UNIT I

Introduction to UML

Syllabus : Importance of modeling, principles of modeling, object oriented modeling, conceptual model of the UML, Architecture, and Software Development Life Cycle.

Object -oriented modeling languages appeared sometime between the mid 1970s and the late 1980s as methodologists, faced with a new genre of object-oriented programming languages and increasingly complex applications, began to experiment with alternative approaches to analysis and design. The number of object-oriented methods increased from fewer than 10 to more than 50 during the period between 1989 and 1994. Many Importance of modeling, principles of modeling, object oriented modeling, conceptual model of the UML, Architecture, and Software Development Life Cycle.

Brief History of the UML

users of these methods had trouble finding a modeling language that met their needs completely, thus fueling the so- called method wars. Learning from experience, new generations of these methods began to appear, with a few clearly prominent methods emerging, most notably Booch, Jacobson's OOSE (Object-Oriented Software Engineering), and Rumbaugh's OMT (Object Modeling Technique). Other important methods included Fusion, Shlaer-Mellor, and Coad-Yourdon. Each of these was a complete method, although each was recognized as having strengths and weaknesses. In simple terms, the Booch method was particularly expressive during the design and construction phases of projects, OOSE provided excellent support for use cases as a way to drive requirements capture, analysis, andhigh-level design, and OMT-2 was most useful for analysis and data-intensive information systems. The behavioral component of many object-oriented methods, including the Booch method and OMT, was the language of statecharts, invented by David Harel. Prior to this object-oriented adoption, statecharts were used mainly in the realm of functional decomposition and structured analysis, and led to the development of executable models and tools that generated full running code.

A critical mass of ideas started to form by the mid 1990s, when Grady Booch (Rational Software Corporation), Ivar Jacobson (Objectory), and James Rumbaugh (General Electric) began to adopt ideas from each other's methods, which collectively were becoming recognized as the leading object-oriented methods worldwide. As the primary authors of the Booch, OOSE, and OMT methods, we were motivated to create a unified modeling language for three reasons. First, our methods were already evolving toward each other independently. It made sense to continue that evolution together rather than apart, eliminating the potential for any unnecessary and gratuitous differences that would further confuse users. Second, by unifying our methods, we could bring some stability to the object-oriented marketplace, allowing projects to settle on one mature modeling language and letting tool builders focus on delivering more useful features. Third, we expected that our collaboration would yield improvements for all three earlier methods, helping us to capture lessons learned and to address problems that none of our methods previously handled well.

As we began our unification, we established three goals for our work:

1. To model systems, from concept to executable artifact, using object- oriented techniques
2. To address the issues of scale inherent in complex, mission-critical systems
3. To create a modeling language usable by both humans and machines

Devising a language for use in object- oriented analysis and design is not unlike designing a programming language. First, we had to constrain the problem: Should the language encompass requirements specification? Should the language be sufficient to permit visual programming? Second, we had to strike a

balance between expressiveness and simplicity. Too simple a language would limit the breadth of problems that could be solved; too complex a language would overwhelm the mortal developer. In the case of unifying existing methods, we also had to be sensitive to the installed base. Make too many changes, and we would confuse existing users; resist advancing the language, and we would miss the opportunity of engaging a much broader set of users and of making the language simpler. The UML definition strives to make the best trade-offs in each of these areas.

The UML effort started officially in October 1994, when Rumbaugh joined Booch at Rational. Our project's initial focus was the unification of the Booch and OMT methods. The version 0.8 draft of the Unified Method (as it was then called) was released in October 1995. Around the same time, Jacobson joined Rational and the scope of the UML project was expanded to incorporate OOSE. Our efforts resulted in the release of the UML version 0.9 documents in June 1996. Throughout 1996, we invited and received feedback from the general software engineering community. During this time, it also became clear that many software organizations saw the UML as strategic to their business. We established a UML consortium, with several organizations willing to dedicate resources to work toward a strong and complete UML definition. Those partners contributing to the UML 1.0 definition included Digital Equipment Corporation, Hewlett-Packard, I-Logix, Intellicorp, IBM, ICON Computing, MCI Systemhouse, Microsoft, Oracle, Rational, Texas Instruments, and Unisys. This collaboration resulted in the UML 1.0, a modeling language that was well-defined, expressive, powerful, and applicable to a wide spectrum of problem domains. UML 1.0 was offered for standardization to the Object Management Group (OMG) in January 1997, in response to their request for proposal for a standard modeling language.

Between January 1997 and July 1997, the original group of partners was expanded to include virtually all of the other submitters and contributors of the original OMG response, including Andersen Consulting, Ericsson, ObjecTime Limited, Platinum Technology, PTech, Reich Technologies, Softeam, Sterling Software, and Taskon. A semantics task force was formed, led by CrisKobryn of MCI Systemhouse and administered by Ed Eykholt of Rational, to formalize the UML specification and to integrate the UML with other standardization efforts. A revised version of the UML (version 1.1) was offered to the OMG for standardization in July 1997. In September 1997, this version was accepted by the OMG Analysis and Design Task Force (ADTF) and the OMG Architecture Board and then put up for vote by the entire OMG membership. UML 1.1 was adopted by the OMG on November 14, 1997.

A successful software organization is one that consistently deploys quality software that meets the needs of its users. An organization that can develop such software in a timely and predictable fashion, with an efficient and effective use of resources, both human and material, is one that has a sustainable business.

There's an important implication in this message: The primary product of a development team is not beautiful documents, world-class meetings, great slogans, or Pulitzer prize— winning lines of source code. Rather, it is good software that satisfies the evolving needs of its users and the business. Everything else is secondary.

Unfortunately, many software organizations confuse "secondary" with "irrelevant." To deploy software that satisfies its intended purpose, you have to meet and engage users in a disciplined fashion, to expose the real requirements of your system. To develop software of lasting quality, you have to craft a solid architectural foundation that's resilient to change. To develop software rapidly, efficiently, and effectively, with a minimum of software scrap and rework, you have to have the right people, the right tools, and the right focus. To do all this consistently and predictably, with an appreciation for the lifetime costs of the system, you must have a sound development process that can adapt to the changing needs of your business and technology.

Modeling is a central part of all the activities that lead up to the deployment of good software. We build models to communicate the desired structure and behavior of our system. We build models to visualize and control the system's architecture. We build models to better understand the system we are building, often exposing opportunities for simplification and reuse. We build models to manage risk.

The Importance of Modeling

Unsuccessful software projects fail in their own unique ways, but all successful projects are alike in many ways. There are many elements that contribute to a successful software organization; one common thread is the use of modeling.

Modeling is a proven and well-accepted engineering technique. We build architectural models of houses and high rises to help their users visualize the final product. We may even build mathematical models in order to analyze the effects of winds or earthquakes on our buildings.

Modeling is not just a part of the building industry. It would be inconceivable to deploy a new aircraft or an automobile without first building models• from computer models to physical wind tunnel models to full-scale prototypes. New electrical devices, from microprocessors to telephone switching systems require some degree of modeling in order to better understand the system and to communicate those ideas to others. In the motion picture industry, storyboarding, which is a form of modeling, is central to any production. In the fields of sociology, economics, and business management, we build models so that we can validate our theories or try out new ones with minimal risk and cost.

What, then, is a model? Simply put,

> *A model is a simplification of reality.*

A model provides the blueprints of a system. Models may encompass detailed plans, as well as more general plans that give a 30,000-foot view of the system under consideration.

Why do we model? There is one fundamental reason.

> *We build models so that we can better understand the system we are developing.*

Through modeling, we achieve four aims.

1. Models help us to visualize a system as it is or as we want it to be.
2. Models permit us to specify the structure or behavior of a system.
3. Models give us a template that guides us in constructing a system.
4. Models document the decisions we have made.

Modeling is not just for big systems. Even the software equivalent of a dog house can benefit from some modeling. However, it's definitely true that the larger and more complex the system, the more important modeling becomes, for one very simple reason:

> *We build models of complex systems because we cannot comprehend such a system in its entirety.*

Principles of Modeling

The use of modeling has a rich history in all the engineering disciplines. That experience suggests four basic principles of modeling. First,

The choice of what models to create has a profound influence on how a problem is attacked and how a solution is shaped.

In other words, choose your models well. The right models will brilliantly illuminate the most wicked development problems, offering insight that you simply could not gain otherwise; the wrong models will mislead you, causing you to focus on irrelevant issues.

Second,

Every model may be expressed at different levels of precision.

If you are building a high rise, sometimes you need a 30,000- foot view• for instance, to help your investors visualize its look and feel. Other times, you need to get down to the level of the studs• for instance, when there's a tricky pipe run or an unusual structural element.

Third,

The best models are connected to reality.

A physical model of a building that doesn't respond in the same way as do real materials has only limited value; a mathematical model of an aircraft that assumes only ideal conditions and perfect manufacturing can mask some potentially fatal characteristics of the real aircraft. It's best to have models that have a clear connection to reality, and where that connection is weak, to know exactly how those models are divorced from the real world. All models simplify reality; the trick is to be sure that your simplifications don't mask any important details.

In software, the Achilles heel of structured analysis techniques is the fact that there is a basic disconnect between its analysis model and the system's design model. Failing to bridge this chasm causes the system as conceived and the system as built to diverge over time. In object-oriented systems, it is possible to connect all the nearly independent views of a system into one semantic whole.

Fourth,

No single model is sufficient. Every nontrivial system is best approached through a small set of nearly independent models.

If you are constructing a building, there is no single set of blueprints that reveal all its details. At the very least, you'll need floor plans, elevations, electrical plans, heating plans, and plumbing plans.

Object-Oriented Modeling

In software, there are several ways to approach a model. The two most common ways are from an algorithmic perspective and from an object-oriented perspective.

The traditional view of software development takes an algorithmic perspective. In this approach, the main building block of all software is the procedure or function. This view leads developers to focus on issues of control and the decomposition of larger algorithms into smaller ones. There's nothing inherently evil about such a point of view except that it tends to yield brittle systems. As requirements change (and they will) and the system grows (and it will), systems built with an algorithmic focus turn out to be very hard to maintain.

The contemporary view of software development takes an object-oriented perspective. In this approach, the main building block of all software systems is the object or class. Simply put, an object is a thing, generally drawn from the vocabulary of the problem space or the solution space; a class is a description of a set of common objects. Every object has identity (you can name it or otherwise distinguish it from other objects), state (there's generally some data associated with it), and behavior (you can do things to the object, and it can do things to other objects, as well).

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 and so 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 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.
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.

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

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

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, 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.

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

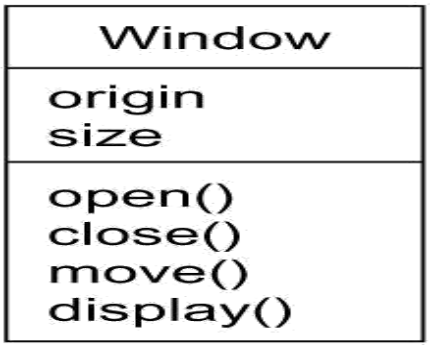
These things are the basic object-oriented building blocks of the UML. You use them to write well-formed models.

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. In all, there are seven kinds of structural things.

First, 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

Figure Classes



Second, 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. Graphically, an interface is rendered as a circle together with its name. An interface rarely stands alone. Rather, it is typically attached to the class or component that realizes the interface, as in Figure.

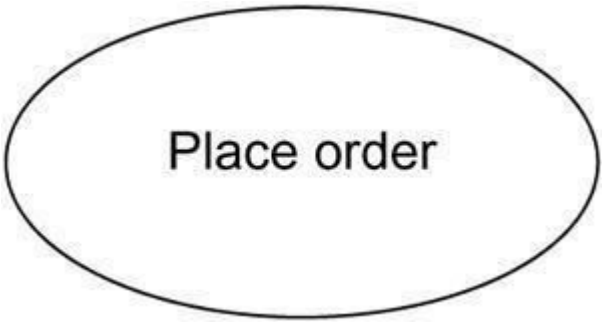
Figure Interfaces

Third, 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. Therefore, collaborations have structural, as well as behavioral, dimensions. A given class 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, usually including only its name, as in Figure .

Figure Collaborations

Fourth, a *use case* is a description of set of sequence of actions that a system performs that yields an observable result 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.

Figure Use Cases

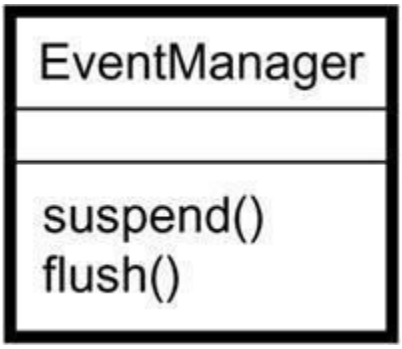


Fifth, an *active class* is a class whose objects own one or more processes or threads and therefore can initiate control activity. An active class is just like a class except that its objects represent elements whose behavior is concurrent with other elements. Graphically, an active class is rendered just like a class, but

with heavy lines, usually including its name, attributes, and operations, as in Figure.

Figure Active Classes

The remaining two elements• component, and nodes• are also different. They represent physical

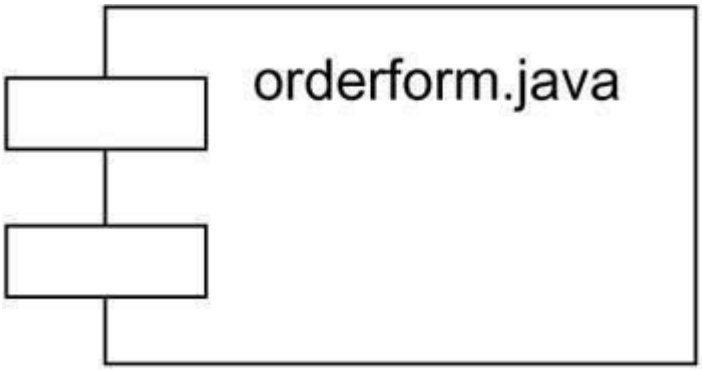


things, whereas the previous five things represent conceptual or logical things.

Sixth, a *component* is a physical and replaceable part of a system that conforms to and provides the realization of a set of interfaces. In a system, you'll encounter different kinds of deployment components, such as COM+ components or Java Beans, as well as components that are artifacts of the development process, such as source code files. A component typically represents the physical packaging of otherwise logical elements, such as classes, interfaces, and collaborations. Graphically, a component is rendered as a rectangle with tabs, usually including only its name, as in Figure.

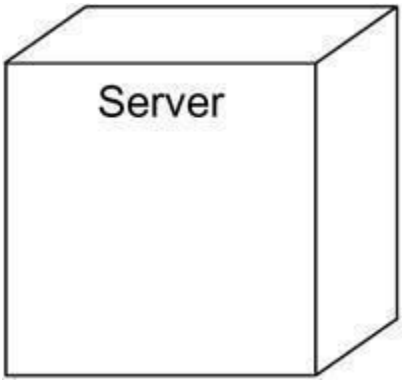
Figure Components

Seventh, a *node* is a physical element that exists at run time and represents a computational resource,



generally having at least some memory and, often, processing capability. A set of components may reside on a node and may also migrate from node to node. Graphically, a node is rendered as a cube, usually including only its name, as in Figure.

Figure Nodes



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

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 two primary kinds of behavioral things.

First, an *interaction* is a behavior that comprises a set of messages exchanged among a set of objects 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, action sequences (the behavior invoked by a message), and links (the connection between objects). Graphically, a message is rendered as a directed line, almost always including the name of its operation, as in Figure.

Figure 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.

Figure States

These two elements• interactions and state machines• 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.

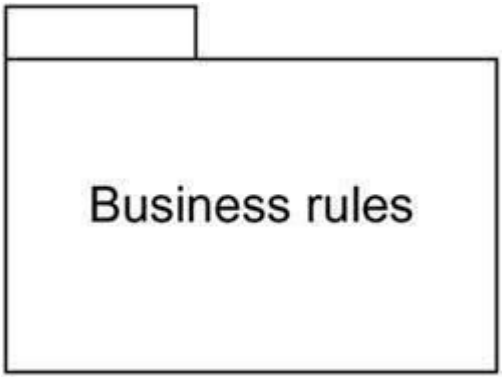
Grouping Things

Grouping things are the organizational parts of UML models. These are the boxes into which amodel can be decomposed. In all, there is one primary kind of grouping thing, namely, packages.

A *package* is a general-purpose mechanism for organizing elements into groups. 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.

Figure Packages

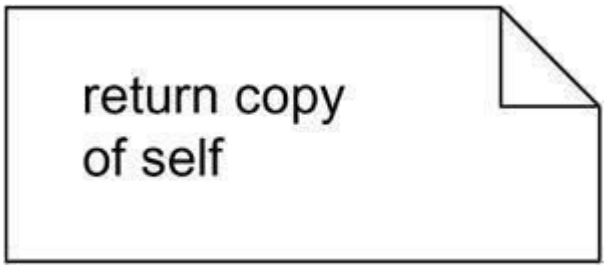


Packages are the basic grouping things with which you may organize a UML model. There are also variations, such as frameworks, models, and subsystems (kinds of packages).

Annotational Things

*Annotationalthings*are the explanatory parts of UML models. These are the comments you mayapply 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 .

Figure Notes



Relationships in the UML

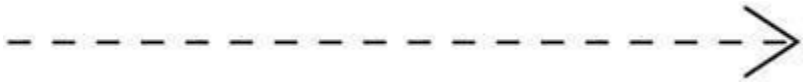
There are four kinds of relationships in the UML:

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.

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

Figure Dependencies

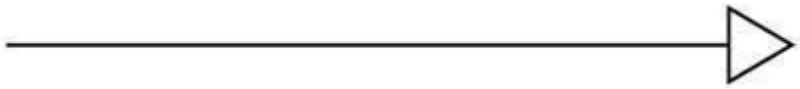


Second, an *association* is a structural relationship that describes a set of links, a link being a connection among objects. Aggregation is a special kind of association, representing a structural relationship between a whole and its parts. Graphically, an association is rendered as a solid line, possibly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

Figure Associations

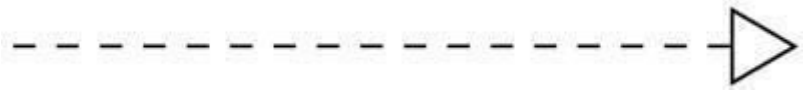
Third, a *generalization* is a specialization/generalization relationship in which objects of the specialized element (the child) are substitutable for objects of the generalized element (the parent). In this way, the child shares the structure and the behavior of the parent. Graphically, a generalization relationship is rendered as a solid line with a hollow arrowhead pointing to the parent, as in Figure .

Figure Generalizations



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.

Figure Realization



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 (for dependencies).

The five views of an architecture are discussed in the following section.

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 arcs (relationships). the UML includes nine such diagrams:

1. Class diagram
2. Object diagram

3. Use case diagram
4. Sequence diagram
5. Collaboration diagram
6. Statechart diagram
7. Activity diagram
8. Component diagram
9. Deployment diagram

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.

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.

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.

Both sequence diagrams and collaboration diagrams are kinds of interaction diagrams. An shows an interaction, consisting of a set of objects and their relationships, 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 *collaboration diagram* is an interaction diagram that emphasizes the structural organization of theobjects that send and receive messages. Sequence diagrams and collaboration diagrams are isomorphic, meaning that you can take one and transform it into the other.

A *statechart diagram* shows a state machine, consisting of states, transitions, events, and activities. Statechart diagrams address the dynamic view of a system. 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.

An *activity diagram* is a special kind of a statechart diagram that shows the flow from activity to activity within a system. 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.

A *component diagram* shows the organizations and dependencies among a set of components. Component diagrams address the static implementation view of a system. They are related to class diagrams in that a component typically maps to one or more classes, interfaces, or collaborations.

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. They are related to component diagrams in that a node typically encloses one or more components.

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 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 ,incomplete, inconsistent.

Common Mechanisms in 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'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.

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.

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.

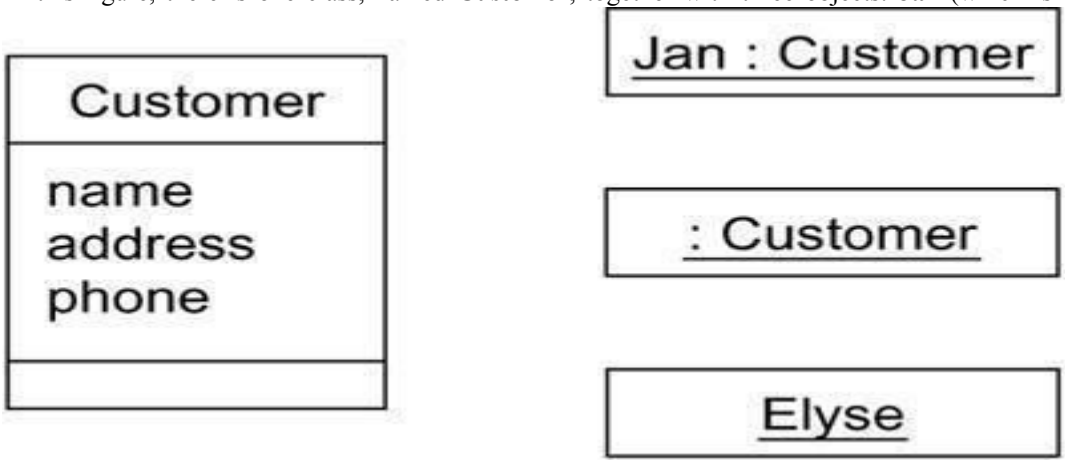
Common Divisions

In modeling object-oriented systems, the world often gets divided in at least a couple of ways.

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.

Figure Classes And Objects

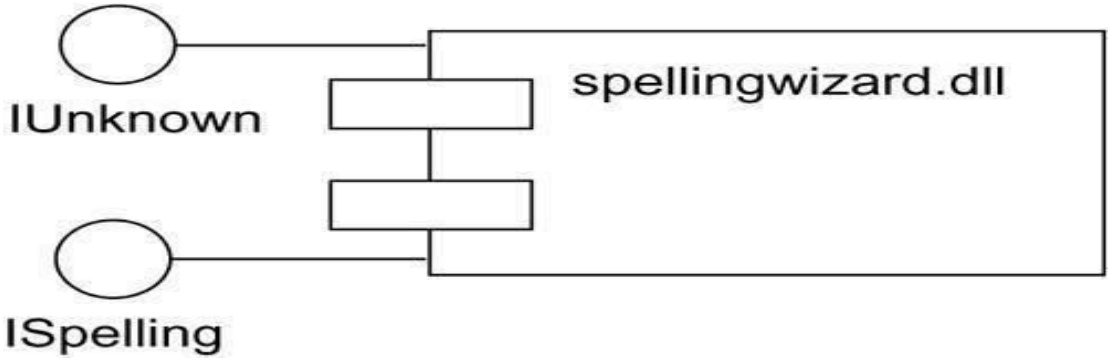
In this figure, there is one class, named **Customer**, together with three objects: **Jan** (which is marked



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).

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.

Figure Interfaces And Implementations



In this figure, there is one component named **spellingwizard.dll** that implements two interfaces, **IUnknown** and **ISpelling.**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.

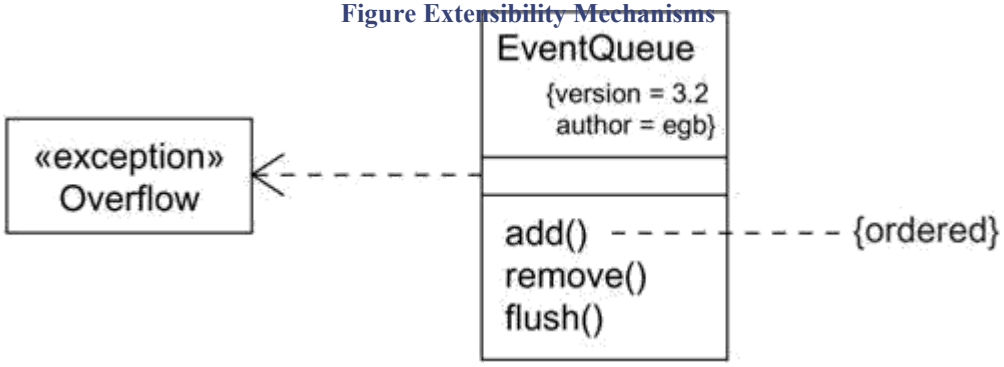
Extensibility Mechanisms

. 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 .

Figure Extensibility Mechanisms



A *tagged value* extends the properties of a UML building block, allowing you to create new information in that element'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, for example, the class**EventQueue**is extended by marking its versionand 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 Figureshows, you can add a constraint that explicitly marks these for the operation **add.**

Architecture

A system's architecture is perhaps the most important artifact that can be used to manage these different viewpoints and so control the iterative and incremental development of a system throughout its life cycle.

Architecture is the set of significant decisions about

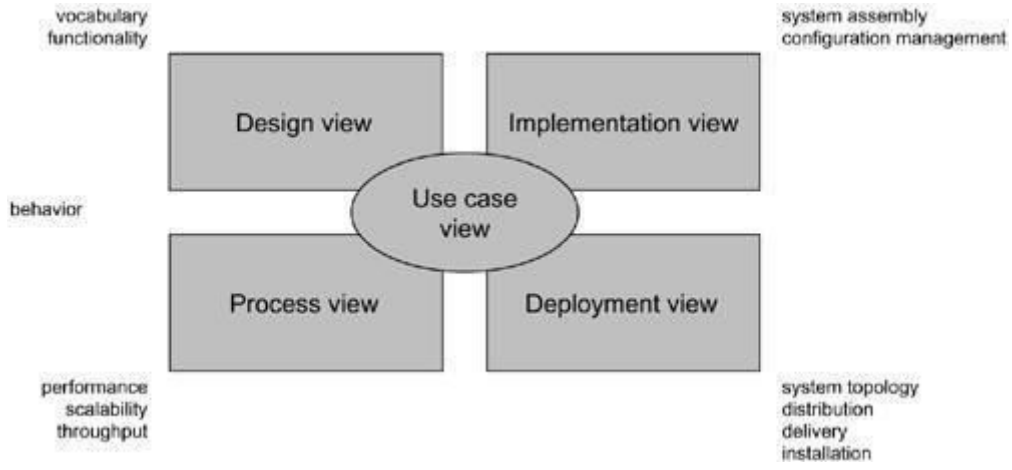
- The organization of a software system

- 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.

As Figure 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 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, statechart 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, statechart diagrams, and activity diagrams.

The *process view* of a system encompasses the threads and processes that form the system's concurrency and synchronization mechanisms. This view primarily addresses the performance, scalability, and throughput of 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 represent these threads and processes.

The *implementation view* of a system encompasses the components and files 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 components and files that can be assembled in various ways to produce a running system. With the UML, the static aspects of this view are captured in component diagrams; the dynamic aspects of this view are captured in interaction diagrams, statechart 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, statechart 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 every one of these five views and their interactions.

Software Development Life Cycle

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 desiredbehavior of the system, for verifying and validating the system's architecture, for testing, and for communicating among the stakeholders of the project.
Architecture-centric means that a system's architecture is used as a primary artifact forconceptualizing, constructing, managing, and evolving the system under development.

An *iterative process* is one that involves managing a stream of executable releases. An is one that involves the continuous integration of the system's architecture to produce these releases, with each new release embodying incremental improvements over the other. Together, an iterative and incremental process is *risk-driven,* meaning that each new release is focused on attacking and reducing the most significant risks to the success of the project.

This use case driven, architecture-centric, and iterative/incremental process can be broken into phases. A *phase* is the span of time between two major milestones of the process, when a well-defined set of objectives are met, artifacts are completed, and decisions are made whether to move into the next phase. As Figure shows, there are four phases in the software development life cycle: inception, elaboration, construction, and transition. In the diagram, workflows are plotted against these phases, showing their varying degrees of focus over time.

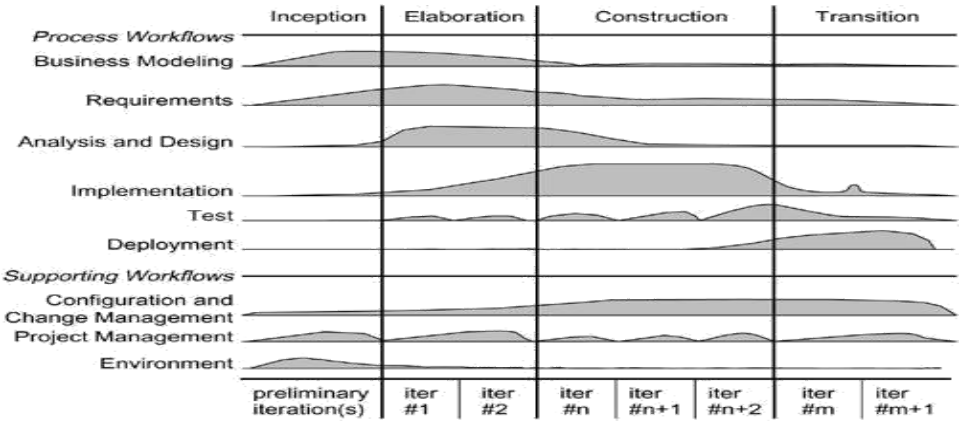


Figure Software Development Life Cycle

Inception is the first phase of the process, when the seed idea for the development is brought upto the point of being• at least internally• sufficiently well-founded to warrant entering into the elaboration phase.

Elaboration is the second phase of the process, when the product vision and its architecture aredefined. In this phase, the system's requirements are articulated, prioritized, and baselined. A system's requirements may range from general vision statements to precise evaluation criteria, each specifying particular functional or nonfunctional behavior and each providing a basis for testing.

Construction is the third phase of the process, when the software is brought from an executablearchitectural baseline to being ready to be transitioned to the user community. Here also, the system's requirements and especially its evaluation criteria are constantly reexamined against the business needs of the project, and resources are allocated as appropriate to actively attack risks to the project.

Transition is the fourth phase of the process, when the software is turned into the hands of theuser community. Rarely does the software development process end here, for even during this phase, the system is continuously improved, bugs are eradicated, and features that didn't make an earlier release are added.

One element that distinguishes this process and that cuts across all four phases is an iteration. An *iteration* is a distinct set of activities, with a baselined plan and evaluation criteria that result in a release, either internal or external. This means that the software development life cycle can be characterized as involving a continuous stream of executable releases of the system's architecture. It is this emphasis on architecture as an important artifact that drives the UML to focus on modeling the different views of a system's architecture.