6th International Conference on Telecommunications
Maja Matijašević and Alen Bažant (eds.)
June 13–15, 2001, Zagreb, Croatia

ConTEL 2001

UML Based Object-oriented Development: Experience with Inexperienced Developers

Mario Kušek
Mario.Kusek@fer.hr

Department of Telecommunications
Faculty of Electrical Engineering and Computing, University of Zagreb
Zagreb, Croatia, HR-10000

Saša Dešić
Sasa.Desic@etk.ericsson.se

Research & Development Centre
Ericsson Nikola Tesla d.d.

Zagreb, Croatia, HR-10000

Darko Gvozdanović
Darko.Gvozdanovic@etk.ericsson.se

Research & Development Centre
Ericsson Nikola Tesla d.d.

Zagreb, Croatia, HR-10000

Abstract

UML is becoming increasingly important in modern software development. Many articles describe UML features, but only very few of them discuss its usability in real projects. This article discusses features and usability of UML in software projects based on experiments and pilot projects. In the analysis some differences between UML and SDL (Specification and Description Language) are emphasized. This article deals with the impact of UML on newcomers in the world of object-oriented software development. The experiment with two groups of students (one trained in UML) was carried out. Their goal was to develop the solution for particular software system. Advantages and disadvantages of UML are also commented with respect to user's level of knowledge, application type and requirements.

1. Introduction

World development and lifestyle increasingly depend on software. Software-intensive systems, as technological achievements, as well as social demands, are growing in size, complexity, distribution and importance. However expansion of these systems changes the limits of software industry know-how. As a result, building and maintenance of software becoming increasingly complex.

Various software development projects fail in different ways but they all share common symptoms. Some of them are: inaccurate understanding of end-user needs; inability to deal with changing requirements; software that is not easy to maintain or extend or late discovery of serious project flaws. Analysis of the symptoms reveal that their root causes are: *ad hoc* management of the requirements; ambiguous and unambiguous communication; overwhelming complexity; undetected inconsistencies in the requirements, design and implementation; uncontrolled change propagation or insufficient testing.

Some of the problems and their causes can be avoided by implementing more rigorous development process. Deployment of notation language, like UML, might facilitate communication between all participants in the development process. These are some of the reasons for UML utilization in software development. Present article analyses the aspects of UML use in the development process.

Second section explains UML roots, causes and purpose. UML diagrams are explained in brief. The third section describes one software development process that utilizes UML. Examples and aspects of UML utilization are shown in the fourth section. In section five are described the experiments with UML and inexperienced developers. Section six gives the summary of the results.

2. UML

The unified modeling language (UML) is a graphical language for visualizing, specifying, constructing, and documenting software-intensive systems. UML provides a standard way of writing system's blueprints, covering conceptual things, classes written in a specific programming language, database schemes and reusable software components. It is a standard notation, used by everyone involved in software production, deployment and maintenance.

UML includes nine diagrams for describing the system:

- Class diagram describes a set of classes, interfaces and their relationships. It shows the view of the system's static design. This diagram is very useful in modeling object-oriented systems.

- Object diagram shows a set of objects and snapshots of instances of the things found in class diagrams.
- Use case diagram shows a set of the use cases, actors and their relationships. This diagram is especially important in organizing and modeling behavior of a system.

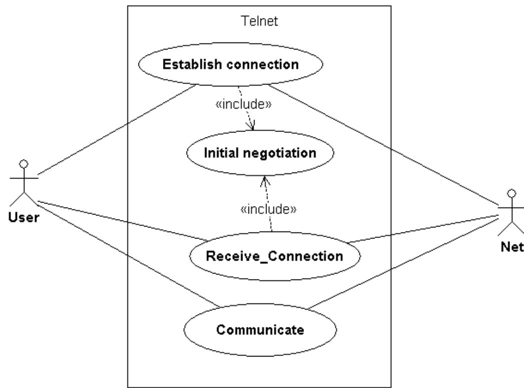

Figure 1. Example of a use case diagram

- Sequence and collaboration diagrams are interaction diagrams, which describe interaction between the objects. They show their relationships, including messages between the objects. Interaction diagram explains dynamic view of the system. Sequence diagram emphasizes the time order of messages. Collaboration diagram emphasizes structural organization of the objects in interaction. Both are isomorphic, meaning that a sequence diagram can be transformed into collaboration diagram and vice versa.
- Statechart diagram shows a state machine consisting of states, transitions, events and activities. It addresses dynamic view of the system. This diagram is very important in behavior modeling.
- Activity diagram is a special kind of statechart diagram. It emphasizes flow from the activity to the activity within the system. It is very useful in tracing concurrent activities in the system.
- Component diagram describes organization and dependencies among the components. It is related to class diagrams with respect to mapping the component to one or more classes, interfaces, or collaboration.
- Deployment diagram explains configuration of run-time processing nodes and of their components. It shows static deployment view of architecture.

3. Development process

UML is a modeling language rather than a method. For a successful project, however, modeling language is not enough. There are several developing methods (e.g. Extreme programming and Feature Driven Programming). Creators of UML have developed Rational Unified Process (RUP), which mostly utilizes UML and incorporates best practices of various projects. This section briefly explains Rational Unified Process.

Rational Unified Process is an iterative and incremental development process (Figure 2.). Software is carefully being built up through the process, step-by-step and functionality-by-functionality. Subset of functionality is designed, implemented and tested after every iteration,

During **inception**, first meetings are held to define project goal and scope and to make rough cost-benefit estimate).

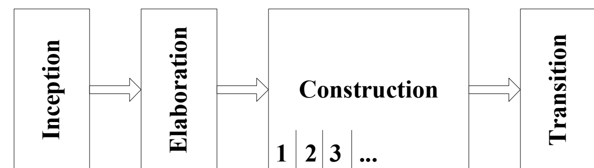

Figure 2. Rational Unified Process

Elaboration specifies more details about requirements and technologies involved, and comprises high-level analyses and design for creating baseline architecture. Plan for the construction phase is worked out at this phase too.

Construction phase consists of iterations; each built, tested and integrated production-quality software satisfying a subset of project requirements. The delivery may be external, to early users, or purely internal. Iterations comprise all usual life cycle analytical steps, design, implementation and testing.

Transition includes beta testing, performance tuning and user training. Optimization reduces clarity and extensibility of the system in order to improve performance.

4. UML in practice

4.1. Experimental

UML system modeling was tested experimentally. Two software systems had to be designed using UML and the appropriate tool. The results were supposed to show UML usability in various types of systems and required levels of detail. For that purpose, we launched two projects.

The first one, called JAMES, was the communication program based on the specific protocol. The program was divided into three main parts (Figure 3):

- USI (User Side Interface) - control procedures for the user side,
- NSI (Network Side Interface) - control procedures for the network side, and
- GUI (Graphical User Interface) - graphical interface.

The user of that application had to be able perform basic communication actions:

- initiate connection establishment;
- communicate through file transfer, sending and receiving the messages and
- release the connection.

That kind of the application was not demanding in architectural design, but required precisely defined behavior. Exactly that situation was not suitable for UML supported design, intentionally chosen to be the part of our experiment.

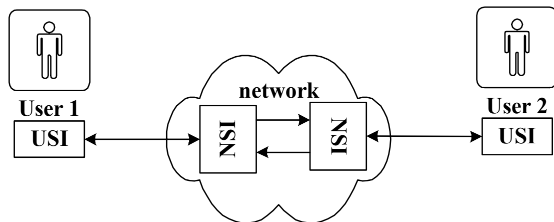

Figure 3. James: Model of the system

The second project, called BOND, was expected to result in a distributed processing simulator (Figure 4). The distributed processing system could be, for example, a Web site with the cluster of Web servers. The main purpose of the project was aimed to create functional prototype of the simulation system. All nodes in the system received the requests. Those from highly loaded nodes could be re-scheduled to other nodes.

The prototype was extended by mobile agents responsible for the system maintenance and monitoring. The agents collected the data about remote sites and saved them on the management site.

In order to run the simulation, network topology and load distribution had to be defined. Load distribution was specified as the incidence of job arrival per node. All that could be done through GUI. During simulation, the user could change simulation parameters (network topology and load distribution), view the system statistics and analyze overall system performance in the course and after simulation.

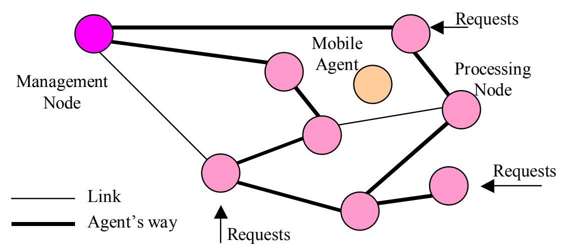

Figure 4. Bond: Model of the system

Unlike JAMES project, BOND was architecturally highly demanding, while behavior was left to the implementers.

It was not that these two kinds of projects enabled the analysis of UML utilization only, but also of the potential use of the whole chain of formal methods, such as UML to SDL converter followed by SDL design.

4.2. Results

Having designed those two systems, we have arrived to the following conclusions: UML is superior in early phases of development. Requirement analysis is heavily supported, but the two types of diagrams differ. Use-case and sequence diagrams provide everything needed for capturing all systems' features, yet simplicity must be preserved for the customer to understand the designed behavior. Misunderstanding between the designers and customers is avoided with the use of UML, but misunderstanding itself usually causes the projects fail. Negotiating practice with the customers (who are frequently prone to changing their mind) has not been very successful and should, therefore, be adjusted to UML. Another good side of having use-cases is that they are valuable source for creating test-case scenarios in testing and validating partly or fully implemented systems.

UML diagrams give the means to express ideas, but ideas come from experience. Less experienced designers may not understand the background of a particular design of a senior designer, but they can understand it and are definitely able to improve themselves on it. UML helps in acquiring object-oriented way of thinking. Less experienced designers very often say: "I can't express it with UML", but in most cases this means there is something wrong with an object-oriented design. Hence, using UML is very helpful to all categories of designers.

Unlike with SDL the developers find UML easy to use in that it has no strict formal structure or terminology, although some basic rules must be respected. In SDL usage, the prerequisite is higher level of expertise. As expected, experienced users were more efficient in describing and designing the system. Previous knowledge of the object-oriented system development was very helpful in using UML.

UML was helpful for understanding and developing object-oriented thinking.

UML has tackled another important issue. Parallel design of different parts of the system is almost obligatory, except of the very simple ones. Precondition for proper system division comprises two things:

- definition of relatively independent subsystems, having appropriate functionality, and
- ensuring full collaboration of these subsystems, once they are designed.

UML addresses them by packages and interfaces. There are no package diagrams, but a designer can use class diagrams instead. By doing so, not only implementation but also design is hidden behind the interfaces, facilitating subsequent changes and maintenance. It is even possible to develop particular parts of the system using different tools (without any UML), and than to make reverse engineering into UML design (Figure 5). This particularly refers to GUI design. Given that redesigning and adding new functionality is often the case (not usual with a brand new system), using legacy code can make substantial savings.

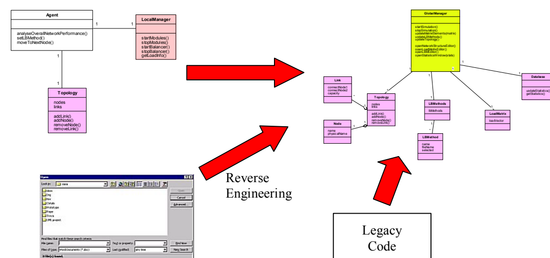

Figure 5. Reverse engineering (GUI and legacy code)

Obviously, UML handles architectural problems well. Defining behavior is its weaker side. There is a whole lot of diagrams (sequence, collaboration, activity and state-chart diagrams), but the designers can use them for their own needs only. They do not contribute at all to automatic code generators. Capturing behavior of the mentioned communication application is worthless in automatic code generation, but not in documenting the way it works. The designers would definitely prefer the code to give some reflection of the captured behavior. Consequently, UML to SDL (Specification and Description Language [4]) converter offered by some vendors is a big step forward. SDL is suitable for the in-depth system design and verification. Also, SDL is also successful in expressing system dynamic behavior. With the use of a translator, a lot of information from UML is kept and transferred into SDL. Consequently, visual design along with all the possibility of system verification and validation offered by the tools, remains available to designers.

System design with UML is very dependent on the type of the system under development. The experiments support the conclusions arrived at. UML is much more appropriate for designing the systems having complex static architecture (e.g. data and GUI-related systems).

SDL, on the other hand, is more appropriate for the systems with much internal communication. GUI systems and all systems involving a lot of communication with the outer world can be developed with SDL, although it is more difficult than with UML. The proposed solution is a combination of UML and SDL. Figure 6 gives an example of such a design process that utilizes this approach (Figure 6).

The applied UML tool has code generators for several most interesting programming languages. Preconditions for code generation are completeness and correctness of the model. Hence, prior to code generation there is always an automatic check of the system design.

Major disadvantage of UML design in code generation is the loss of much information. Code generators use only class diagrams. Complete behavior has to be implemented manually. In some cases, like in JAMES project, most of the work is done through sequence and statechart diagrams. Class diagrams are very simple, representing static relations rather than dynamic behavior. In our case, code generation is practically absent. However, it does not mean that all other diagrams are worthless. They are used in describing and documenting the system.

Otherwise generated code is very clean, readable, well structured and documented. Notes added to UML diagrams are included into generated code as comments. Code is clearly classified into the sections for attributes, constructors, and user defined methods. Another practical feature is that the generated code clearly reflects the system design (relationship between classes, their associations, role names, etc.).

UML tools in the development and system analysis help us to test the system manually. SDL tools, though, have powerful capabilities for automatic inspection of the newly created systems. UML tools do not support processes of validation and verification available by SDL tools. They merely provide basic structural check. This is another good reason for the use of UML and SDL approaches (Figure 6).

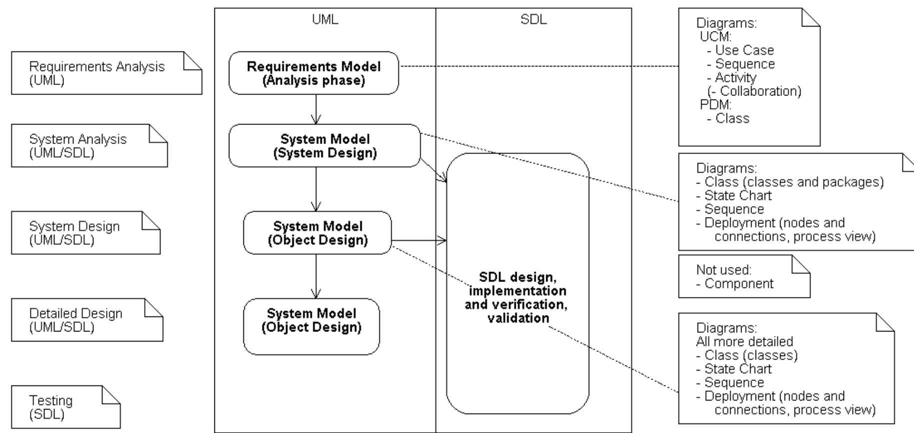

Figure 6. Software design process supported by UML/SDL combination

5. Experiments with inexperienced developers

5.1. Experimental

Many articles discuss great features of UML, but just a few of them describe experimental results with UML in real-life projects. It is hard to quantitatively present the advantages of UML over common way of software development. An experiment with a group of students was carried out to demonstrate UML usefulness. All students attended the courses (during their university education) on C++, SDL basics, Java basics and C.

The students were divided in two groups. The UML group attended UML presentation and the non-UML group did not. Both groups were divided in three subgroups. Each subgroup was told to design a small system. The systems were: Web service for exam appliance, cache machine and automatic door opening system. The non-UML group, having no information about UML, had to design the system using previous knowledge and sense. The key issue was that nobody was experienced in the object-oriented system design and development.

5.2. Results

It is hard to quantitatively measure the design and to compare the progress of two different groups. One of the measurement parameters was the amount of created diagrams and materials. The non-UML group made a few rough descriptions of the system operation and some flow diagrams (Figure 7). Most of the time they were trying to understand the internal structure and architecture of the system. The UML group was better. It created mostly use case, sequence (Figure 8) and activity diagrams, and even class diagrams that show classes, methods and attributes. Judging from the amount (quantity) of the developed documents, UML was apparently extremely supported in designing (especially in meeting the requirements and understanding the overall system architecture).

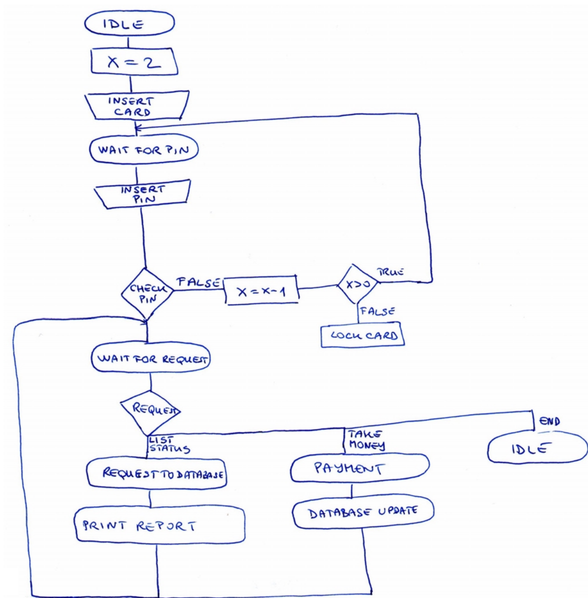

Figure 7. Flow diagram created by Non-UML group

Implementation as a final goal of every development process requires that many details be defined. The level of detail was a parameter in comparing two groups and their work. The designs of the non-UML group were on a very high level of abstraction and with few details. The programmer, responsible for implementation, could not build a system with such input documents.

On the other hand, the UML group used class diagrams containing many details and it prepared a very good base for implementation. Other created documents are also useful for system visualization and understanding of its function.

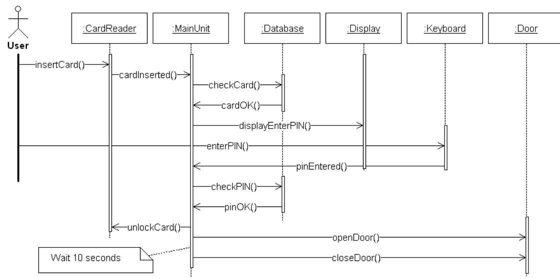

Figure 8. Sequence diagram created by UML group

The non-UML documents clearly show that the group was slightly confused and disoriented. Unlike them, the UML students were lead by UML diagrams and stayed more concentrated and effective. The logical flow of diagram usage helped maintain the right direction of design efforts.

The non-UML group failed to determine the system classes, their methods and attributes. The UML group succeeded by using powerful tools, such as sequence and class diagrams (Figure 9). In that way, they were driven to finding appropriate solution for system classes and associations among them.

At the early stages of development, it is very important to examine various scenarios and possible irregularities in the system. The analysis employing case and sequence diagrams can help find, develop and explain the system states, where more then one option is possible. Unfortunately, UML group payed no attention to that. It was probably due to participating in the UML project for the first time, or time restriction.

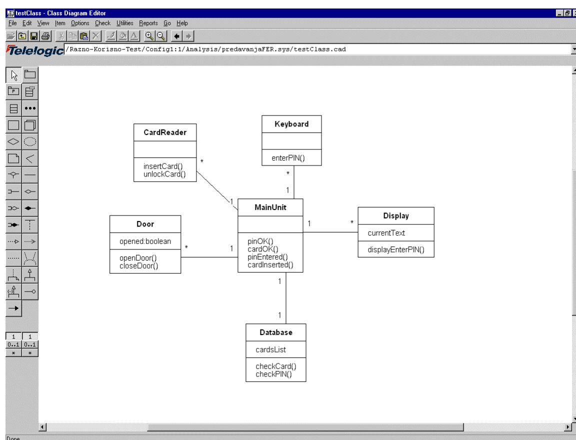

Figure 9. Class diagram created with UML tool

The non-UML group used flow diagrams that helped system visualization. It is known to be a very useful method for facilitating development process. Like flow diagrams, UML diagrams help visualize the system. Flow diagrams visualize only behavior of the system, whereas UML diagrams visualize all its aspects.

Results of the experiment can be summarized in one figure (Figure 10). The achievements of both groups are shown with respect to several viewpoints of development process. The results per areas are scored 1-10. Predicted results for full UML usage (UML by definition) are added to the graph to emphases UML superiority.

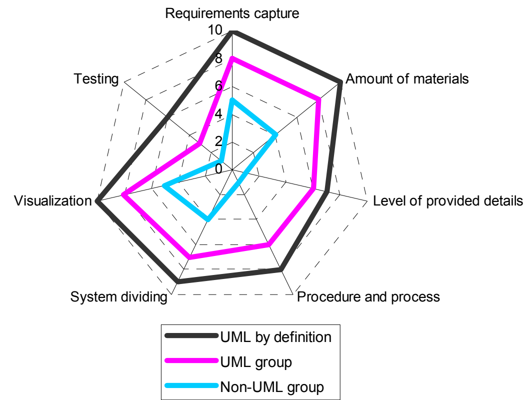

Figure 10. Summary of experimental results

6. Conclusions

UML offers many advantages. New projects have better survival prospects with UML. The experiment has shown that UML has a very strong impact on newcomers and that it can markedly increase their work and design capabilities.

Utilization of UML tools can additionally improve software design process. With these tools, documentation process can be included into development process, because the documentation is created during design time.

Practical application of UML proved to be successful. Requirement analysis and architectural design benefit from UML. Additionally, UML to SDL translation will also strengthen the process of the systems' dynamic structure design.

References

[1] G. Booch, J.Rumbaugh, I. Jacobson, *The Unified Modeling Language Use Guide,* Addison-Wesley, 1999.

[2] M. Fowler, Kendall Scott, *UML Distilled*, 2nd edition Addison-Wesley, 2000.

[3] P. Kruchten, *The Rational Unified Process*, 2nd edition Addison-Wesley, 2000.

[4] J. Ellsberger, D. Hogrefe, A. Sarma, *SDL Formal Object-oriented Language for Communicating Systems*, Prentice Hall 2000.

[5] O. Laitenberger, C. Atkinson, M. Sclich, K. El Amam, *An Experimental Comparison of Reading Techniques for Defect Detection in UML Design Documentation*, Fraunhofer, Kaiserslautern, 1999.