systems. Pressman (1992) defines software engineering as the establishment and use of sound engineering principles in order to obtain economically software that is reliable and works efficiently on real machines. Tradition has formed a classical life cycle for software engineering to consist of systems engineering, analysis, design, code, testing, and maintenance, cf. (Pressman, 1992; Sommerville, 1992). Modelling and constructing the future computer-based system take primarily place in the analysis, design, and coding phases where the objective is to describe the future computer-based systems in terms of software components. Various methodologies have proposed specific solutions to these three phases like recent object-oriented methodologies, cf. (Booch ,1994; Coad and Yourdon , 1991a; Coad and Yourdon, 1991b; Jacobson et.al., 1992; Martin and Odell, 1993; Rumbaugh et.al., 1991; Shlaer and Mellor, 1988; Shlaer and Mellor. 1992; Wirfs-Brock et.al., 1990). These methodologies have primarily been designed and created upon experiences from the development of traditional computer-based systems, e.g. administrative systems such as a bank account system.

With the evolvement of hardware technologies, new types of computer-based systems have emerged during the last years. One of them being multimedia systems. Multimedia systems share many similarities with other kinds of computer-based systems. Yet there are also fundamental differences. Compared to typical software systems, multimedia systems are often more interactive and involve a considerable amount of elements that produced in other ways. These elements provide content in the multimedia system, and examples are graphics, sound, video clips etc. They are often denoted as the assets of a multimedia system. A high level of interaction between the user and the system is characteristic to most multimedia systems. The system process and presents a large amount of information to the user by means of several media, and the user is expected to continuously respond to that information. It has been suggested that multimedia system development should start by defining the fundamental requirements to the system in question. After this initial phase, development should then proceed in two parallel tracks, where the first track deals with development of software, and the second track with production of

assets. Finally, the last phase should involve integration of assets into the software system.

The development of multimedia systems can be viewed from different design perspectives, e.g. moviemaking, storytelling, or as a software engineering design process (Eriksen, Skov and Stage, 2000). In this chapter we hold the latter view. Research within multimedia systems development shows that contemporary multimedia systems are designed and created primarily by intuition and, thereby, lacking method support and systematic approaches to work practices, cf. (Sutcliffe and Faraday, 1994; Sutcliffe and Faraday, 1997). A fundamental lesson learned through many studies and experiments in software engineering is that improvements in design and quality assurance processes require systematic work practices that involve well-founded methods Sommerville (1992).

In this chapter, we focus specifically on the software aspect of multimedia system development. Skov and Stage (1996) state that software designers who employ software engineering techniques and notations during the modelling process are able to gain and express a fundamental understanding by means of static, structural diagrams, but at the same time they face a key challenge: it is difficult to understand and design the dynamic collection of objects that are collaborating during the execution of a software system and what properties the system as a whole will possess. In the development of software systems with a strong element of interaction, like multimedia systems, this challenge becomes a key characteristic of the design activity. A high level of interaction increases the dynamic nature of the collection of objects and their collaboration. Probably the design and implementation of multimedia systems is a fundamental challenging process with a particular focus on the dynamics of the interaction. For these reasons we have set up two questions to be answered in this chapter:

- What are the key characteristics and challenges of designing multimedia systems?
- How does experienced software designers work with the task of designing the dynamic element of an interactive software system?

In the first question, we will address the characteristics and complexities imposed by the development of multimedia systems in order to set up an experiment, which addresses the second questions. The chapter provides qualitative interpretations from an empirical study of three software designers. Section 2 addresses the first question by discussing key characteristics of typical multimedia systems. The next section provides an overview of the empirical study; the complete documentation is available on the WWW (Skov and Stage 1995). The results are presented in the following four sections where each section presents a key observation that in part answers the second question raised above. Finally, the last section summarizes the lessons learned from this limited study and points out avenues for further research.

2. Designing Multimedia Systems

Multimedia systems can be compared with traditional software systems in order to emphasise similarities and differences. Two key differences that emerge from such a comparison are that multimedia systems generally impel the user's senses more intensively and involve a higher degree of interaction. Below, these two aspects are illustrated by three examples.

The first example of a multimedia system is used by people who are learning to drive a car for training them in certain kinds of traffic behaviour (Bergman and Møller 1998). Thus the purpose of this system is education and training. The system presents the user with videoclips or animated sketches of realistic traffic situations, and the user must act to handle the situations that arise. The output from the system is video and sound that presents a traffic situation to the user. The input may take different forms; in a simple version, the user may select an action among a set of options that are presented by a kind of menu, whereas in a more elaborate version the user may be able to operate devices similar to the wheel and pedals of a real car. Other examples of this category are systems for training and evaluating decision-makers in an organisation or umpires in a sailing match, cf. Eriksen, Skov, and Stage (2000). This example illustrates how multimedia systems are impelling the user's senses much stronger than traditional software systems. The use of graphics, animations, and video transcends the potentials of traditional screen-based output, and the combined use of sound adds a completely new dimension.

The second example is a game where the user moves around in a virtual world that is created by exposing the user to different images and sound (Andersen and Callesen 2000). Thus the purpose of this system is entertainment. The user controls a certain character or a group of characters by means of an input device and thereby acts in the virtual world. The action of the controlled character often requires considerable input from the user. This example illustrates how a multimedia system requires a high degree of interaction with the user. If the user wants to stay in the game, it is necessary to continuously respond to the changes that are displayed on the screen, and this response clearly influences the further development of the game.

The third example is a system for teaching children about wildlife (Hansen et al. 1999). The purpose of this system is edutainment, i.e. simultaneous education and entertainment. Thus the system was developed as a prototype for a zoo in order to illustrate how multimedia technology could be used to teach children between five and ten years of age about the animals in the zoo. The system enables the user to view photos, drawings, and videoclips of animals and obtain various information about them. Moreover, users can exercise their knowledge by answering quiz-like questions on the animals.

The basic characteristics of these and other multimedia systems are summarised in this definition (inspired by Eriksen and Skov, 1998): *A multimedia system is a computer-based system that integrates a multitude of assets to facilitate user immersion and activity in a virtual situation. The assets are representing fragments of the virtual situation and are based on modalities such as text, graphics, pictures, video, animations, sound, tactile information, and motion. The activity involves interaction with objects in the virtual situation and it is limited by certain temporal and spatial structures. The development of the virtual situation is defined by a plot*. This definition emphasises the main challenges of designing multimedia systems. First, the plot, the virtual world, and the limitations imposed by temporal and spatial structures must be created and described. The result is a design of the story that provides the foundation of the system. Second, a large amount of assets must be specified, typically in detail. Third, the software system that integrates the assets, handles their interplay, and enables interaction with the user must be designed.

This design task requires both a static and a dynamic conception. The static conception is necessary in order to identify objects and their properties. In software design, this process of identifying basic elements is referred to as abstraction (Booch 1994). The behavior of the system as a whole can be designed by describing

how objects are created and disposed, and how they interact with each other. This process requires a dynamic conception where the focus is on the collection of objects and their interplay. Many of the existing software development methods provide considerable support to the design and understanding of the static aspects; for example, this applies to the object-oriented methods that were referenced above in the introduction. Even though different traditions emphasise and employ different concepts, they share a strong focus on static aspects. The existing methods provide much less support the dynamic aspect of software design. Due to the dynamic nature of multimedia systems, the limitations of the existing concepts and notations reduce the advantage of using a software development method. This fundamental problem leaves us with two different research approaches.

One approach is to employ a more experimental approach. In the design of McPie, the developers engaged in a number of cycles where different prototypes of the system were developed and evaluated Horn, Svendsen, and Madsen (2000). A second approach is to employ either concepts or notations. In the design of a training and evaluation multimedia system, the developers produced descriptions that were inspired by storytelling and movie production (Eriksen, Skov and Stage 2000).

Our aim with this chapter is to contribute to the second option. In the following sections, we focus on an empirical study of three software designers. The aim of this study was to identify the basic concepts that the designers used to design and understand the dynamic aspects of a software system and their usefulness in that design process. The details of this study are described in the following section.

3. Empirical Study

The empirical study reported in this chapter was designed to explore the second question raised in the introduction. The question is of a qualitative nature since no variables are defined a priori. The study can be characterized as being a descriptive, in vitro qualitative case study based on observation of an individual designer Basili (1996). The detailed design of the study was inspired by an earlier, more general experiment by Guindon et.al. (1987). A complete description of our study is publicly available on the WWW Skov and Stage (1995). Three different software engineering design approaches were chosen for the study representing different design perspectives and abstractions. The three approaches were an object-oriented approach, an operating systems approach, and a mathematical-logical approach. The first is primarily founded as a practical approach widely used in the systems development industry, while the two others are more formal approaches building on strict semantic deifintions and notations. The study involved three designers with the following personal characteristics:

- *OO Designer (object-oriented)*: He has a Ph.D. in Computer Science; more than 10 years of experience with research, teaching, and software development with emphasis on programming, programming languages, and programming environments.

  He applied no specific method and used only an informal notation inspired by Smalltalk Goldberg and Robson (1989) and Beta Madsen et.al. (1993). Smalltalk and Beta are based on traditional object-oriented concepts such as modularity, information hiding etc. and objects constitutes a natural medium for analyzing and describing concurrent processes.

- *OS Designer (operating system)*: He has a Ph.D. in Computer Science; more than 20 years of experience with research, teaching, and software development with emphasis on operating systems and distributed systems.

  He applied the Phase Web notation Manthey (1988), Manthey (1994) that is inspired by the Actor model Satoh and Tokoro (1992). The Phase Web notation is a method for analyzing and describing concurrent and communicating processes. The Phase Web notation uses entities called sensors in the description of different states of the entire system. The Phase Web notation is highly based on aspects of synchronization and co-exclusion.

- *ML Designer (mathematical-logical)*: He is a Ph.D. student in Computer Science; about 3 years of experience with research and application of formal methods related to protocol design and verification.

  He applied the Calculus for Communicating Systems (CCS). CCS is a general calculus or theory for analyzing and describing concurrent and communicating processes. However, CCS is not only applicable for describing concurrency and communication, but can also be applied for studying machines, architectures, programming methods and languages in general Milner (1989).

Each designer was given the task of designing the process architecture for a interactive software system controlling a set of elevators in a building. In solving this classical standard problem from software specification and software requirements research Guindon et.al. (1987), they were required only to use their "own" paradigm as a development methodology.

The lift control problem.

An N-elevator system is to be installed in a building with M floors. Your assignment is to design the process architecture for a software system which controls the movement of the elevators. The processors can described as follows:

- There is one processor at each of the N elevators. This processor controls the engine, the doors, and reads the pushbuttons and sensors associated with every elevator.
- There is one processor at each of the M floors which reads the push buttons associated with that floor.
- There is one processor, which is able to control all requests from all pushbuttons.

It is your choice whether you want to design a centralized or a decentralized solution to the problem. The design has to be elaborated according to the following rules:

1. Each lift has a set of buttons with 1 button for each floor. These illuminate when pressed and cause the lift to visit the corresponding floor. The illumination is cancelled when the corresponding floor is visited (i.e., stopped at) by the lift.
2. Each floor has 2 buttons (except ground and top), one to request an up-lift and one to request a down-lift. These buttons illuminate when pressed. The buttons are cancelled when a lift visits the floor either while travelling in the desired direction, or having no requests outstanding. In the latter case, if both floor request buttons are illuminated, only 1 should be cancelled. The algorithm used to decide which of these requests to service first should minimize the waiting time for both of them.

---

3. When a lift has no requests to service, it should remain at its last destination with its doors closed and await further requests (or model a ``holding'' floor).
4. All requests for lifts from floors must be serviced eventually, with all floors given equal priority (can this be proved or demonstrated?).
5. All requests for floors within lifts must be serviced eventually, with floors being serviced sequentially in the direction of travel (can this be proved or demonstrated?).
6. Each lift has an emergency button. When pressed, it causes a warning signal to be sent to the site manager. The lift is then deemed ``out of service''. Each lift has a mechanism to cancel its ``out of service'' status.

It is a requirement that the task is solved using object-orientation, Phase Web, or CCS. A final solution shall be delivered. It shall be possible for another person to evaluate this solution.

None of the three designers knew the problem in advance. They were given two hours to solve it and they were required to produce a solution that could be handed over to another person for later evaluation.

The designers were instructed to think aloud during their design process. During the whole session, they only used pencil and paper. All three design sessions were videotaped and the paper-sheets produced were enumerated to enable later identification and relation to the video recordings.

In the data analysis, the videotapes were examined with the purpose of describing each designer's process as a sequence of activities where a change from one activity to another was identified as a situation in which the designer broke the ongoing line of reasoning Guindon et.al. (1987). Furthermore, the activities were characterized in terms of the problem considered, the concepts used to analyze and solve this problem, and the approach taken by the designer. This procedure was developed because of the highly dense and complex nature of video material as a documentation medium. For each of the three sessions, this transcript was elicited by viewing the videotape five to six times, and it amounts to approximately six pages of text. These transcripts are available on the web, see Skov and Stage (1995).

The second part of the data analysis focussed only on the solutions produced by the three designers. Their solutions were evaluated by two independent reviewers who were both associate professors in computer science and had many years of experience in areas related to the focus of the experiment. The documentation produced by the three designers was examined and evaluated in about an hour by each of these reviewers. To summarize their evaluation, they were also asked to mark the three solutions with a grade representing how well the assignment had been solved. These reviews are also available on the web, see Skov and Stage (1995).

3. Conceptual Basis

The first observation focusses on the concepts that were used by the three designers throughout the process architecture design.

> *The OO Designer uses a broad variety of different concepts with unclear mutual relations. In several situations, the same aspect of the problem is characterized in terms of two or three concepts with almost the same meaning.*

The OS Designer and the ML Designer both use a limited set of well-defined concepts with clear mutual relations.

The OO Designer used a broad variety of different concepts during his design of the process architecture. To analyze this in detail, we distinguish between the three levels of using concepts that are specified in Table 1.

Table 1 shows that the OO Designer used a total of 25 different concepts during the design process. Level 3 include concepts that were used to analyze the problem or express a solution in an explicit manner and were consistent with the notation he used. We have attributed 11 of the concepts to this level. Level 2 include concepts that were stated explicitly and used to characterize the problem or evaluate alternatives but they were not used to express a solution. We have attributed 10 concepts to this level. Finally, level 1 includes concepts that were used in thinking about the problem but not used explicitly in analyzing the problem or expressing a solution. We have attributed 4 concepts to this level. It is characteristic that the OO Designer has less than half of the concepts on level 3. A total of 14 concepts are on level 1 or 2 which indicate a more shallow use of these concepts, and one sixth of the total number of concepts used are only on level 1.

Level	Description	Legend	*OO Designer*	*OS Designer*	*ML Designer*
3	The concept was used as an integrated element of the paradigm	•••	11	9	6
2	The concept was stated explicitly during the process	••	10	7	7
1	The concept was not stated, but was used implicitly in the process	•	4	2	1
	Total		25	18	14

Table 1. The number of concepts used during process architecture design

Table 2 shows the distribution of the concepts that were used by the OO Designer on a selection of essential issues in process architecture design. These issues have been used by either of the three designers or serve as a key concept in a major object-oriented analysis and design methods. These issues have then been organized under the three overall concepts that are indicated in the table. The table shows only 15 out of the total of 25 concepts that were applied. Ten concepts, including robustness, time aspects, and independence, have been left out for brevity. The bullets in the table refer to the three levels of use that are shown in Table 1. A parenthesis indicates that the concept was only used upon request of the observer.

Table 2 emphasizes three important characteristics. First, it shows that the OO Designer focussed primarily on communication issues. Communication between objects was analyzed and specified explicitly. There are fewer concepts on concurrency and synchronization and about half of these have only been introduced on behalf of the observer. Second, it shows that three of the essential concepts are used only on level 1. Some of these concepts embody aspects that are critical to the success of the solution. Third, it shows that synchronization received least attention. Only the overall concept was used on level 3 whereas the concepts reflecting more specific aspects of this issue were at most treated on level 2. This include concepts like deadlock, starvation, etc. These concepts played only a mi-

nor role in the design process since the OO Designer did not employ them for systematic reflection.

The multitude of concepts used by the OO Designer were characterized by unclear mutual relations. In several cases, he did not define the concepts he used, nor did he express their semantics clearly. For example, he did not make clear what active objects versus processes meant to him:

> OO Designer: *I view all elevator objects as active objects ... there may be introduced some processes later on, but I will not treat such aspects on this design level.*

The OO Designer also used different concepts with almost the same meaning to describe the same aspect of the problem. For example, he did not distinguish between process, processor, and active object in his solution:

> OO Designer: *I do not make a distinction between processes and processors at this design level.*

	OO Designer	OS Designer	ML Designer
Concurrency	•••	•••	•••
- Parallelism	(•)		•••
- Active Objects	(•••)		
- Process	•••	•••	•••
- Processor	(••)	••	••
- Multitasking		••	
Communication	•••	•••	•••
- Message	•		•
- Broadcast	••	•	••
- Interrupt	••	••	
- Polling	•••	••	
Synchronization	•••	(•••)	•••
- Mutual Exclusion		•••	
- Starvation	(••)	(••)	(••)
- Deadlock	(••)		
- Rendezvous	••		
- Race Condition	•		

Table 2. The specific concepts used during process architecture design

In fact, the three concepts were used interchangeably. In the beginning, he used the concept of processor for the entities in his solution. Later on, he began to use the concept of process for the same entities and, finally, he mentioned that all these processes could be compared to active objects. In all three cases, he did not provide an explicit definition, nor did he consider how it might affect his solution.

The OS Designer and the ML Designer both used considerably fewer concepts than the OO Designer. Table 1 shows that the OS Designer used a total of 18 concepts of which 9 were on level 3. The ML Designer used a total of 14 concepts of which 6 were on level 3. Compared to the OO Designer, they also have significantly fewer concepts attributed to level 1.

The use of specific concepts is shown in Table 2. The overall concepts, concurrency, communication, and synchronization, are used actively by both designers. These concepts served as natural and well-defined components of their design processes. However, both designers seemed to emphasize concurrency and communication more than synchronization.

Another characteristic of the two paradigms is that the key concepts have generally accepted definitions and their mutual relations are usually clear. For instance, the concept of process has a well-defined meaning in both paradigms. This was very apparent in the study since none of these two designers switched randomly between different concepts in their description of a certain aspect.

5 Abstraction and Complexity

The second observation focusses on on the role of abstraction in the whole design process and the extent to which the descriptions made reflected their intuitive understanding of the problem.

> *The OO Designer changes rapidly and unsystematically between different aspects of the problem and between different levels of abstraction. In this sense, his approach is controlled by the nature of the problem. The description made is closely related to his intuitive understanding of the problem.*

> *The OS Designer and the ML Designer both approach the problem systematically, often working continuously with the same problem and on the same level of abstraction. In this sense, their approaches are controlled by the nature of their paradigms. They are forced to introduce inexpedient reductions because the complexity of their descriptions is growing dramatically with the size of the task.*

The entire approach employed by the OO Designer reflects frequent and unsystematic changes between different aspects of the problem. Typically, he immediately began to perform a test once he had specified a certain aspect of the problem. During the execution of these tests, he often discovered a related problem in his

solution. This initiated an attempt to design a modified solution that also handled the new problem. The following scenario illustrates approximately fifteen minutes of the design process: (1) he specifies the static aspects of an elevator object; (2) he begins to test whether an elevator moving in one direction is able to satisfy a request in the opposite direction; (3) he tries to describe the various states of an elevator; (4) he realizes that there may be a potential problem with the pushbuttons in the elevator; and (5) he starts to specify actions for the pushbuttons. While coding the solution, he still returned to further specification and experimentation with the solutions that were expressed in the code.

In the early analysis, it was characteristic that the OO Designer relied only to a very limited extent on specific object-oriented concepts. For example, he had worked for fifteen minutes before he introduced even the first object-oriented concept; this happened when he turned his focus to definition of processes and objects. Similarly, his overall design choices were governed more by his general knowledge and intuitive understanding than by specific object-oriented concepts or perspectives.

Two additional characteristics contribute to the impression that his approach was controlled by the nature of the problem rather than the nature of his paradigm. First, it seemed random when he turned from specifying to simulation. Sometimes this change was triggered by a recognition of a potential problem or an aspect he could not describe clearly. In several cases, the latter was caused by the inherent concurrency of the problem. Second, after testing a partial solution he often returned to a different problem.

While working with the task, the OO Designer was able to specify on a level of detail that was appropriate for his momentary intentions. During his first analysis, the OO Designer tried to obtain a general overview of the elevator problem by drawing a sketch of a building with a number of floors and a number of elevators. He defined elevator and floor objects without describing them in greater detail. Thereby, he obtained a first intuitive understanding of the problem at hand. Later on, elevators and floors constituted the objects upon which he specified more detailed aspect and evaluated potential solutions. This was possible because the object-oriented concepts do not require description at a specific level of detail. The adding of details can be done continuously during the design phase. Moreover, his intuitive model resembled the real world complexity when considering objects and static relations.

The OS Designer and the ML Designer both began their design process by drawing a sketch of a building with a number of floors and elevators. Once this frame for an intuitive understanding was established, they changed to a mode of operation that was far more rational than the experimental process conducted by the OO Designer. In some situations, they *did* test partial solutions, implying that they

changed from the rational mode in which they were specifying in terms of their paradigm to a more experimental mode where they were testing partial solutions. But it was on a much smaller scale, and they never switched randomly between different issues or levels of abstraction. Both designers spent significantly more time specifying the actions for the solution, and the majority of their effort was spent on systematic work that was carried out in a mostly sequential order. The paradigms seemed to force the focus of the designers in certain directions and maintain this focus for longer periods of time.

The OS Designer and the ML Designer both faced severe problems with complexity as their descriptions grew radically with the size of the task. This lead them to reduce the task considerably by imposing restrictions on the number of elevators and floors. The ML Designer chose to restrict the task to three floors and only one elevator. The OS Designer initially reduced the problem to three floors and three elevators and, later on, he considered to reduce the task further by dealing only with two floors and two elevators:

> OS Designer: *Normally I would not solve such a task with three floors. I would only involve two floors in order to reduce complexity*.

This problem was emphasized by Reviewer R2 who stated that the solutions made by both the OS Designer and the ML Designer were characterized by an enormous complexity. The problem was that the design of a solution is described by a set of rules that are expressed in the notations defined by the paradigms. These rules are not intended for manual execution by people, but have been made to support automated processing. The OS Designer stated it this way:

> OS Designer: *Solutions made with the Phase Web paradigm quickly become complex and enormous because of the many actions*.

The OS Designer was very familiar with his paradigm. Thus he was able to foresee problems that would emerge in a solution as a consequence of certain design choices. For example, he had no problem envisioning the actions needed to move an elevator from one floor to another. Due to more limited experience with the paradigm, the ML Designer had a less clear relation between his thinking and the nota-

tion used for expressing it. Nevertheless, he never seemed to have problems that were comparable to the OO Designer.

Despite the fact that both the OS Designer and the ML Designer had to restrict the task in order to reduce complexity, it seemed that they used the relevant concepts and their notations coherently. Both designers were able to think of the problem in terms of their respective paradigms. In this sense, there was no leap or semantic gap between their thinking and notation. They were able to use the notation in accordance with their thinking, once the problem had been reduced to a size that was practical compared to the capability of the notation.

A more severe leap occurred in the relation between the descriptions and solutions on the one hand, and an intuitive understanding of the real world problem on the other hand. The task was reduced considerably at an early point in time. The restrictions that followed from this reduction gradually narrowed the scope of the designers' work and in the end, both designers turned out to have solved a very limited problem. This was expressed very clearly by Reviewer R2 who claimed that the intuitive understanding of the actions in the paradigms was not clear to him:

> Reviewer R2: *It easily becomes signs and strange actions, where you cannot associate anything with these widgets. A semantic gap is introduced between your intuitive understanding of the problem and the actions in the paradigm.*

This statement illustrates the problems that arise when a paradigm forces designers to reduce the task inexpediently. The reduction is necessary due to complexity of the descriptions made but it seems to imply that the semantic contents of these descriptions contradicts an intuitive understanding of the problem. The reduction introduces a leap between the intuitive understanding and the descriptions made.

6 Identifying the Problem

The third observation focusses on the part of the design process in which the designers identified whether concurrency was an issue and the nature of this potential issue.

The OO Designer identifies the inherent concurrency of the problem through experiments and simulations. In this way, he uses a considerable amount of time obtaining a clear understanding of the nature of the concurrency involved.

The OS Designer and the ML Designer both identify the inherent concurrency of the problem in terms of mechanisms from their paradigms. In this way, they quickly achieve a sufficient understanding of the nature of the concurrency involved.

One of the major difficulties during process architecture design is the identification of potential sources of concurrency Jacobson et.al. (1992). Early in the design process, the OO Designer was able to identify that the elevator problem involved inherent concurrency. However, this identification was not based on a systematic and methodical analysis but rather on an intuitive understanding of the problem. Moreover, he was unable to indicate where concurrency could turn out to be a problem and what consequences it would imply for his solution.

The process conducted by the OO Designer was characterized by frequent and random changes between specifying and simulation. Now and then he was specifying actual code of his solution; now and then he was testing a partial solution by performing simulations. Typically, he was working approximately ten minutes on specifying and then about five minutes on testing. During the entire design process, he made this kind of change between specifying and testing six or seven times.

This approach to identification of concurrency can be characterized as being highly experimental. Potential problems caused by the inherent concurrency were discovered through execution of simulations on partial solutions. Another example of this occurred when he performed minor simulations on his solution in order to determine whether an elevator was able to satisfy an up-request when it was going down. He also simulated what would happen when an elevator was waiting for requests. A consequence of this approach was that he used a considerable amount of time on the identification of concurrency in order to gain the necessary overview. In fact, aspects of concurrency were identified and handled every now and then during

the entire design process. For example, after having made his first overall design choice, he spent six to seven minutes trying to apprehend a very minor and simple concurrency problem. Thus the overall impression is a very ineffective and irrational mode of operation.

The OO Designer also faced problems when he attempted to evaluate the consequences of his design choices. At first, he chose a centralized unit for coordination of requests. Through simulations, he later realized that this centralized unit was a problem because the elevator had to delete a request when it arrived at a floor. Instead, he began to design a decentralized solution:

> OO Designer: *A decentralized solution attracts me because it is more robust. Such a system (the lift control system) is of course subject to breakdowns.*

This redesign caused other problems as he could no longer guarantee that only one elevator would service a request from a floor:

> OO Designer: *I have some problems with the concurrency involved here, but I will rather have two elevators trying to service a request from a floor than none.*

This approach to identification of concurrency indicates that the OO Designer never became totally certain whether the problem was fully understood or whether important concurrency issues still had to be addressed in order to design a satisfactory solution.

Both the OS Designer and the ML Designer quickly identified the key problem of the task. They benefited from their paradigms which respectively enabled them to identify and describe the inherent concurrency in terms of the mechanisms that were provided by the paradigms that they applied.

Both paradigms represent a long tradition of dealing with concurrency and related issues concepts, and they have been designed specifically to handle such aspects. As a result, the OS Designer and the ML Designer could avoid dealing explicitly with many detailed aspects of the concurrency involved. All they had to do was to specify the conditions that have to hold during the execution of the system. In fact, it was often hard to see that they addressed the

problem at all since their paradigms handled the concurrency:

> ML Designer: *It is a formalism where you can put things in parallel for free. Communication and concurrency have been built into the language.*

The two designers with backgrounds in classical concurrency paradigms employed a conception of concurrency that is very different from the view held by the OO Designer. Their identification of sources of concurrency were more systematic because their paradigms imposed a sequential order on the issues they had to deal with. Both paradigms can be seen as media for capturing and describing concurrency aspects and this enables users of the paradigms to invest a very modest effort but still achieve a sufficient understanding of the concurrency involved.

7 Expressing the Solution

The fourth observation focusses on the part of the design process in the designers developed and expressed their solution to the problem.

> *The OO Designer has difficulties describing how communication and synchronization are handled because he lacks a notation to describe the dynamic behaviour of objects on an overall level.*

> *The OS Designer and the ML Designer both describe communication and synchronization without any severe problems as they simply express themselves in terms of the mechanisms that are available in their paradigms.*

When the OO Designer started drawing a sketch of the real world situation, his focus was primarily on objects and their dynamic behaviour. For instance, he thought of and described the dynamic behaviour of an elevator: what happens when it reaches a specific floor, what happens when it awaits requests, etc. This lead him to determine the heart of the lift control problem as being the design of a good algorithm to control the behaviour of the lifts:

> OO Designer: ... *all rules in the problem statement imply that the algorithm for the elevators has to be rational ... the design of the algorithm is*

the core of this task.

His attempts to map this behaviour into an object-oriented description caused fundamental problems. He tried to express the dynamic behaviour of the whole system as procedural code in the abstract classes of the objects involved. After having spent only 31 minutes on the task, he started coding both essential and unessential aspects on a very detailed level. His third and fourth sheet of paper already contains program statements; for example, he specifies a constructor function that initializes an elevator object to be generated on floor 1. He worried about such details even though he still lacked a coherent overall understanding of the problem. He gave the following reason for specifying with code:

> OO Designer: *It is easier for me to see how the objects interact when I specify them with actual code*.

This statement can, however, be questioned since, later on, he had difficulties describing certain aspects of the dynamic interaction between objects. For example, he had to incorporate additional attributes to represent the motion of an elevator going up or down and this in turn introduced difficulties in realizing whether his solution would be able to service the various request from pushbuttons in the elevators and on the floors.

The heart of the difficulty he faced was simply caused by the lack of an overall notation for analyzing and designing solutions in terms of specific objects and on a relevant level of detailing. Instead, he became burdened with isolated and unessential issues that occurred only because he had to express himself in detailed code belonging to the individual classes. As he gained new insight and wanted to modify his design on the overall level, he had to express these changes in several different fragments of code. This also implied that he modified the same part of the code several times.

To circumvent the lack of a suitable means of expression, he tried to employ concrete knowledge about the hardware and software that would be available on the underlying implementation platform. In designing the experiment, we had anticipated this problem. The problem statement of this study was originally taken from Guindon et.al. (1987). Yet this description lacks specific information about the facilities of the technical platform. In order to support the designers in developing an appropriate solution, we had extended the original problem statement with a detailed specification of the available hardware. However, two contradicting characteristics illustrate that this description was of very limited value to the OO Designer. On

the one hand, he did not distinguish between processors and processes, cf. the first observation on conceptual basis. This indicates that he relied on the assumption that it would be a simple task to map the processes he designed onto the processors that constituted the technical platform. By making this assumption, he was able to refrain from solving a key problem of the process architecture design and this in turn allowed him to ignore the information about the available hardware that was provided in the problem statement. On the other hand, he faced several situations in which he requested even more specific and detailed information about the features of the platform:

> OO Designer: *I assume that this is handled by interrupts, but I cannot specify it in more detail without additional knowledge of the underlying hardware*.

The lack of a specific platform with a distinct semantics was obviously a problem to the OO Designer. He virtually chose to ignore the problems of describing how communication and synchronization should be handled. In Table 2, this is illustrated by his limited focus on concepts like broadcast, interrupts, starvation, deadlock, race conditions, etc. These issues were only treated marginally during the design process even though some of them may significantly influence the success of the final solution. Later on, he realized that a problem could arise with his solution when an elevator was trying to service a request. The problem was that an elevator could move to a floor in vain because another elevator had already serviced the same request:

> OO Designer: *Some of the elevators may move in vain ... this is a synchronization problem ... some communication is, of course, taking place between the processors, but I need more information of the hardware to specify such aspects.*

These characteristics clearly emphasize that the object-oriented paradigm lacks means to express overall aspects of communication and synchronization. Instead, users of the paradigm are forced to deal with the actual features of the underlying hardware and software.

The paradigms employed by the OS Designer and the ML Designer

have been built deliberately to handle key aspects of communication and synchronization. Of course, this influenced the experiment. Both designers were able to describe and express aspects of communication and synchronization in terms of the notations provided by their paradigms. This was clearly in contrast to the work of the OO Designer because he had to deal explicitly with these topics.

None of the two designers made any use of the information that was given in the problem statement about the available technical platform. Instead, they made assumptions about features that had to be available:

> OS Designer: *I assume that features of the underlying hardware make it possible for me to turn off the lights in the pushbuttons when a request has been serviced.*

Both designers expressed that problems concerning communication and synchronization played a very minor role in their designs because their paradigms would take care of such aspects:

> OS Designer: *I know you might think that I am getting over the problem easily, but I am sitting on a synchronization machine which solves a lot of my problems. All I have to do is to write down the synchronization conditions that have to hold.*

The ML Designer described the key aspects of synchronization by means of labels on his entities. In order to synchronize an elevator and a floor, he could specify a specific action on each of the entities involved. The actual synchronization of the two would then be handled by the paradigm.

8 Conclusion

This chapter has explored how three experienced designers worked with a realistic and complex problem related to designing the dynamic element of a software system. The preceding sections have expressed the results of this exploratory approach in terms of four observations that are summarized in Table 3.

		OO Designer		*OS and ML Designers*
Conceptual Basis	–	Many similar concepts with unclear mutual relations	+	Few well-defined concepts with clear mutual relations
Abstraction and Complexity	–	Rapid changes between different aspects of the problem and different levels of abstraction	+/–	Continuous but unreflected work on the same aspect of the problem and on the same level of abstraction
	+	Close relation between description and intuitive understanding	–	Reduced description due to dramatic growth in complexity
Identifying the Problem	–	Unstructured and experimental approach, slowly accomplished	+	Structured and systematic approach, quickly accomplished
Expressing the Solution	–	No abstract notation for specifying communication and synchronization	+	Simple, abstract mechanisms for specifying communication and synchronization

Table 3. Summary of observations

The OO Designer faced several situations in which he needed stronger conceptual or methodical support. His design process conveyed four essential deficiencies. First, he wasted much effort because he lacked a coherent conceptual framework and a related set of methodological guidelines to support the design process. Second, a notation for relating the dynamic behaviour of objects to the static definitions of classes would have improved his design process and solution significantly. Third, his design suffered from the lack of an abstract machine that served as the underlying technical platform of a design solution. Fourth, his solution describes the structuring of processes but the overall design is largely missing.

These observations are based on a small experiment with only one object-oriented designer and a comparison with two other designers; and all three designers represent fundamentally different and very heterogeneous paradigms. Thus our experiment cannot form the basis of quantitative and statistically valid conclusions concerning all object-oriented designers. The advantage of this limited experiment is that it has facilitated a rich, qualitative insight into the problems and breakdowns faced by each individual designer and the reasons why they occurred (Basili 1996). Video observation and exhaustive, qualitative data analysis of a large number of designers is practically impossible. The qualitative exploration is a first step towards a bet-

ter understanding of object-oriented process architecture design. Later on, the four observations can be examined quantitatively in a more ambitious study that involves more designers and a varied selection of process architecture problems.

This exploratory approach also opens other avenues for further research. A related effort would be to study the extent to which specific object-oriented methods support the process architecture design activity. Finally, it could be investigated how the process architecture has been designed in a number of complex software systems. All of these efforts would contribute to improve the work practices of software designers dealing with realistic concurrency problems.

9 Acknowledgements

The research behind this article has received financial support from the Danish Natural Science Research Council under grant No. 9400911. We owe a special thank to the five participants of the experiment: Kasper Østerbye, Michael J. Manthey, Kaare J. Kristoffersen, Arne Skou, and Bent Bruun Kristensen. Without their participation, the experiment would not have been possible. We are also grateful to Lars Mathiassen. Peter Axel Nielsen, Dan Sletten, and Heinz Züllighoven for their comments and suggestions to different versions of this article.

References

Andersen, P. B. and Callesen, J. (2000). Agents as Actors. (in this volume)

Basili, V. (1996) The Role of experimentation in software engineering: Past, current, and future. In Proceedings of the 18th International Conference on Software Engineering, pages 442-449.

Booch, G. (1994) Object-Oriented Analysis and Design with Applications. Benjamin/ Cummings, Redwood City, California.

Booch, G, Jacobson, I., and Rumbaugh, I. (1997) The Unified Modeling Language Version 1.0. Rational Software Corporation, Santa Clara, California.

Coad, P. and Yourdon, E. (1991a) Object-Oriented Analysis. 2nd edition, Prentice-Hall, Englewood Cliffs, New Jersey.

Coad, P. and Yourdon, E. (1991b) Object-Oriented Design. Prentice-Hall, Englewood Cliffs, New Jersey.

Eriksen, L.B., and Skov, M. (1998). A Critical Look at OOA&D in Multimedia Systems Development. In proceedings of the 21th Information Systems research seminar In Scandinavia. 8 - 11 August, Sæby, Denmark

Eriksen, L. B., Skov, M., and Stage, J. (1998b). A Multimedia System Development Project: Documentation. Available through the WWW at the following URL: http://www.cs.auc.dk/~dubois/manager/

Eriksen, L. B., Skov, M., and Stage, J. (2000 Multimedia Systems Development Methodologies: Experiences and Requirements. Submitted for publication.

Goldberg, A. and Robson, D. (1989) Smalltalk-80. The Language. Addison-Wesley, Reading, Massachussetts.

Guindon, R., Krasner, H., and Curtis, B. (1987). Breakdowns and Processes during the early Activities of Software Design by Professionals. In DeMarco, T. and Lister, T. (eds) Software State-of-the-art: Selected Papers. Dorset House Publishing, New York, pp. 455 - 475.

Hansen, K. K., Harbøll, B., Høegh, R. T., Lorentzen, K. H., Madsen, R. Ø., and Pedersen, M. S. (1999). Zoomedia. A Multimedia System Developed for Aalborg Zoo (in Danish). Aalborg University.

Horn, G, Svendsen, E.H., and Madsen, K.H. (2000) Experimental Design of Multimedia. (in this volume)

Jacobson, I., Christerson, M., Jonsson, P., and Övergaard, G. (1992). Object-Oriented Software Engineering. Addison-Wesley, Wokingham.

Jackson, M. (1983). Systems Developmentoftware. Prentice-Hall, New Jersey.

Madsen, O.L. and Møller-Pedersen, B. (1993) Object-Oriented Programming in the Beta Programming Language. Addison-Wesley, Reading, Massachussetts.

Manthey, M., Andersen, L.U., Arent, J., Christiansen, H., Nielsen, T.K., Simonsen, J., and Sørensen, T.B. (1994) A topsy example. Aalborg University, Denmark

Manthey, M (1988) Maskinel II. Technical report. Aalborg University, Denmark

Milner, R. (1989) Communication and Concurrency. Prentice-Hall.

Pressman, R. S. (1992). Software Engineering: A Practitioners Approach. McGraw Hill.

Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, S., and Lorensen W. (1991). Object-Oriented Modelling and Design. Prentice-Hall, Engelwood Cliffs, New Jersey.

Satoh, I. and Tokoro, M. (1992) A Formalism for Real-time Concurrent Object-Oriented Computing. ACM Sigplan Notices: OOPSLA. 27(10), October, pp. 315 - 326

Shlaer, S. and Mellor, S. J. (1988) Object-Oriented Systems Analysis: Modeling the World in Data. Yourdon Press, Englewood Cliffs, New Jersey

Skov, M. and Stage, J. (1995) Object-Oriented Design of Process Architecture: An Exploratory Study - Documentation. Available through the WWW at the following URL: http://www.cs.auc.dk/~jans/procarch/

Skov, M. and Stage, J. (1996) Object-Oriented Design of Process Architecture: An Exploratory Study. In Proceedings of the 19th Information Systems research seminar In Scandinavia. 10 - 13 August, Lökeberg, Sweden, pp. 975-1000

Sommerville, I. (1992). Software Engineering. 4th edition. Addison-Wesley, Workingham.

Sutcliffe, A.G. and Faraday, P. (1994). Designing Presentation in Multimedia Interfaces. In: Adelson, B., Dumais, and Olson, J. (Eds.), Proceedings of Computer-Human Interaction Conference ' 94. 92-98.

Sutcliffe, A.G. and Faraday, P. (1997). Designing Effective Multimedia Presentations. In: Ware, C. and Wixon, D. (Eds.), Proceedings of Computer-Human Interaction Conference ' 97.

Wirfs-Brock, R., Wilkerson, B., and Wiener, L. (1990) Designing Object-Oriented Software. Prentice Hall, Englewood Cliffs, New Jersey