Unified Modeling Language: The Teen Years and Growing Pains

John Erickson[1] and Keng Siau[2]

[1] Department of Marketing and Management,
College of Business Administration,
University of Nebraska at Omaha, Omaha, NE 68182, USA
johnerickson@unomaha.edu

[2] Department of Business & Information Technology,
Missouri University of Science and Technology, Rolla, MO 65409, USA
siauk@mst.edu

Abstract. Unified Modeling Language (UML) is adopted by the Object Management Group as a standardized general-purpose modeling language for object-oriented software engineering. Despite its status as a standard, UML is still in a development stage and many studies have highlighted its weaknesses and challenges - including those related to human factor issues. Further, UML has grown considerably more complex since its inception. This paper traces the history of Unified Modeling Language (UML) from its formation to its current state and discusses the current state of the UML language. The paper first introduces UML and its various diagrams, and discusses its characteristics and features. The paper then looks at UML's strengths, challenges, and possible future development. The human factor issues with using UML are discussed and elaborated. Potential research questions related to UML are also highlighted.

Keywords: Unified Modeling Language, Human Factors, Systems Analysis and Design, Object Orientation.

1 Object-Orientation and UML's Genesis

1.1 Introduction

The continuing proliferation and development of information systems has proceeded at a pace amazing to even those intimately involved in the creation of such systems. The number of large companies building, revising or re-engineering their information systems seems to be ever increasing. Fortune 500 companies currently engaged upon multi-hundred million dollar IS/IT projects include Union Pacific Railroad, ConAgra Foods, Walmart, eBay/PayPal, and Blue-Cross Blue Shield to name only a very few. Given the increasing use of Cloud-based services, there appears to be some moves toward modularization and standardization of software and systems. Even with much progress, it appears, however, that software engineering is not keeping pace with the advances in hardware and general technological capabilities.

While technological change continuously swirls around many businesses and organizations , systems development often still adheres to the general ADCT (Analyze, Design, Code, Test) rubric, and utilizes such specific methodologies as the Waterfall Method, the Spiral Method, the System Life Cycle (alternatively known as the System Development Lifecycle, or SDLC), Prototyping, Rapid Application Development (RAD), Joint Application Development (JAD), End-User development, Outsourcing in various forms, or 'simply' buying pre-designed software from vendors (e.g., SAP, J. D. Edwards, Oracle, and People Soft).

In the past, systems and software development methods did not require that developers adhere to a specific approach to building systems, and while this may have been beneficial in that it allowed developers the freedom to choose a method that they were most comfortable with and knowledgeable about, such open-ended approaches can affect and constrain the system in unexpected ways or even result in failure. For example, system development and implementation failure rates remain stubbornly high. Cost overruns and time overruns are still the norm, rather than the exception. Also, open-ended approaches sometimes result in maintenance issues as not all systems analysts are trained in all methods. . In the current development environment, a different approach to systems development, one that provides close integration between analysis, design, and coding, would appear to be necessary. Further, many of the information system project failures are the result of human issues and human factors. Thus, not only do we need to enhance systems analysis and design methods, but we also need to enhance them with the human factor issues in mind. In this paper, we explore the role of the Unified Modeling Language (UML) as a modeling language that enables such an approach.

The first section of the paper explores the concept of object-orientation, including object-oriented systems analysis and design, the idea of modeling and modeling languages, and the history of UML. The following section, Section 2, covers the basic UML constructs, and Section 3 examines UML from a practical or practitioner perspective, while Section 4 discusses the future of UML and the human factor issues that need to be studied in enhancing UML.

1.2 Object-Orientation

Over the past two to three decades, object-oriented programming languages have emerged as the approach that many developers prefer to use during the Coding part of the systems development life cycle. However, in most cases the Analysis and Design steps have continued to proceed in the traditional style. This has often created tension, since traditional analysis and design are process-oriented instead of being object-oriented.

Object-oriented systems analysis and design (OOSAD) methods were developed to close the gap between the different stages, the first methods appearing in the 1980s. By the early 1990s, a virtual explosion in the number of OOSAD approaches began to flood the new paradigmatic environment. Between 1989 and 1994 the number of OO development methods grew from around ten to more than fifty (Booch, Rumbaugh and Jacobson, 1999). Two of these modeling languages are of particular interest for the purposes of this paper, Booch and Jacobson's OOSE (Object-Oriented Software Engineering), and Rumbaugh's OMT (Object Modeling Technique). A partial listing of methods/languages is shown below:

Table 1. Examples of Object Oriented Method/Language

• Bailin	• Hood
• Berard	• Jacobson
• Booch	• Martin-Odell
• Coad-Yourdon	• Rumbaugh
• Colbert	• Schlaer-Mellor
• Embley	• Seidewitz
• Firesmith	• UML
• Gibson	• Wirfs-Brock

1.3 The Emergence of UML

Prominent developers of different object-oriented modeling approaches joined forces to create UML, which was originally based on the two distinct OO modeling languages mentioned above: Booch and Jacobson's OOSE (Object-Oriented Software Engineering), and Rumbaugh's OMT (Object Modeling Technique). Development began in 1994 and continued through 1996, culminating in the January 1997 release of UML version 1.0 (Booch, Rumbaugh and Jacobson, 1999). The Object Management Group (OMG) adopted UML 1.1 as a standard modeling language in November of 1997. Version 2.4.1 is the most current release.

2 Current UML Models and Extensibility Mechanisms

2.1 Modeling

UML, as its name implies, is really all about creating models of software systems. Models are an abstraction of reality, meaning that we cannot, and really do not care to model in total reality settings, simply because of the complexity that such models would entail. Without abstraction, models would consume far more resources than any benefit gained from their construction. For the purposes of this paper, a model constitutes a view into the system. UML originally proposed a set of nine distinct modeling techniques representing nine different models or views of the system. With the release of UML 2.0 in July, 2005, five additional diagramming techniques were incorporated into the language. The techniques can be separated into structural (static) and behavioral (dynamic) views of the system. UML 2.4.1, the latest version of the modeling language, includes additional diagram types for model management.

Structural Diagrams

Profile Diagrams, Class Diagrams, Object Diagrams, Component Diagrams, Deployment Diagrams, Composite Structure Diagrams, and Package Diagrams comprise the static models of UML. Static models represent snapshots of the system at a given point or points in time, and do not relate information about how the system achieved the condition or state that it is in at each snapshot.

Class diagrams represent the basis of the OO paradigm to many adherents and depict class models. Class diagrams specify the system from both an analysis and design perspective. They depict what the system can do – analysis, and provide a blueprint showing how the system will be built – design (Ambler, 2000). Class diagrams are self-describing, and include a listing of the attributes, behaviors, and responsibilities of the system classes. Properly detailed class diagrams can be directly translated into physical (program code) form. In addition, correctly developed class diagrams can guide the software engineering process, as well as provide detailed system documentation (Lago, 2000).

Object Models and Diagrams represent specific occurrences or instances of class diagrams, and as such are generally seen as more concrete than the more abstract class diagrams.

Component diagrams depict the different parts of the software that constitute a system. This would include the interfaces of and between the components as well as their interrelationships. Ambler (2000) and Booch, Rumbaugh and Jacobson, (1999) defined component diagrams as class diagrams at a more abstract level.

Deployment diagrams can also be seen as a special case of class diagrams. In this case, the diagram models how the run-time processing units are connected and work together. The primary difference between component and deployment diagrams is that component diagrams focus on software units, while deployment diagrams depict the hardware arrangement for the proposed system.

Profile Diagrams are structure diagrams that describe lightweight extension mechanisms to the UML by defining custom stereotypes, tagged values, and constraints. Profiles allow adaptation of the UML metamodel for different platforms and domains.

Composite Structure Diagrams depict the relationships and communications between the functional parts of a system. These diagrams depict high level and abstract views of the system being modeled.

Package Diagrams are essentially a subtype of Class or Object Diagrams. They are intended to depict or show a grouping of related UML elements. Package Diagrams make it easier to see dependencies among different parts of the system being modeled (Pilone and Pitman, 2005). As such, Package Diagrams represent a high level view of the systems being modeled and therefore a good possibility for conveying understanding to users and other interested parties.

Behavioral Diagrams

Use Case Diagrams, Activity Diagrams, State Diagrams are the primary types of Behavioral Diagrams. The set of four Interaction Diagrams form a sub-type of Behavioral Diagram: Sequence Diagrams, Communication Diagrams, Interaction

Overview Diagrams, and Timing Diagrams. In contrast to the static diagrams, the dynamic diagrams in UML are intended to depict the behavior of the system as it transitions between states, interacts with users or internal classes and objects, and moves through the various activities that it is designed to accomplish.

Use Case Diagrams. While class models and diagrams represent the basis of OO as previously discussed, use case models and diagrams portray the system from the perspective of an end-user, and represent tasks that the system and users must execute in performance of their jobs (Pooley and Stevens, 1999). Use case models and the resulting use case diagrams consist of actors (those persons or systems outside the system of interest that need to interact with the system under development), use cases, and relationships among the actors and use cases. Booch, Rumbaugh, and Jacobson (1999) proposed that developers begin the analysis process with use cases. By that, they mean that developers should begin the analysis process by interviewing end users, perusing the basic legacy system documentation, etc. and creating from those interviews and documents the use cases that drive the class model development as well as the other models of the system. Dobing and Parsons (2006) propose that Use Case Narratives, which are written descriptions of Use Cases, are also heavily employed by developers to aid in assembling and understanding Use Case Diagrams.

Activity diagrams model the flow of control through activities in a system and, as such, are really just flow charts. In addition, activity diagrams are special instances of statechart diagrams.

State diagrams model state machines. State machines model the transition between states within an object, and the signals or events that trigger or elicit the change in state from one value to another (Booch, Rumbaugh and Jacobson, 1999). For example, a change in air temperature triggers a thermostat to activate a heating or cooling system that regulates the temperature in a room or building. A rise in air temperature in this case would be sensed by the thermostat and would cause the cooling system to change states from inactive (or idle) to active, and begin the cooling process. Once the ideal temperature is reached, the thermostat would sense that and trigger a state change in the cooling system back to inactive.

Interaction Diagrams are intended to depict communications, such as messages and events between and among objects. UML 2.X substantially enhanced flow of control in Interaction Diagrams over UML 1.X. The old isomorphic Sequence and Collaboration Diagrams from UML 1.X have been supplemented by two new diagrams -- Interaction Overview Diagrams and Timing Diagrams. Collaboration Diagrams are known as Communication Diagrams in UML 2.X. The four interaction diagrams are briefly described below.

Sequence Diagrams portray and validate in detail the logical steps of use cases (Ambler, 2000). Sequence diagrams depict the time ordering of messages between objects in the system, and as such include lifelines for the objects involved in the sequence as well as the focus of control at points in time (Booch, Rumbaugh and Jacobson, 1999).

Interaction Overview Diagrams are a simplification of and a sub-type of Activity Diagrams. These diagrams can aid the user in understanding the flow of control as a system operates, but they suppress the details of the messages and information the messages pass among objects. These are high level diagrams that are not intended to convey the specifics or details of how a system interacts with other systems or subsystems.

Communication Diagrams stress or depict the items involved in the interactions as opposed to the sequencing and control flows. There is some level of isomorphism present between Communication and Sequence Diagrams because one can easily be converted to the other. However, the mapping is not one to one (a formal isomorphism). In other words, some details can be lost when converting from Sequence Diagrams to Communication Diagrams.

Timing Diagrams are most often used with real-time systems and attempt to convey the timing element related to the massages being passed throughout the system being modeled. Timing Diagrams show a lifeline and the events that occur temporally as the system works at run time. The details of temporal constraints included in messages are the elements highlighted in these diagrams.

2.2 Extensibility Mechanisms

UML is intended to be a fully expressive modeling language. As such, UML possesses a formal grammar, vocabulary, and syntax for expressing the necessary details of the system models through the nine diagramming techniques. Even though UML represents a complete and formal modeling development language, there is no realistic way that it can suffice for all models across all systems.

In order to help deal with the dual problems of the general nature of UML, and the necessity to make it also domain specific, UML 2.X was developed to create a modified version of the language that can extend the language so that it also covers specific domains, rather like SAP's "industry solutions" for customizing their ERP product to specific industries. Similarly, UML allows tool developers to create profiles that might be tailored to a specific type of system, real-time, for example. UML 2.x continues with the three Extensibility mechanisms from UML 1.x While UML provides for four commonly used mechanisms, Specifications, Adornments, Common Divisions, and Extensibility (Booch, Rumbaugh and Jacobson, 1999), we will concern ourselves only with Extensibility for the purposes of this exposition.

Stereotypes

In a basic sense, Stereotypes simply add new "words" to UML's vocabulary. Stereotypes are generally derivations from existing structures already found within UML, but yet are different enough to be specific to a particular context. Booch, Rumbaugh and Jacobson (1999) used the example of modeling exceptions in C++ and Java as classes, with special types of attributes and behaviors.

Tagged Values

Providing information regarding version numbers or releases is an example of tagged values in use (Booch, Rumbaugh and Jacobson, 1999). Tagged values can be added to any UML building block to provide clarity to developers and users during and after the development cycle.

Constraints

Constraints are simply that – constraints. UML allows developers to add constraints to systems that modify or extend rules to apply (or not apply) under conditions and triggers that might be exceptions to those rules.

3 The Current State of UML

3.1 UML's Teen Years: The Positives

The characteristics of UML described in the preceding discussion have helped it gain broad acceptance and support among the developer community. The widespread adoption and use of UML as a primary, modeling language for OO systems development efforts can be seen as at least indirect evidence of the usability of the language in analysis, design, and implementation tasks.

UML presents a standard way of modeling object-oriented systems that enhances systems development efforts, and future enhancements to UML will provide even greater standardization and interoperability. UML also provides a vital and much needed communication connection (Fowler, 2000) between users and designers by incorporating use case modeling and diagramming in its repertoire.

UML can be used with a variety of development methodologies, and is not shackled to only one approach. This can only broaden its appeal and overall usefulness to developers in the industry. In a nutshell, UML has provided some vital and much needed stability in the modeling arena, and the software development community as a whole can only benefit from that (Siau and Cao, 2001; Siau and Loo 2006; Siau and Tian 2009).

Selic, Ramakers, and Kobryn (2002) propose that as information systems become ever more complex, modeling software for constructing understandable representations of those systems will become correspondingly more important for developers in such complex and quickly changing environments. As such, UML is still positioned to provide modeling support for developers. The continuing push toward MDA (Model-Driven Architecture) is evidence that, if a developer wishes, executable models are ever closer and more practical.

3.2 UML's Teen Years: the Negatives

Use cases are more process than object-oriented. Thus, the use case-centric approach has been criticized because it takes a process-oriented rather than an object-oriented view of system. This is a point of controversy among researchers and developers alike. Dobing and Parsons (2000) go so far as to propose that since use cases, and resulting use case diagrams, are process-oriented, their role in object-oriented systems development should be questioned, and possibly removed from use in OO development methods for that very reason.

However, since nearly all businesses are process-oriented, or at least are seen that way by most end users, it might be desirable, even necessary, for developers to capture essential end-user requirements by means of their process-based descriptions of the tasks that they and the system must perform. Other proposals to extend UML include the one by Tan, Alter, and Siau (2011). They propose that service responsibility table be used to supplement UML in system analysis. Another proposal is by Siau and Tan (2006) to use cognitive mapping techniques to supplement UML.

In addition, UML has been criticized for being overly complex, too complex for mere mortals to understand or learn to use in a reasonable time (Siau and Cao 2001). Some research has been done with regard to complexity. Rossi and Brinkkemper's (1996) study established a set of metrics for measuring diagram complexity, while Siau and Cao (2001) apply the metrics to UML and other modeling techniques. Their results indicated that although none of the individual UML diagrams is more complex than those used in other techniques, UML as a whole is much more complex. Since UML 2.0 was released in 2005, the language reached new levels of complexity.

Siau, Erickson, and Lee (2005), extending the work of Siau and Cao (2001), argue that there is a different between theoretical and practical complexity. Siau and Cao (2001) study the theoretical complexity of UML. In practice, not all the constructs will be used and some constructs are more important than others. Therefore, the practical complexity of UML is not as great as that computed by Siau and Cao (2001).

However, Duddy (2002) takes a more pessimistic view of both current and future versions of UML. He believes that even though UML 2.X provides coverage for development tools and paradigms currently in use, there is no way that it can provide support for emerging application development approaches or tools, such as application servers, loosely coupled messaging, and Web services. To be fair, however, it is also somewhat unrealistic to expect any tool to be a panacea for whatever methodologies, approaches or paradigms capture the attention of developers at any given point in time.

Finally, with the appearance of aspect-oriented programming, it is entirely possible that the entire object-oriented approach could be supplanted with a new paradigm. If that were to happen, it might become problematic in that UML is constrained by its limited extensibility mechanisms. In other words, we must ask whether or not UML is robust enough to adapt, or be adapted to a radically new paradigm?

3.3 UML's Teen Years: Growing Pains and Acne

UML's complexity discussed above means one of two things: companies considering the use of UML will either have to provide extensive (that is, expensive) training if they plan to develop in house, or they will have to hire UML trained (that is, also expensive) consultants to carry out their systems development efforts. Either way, this indicates that UML-based systems development projects will probably not get any less expensive in the future. However, the same criticism could be leveled at most other modeling tools as well.

As evidenced by several research streams since UML 2.0 was released, many developers and projects make use of UML simply as a post hoc documentation tool (Dobing and Parsons, 2005, Erickson & Siau, 2008). This basically means that there is a dichotomous split between two different types of UML users, those who support

MDA and fully executable modeling, and those who use UML simply as a documentation tool. These behaviors beg the following question: if most systems developers do not use many of the features and capabilities of UML, is it worthwhile to maintain in the language those capabilities and features that are rarely used? Only research into the issue will be able to answer that question. For more information regarding this issue see Siau and Halpin (2001).

None of this should be surprising, since no modeling tool will be adopted by everyone. However, a critical point here is the middle group, those that use UML, but not in a formal fashion, may do so by changing the diagramming techniques as they feel necessary in the pursuit of their projects. Valid questions are, do they not fully use the capabilities of UML because the tools are too expensive, because fully using UML is too complex, or for any of a variety of other reasons?

While these questions and suppositions are based on anecdotal evidence, at least one highlights an issue with UML. Do developers feel that UML is too complex for them to use easily? If the development tool is extremely difficult to use, then it appears that perhaps more effort is expended toward understanding and using the tools than toward developing the system, which is the primary goal in the first place. If so, and this would need research, then does or will future versions of UML be improvements from a usage perspective?

The paper by Siau and Tian (2009) points out some of the human factor issues in UML graphical notations. Human factor studies seem necessary to evaluate the usability of many of the UML constructs and propose modifications to the existing UML constructs to make them more useable and user friendly.

4 Conclusions

Neither UML nor any other modeling language, development method, or methodology has proven to be a panacea for the Analysis, Design, and Implementation of information systems. As suggested by Brooks (1987), this is not surprising because the inherent, essential characteristics of software development make it a fundamentally complex activity. UML is not perfect but it integrates many important software engineering practices that are important enhancements to systems development, and it does so in a way that, if not clear to everyone, is at least enlightening to developers.

Finally, looking back at the past 40 years of systems development chaos and woes, it appears that UML can and should be seen, problems notwithstanding, as one of the most important innovations in systems development since the advent of the structured approaches.

References

1. Ambler, S.: How the UML Models Fit Together (2000), `http://www.sdmagazine.com/articles/2000/003/003z/003z1.htmp?topic=uml`
2. Booch, G., Rumbaugh, J., Jacobson, I.: The Unified Modeling Language User Guide. Addison-Wesley, MA (1999)

3. Brooks, F.: No Silver Bullet: Essence and Accidents of Software Engineering. IEEE Computer 20(4), 10–19 (1987)
4. Dobing, B., Parsons, J.: Understanding the Role of Use Cases in UML: A Review and Research Agenda. Journal of Database Management 11(4), 28–36 (2000)
5. Dobing, B., Parsons, B.: How UML is Used. Communications of the ACM 49(5), 109–113 (2006)
6. Duddy, K.: UML2 Must Enable a Family of Languages. Communications of the ACM 45(11), 73–75 (2002)
7. Erickson, J., Siau, K.: Theoretical and Practical Complexity of Modeling Methods. Communications of the ACM 50(8), 46–51 (2007)
8. Fowler, M.: Why Use the UML? (2000), `http://www.sdmagazine.com/articles/2000/003/003z/003z3.htmp?topic=uml`
9. Kobryn, C.: What to Expect from UML 2.0. SD Times (2002)
10. Lago, P.: Rendering Distributed Systems in UML. In: Siau, K., Halpin, T. (eds.) Unified Modeling Language: Systems Analysis, Design, and Development Issues. Idea Group Publishing, Hershey (2000)
11. Mellor, S.: Make Models Be Assets. Communications of the ACM 45(11), 76–78 (2002)
12. Miller, J.: What UML Should Be. Communications of the ACM 45(11), 67–69 (2002)
13. Pilone, D., Pitman, N.: UML 2.0 in a Nutshell. O'Reilly Media (2005)
14. Pooley, R., Stevens, P.: Using UML: Software Engineering with Objects and Components. Addison Wesley Longman Limited, Harlow (1999)
15. Rossi, M., Brinkkemper, S.: Complexity Metrics for Systems Development Methods and Techniques. Information Systems 21(2), 209–227 (1996)
16. Selic, B., Ramackers, G., Kobryn, C.: Evolution, Not Revolution. Communications of the ACM 45(11), 70–72 (2002)
17. Siau, K., Cao, Q.: Unified Modeling Language - A Complexity Analysis. Journal of Database Management 12(1), 26–34 (2001)
18. Siau, K., Erickson, J., Lee, L.: Theoretical versus Practical Complexity: The Case of UML. Journal of Database Management 16(3), 40–57 (2005)
19. Siau, K., Lee, L.: Are Use Case and Class Diagrams Complementary in Requirements Analysis? – An Experimental Study on Use Case and Class Diagrams in UML. Requirements Engineering 9(4), 229–237 (2004)
20. Siau, K., Loo, P.: Identifying Difficulties in Learning UML. Information Systems Management 23(3), 43–51 (2006)
21. Siau, K., Halpin, T.: Unified Modeling Language: Systems Analysis, Design, and Development Issues. Idea Group Publishing, Hershey (2001)
22. Siau, K., Tan, X.: Using Cognitive Mapping Techniques to Supplement UML and UP in Information Requirements Determination. Journal of Computer Information Systems 46(5), 59–66 (2006)
23. Siau, K., Tian, Y.: A Semiotics Analysis of UML Graphical Notations. Requirements Engineering 14(1), 15–26 (2009)
24. Sieber, T., Siau, K., Nah, F., Sieber, M.: SAP Implementation at the University of Nebraska. Journal of Information Technology Cases and Applications 2(1), 41–72 (2000)
25. Tan, X., Alter, S., Siau, K.: Using Service Responsibility Tables to Supplement UML in Analyzing e-Service Systems. Decision Support Systems 51(3), 350–360 (2011)
26. Zhao, L., Siau, K.: Component-Based Development Using UML. Communications of the AIS 9, 207–222 (2002)