UML Overview

This chapter is a quick overview of UML and what it is good for.

**Brief Summary of UML**

The Unified Modeling Language (UML) is a general-purpose visual modeling language that is used to specify, visualize, construct, and document the artifacts of a software system. It captures decisions and understanding about systems that must be constructed. It is used to understand, design, browse, configure, maintain, and control information about such systems. It is intended for use with all development methods, lifecycle stages, application domains, and media. The modeling language is intended to unify past experience about modeling techniques and to incorporate current software best practices into a standard approach. UML includes semantic concepts, notation, and guidelines. It has static, dynamic, environmental, and organizational parts. It is intended to be supported by interactive visual modeling tools that have code generators and report writers. The UML specification does not define a standard process but is intended to be useful with an iterative development process. It is intended to support most existing object-oriented development processes.

The UML captures information about the static structure and dynamic behavior of a system. A system is modeled as a collection of discrete objects that interact to perform work that ultimately benefits an outside user. The static structure defines the kinds of objects important to a system and to its implementation, as well as the relationships among the objects. The dynamic behavior defines the history of objects over time and the communications among objects to accomplish goals. Modeling a system from several separate but related viewpoints permits it to be understood for different purposes.

The UML also contains organizational constructs for arranging models into packages that permit software teams to partition large systems into workable pieces, to understand and control dependencies among the packages, and to
manage the versioning of model units in a complex development environment. It contains constructs for representing implementation decisions and for organizing run-time elements into components.

UML is not primarily a programming language. It can be used to write programs, but it lacks the syntactic and semantic conveniences that most programming languages provide to ease the task of programming. Tools can provide code generators from UML into a variety of programming languages, as well as construct reverse-engineered models from existing programs. The UML is not a highly formal language intended for theorem proving. There are a number of such languages, but they are not easy to understand or to use for most purposes. The UML is a general-purpose modeling language. For specialized domains, such as GUI layout, VLSI circuit design, or rule-based artificial intelligence, a more specialized tool with a special language might be appropriate. UML is a discrete modeling language. It is not intended to model continuous systems such as those found in engineering and physics. UML is intended to be a universal general-purpose modeling language for discrete systems such as those made of software, firmware, or digital logic.

UML History

UML was developed in an effort to simplify and consolidate the large number of object-oriented development methods that had emerged.

Object-oriented development methods

Development methods for traditional programming languages, such as Cobol and Fortran, emerged in the 1970s and became widespread in the 1980s. Foremost among them was Structured Analysis and Structured Design [Yourdon-79] and its variants, such as Real-Time Structured Design [Ward-85] and others. These methods, originally developed by Constantine, DeMarco, Mellor, Ward, Yourdon, and others, achieved some penetration into the large system area, especially for government-contracted systems in the aerospace and defense fields, in which contracting officers insisted on an organized development process and ample documentation of the system design and implementation. The results were not always as good as hoped for—many computer-aided software engineering (CASE) systems were little more than report generators that extracted designs after the implementation was complete—but the methods included good ideas that were occasionally used effectively in the construction of large systems. Commercial applications were more reluctant to adopt large CASE systems and development methods. Most businesses developed software internally for their own needs, without the adversarial relationship between customer and contractors that characterized large government projects. Commercial systems were perceived to be
simpler, whether or not this was actually true, and there was less need for review by outside organizations.

The first object-oriented language is generally acknowledged to be Simula-67 [Birtwistle-75], developed in Norway in 1967. This language never had a significant following, although it greatly influenced the developers of several of the later object-oriented languages. The work of Dahl and Nygaard had a profound influence on the development of object orientation. The object-oriented movement became active with the widespread availability of Smalltalk in the early 1980s, followed by other object-oriented languages, such as Objective C, C++, Eiffel, and CLOS. The actual usage of object-oriented languages was limited at first, but object-orientation attracted a lot of attention. About five years after Smalltalk became widely known, the first object-oriented development methods were published by Shlaer/Mellor [Shlaer-88] and Coad/Yourdon [Coad-91], followed soon thereafter by Booch [Booch-94], Rumbaugh/Blaha/Premerlani/Eddy/Lorensen [Rumbaugh-91] (updated as [Blaha-05]), and Wirfs-Brock/Wilkerson/Wiener [Wirfs-Brock-90] (note that copyright years often begin in July of the previous calendar year). These books, added to earlier programming-language design books by Goldberg/Robson [Goldberg-83], Cox [Cox-86], and Meyer [Meyer-88], started the field of object-oriented methodology. The first phase was complete by the end of 1990. The Objectory book [Jacobson-92] was published slightly later, based on work that had appeared in earlier papers. This book took a somewhat different approach, with its focus on use cases and the development process.

Over the next five years, a plethora of books on object-oriented methodology appeared, each with its own set of concepts, definitions, notation, terminology, and process. Some added useful new concepts, but overall there was a great similarity among the concepts proposed by different authors. Many of the newer books started from one or more of the existing methods and made extensions or minor changes. The original authors were not idle either; most of them updated their original work, often incorporating good ideas from other authors. In general, there emerged a pool of common core concepts, together with a wide variety of concepts embraced by one or two authors but not widely used. Even in the core concepts, there were minor discrepancies among methods that made detailed comparison somewhat treacherous, especially for the casual reader.

Unification effort

There were some early attempts to unify concepts among methods. A notable example was Fusion by Coleman and his colleagues [Coleman-94], which included concepts from OMT [Rumbaugh-91], Booch [Booch-94], and CRC [Wirfs-Brock-90]. As it did not involve the original authors, it must be regarded as another new method rather than as a replacement of several existing methods. The first successful attempt to combine and replace existing approaches came when
Rumbaugh joined Booch at Rational Software Corporation in 1994. They began combining the concepts from the OMT and Booch methods, resulting in a first proposal in 1995. At that time, Jacobson also joined Rational and began working with Booch and Rumbaugh. Their joint work was called the Unified Modeling Language (UML). The momentum gained by having the authors of three of the top methods working together to unify their approaches shifted the balance in the object-oriented methodology field, where there had previously been little incentive (or at least little willingness) for methodologists to abandon some of their own concepts to achieve harmony.

In 1996, the Object Management Group (OMG) issued a request for proposals for a standard approach to object-oriented modeling. UML authors Booch, Jacobson, and Rumbaugh began working with methodologists and developers from other companies to produce a proposal attractive to the membership of OMG, as well as a modeling language that would be widely accepted by tool makers, methodologists, and developers who would be the eventual users. Several competing efforts also were started. Eventually, all the proposals coalesced in the final UML proposal that was submitted to the OMG in September 1997. The final product is a collaboration among many people. We began the UML effort and contributed a few good ideas, but the ideas in it are the product of many minds.

**Standardization**

The Unified Modeling Language was adopted unanimously by the membership of the OMG as a standard in November 1997 [UML-98]. The OMG assumed responsibility for the further development of the UML standard. Even before final adoption, a number of books were published outlining the highlights of the UML. Many tool vendors announced support or planned support for the UML, and several methodologists announced that they would use UML notation for further work. UML has now supplanted most, if not all, of the previous modeling notations in development processes, modeling tools, and articles in the technical literature. The emergence of the UML appears to have been attractive to the general computing public because it consolidates the experiences of many authors with an official status that has reduced gratuitous divergence among tools.

Noteworthy is a series of international research conferences with the title UML yyyy, where yyyy is a year starting with 1998 and continuing annually [UMLConf]. Also note the yearly [ECOOP] and [OOPSLA] conferences dealing with object-oriented technology in general.

**UML2**

After several years of experience using UML, the OMG issued requests for proposals to upgrade UML to fix problems uncovered by experience of use and to extend
it with additional capabilities that were desired in several application domains. Proposals were developed from November 2000 to July 2003, with a specification of UML version 2.0 being adopted by the OMG membership shortly thereafter [UML-04]. The adopted specification underwent the normal OMG finalization process to fix bugs and problems encountered in initial implementation, with a final available specification expected at the end of 2004 or beginning of 2005.

In this book, we use the term UML1 to refer to UML specification versions 1.1 to 1.5 and UML2 to refer to UML specification versions 2.0 and higher.

New features. In general, UML2 is mostly the same as UML1, especially regarding the most commonly used, central features. Some problem areas have been modified, a few major enhancements have been added, and many small bugs have been fixed, but users of UML1 should have little trouble using UML2. The new version may be considered like a new version of a programming language or an application. Some of the most important changes visible to users are:

- Sequence diagram constructs and notation based largely on the ITU Message Sequence Chart standard, adapted to make it more object-oriented.
- Decoupling of activity modeling concepts from state machines and the use of notation popular in the business modeling community.
- Unification of activity modeling with the action modeling added in UML version 1.5, to provide a more complete procedural model.
- Contextual modeling constructs for the internal composition of classes and collaborations. These constructs permit both loose and strict encapsulation and the wiring of internal structures from smaller parts.
- Repositioning of components as design constructs and artifacts as physical entities that are deployed.

Internal mechanisms. Other changes affect the internal representation of UML constructs (the metamodel) and its relationship to other specifications. These changes will not concern most users directly, but they are important to toolmakers because they affect interoperability across multiple specifications, therefore they will affect users indirectly:

- Unification of the core of UML with the conceptual modeling parts of MOF (Meta-Object Facility). This permits UML models to be handled by generic MOF tools and repositories.
- Restructuring of the UML metamodel to eliminate redundant constructs and to permit reuse of well-defined subsets by other specifications.
- Availability of profiles to define domain and technology-specific extensions of UML.
Other sources

In addition to the various development methods cited above and a number of others that came a bit later, certain UML views show strong influences from particular non-object-oriented sources.

The static view, with classes connected by various relationships, is strongly influenced by Peter Chen’s Entity-Relationship (ER) model originally developed in 1976. The influence came into UML through most of the early object-oriented methods. The ER model also heavily influenced database systems. The programming language world and the database world have unfortunately mostly gone their separate ways.

State machine models have been used in computer science and electrical engineering for many years. David Harel’s statecharts are an important extension to classical state machines that add the concept of nested and orthogonal states. Harel’s ideas were adapted by OMT, and from there into other methods and eventually into UML, where they form the basis of the state machine view.

The sequence diagram notation of UML2 is taken from the ITU Message Sequence Chart (MSC) standard [ITU-T Z.120], adapted to make it match other UML concepts better. This standard, which has been widely used in the telecommunications industry, replaces the sequence diagram notation of UML1 by adding a number of structured constructs to overcome problems in the previous UML1 notation. The ITU is considering whether to adopt some or all of the changes into the ITU standard.

The structured classifier concepts of UML2 were strongly influenced by the real-time engineering constructs of SDL [ITU-T Z.100], MSC, and the ROOM method [Selic-94].

The activity diagram notation of UML1, and even more that of UML2, is heavily influenced by various business process modeling notations. Because no single business process modeling notation was dominant, the UML notation was selected from various sources.

There are many other influences of UML, and often the original source of an idea precedes the person who is famous for popularizing it. About 20 persons were major contributors to the UML1 specification, with many others participating in a lesser way. Maybe 30 or so played major roles in the development of UML2, with scores of others submitting suggestions, reviewing proposals, and writing books. It is impossible to list everyone who contributed to UML, and the brief references that we have included undoubtedly overlook some important contributors, for which we ask understanding.
What does unified mean?

The word *unified* has the following relevant meanings for UML.

**Across historical methods and notations.** The UML combines the commonly accepted concepts from many object-oriented methods, selecting a clear definition for each concept, as well as a notation and terminology. The UML can represent most existing models as well as or better than the original methods can.

**Across the development lifecycle.** The UML is seamless from requirements to deployment. The same set of concepts and notation can be used in different stages of development and even mixed within a single model. It is unnecessary to translate from one stage to another. This seamlessness is critical for iterative, incremental development.

**Across application domains.** The UML is intended to model most application domains, including those involving systems that are large, complex, real-time, distributed, data or computation intensive, among other properties. There may be specialized areas in which a special-purpose language is more useful, but UML is intended to be as good as or better than any other general-purpose modeling language for most application areas.

**Across implementation languages and platforms.** The UML is intended to be usable for systems implemented in various implementation languages and platforms, including programming languages, databases, 4GLs, organization documents, firmware, and so on. The front-end work should be identical or similar in all cases, while the back-end work will differ somewhat for each medium.

**Across development processes.** The UML is a modeling language, not a description of a detailed development process. It is intended to be usable as the modeling language underlying most existing or new development processes, just as a general-purpose programming language can be used in many styles of programming. It is particularly intended to support the iterative, incremental style of development that we recommend.

**Across internal concepts.** In constructing the UML metamodel, we made a deliberate effort to discover and represent underlying relationships among various concepts, trying to capture modeling concepts in a broad way applicable to many known and unknown situations. This process led to a better understanding of the concepts and a more general applicability of them. This was not the original purpose of the unification work, but it was one of the most important results.
Goals of UML

There were a number of goals behind the development of UML. First and most important, UML is a general-purpose modeling language that all modelers can use. It is nonproprietary and based on common agreement by much of the computing community. It is meant to include the concepts of the leading methods so that it can be used as their modeling language. At the very least, it was intended to supersede the models of OMT, Booch, and Objectory, as well as those of other participants of the proposal. It was intended to be as familiar as possible; whenever possible, we used notation from OMT, Booch, Objectory, and other leading methods. It is meant to support good practices for design, such as encapsulation, separation of concerns, and capture of the intent of a model construct. It is intended to address current software development issues, such as large scale, distribution, concurrency, patterns, and team development.

UML is not intended to be a complete development method. It does not include a step-by-step development process. We believe that a good development process is crucial to the success of a software development effort, and we propose one in a companion book [Jacobson-99]. It is important to realize that UML and a process for using UML are two separate things. UML is intended to support all, or at least most, of the existing development processes. UML includes the concepts that we believe are necessary to support a modern iterative process based on building a strong architecture to solve user-case–driven requirements.

A final goal of UML was to be as simple as possible while still being capable of modeling the full range of practical systems that need to be built. UML needs to be expressive enough to handle all the concepts that arise in a modern system, such as concurrency and distribution, as well as software engineering mechanisms, such as encapsulation and components. It must be a universal language, like any general-purpose programming language. Unfortunately, that means that it cannot be small if we want it to handle things other than toy systems. Modern languages and modern operating systems are more complicated than those of 50 years ago because we expect much more of them. UML has several kinds of models; it is not something you can master in one day. It is more complicated than some of its antecedents because it is intended to be more comprehensive. But you don’t have to learn it all at once, any more than you would a programming language, an operating system, or a complex user application, not to mention a natural language or skill.

Complexity of UML

UML is a large and varied modeling language intended for use on many different levels and at many stages of the development lifecycle. It has been criticized for being large and complex, but complexity is inherent in any universal application that is intended for realistic use on real-world problems, such as operating systems,
programming languages, multimedia editing applications, spreadsheet editors, and desktop publishing systems. Such applications can be kept small only at the cost of making them toys, and the developers of UML did not wish it to be a toy. The complexity of UML must be understood in light of its history:

- UML is a product of consensus of persons with varied goals and interests. It shares the qualities of the product of a democratic process. It is not as clean or coherent as the product of a single will. It contains superfluous features (but different persons might disagree about exactly what is superfluous). It contains overlapping features that are not always well integrated. Most of all, it lacks a consistent viewpoint. Unlike a programming language, which has a fairly narrow usage, it is intended for all kinds of things, from business modeling to graphical programming. Wide breadth of applicability usually comes at the expense of specificity.

- It was originally the merger of four or five leading modeling approaches, and later has been the target for accommodating a number of existing notations, such as SDL (Specification and Description Language, [ITU-T Z.100]), various business modeling languages (which themselves had no single standard), action languages, state machine notations, and so on. The desire to preserve previous notation often creates inconsistencies across features and includes redundant notation intended to cater to the familiarities of certain usage groups.

- The official specification documents have been written by teams of uneven ability. There is a wide variation in style, completeness, precision, and consistency among various sections of the documents.

- UML is not a precise specification in the manner of a formal language. Although the computer science community holds formality to be a virtue, few mainstream programming languages are precisely defined, and formal languages are often inaccessible even to experts. It should also be noted that modeling is not the same as coding. In the construction industry, blueprints are written in an informal style using many conventions that depend on the common sense of the craftsperson, but buildings are built from them successfully.

- The semantics sections sometimes contain vague statements without adequate explanation and examples. Terms are introduced in metamodels and not well distinguished from other terms. There are too many fine distinctions that someone thought important but did not explain clearly.

- There is far too much use of generalization at the expense of essential distinctions. The myth that inheritance is always good has been a curse of object-orientation from its earliest days.

- There is a tension between concepts for conceptual modeling and programming language representation, with no consistent guidelines.
UML Assessment

- UML is messy, imprecise, complex, and sprawling. That is both a fault and a virtue. Anything intended for such widespread usage is going to be messy.
- You don’t have to know or use every feature of UML any more than you need to know or use every feature of a large software application or programming language. There is a small set of central concepts that are widely used. Other features can be learned gradually and used when needed.
- UML can be and has been used in many different ways in real-world development projects.
- UML is more than a visual notation. UML models can be used to generate code and test cases. This requires an appropriate UML profile, use of tools matched to the target platform, and choices among various implementation trade-offs.
- It is unnecessary to listen too much to UML language lawyers. There is no single right way to use it. It is one of many tools that a good developer uses. It doesn’t have to be used for everything. You can modify it to suit your own needs provided you have the cooperation of your colleagues and software tools.

UML Concept Areas

UML concepts and models can be grouped into the following concept areas.

Static structure. Any precise model must first define the universe of discourse, that is, the key concepts from the application, their internal properties, and their relationships to each other. This group of constructs is the static view. Application concepts are modeled as classes, each of which describes discrete objects that hold information and communicate to implement behavior. The information they hold is modeled as attributes; the behavior they perform is modeled as operations. Several classes can share their common structure using generalization. A child class adds incremental structure and behavior to the structure and behavior that it obtains by inheritance from the common parent class. Objects also have run-time connections to other individual objects. Such object-to-object relationships are modeled as associations among classes. Some relationships among elements are grouped together as dependency relationships, including relationships among levels of abstraction, binding of template parameters, granting of permission, and usage of one element by another. Classes may have interfaces, which describe their externally-visible behavior. Other relationships are include and extend dependencies of use cases. The static view is notated using class diagrams and its variants. The static view can be used to generate most data structure declarations in a program. There are several other kinds of elements in UML diagrams, such as interfaces, data types, use cases, and signals. Collectively, these are called classifiers,
and they behave much like classes with certain additions and restrictions on each kind of classifier.

**Design constructs.** UML models are meant for both logical analysis and designs intended for implementation. Certain constructs represent design items. A structured classifier expands a class into its implementation as a collection of parts held together by connectors. A class can encapsulate its internal structure behind externally visible ports. A collaboration models a collection of objects that play roles within a transient context. A component is a replaceable part of a system that conforms to and provides the realization of a set of interfaces. It is intended to be easily substitutable for other components that meet the same specification.

**Deployment constructs.** A node is a run-time computing resource that defines a location. An artifact is a physical unit of information or behavior description in a computing system. Artifacts are deployed on nodes. An artifact can be a manifestation, that is, an implementation, of a component. The deployment view describes the configuration of nodes in a running system and the arrangement of artifacts on them, including manifestation relationships to components.

**Dynamic behavior.** There are three ways to model behavior. One is the life history of one object as it interacts with the rest of the world; another is the communication patterns of a set of connected objects as they interact to implement behavior; the third is the evolution of the execution process as it passes through various activities.

The view of an object in isolation is a state machine—a view of an object as it responds to events based on its current state, performs actions as part of its response, and transitions to a new state. State machines are displayed in state machine diagrams.

An interaction overlays a structured classifier or collaboration with the flow of messages between parts. Interactions are shown in sequence diagrams and communication diagrams. Sequence diagrams emphasize time sequences, whereas communication diagrams emphasize object relationships.

An activity represents the execution of a computation. It is modeled as a set of activity nodes connected by control flows and data flows. Activities can model both sequential and concurrent behavior. They include traditional flow-of-control constructs, such as decisions and loops. Activity diagrams may be used to show computations as well as workflows in human organizations.

Guiding all the behavior views is a set of use cases, each a description of a slice of system functionality as visible to an actor, an external user of the system. The use case view includes both the static structure of the use cases and their actors as well as the dynamic sequences of messages among actors and system, usually expressed as sequence diagrams or just text.
Model organization. Computers can deal with large flat models, but humans cannot. In a large system, the modeling information must be divided into coherent pieces so that teams can work on different parts concurrently. Even on a smaller system, human understanding requires the organization of model content into packages of modest size. Packages are general-purpose hierarchical organizational units of UML models. They can be used for storage, access control, configuration management, and constructing libraries that contain reusable model fragments. A dependency between packages summarizes the dependencies among the package contents. A dependency among packages can be imposed by the overall system architecture. Then the contents of the packages must conform to the package dependencies and to the imposed system architecture.

Profiles. No matter how complete the facilities in a language, people will want to make extensions. UML contains a limited extensibility capability that should accommodate most of the day-to-day needs for extensions, without requiring a change to the basic language. A stereotype is a new kind of model element with the same structure as an existing element, but with additional constraints, a different interpretation and icon, and different treatment by code generators and other back-end tools. A stereotype defines a set of tagged values. A tagged value is a user-defined attribute that applies to model elements themselves, rather than objects in the run-time system. For example, tagged values may indicate project management information, code generator guidance, and domain-specific information. A constraint is a well-formedness condition expressed as a text string in some constraint language, such as a programming language, special constraint language, or natural language. UML includes a constraint language called OCL. A profile is a set of stereotypes and constraints for a particular purpose that can be applied to user packages. Profiles can be developed for particular purposes and stored in libraries for use in user models. As with any extensibility mechanism, these mechanisms must be used with care because of the risk of producing a private dialect unintelligible to others. But they can avoid the need for more radical changes.