

Introducing the UML

The Unified Modeling Language (UML) is a standard language for writing software blueprints. The UML may be used to visualize, specify, construct, and document the artifacts of a software-intensive system.

The UML is appropriate for modeling systems ranging from enterprise information systems to distributed Web-based applications and even to hard real time embedded systems. It is a very expressive language, addressing all the views needed to develop and then deploy such systems. Even though it is expressive, the UML is not difficult to understand and to use. Learning to apply the UML effectively starts with forming a conceptual model of the language, which requires learning three major elements: the UML's basic building blocks, the rules that dictate how these building blocks may be put together, and some common mechanisms that apply throughout the language.

The UML is only a language, so it is just one part of a software development method. The UML is process independent, although optimally it should be used in a process that is use case driven, architecture-centric, iterative, and incremental.

An Overview of the UML

The UML is a language for

- Visualizing
- Specifying
- Constructing
- Documenting

the artifacts of a software-intensive system.

The UML Is a Language

A language provides a vocabulary and the rules for combining words in that vocabulary for the purpose of communication. A *modeling language* is a language whose vocabulary and rules focus on the conceptual and physical representation of a system. A modeling language such as the UML is thus a standard language for software blueprints.

The basic principles of modeling are discussed in this chapter.

Modeling yields an understanding of a system. No one model is ever sufficient. Rather, you often need multiple models that are connected to one another to understand anything but the most trivial system. For software-intensive systems, this requires a language that addresses the different views of a system's architecture as it evolves throughout the software development life cycle.

The vocabulary and rules of a language such as the UML tell you how to create and read well-formed models, but they don't tell you what models you should create and when you should create them. That's the role of the software development process. A well-defined process will guide you in deciding what artifacts to produce, what activities and what workers to use to create them and manage them, and how to use those artifacts to measure and control the project as a whole.

The UML Is a Language for Visualizing

For many programmers, the distance between thinking of an implementation and then pounding it out in code is close to zero. You think it, you code it. In fact, some things are best cast directly in code. Text is a wonderfully minimal and direct way to write expressions and algorithms.

In such cases, the programmer is still doing some modeling, albeit entirely mentally. He or she may even sketch out a few ideas on a white board or on a napkin. However, there are several problems with this. First, communicating those conceptual models to others is error-prone unless everyone involved speaks the same language. Typically, projects and organizations develop their own language, and it is difficult to understand what's going on if you are an outsider or new to the group. Second, there are some things about a software system you can't understand unless you build models that transcend the textual programming language. For example, the meaning of a class hierarchy can be inferred, but not directly grasped, by staring at the code for all the classes in the hierarchy. Similarly, the physical distribution and possible migration of the objects in a Web-based system can be inferred, but not directly grasped, by studying the system's code. Third, if the developer who cut the code never wrote down the models that are in his or her head, that information would be lost forever or, at best, only partially recreatable from the implementation once that developer moved on.

Writing models in the UML addresses the third issue: An explicit model facilitates communication.

Some things are best modeled textually; others are best modeled graphically. Indeed, in all interesting systems, there are structures that transcend what can

The complete semantics of the UML are discussed in *The Unified Modeling Language Reference Manual*.

be represented in a programming language. The UML is such a graphical language. This addresses the second problem described earlier.

The UML is more than just a bunch of graphical symbols. Rather, behind each symbol in the UML notation is a well-defined semantics. In this manner, one developer can write a model in the UML, and another developer, or even another tool, can interpret that model unambiguously. This addresses the first issue described earlier.

The UML Is a Language for Specifying

In this context, *specifying* means building models that are precise, unambiguous, and complete. In particular, the UML addresses the specification of all the important analysis, design, and implementation decisions that must be made in developing and deploying a software-intensive system.

The UML Is a Language for Constructing

The UML is not a visual programming language, but its models can be directly connected to a variety of programming languages. This means that it is possible to map from a model in the UML to a programming language such as Java, C++, or Visual Basic, or even to tables in a relational database or the persistent store of an object-oriented database. Things that are best expressed graphically are done so graphically in the UML, whereas things that are best expressed textually are done so in the programming language.

Modeling the structure of a system is discussed in Parts 2 and 3.

This mapping permits forward engineering—the generation of code from a UML model into a programming language. The reverse is also possible: You can reconstruct a model from an implementation back into the UML. Reverse engineering is not magic. Unless you encode that information in the implementation, information is lost when moving forward from models to code. Reverse engineering thus requires tool support with human intervention. Combining these two paths of forward code generation and reverse engineering yields round-trip engineering, meaning the ability to work in either a graphical or a textual view, while tools keep the two views consistent.

Modeling the behavior of a system is discussed in Parts 4 and 5.

In addition to this direct mapping, the UML is sufficiently expressive and unambiguous to permit the direct execution of models, the simulation of systems, and the instrumentation of running systems.

The UML Is a Language for Documenting

A healthy software organization produces all sorts of artifacts in addition to raw executable code. These artifacts include (but are not limited to)

- Requirements
- Architecture
- Design
- Source code
- Project plans
- Tests
- Prototypes
- Releases

Depending on the development culture, some of these artifacts are treated more or less formally than others. Such artifacts are not only the deliverables of a project; they are also critical in controlling, measuring, and communicating about a system during its development and after its deployment.

The UML addresses the documentation of a system's architecture and all of its details. The UML also provides a language for expressing requirements and for tests. Finally, the UML provides a language for modeling the activities of project planning and release management.

Where Can the UML Be Used?

The UML is intended primarily for software-intensive systems. It has been used effectively for such domains as

- Enterprise information systems
- Banking and financial services
- Telecommunications
- Transportation
- Defense/aerospace
- Retail
- Medical electronics
- Scientific
- Distributed Web-based services

The UML is not limited to modeling software. In fact, it is expressive enough to model nonsoftware systems, such as workflow in the legal system, the structure and behavior of a patient healthcare system, software engineering in aircraft combat systems, and the design of hardware.

A Conceptual Model of the UML

To understand the UML, you need to form a conceptual model of the language, and this requires learning three major elements: the UML's basic building blocks, the rules that dictate how those building blocks may be put together, and some common mechanisms that apply throughout the UML. Once you have grasped these ideas, you will be able to read UML models and create some basic ones. As you gain more experience in applying the UML, you can build on this conceptual model, using more advanced features of the language.

Building Blocks of the UML

The vocabulary of the UML encompasses three kinds of building blocks:

1. Things
2. Relationships
3. Diagrams

Things are the abstractions that are first-class citizens in a model; relationships tie these things together; diagrams group interesting collections of things.

Things in the UML There are four kinds of things in the UML:

1. Structural things
2. Behavioral things
3. Grouping things
4. Annotational things

These things are the basic object-oriented building blocks of the UML. You use them to write well-formed models.

Structural Things *Structural things* are the nouns of UML models. These are the mostly static parts of a model, representing elements that are either conceptual or physical. Collectively, the structural things are called *classifiers*.

Classes are discussed in Chapters 2 and 6.

A *class* is a description of a set of objects that share the same attributes, operations, relationships, and semantics. A class implements one or more interfaces. Graphically, a class is rendered as a rectangle, usually including its name, attributes, and operations, as in Figure 1-1.

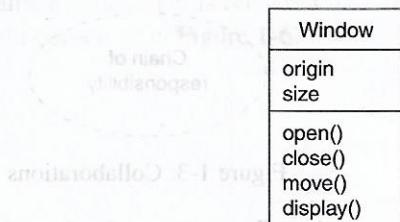


Figure 1-1: Classes

Interfaces are discussed in Chapter 7.

An *interface* is a collection of operations that specify a service of a class or component. An interface therefore describes the externally visible behavior of that element. An interface might represent the complete behavior of a class or component or only a part of that behavior. An interface defines a set of operation specifications (that is, their signatures) but never a set of operation implementations. The declaration of an interface looks like a class with the keyword «interface» above the name; attributes are not relevant, except sometimes to show constants. An interface rarely stands alone, however. An interface provided by a class to the outside world is shown as a small circle attached to the class box by a line. An interface required by a class from some other class is shown as a small semicircle attached to the class box by a line, as in

Figure 1-2.

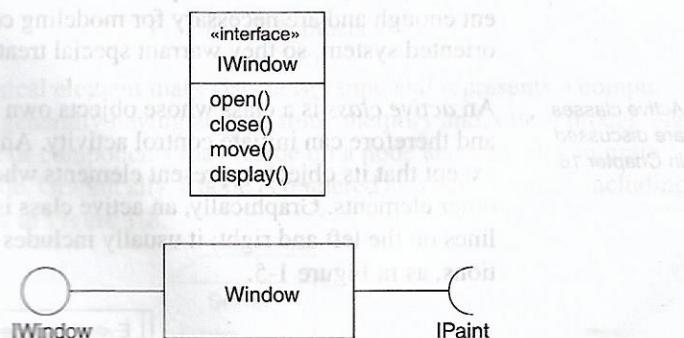


Figure 1-2: Interfaces

Collaborations are discussed in Chapter 19.

A *collaboration* defines an interaction and is a society of roles and other elements that work together to provide some cooperative behavior that's bigger than the sum of all the elements. Collaborations have structural, as well as behavioral, dimensions. A given class or object might participate in several collaborations. These collaborations therefore represent the *implementation* of patterns that make up a system. Graphically, a collaboration is rendered as an ellipse with dashed lines, sometimes including only its name, as in

Figure 1-3.

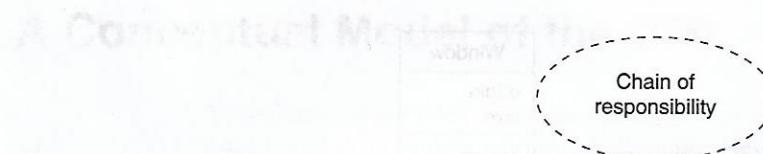


Figure 1-3: Collaborations

Use cases are discussed in Chapter 12.

A *use case* is a description of sequences of actions that a system performs that yield observable results of value to a particular actor. A use case is used to structure the behavioral things in a model. A use case is realized by a collaboration. Graphically, a use case is rendered as an ellipse with solid lines, usually including only its name, as in Figure 1-4.

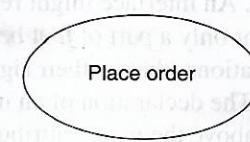


Figure 1-4: Use Cases

Active classes are discussed in Chapter 16.

The remaining three things—active classes, components, and nodes—are all class-like, meaning they also describe sets of entities that share the same attributes, operations, relationships, and semantics. However, these three are different enough and are necessary for modeling certain aspects of an object-oriented system, so they warrant special treatment.

An *active class* is a class whose objects own one or more processes or threads and therefore can initiate control activity. An active class is just like a class except that its objects represent elements whose behavior is concurrent with other elements. Graphically, an active class is rendered as a class with double lines on the left and right; it usually includes its name, attributes, and operations, as in Figure 1-5.

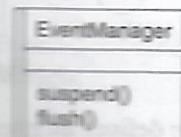


Figure 1-5: Active Classes

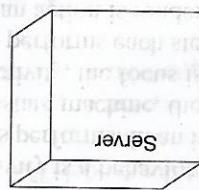
Components and internal structure are discussed in Chapter 10.

A *component* is a modular part of the system design that hides its implementation behind a set of external interfaces. Within a system, components sharing the same interfaces can be substituted while preserving the same logical behavior. The implementation of a component can be expressed by wiring together parts and connectors; the parts can include smaller components.

of artifacts).

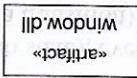
These elements—classes, interfaces, collaborations, use cases, active classes, components, artifacts, and nodes—are the basic structural things that you may include in a UML model. There are also variations on these, such as actors, signals, and utilities (kinds of classes); processes and threads (kinds of active classes), and documents, files, libraries, pages, and tables (kinds of artifacts).

Figure 1-8: Nodes



A node is a physical element that exists at run time and represents a computer from node to node. Graphically, a node is rendered as a cube, usually including a set of components that have at least some memory and, often, processing capability. A set of components may reside on a node and may also migrate from one source to another. Generally having at least some memory and, often, processing capability, a node typically has a name, as in Figure 1-8.

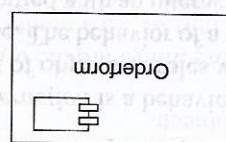
Figure 1-7: Artifacts



An artifact is a physical and replaceable part of a system that contains physical information (“bits”). In a system, you’ll encounter different kinds of deployment artifacts, such as source code files, executables, and scripts. An artifact typically represents the physical packaging of source or run-time information. Graphically, an artifact is rendered as a rectangle with the key word “artifact” above the name, as in Figure 1-7.

The remaining two elements—artifacts and nodes—are also different. They represent physical things, whereas the previous five things represent conceptual or logical things.

Figure 1-6: Components



Graphically, a component is rendered like a class with a special icon in the upper right corner, as in Figure 1-6.

Use cases, which are used to structure the behavioral things in a model, are discussed in Chapter 12; interactions are discussed in Chapter 11.

State machines are discussed in Chapter 24.

Behavioral Things *Behavioral things* are the dynamic parts of UML models. These are the verbs of a model, representing behavior over time and space. In all, there are three primary kinds of behavioral things.

First, an *interaction* is a behavior that comprises a set of messages exchanged among a set of objects or roles within a particular context to accomplish a specific purpose. The behavior of a society of objects or of an individual operation may be specified with an interaction. An interaction involves a number of other elements, including messages, actions, and connectors (the connection between objects). Graphically, a message is rendered as a directed line, almost always including the name of its operation, as in Figure 1-9.

display

Figure 1-9: Messages

Second, a *state machine* is a behavior that specifies the sequences of states an object or an interaction goes through during its lifetime in response to events, together with its responses to those events. The behavior of an individual class or a collaboration of classes may be specified with a state machine. A state machine involves a number of other elements, including states, transitions (the flow from state to state), events (things that trigger a transition), and activities (the response to a transition). Graphically, a state is rendered as a rounded rectangle, usually including its name and its substates, if any, as in Figure 1-10.

Waiting

Figure 1-10: States

Third, an *activity* is a behavior that specifies the sequence of steps a computational process performs. In an interaction, the focus is on the set of objects that interact. In a state machine, the focus is on the life cycle of one object at a time. In an activity, the focus is on the flows among steps without regard to which object performs each step. A step of an activity is called an *action*. Graphically, an action is rendered as a rounded rectangle with a name indicating its purpose. States and actions are distinguished by their different contexts.

process order

Figure 1-11: Actions

These three elements—interactions, state machines, and activities—are the basic behavioral things that you may include in a UML model. Semantically, these elements are usually connected to various structural elements, primarily classes, collaborations, and objects.

Grouping Things *Grouping things* are the organizational parts of UML models. These are the boxes into which a model can be decomposed. There is one primary kind of grouping thing, namely, packages.

Packages are discussed in Chapter 8.

A *package* is a general-purpose mechanism for organizing the design itself, as opposed to classes, which organize implementation constructs. Structural things, behavioral things, and even other grouping things may be placed in a package. Unlike components (which exist at run time), a package is purely conceptual (meaning that it exists only at development time). Graphically, a package is rendered as a tabbed folder, usually including only its name and, sometimes, its contents, as in Figure 1-12.

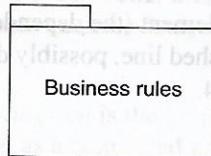


Figure 1-12: Packages

Packages are the basic grouping things with which you may organize a UML model. There are also variations, such as frameworks, models, and subsystems (kinds of packages).

Notes are discussed in Chapter 3.

Annotational Things *Annotational things* are the explanatory parts of UML models. These are the comments you may apply to describe, illuminate, and remark about any element in a model. There is one primary kind of annotational thing, called a note. A *note* is simply a symbol for rendering constraints and comments attached to an element or a collection of elements. Graphically, a note is rendered as a rectangle with a dog-eared corner, together with a textual or graphical comment, as in Figure 1-13.

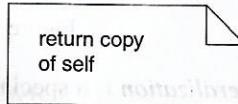


Figure 1-13: Notes

This element is the one basic annotational thing you may include in a UML model. You'll typically use notes to adorn your diagrams with constraints or

comments that are best expressed in informal or formal text. There are also variations on this element, such as requirements (which specify some desired behavior from the perspective of outside the model).

Relationships in the UML

There are four kinds of relationships in the UML:

1. Dependency
2. Association
3. Generalization
4. Realization

These relationships are the basic relational building blocks of the UML. You use them to write well-formed models.

*Dependencies
are discussed
in Chapters 2
and 6.*

First, a *dependency* is a semantic relationship between two model elements in which a change to one element (the independent one) may affect the semantics of the other element (the dependent one). Graphically, a dependency is rendered as a dashed line, possibly directed, and occasionally including a label, as in Figure 1-14.

Figure 1-14: Dependencies

*Associations
are discussed
in Chapters 2
and 6.*

Second, an *association* is a structural relationship among classes that describes a set of links, a link being a connection among objects that are instances of the classes. Aggregation is a special kind of association, representing a structural relationship between a whole and its parts. Graphically, an association is rendered as a solid line, possibly directed, occasionally including a label, and often containing other adornments, such as multiplicity and end names, as in Figure 1-15.

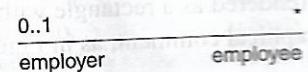


Figure 1-15: Associations

*Generaliza-
tions are
discussed in
Chapters 2
and 6.*

Third, a *generalization* is a specialization/generalization relationship in which the specialized element (the child) builds on the specification of the generalized element (the parent). The child shares the structure and the behavior of the parent. Graphically, a generalization relationship is rendered as a solid line with a hollow arrowhead pointing to the parent, as in Figure 1-16.



Figure 1-16: Generalizations

Realizations
are discussed
in Chapter 6.

Fourth, a *realization* is a semantic relationship between classifiers, wherein one classifier specifies a contract that another classifier guarantees to carry out. You'll encounter realization relationships in two places: between interfaces and the classes or components that realize them, and between use cases and the collaborations that realize them. Graphically, a realization relationship is rendered as a cross between a generalization and a dependency relationship, as in Figure 1-17.



Figure 1-17: Realizations

The five views
of an architec-
ture are
discussed later
in this chapter.

These four elements are the basic relational things you may include in a UML model. There are also variations on these four, such as refinement, trace, include, and extend.

Diagrams in the UML A *diagram* is the graphical presentation of a set of elements, most often rendered as a connected graph of vertices (things) and paths (relationships). You draw diagrams to visualize a system from different perspectives, so a diagram is a projection into a system. For all but the most trivial systems, a diagram represents an elided view of the elements that make up a system. The same element may appear in all diagrams, only a few diagrams (the most common case), or in no diagrams at all (a very rare case). In theory, a diagram may contain any combination of things and relationships. In practice, however, a small number of common combinations arise, which are consistent with the five most useful views that comprise the architecture of a software-intensive system. For this reason, the UML includes thirteen kinds of diagrams:

1. Class diagram
2. Object diagram
3. Component diagram
4. Composite structure diagram
5. Use case diagram
6. Sequence diagram
7. Communication diagram
8. State diagram
9. Activity diagram
10. Deployment diagram
11. Package diagram

12. Timing diagram
13. Interaction overview diagram

Class diagrams are discussed in Chapter 5.

A *class diagram* shows a set of classes, interfaces, and collaborations and their relationships. These diagrams are the most common diagram found in modeling object-oriented systems. Class diagrams address the static design view of a system. Class diagrams that include active classes address the static process view of a system. Component diagrams are variants of class diagrams.

Object diagrams are discussed in Chapter 9.

An *object diagram* shows a set of objects and their relationships. Object diagrams represent static snapshots of instances of the things found in class diagrams. These diagrams address the static design view or static process view of a system as do class diagrams, but from the perspective of real or prototypical cases.

Component diagrams and internal structure are discussed in Chapter 10.

A *component diagram* shows an encapsulated class and its interfaces, ports, and internal structure consisting of nested components and connectors. Component diagrams address the static design implementation view of a system. They are important for building large systems from smaller parts. (UML distinguishes a *composite structure diagram*, applicable to any class, from a component diagram, but we combine the discussion because the distinction between a component and a structured class is unnecessarily subtle.)

Use case diagrams are discussed in Chapter 12.

A *use case diagram* shows a set of use cases and actors (a special kind of class) and their relationships. Use case diagrams address the static use case view of a system. These diagrams are especially important in organizing and modeling the behaviors of a system.

Interaction diagrams are discussed in Chapter 13.

Both sequence diagrams and communication diagrams are kinds of interaction diagrams. An *interaction diagram* shows an interaction, consisting of a set of objects or roles, including the messages that may be dispatched among them. Interaction diagrams address the dynamic view of a system. A *sequence diagram* is an interaction diagram that emphasizes the time-ordering of messages; a *communication diagram* is an interaction diagram that emphasizes the structural organization of the objects or roles that send and receive messages. Sequence diagrams and communication diagrams represent similar basic concepts, but each diagram emphasizes a different view of the concepts. Sequence diagrams emphasize temporal ordering, and communication diagrams emphasize the data structure through which messages flow. A *timing diagram* (not covered in this book) shows the actual times at which messages are exchanged.

State diagrams are discussed in Chapter 17.

A *state diagram* shows a state machine, consisting of states, transitions, events, and activities. A state diagram shows the dynamic view of an object. They are especially important in modeling the behavior of an interface, class, or collaboration and emphasize the event-ordered behavior of an object, which is especially useful in modeling reactive systems.

Activity
diagrams are
discussed in
Chapter 14.

An *activity diagram* shows the structure of a process or other computation as the flow of control and data from step to step within the computation. Activity diagrams address the dynamic view of a system. They are especially important in modeling the function of a system and emphasize the flow of control among objects.

Deployment
diagrams are
discussed in
Chapter 22.

A *deployment diagram* shows the configuration of run-time processing nodes and the components that live on them. Deployment diagrams address the static deployment view of an architecture. A node typically hosts one or more artifacts.

Artifact
diagrams are
discussed in
Chapter 21.

An *artifact diagram* shows the physical constituents of a system on the computer. Artifacts include files, databases, and similar physical collections of bits. Artifacts are often used in conjunction with deployment diagrams. Artifacts also show the classes and components that they implement. (UML treats artifact diagrams as a variety of deployment diagram, but we discuss them separately.)

Package dia-
grams are dis-
cussed in
Chapter 8.

A *package diagram* shows the decomposition of the model itself into organization units and their dependencies.

A *timing diagram* is an interaction diagram that shows actual times across different objects or roles, as opposed to just relative sequences of messages. An *interaction overview diagram* is a hybrid of an activity diagram and a sequence diagram. These diagrams have specialized uses and so are not discussed in this book. See the *UML Reference Manual* for more details.

This is not a closed list of diagrams. Tools may use the UML to provide other kinds of diagrams, although these are the most common ones that you will encounter in practice.

Rules of the UML

The UML's building blocks can't simply be thrown together in a random fashion. Like any language, the UML has a number of rules that specify what a well-formed model should look like. A *well-formed model* is one that is semantically self-consistent and in harmony with all its related models.

The UML has syntactic and semantic rules for

- Names What you can call things, relationships, and diagrams
- Scope The context that gives specific meaning to a name
- Visibility How those names can be seen and used by others
- Integrity How things properly and consistently relate to one another
- Execution What it means to run or simulate a dynamic model

Models built during the development of a software-intensive system tend to evolve and may be viewed by many stakeholders in different ways and at different times. For this reason, it is common for the development team to not only build models that are well-formed, but also to build models that are

- Elided Certain elements are hidden to simplify the view
- Incomplete Certain elements may be missing
- Inconsistent The integrity of the model is not guaranteed

These less-than-well-formed models are unavoidable as the details of a system unfold and churn during the software development life cycle. The rules of the UML encourage you—but do not force you—to address the most important analysis, design, and implementation questions that push such models to become well-formed over time.

Common Mechanisms in the UML

A building is made simpler and more harmonious by the conformance to a pattern of common features. A house may be built in the Victorian or French country style largely by using certain architectural patterns that define those styles. The same is true of the UML. It is made simpler by the presence of four common mechanisms that apply consistently throughout the language.

1. Specifications
2. Adornments
3. Common divisions
4. Extensibility mechanisms

Specifications The UML is more than just a graphical language. Rather, behind every part of its graphical notation there is a specification that provides a textual statement of the syntax and semantics of that building block. For example, behind a class icon is a specification that provides the full set of attributes, operations (including their full signatures), and behaviors that the class embodies; visually, that class icon might only show a small part of this specification. Furthermore, there might be another view of that class that presents a completely different set of parts yet is still consistent with the class's underlying specification. You use the UML's graphical notation to visualize a system; you use the UML's specification to state the system's details. Given this split, it's possible to build up a model incrementally by drawing diagrams and then adding semantics to the model's specifications, or directly by creating a specification, perhaps by reverse engineering an existing system, and then creating diagrams that are projections into those specifications.

The UML's specifications provide a semantic backplane that contains all the parts of all the models of a system, each part related to one another in a consistent fashion. The UML's diagrams are thus simply visual projections into that backplane, each diagram revealing a specific interesting aspect of the system.

Notes and other adornments are discussed in Chapter 3.

Adornments Most elements in the UML have a unique and direct graphical notation that provides a visual representation of the most important aspects of the element. For example, the notation for a class is intentionally designed to be easy to draw, because classes are the most common element found in modeling object-oriented systems. The class notation also exposes the most important aspects of a class, namely its name, attributes, and operations.

A class's specification may include other details, such as whether it is abstract or the visibility of its attributes and operations. Many of these details can be rendered as graphical or textual adornments to the class's basic rectangular notation. For example, Figure 1-18 shows a class, adorned to indicate that it is an abstract class with two public, one protected, and one private operation.

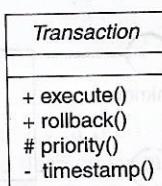


Figure 1-18: Adornments

Every element in the UML's notation starts with a basic symbol, to which can be added a variety of adornments specific to that symbol.

Common Divisions In modeling object-oriented systems, the world often gets divided in several ways.

Objects are discussed in Chapter 8.

First, there is the division of class and object. A class is an abstraction; an object is one concrete manifestation of that abstraction. In the UML, you can model classes as well as objects, as shown in Figure 1-19. Graphically, the UML distinguishes an object by using the same symbol as its class and then simply underlying the object's name.

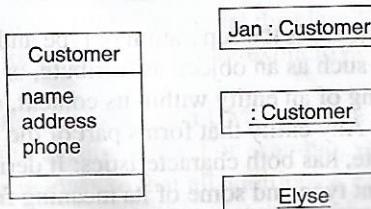


Figure 1-19: Classes and Objects

In this figure, there is one class, named *Customer*, together with three objects: *Jan* (which is marked explicitly as being a *Customer* object), *:Customer* (an anonymous *Customer* object), and *Elyse* (which in its specification is marked as being a kind of *Customer* object, although it's not shown explicitly here).

Almost every building block in the UML has this same kind of class/object dichotomy. For example, you can have use cases and use case executions, components and component instances, nodes and node instances, and so on.

Interfaces are discussed in Chapter 7.

Second, there is the separation of interface and implementation. An interface declares a contract, and an implementation represents one concrete realization of that contract, responsible for faithfully carrying out the interface's complete semantics. In the UML, you can model both interfaces and their implementations, as shown in Figure 1-20.

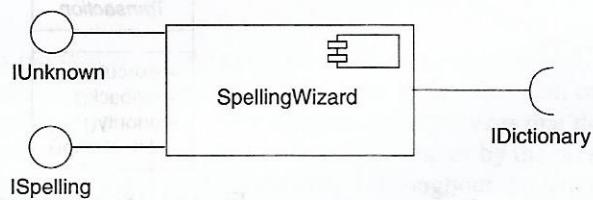


Figure 1-20: Interfaces and Implementations

In this figure, there is one component named *SpellingWizard.dll* that provides (implements) two interfaces, *IUnknown* and *ISpelling*. It also requires an interface, *IDictionary*, that must be provided by another component.

Almost every building block in the UML has this same kind of interface/implementation dichotomy. For example, you can have use cases and the collaborations that realize them, as well as operations and the methods that implement them.

Third, there is the separation of type and role. The type declares the class of an entity, such as an object, an attribute, or a parameter. A role describes the meaning of an entity within its context, such as a class, component, or collaboration. Any entity that forms part of the structure of another entity, such as an attribute, has both characteristics: It derives some of its meaning from its inherent type and some of its meaning from its role within its context (Figure 1-21).

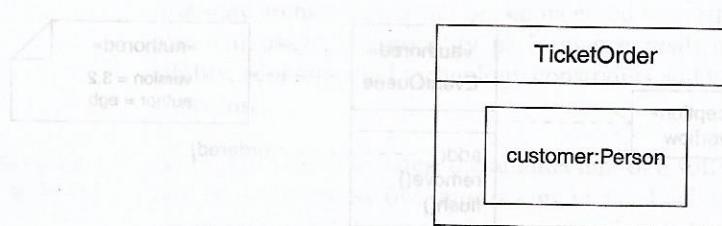


Figure 1-21: Part with role and type

The UML's extensibility mechanisms are discussed in Chapter 3.

Extensibility Mechanisms The UML provides a standard language for writing software blueprints, but it is not possible for one closed language to ever be sufficient to express all possible nuances of all models across all domains across all time. For this reason, the UML is opened-ended, making it possible for you to extend the language in controlled ways. The UML's extensibility mechanisms include

- Stereotypes
- Tagged values
- Constraints

A *stereotype* extends the vocabulary of the UML, allowing you to create new kinds of building blocks that are derived from existing ones but that are specific to your problem. For example, if you are working in a programming language, such as Java or C++, you will often want to model exceptions. In these languages, exceptions are just classes, although they are treated in very special ways. Typically, you only want to allow them to be thrown and caught, nothing else. You can make exceptions first-class citizens in your models—meaning that they are treated like basic building blocks—by marking them with an appropriate stereotype, as for the class *Overflow* in Figure 1-19.

A *tagged value* extends the properties of a UML stereotype, allowing you to create new information in the stereotype's specification. For example, if you are working on a shrink-wrapped product that undergoes many releases over time, you often want to track the version and author of certain critical abstractions. Version and author are not primitive UML concepts. They can be added to any building block, such as a class, by introducing new tagged values to that building block. In Figure 1-19, for example, the class *EventQueue* is extended by marking its version and author explicitly.

A *constraint* extends the semantics of a UML building block, allowing you to add new rules or modify existing ones. For example, you might want to constrain the *EventQueue* class so that all additions are done in order. As Figure 1-22 shows, you can add a constraint that explicitly marks these for the operation *add*.

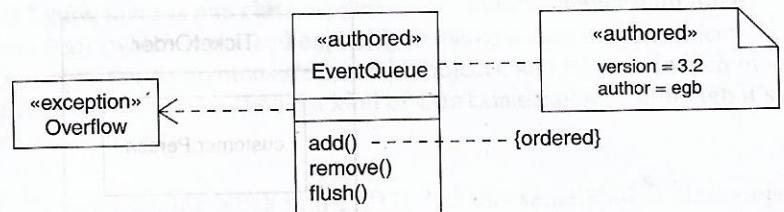


Figure 1-22: Extensibility Mechanisms

Collectively, these three extensibility mechanisms allow you to shape and grow the UML to your project's needs. These mechanisms also let the UML adapt to new software technology, such as the likely emergence of more powerful distributed programming languages. You can add new building blocks, modify the specification of existing ones, and even change their semantics. Naturally, it's important that you do so in controlled ways so that through these extensions, you remain true to the UML's purpose—the communication of information.

Architecture

The need for viewing complex systems from different perspectives is discussed in this Chapter.

Visualizing, specifying, constructing, and documenting a software-intensive system demands that the system be viewed from a number of perspectives. Different stakeholders—end users, analysts, developers, system integrators, testers, technical writers, and project managers—each bring different agendas to a project, and each looks at that system in different ways at different times over the project's life. A system's architecture is perhaps the most important artifact that can be used to manage these different viewpoints and thus control the iterative and incremental development of a system throughout its life cycle.

Architecture is the set of significant decisions about

- The organization of a software system
- The selection of the structural elements and their interfaces by which the system is composed
- Their behavior, as specified in the collaborations among those elements
- The composition of these structural and behavioral elements into progressively larger subsystems
- The architectural style that guides this organization: the static and dynamic elements and their interfaces, their collaborations, and their composition

Software architecture is not only concerned with structure and behavior but also with usage, functionality, performance, resilience, reuse, comprehensibility, economic and technology constraints and trade-offs, and aesthetic concerns.

Modeling the architecture of a system is discussed in Chapter 26.

As Figure 1-23 illustrates, the architecture of a software-intensive system can best be described by five interlocking views. Each view is a projection into the organization and structure of the system, focused on a particular aspect of that system.

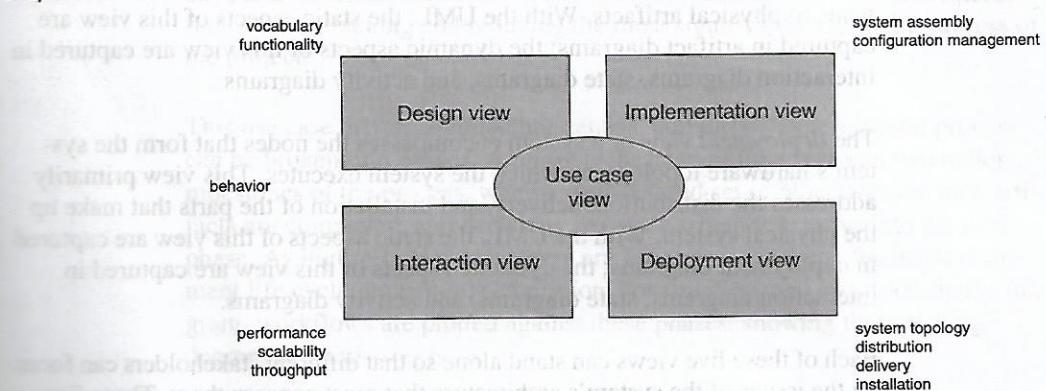


Figure 1-23: Modeling a System's Architecture

The *use case view* of a system encompasses the use cases that describe the behavior of the system as seen by its end users, analysts, and testers. This view doesn't really specify the organization of a software system. Rather, it exists to specify the forces that shape the system's architecture. With the UML, the static aspects of this view are captured in use case diagrams; the dynamic aspects of this view are captured in interaction diagrams, state diagrams, and activity diagrams.

The *design view* of a system encompasses the classes, interfaces, and collaborations that form the vocabulary of the problem and its solution. This view primarily supports the functional requirements of the system, meaning the services that the system should provide to its end users. With the UML, the static aspects of this view are captured in class diagrams and object diagrams; the dynamic aspects of this view are captured in interaction diagrams, state diagrams, and activity diagrams. The internal structure diagram of a class is particularly useful.

The *interaction view* of a system shows the flow of control among its various parts, including possible concurrency and synchronization mechanisms. This view primarily addresses the performance, scalability, and throughput of the

and involved by the system. With the UML, the static and dynamic aspects of this view are captured in the same kinds of diagrams as for the design view, but with a focus on the active classes that control the system and the messages that flow between them.

The *implementation view* of a system encompasses the artifacts that are used to assemble and release the physical system. This view primarily addresses the configuration management of the system's releases, made up of somewhat independent files that can be assembled in various ways to produce a running system. It is also concerned with the mapping from logical classes and components to physical artifacts. With the UML, the static aspects of this view are captured in artifact diagrams; the dynamic aspects of this view are captured in interaction diagrams, state diagrams, and activity diagrams.

The *deployment view* of a system encompasses the nodes that form the system's hardware topology on which the system executes. This view primarily addresses the distribution, delivery, and installation of the parts that make up the physical system. With the UML, the static aspects of this view are captured in deployment diagrams; the dynamic aspects of this view are captured in interaction diagrams, state diagrams, and activity diagrams.

Each of these five views can stand alone so that different stakeholders can focus on the issues of the system's architecture that most concern them. These five views also interact with one another: Nodes in the deployment view hold components in the implementation view that, in turn, represent the physical realization of classes, interfaces, collaborations, and active classes from the design and process views. The UML permits you to express each of these five views.

Software Development Life Cycle

The Rational Unified Process is summarized in Appendix B; a more complete treatment of this process is discussed in The Unified Software Development Process and The Rational Unified Process.

The UML is largely process-independent, meaning that it is not tied to any particular software development life cycle. However, to get the most benefit from the UML, you should consider a process that is

- Use case driven
- Architecture-centric
- Iterative and incremental

Use case driven means that use cases are used as a primary artifact for establishing the desired behavior of the system, for verifying and validating the system's architecture, for testing, and for communicating among the stakeholders of the project.