Unified Modeling Language (UML) — A Complexity Analysis

KENG SIAU, University of Nebraska-Lincoln
QING CAO, University of Nebraska-Lincoln

Unified Modeling Language (UML) has emerged as the software industry’s dominant modeling language. It is the de facto modeling language standard for specifying, visualizing, constructing, and documenting the components of software systems. Despite its prominence and status as the standard modeling language, UML has its critics. Opponents argue that it is complex and difficult to learn. Some question the rationale of having nine diagramming techniques in UML and the raison d’être of those nine techniques in UML. Others point out that UML lacks a comprehensive methodology to guide its users, which makes the language even more convoluted. A few studies on UML can be found in the literature. However, no study exists to provide a quantitative measure of UML complexity or to compare UML with other object-oriented techniques. In this research, we evaluate the complexity of UML using complexity metrics. The objective is to provide a reliable and accurate quantitative measure of UML complexity. A comparison of the complexity metrical values of UML with other object-oriented techniques was also carried out. Our findings suggest that each diagram in UML is not distinctly more complex than techniques in other OO methods. But as a whole, UML is very complex — 2-11 times more complex than other OO methods.

INTRODUCTION

Unified Modeling Language (UML) is a visual modeling language for modeling system requirements, describing designs, and depicting implementation details. Grady Booch, Jim Rumbaugh, and Ivars Jacobson, known collectively as the “three Amigos” at Rational Software Corp, spearheaded development of UML in the mid-1990s. Unified Modeling Language (UML) borrows concepts from a large number of different methods, and is tailored specifically for object-oriented systems development. Soon after its inception, UML emerged as the software industry’s dominant modeling language. UML is not only the de facto modeling language standard for specifying, visualizing, constructing, and documenting the components of software systems, but it has also been accepted by the Object Management Group (OMG) as a standard language for object-oriented analysis and design. In addition, UML has been proposed for standardization by the International Standards Organization (ISO) and approval is anticipated sometime in 2001.

Many of the language’s supporters claim that UML’s simplicity is its chief benefit (Kobryn, 1999) and argue that UML uses simple, intuitive notations that are understandable by nonprogrammers. This is in line with a recent study by Fedorowicz and Villeneuve (1999) who surveyed vendors and developers regarding OO systems analysis, design, and programming. Their results suggested that the OO approach is intuitive and easy to use/comprehend for vendors and developers alike. By offering a common blueprint language, UML also relieves developers of the proprietary ties that are so common in this industry. Major vendors including IBM, Microsoft, and Oracle are brought together under the UML umbrella. If developers, customers, and implementers can all understand a single modeling language instead of the few dozen OO techniques that existed before UML (Siau, 1999), they are more likely to agree on the intended functionality of the modeling techniques and constructs, thereby improving the communication process among the stakeholders and enhancing their chances of creating an application that truly addresses business problems.

UML, nevertheless, has its fair share of critics. Halpin and Bloesch (1999) pointed out that although UML facilitates the transition to object-oriented code, its implementation concerns render it less suitable for developing and validating a conceptual model with domain experts. Beckwith and Moore (1998) studied the use of deployment diagrams in architecture modeling and discovered problems with multitasking. In contrast to Kobryn (1999) who stated that UML is simple to use, Douglass (1998) argued that UML is somewhat large and complex, which can be daunting to novice users. He called for future research to examine the questions: (i) How easy is it to understand UML? and (ii) How easy is it to use UML?
This study investigates the complexity of UML. Complexity is a key measure of the effectiveness of a language because complexity directly affects the learn-ability and ease-of-use of the language. As mentioned by Booch et al. (1999a), UML is still in its infancy and there is a need for improvement in the analysis of UML and in understanding its use and functionality.

The rest of the paper is organized as follows: the next section reviews UML which is followed by a section discussing various approaches to measuring complexity, including empirical studies and different complexity metrics. The complexity of UML is then analyzed using the complexity metrics proposed by Rossi and Brinkkemper (1994). The research findings are discussed and a comparison is made between the complexity of UML and other OO techniques. The last section summarizes the research, lists the limitations of the study, and suggests future research directions.

OBJECT ORIENTATION AND UML

Object-orientation (OO) is the new paradigm for systems development (Johnson and Hardgrave, 1999). The first few years of the 1990s saw the blossoming of nearly 50 different object oriented methods. The abundance of OO methods caused great confusion among users. Standardization of OO was needed to curtail the chaos and the Unified Modeling Language (UML) was the result.

The Unified Modeling Language (UML) includes nine modeling diagrams. They are class diagrams, use-case diagrams, state-chart diagrams, activity diagrams, sequence diagrams, collaboration diagrams, object diagrams, components diagrams, and deployment diagrams (Booch et al., 1999a). These diagrams model both the static view and the dynamic view of the system. While class diagrams, object diagrams, component diagrams, and deployment diagrams model the static view of the system, statechart diagrams, activity diagrams, sequence diagrams, and collaboration diagrams depict the dynamic view of the system. Each diagram shows a different aspect of the system, and is by design incomplete.

A class diagram shows a set of classes, interfaces, and collaborations and their relationships. An object diagram depicts static “snapshots” of the elements within a system, showing objects’ structure, attributes, and relationships to one another. An activity diagram shows the flow of control from one activity to the next, and a use-case diagram illustrates how elements outside the system use the system. For instance, the internal workings of a new payroll system would be shown in an activity diagram, whereas external actors, such as the mail order department, would appear in a use-case diagram. Sequence and collaboration diagrams show interactive processes: developers see not only objects and classes, but also the messages that pass between them. A statechart diagram shows a state machine, consisting of states, transitions, events, and activities. Finally, component and deployment diagrams show the physical or implementation view of the system (including executables, libraries, and interfaces).

In this study, we measure the complexity of each diagram in UML and the complexity of UML as a whole.

MEASURING COMPLEXITY

Siau and Rossi (1998) completed a comprehensive review of the approaches for evaluating modeling methods. They broadly categorized evaluation methods into two categories – empirical versus non-empirical. Empirical studies include those conducted on ease-of-use of data modeling methods (e.g., Chan and Lim, 1998; Chan et al., 1998; Siau et al., 1997; Siau et al., 1995; Chan et al., 1993; Batra et al., 1990). For instance, Chan et al. (1993) conducted an experiment to compare the ER and relational models using textual query languages specifically designed for each model respectively. Siau et al. (1995) compared the relational and ER models using visual query languages. Non-empirical evaluation approaches include feature comparison, metamodeling, paradigmatic analysis, and metrics analysis (e.g., Siau 2000, Siau 1999, Rossi and Brinkkemper 1994, Wand and Weber, 1993). In this study, we use the metrics analysis to evaluate UML.

Complexity Measures

The quality of a complexity measure rests on its explanatory power and applicability. Explanatory power refers to the measure’s ability to explain the interrelationships among complexity, quality, and other programming and design parameters. Applicability refers to the degree to which the measure can be applied to improve the quality of work during the design, coding, and testing stages. There are several major complexity measures in the literature.

Lines of Code

Lines-of-code (LOC) is a count of instruction statements. Because LOC count represents the program size and complexity, it is not a surprise that the more lines of code there are in a program, the more defects should be expected. A concave relationship between number of defects and module size was suggested. Withrow (1990) examined modules written in Ada for a large project at Unisys and confirmed the concave relationship between defect density and module size. The author argued that there might be an optimal program size that could lead to the lowest defect rate. As module size becomes large, the complexity increases to a level beyond a programmer’s immediate span of control and total comprehension.

Halstead’s Software Science

Halstead (1977) developed a system of equations expressing the total vocabulary, overall program length, potential minimum volume for an algorithm, program level (a measure of software complexity), program difficulty, and