2018-19 Onwards (MR-18)	MALLA REDDY ENGINEERING COLLEGE (Autonomous)	B.Tech. IV Semester		
Code:80523	OBJECT ORIENTED ANALYSIS AND DESIGN [Professional Elective - I]	L	T	P
Credits: 3		3	-	-

Prerequisites: NIL

Course Objectives: The student will be able to understand the Unified Modeling Language Principles and learns fundamental process pattern for object-oriented analysis and design.

Module I: UML [09 Periods]

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

Module II: Behavioral and structural Modeling [09 Periods] Basic Behavioral Modeling-I : Use cases, Use case Diagrams, Activity Diagrams.
Basic Structural Modeling: Classes, Relationships, common Mechanisms, and diagrams.

Module III: Behavioral Model II [12 Periods] A: Advanced Structural Modeling: Advanced classes, advanced relationships, Interfaces, Types and Roles, Packages.
B: Class & Object Diagrams: Terms, concepts, modeling techniques for Class & Object Diagrams.
Basic Behavioral Modeling-II : Interactions, Interaction diagrams

Module IV: Advanced Behavioral Modeling [09 Periods] Advanced Behavioral Modeling: Events and signals, state machines, processes and Threads, time and space, state chart diagrams.

Module V: Architecture Modeling [09 Periods] Architectural Modeling: Component, Deployment, Component diagrams and Deployment diagrams.
Case Study: The Unified Library application.

TEXT BOOKS:

1. Grady Booch, James Rumbaugh, Ivar Jacobson: The Unified Modeling Language User Guide, Pearson Education.
2. Hans-Erik Eriksson, Magnus Penker, Brian Lyons, David Fado: UML 2 Toolkit, WILEY-Dreamtech India Pvt. Ltd.

REFERENCES:

1. Meilir Page-Jones: Fundamentals of Object Oriented Design in UML, Pearson Education.
2. Atul Kahate: Object Oriented Analysis & Design, The McGraw-Hill.
3. Mark Priestley: Practical Object-Oriented Design with UML,TATA McGrawHill.
4. Appling UML and Patterns: An introduction to Object – Oriented Analysis and Design and Unified Process, Craig Larman, PearsonEducation.

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

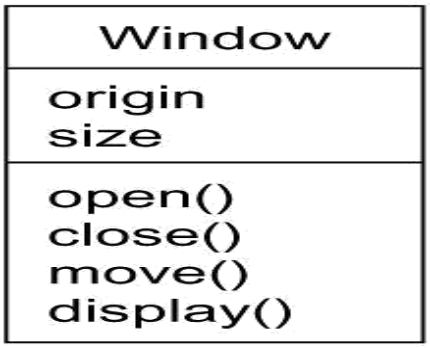
These things are the basic object-oriented building blocks of the UML. You use them to write well-formed models.

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.

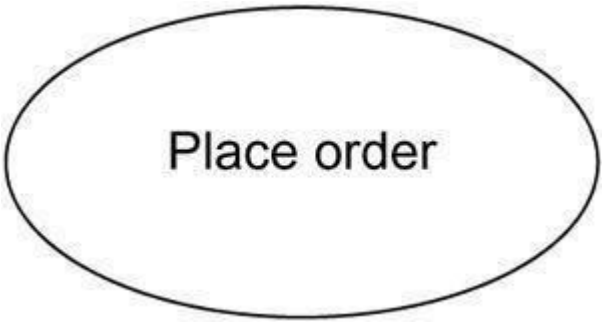
Figure Interfaces

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.

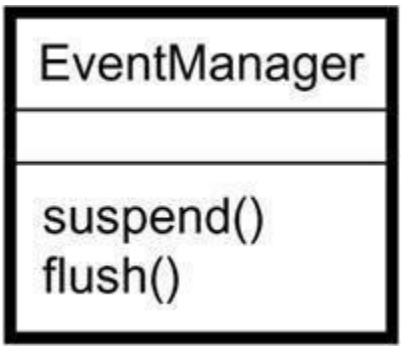
Figure Use Cases



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.

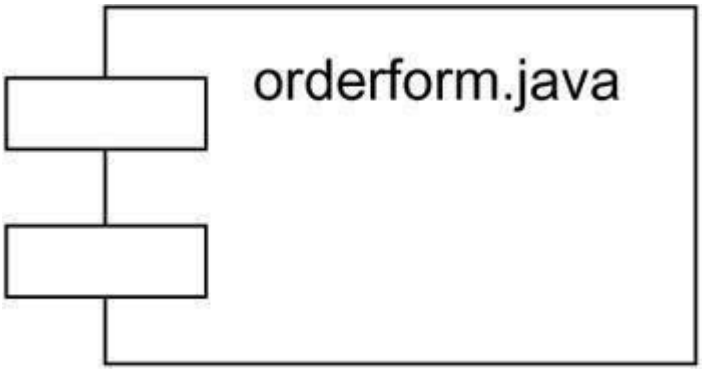
Figure Active Classes



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

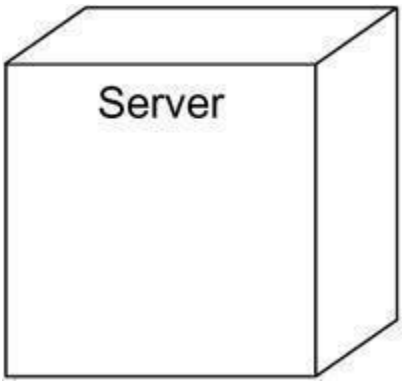
Sixth, a *component* is a physical and replaceable part of a system that conforms to and provides the realization of a set of interfaces. In a system, you'll encounter different kinds of deployment components, such as COM+ components or Java Beans, as well as components that are artifacts of the development 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.

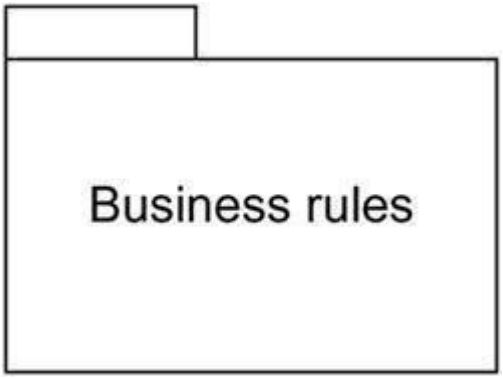
Grouping Things

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

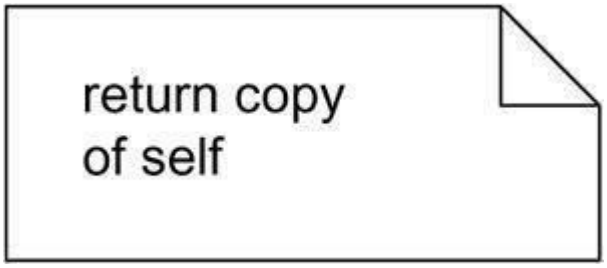


Packages are the basic grouping things with which you may organize a UML model. There are also variations, such as frameworks, models, and subsystems (kinds of packages).

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

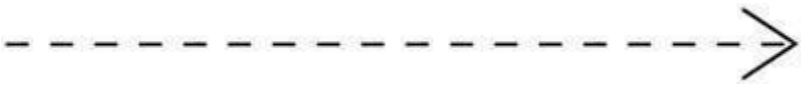
There are four kinds of relationships in the UML:

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.

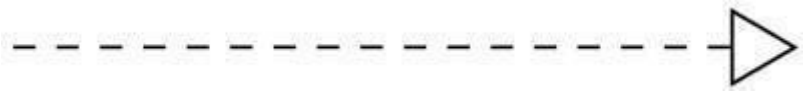
Figure Associations

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.

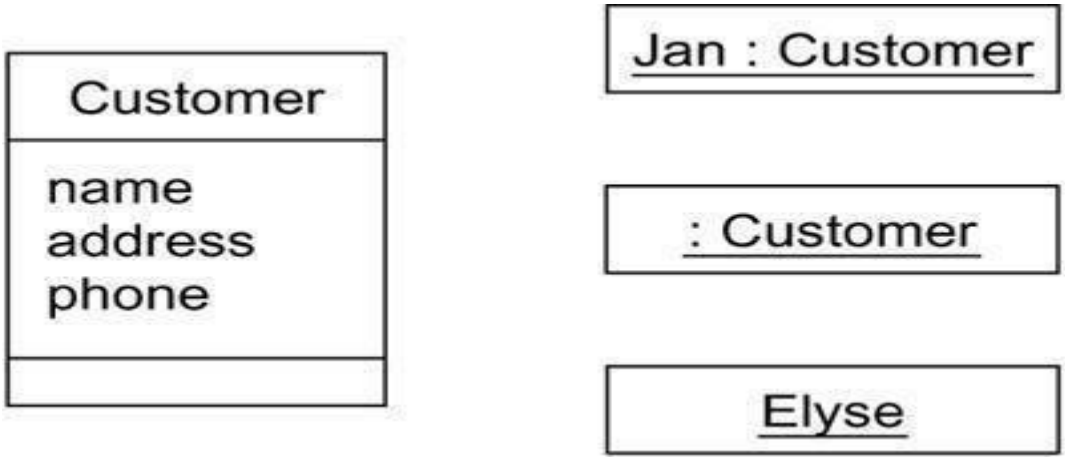
Every element in the UML's notation starts with a basic symbol, to which can be added a variety of adornments specific to that symbol.

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.

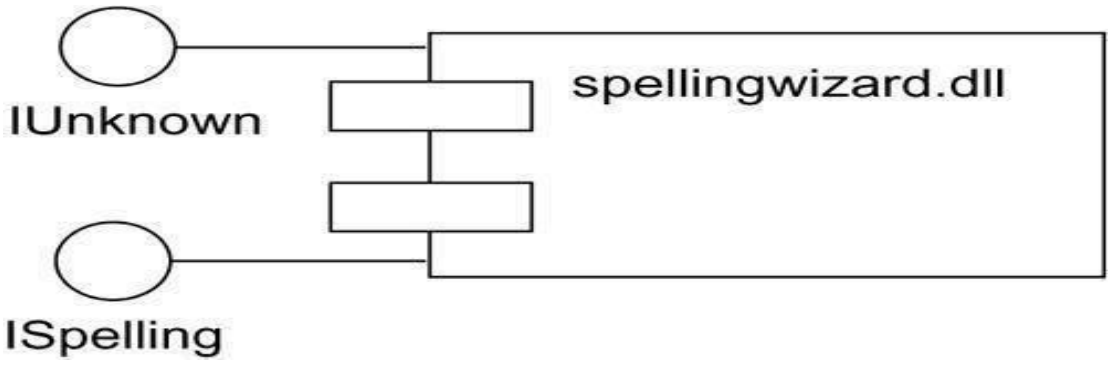
Figure Classes And Objects



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.

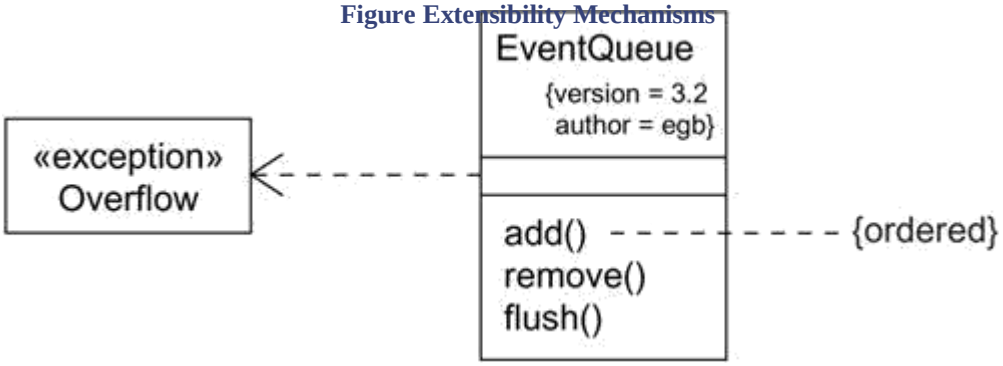
Extensibility Mechanisms

. 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

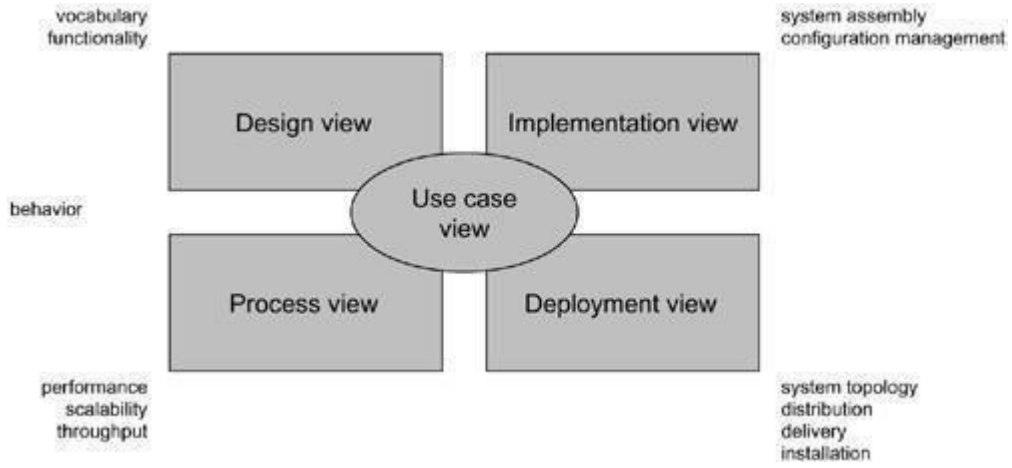
- The organization of a software system

- 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

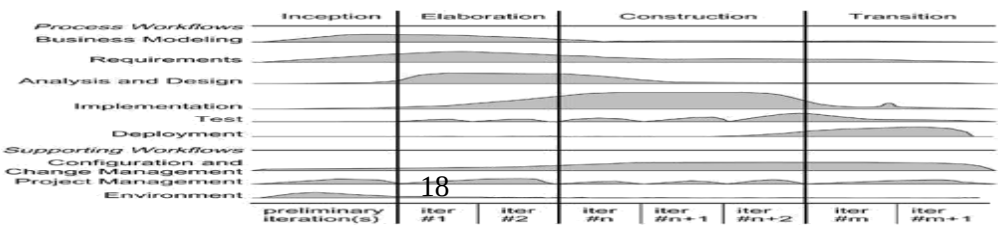
- Use case driven
- Architecture-centric
- Iterative and incremental

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.

UNIT II

Basic Structural Modeling and Advanced Structural Modeling

Syllabus :Classes, Relationships, common Mechanisms and diagrams.Advanced classes, advanced relationships, Interfaces, Types and Roles, Packages.Classes, Terms, concepts, modeling techniques for Class & Object Diagrams.

Class

Classes are the most important building block of any object-oriented system. 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.

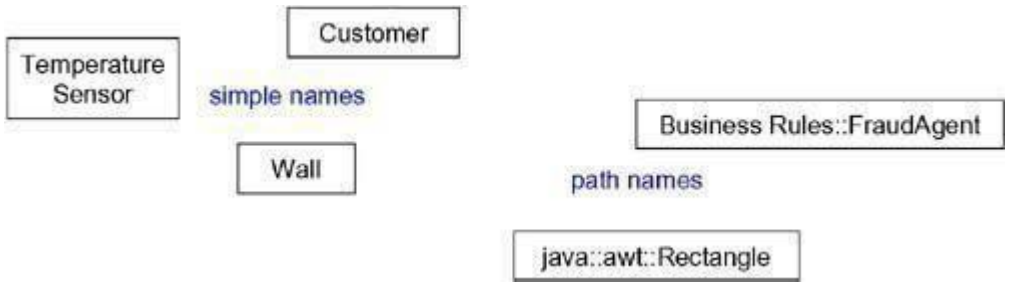
Terms and Concepts

A *class* is a description of a set of objects that share the same attributes, operations,relationships, and semantics. Graphically, a class is rendered as a rectangle.

Names

Every class must have a name that distinguishes it from other classes. A *name* is a textual string. That name alone is known as a *simple name;* a *path name* is the class name prefixed by the name of the package in which that class lives. A class may be drawn showing only its name, as
Figure shows.

Figure Simple and Path Names



Attributes

An *attribute* is a named property of a class that describes a range of values that instances of the property may hold. A class may have any number of attributes or no attributes at all. An attribute represents some property of the thing you are modeling that is shared by all objects of that class

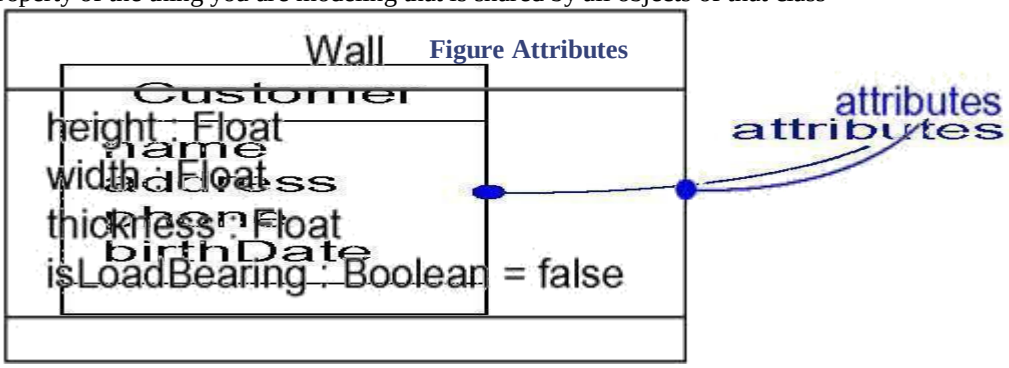


Figure Attributes

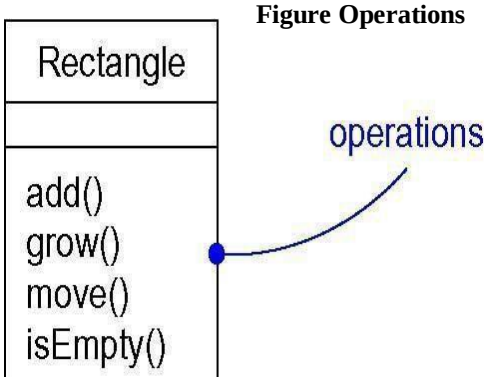
You can further specify an attribute by stating its class and possibly a default initial value, as shown Figure

FigureAttributes and Their Class

Operations

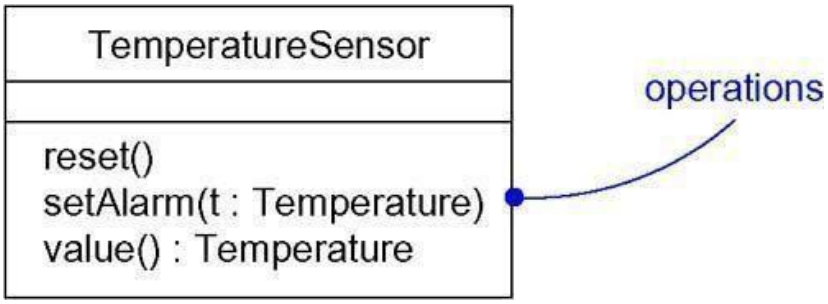
An *operation* is the implementation of a service that can be requested from any object of the class to affect behavior. In other words, an operation is an abstraction of something you can do to an object and that is shared by all objects of that class. A class may have any number of operations or no operations at all.
For example, in a windowing library such as the one found in Java's **awt** package, all objects of the class **Rectangle** can be moved, resized, or queried for their properties. Often (but not always), invoking an operation on an object changes the object's data or state. Graphically, operations are listed in a compartment just below the class attributes. Operations may be drawn showing only their names, as in Figure.

Figure Operations



You can specify an operation by stating its signature, covering the name, type, and default value of all parameters and (in the case of functions) a return type, as shown in Figure.

Figure Operations and Their Signatures

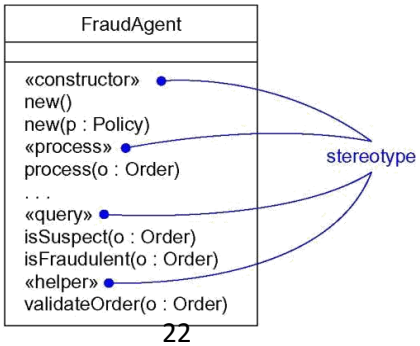


Organizing Attributes and Operations

When drawing a class, you don't have to show every attribute and every operation at once. In fact, in most cases, you can't (there are too many of them to put in one figure) and you probably shouldn't (only a subset of these attributes and operations are likely to be relevant to a specific view). For these reasons, you can elide a class, meaning that you can choose to show only some or none of a class's attributes and operations. An empty compartment doesn't necessarily mean there are no attributes or operations, just that you didn't choose to show them. You can explicitly specify that there are more attributes or properties than shown by ending each list with an ellipsis ("**...**").

To better organize long lists of attributes and operations, you can also prefix each group with a descriptive category by using stereotypes, as shown in Figure .

Figure Stereotypes for Class Features



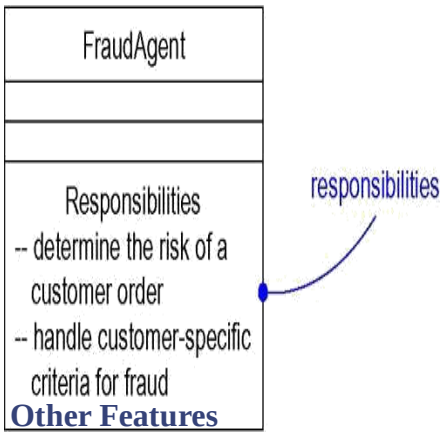
Responsibilities

A *responsibility* is a contract or an obligation of a class. When you create a class, you are making a statement that all objects of that class have the same kind of state and the same kind of behavior. At a more abstract level, these corresponding attributes and operations are just the features by which the class's responsibilities are carried out. A **Wall** class is responsible for knowing about height, width, and thickness; a **FraudAgent** class, as you might find in a credit card application, is responsible for processing orders and determining if they are legitimate, suspect, or fraudulent; a **TemperatureSensor** class is responsible for measuring temperature and raising an alarm if the temperature reaches a certain point.

When you model classes, a good starting point is to specify the responsibilities of the things in your vocabulary. Techniques like CRC cards and use case-based analysis are especially helpful here. A class may have any number of responsibilities, although, in practice, every well-structured class has at least one responsibility and at most just a handful. As you refine your models, you will translate these responsibilities into a set of attributes and operations that best fulfill the class's responsibilities.

Graphically, responsibilities can be drawn in a separate compartment at the bottom of the class icon, as shown in Figure.

Figure Responsibilities



Other Features

Attributes, operations, and responsibilities are the most common features you'll need when you create abstractions. In fact, for most models you build, the basic form of these three features will be all you need to convey the most important semantics of your classes.

When you build models, you will soon discover that almost every abstraction you create is some kind of class. Sometimes, you will want to separate the implementation of a class from its specification, and this can be expressed in the UML by using interfaces.

When you start building more complex models, you will also find yourself encountering the same kinds of classes over and over again, such as classes that represent concurrent processes and threads, or classes that represent physical things, such as applets, Java Beans, COM+ objects, files, Web pages, and hardware. Because these kinds of classes are so common and because they represent important architectural abstractions, the UML provides active classes (representing processes and threads), components (representing physical software components), and nodes (representing hardware devices).

Finally, classes rarely stand alone. Rather, when you build models, you will typically focus on groups

of classes that interact with one another. In the UML, these societies of classes form collaborations and are usually visualized in class diagrams.

Common Modeling Techniques

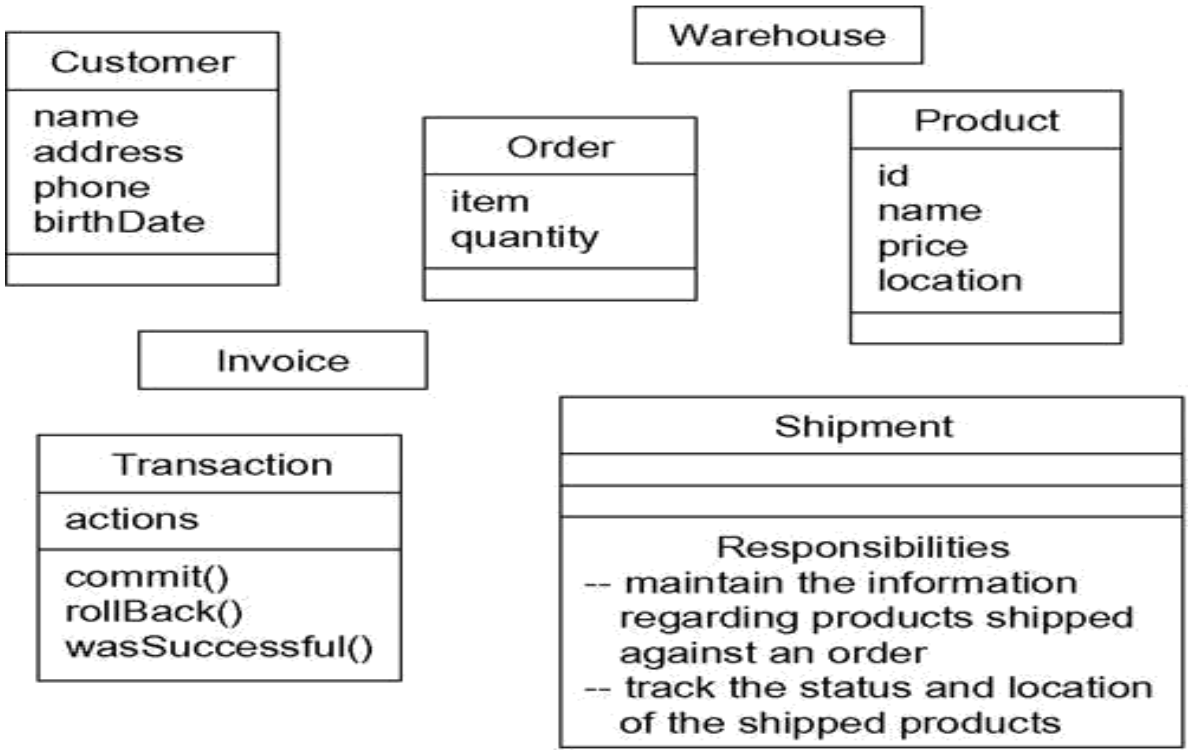
Modeling the Vocabulary of a System

To model the vocabulary of a system,

- Identify those things that users or implementers use to describe the problem or solution. Use CRC cards and use case-based analysis to help find these abstractions.
- For each abstraction, identify a set of responsibilities. Make sure that each class is crisply defined and that there is a good balance of responsibilities among all your classes.
- Provide the attributes and operations that are needed to carry out these responsibilities for each class.

Figureshows a set of classes drawn from a retail system, including**Customer**,**Order**, and**Product.** This figure includes a few other related abstractions drawn from the vocabulary of theproblem, such as **Shipment** (used to track orders), **Invoice** (used to bill orders), and **Warehouse** (where products are located prior to shipment). There is also one solution-relatedabstraction, **Transaction**, which applies to orders and shipments.

Figure Modeling the Vocabulary of a System



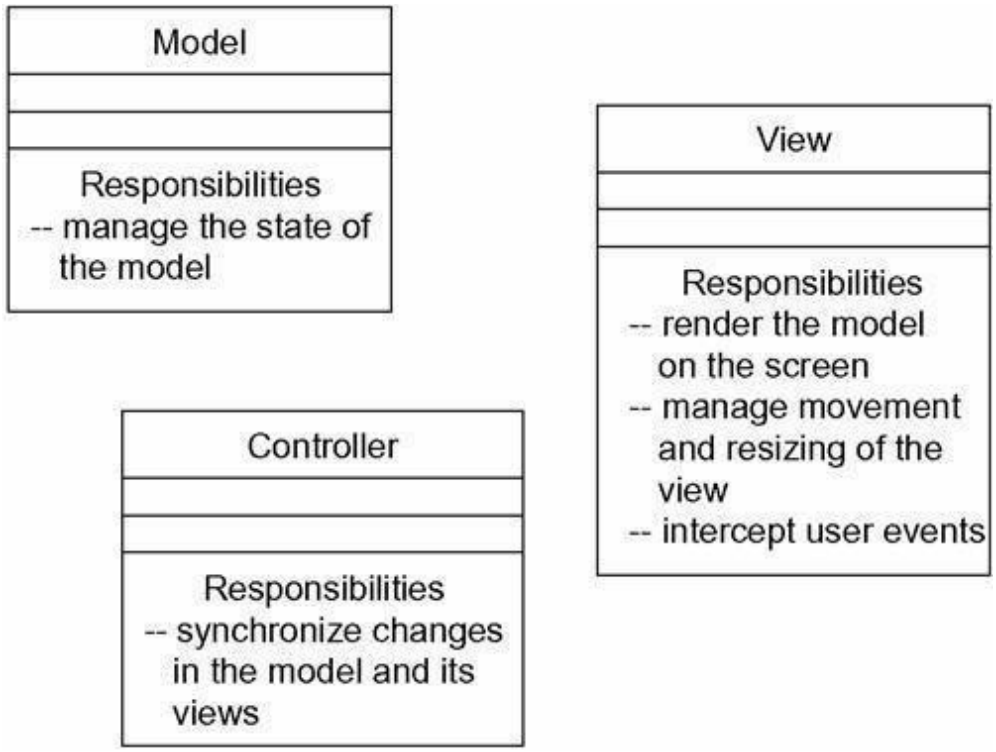
Modeling the Distribution of Responsibilities in a System

To model the distribution of responsibilities in a system,

- Identify a set of classes that work together closely to carry out some behavior.
- Identify a set of responsibilities for each of these classes.
- Look at this set of classes as a whole, split classes that have too many responsibilities into smaller abstractions, collapse tiny classes that have trivial responsibilities into larger ones, and reallocate responsibilities so that each abstraction reasonably stands on its own.
- Consider the ways in which those classes collaborate with one another, and redistribute their responsibilities accordingly so that no class within a collaboration does too much or too little.

For example, Figure shows a set of classes drawn from Smalltalk, showing the distribution of responsibilities among **Model**, **View**, and **Controller** classes. Notice how all these classes work together such that no one class does too much or too little.

Figure Modeling the Distribution of Responsibilities in a System



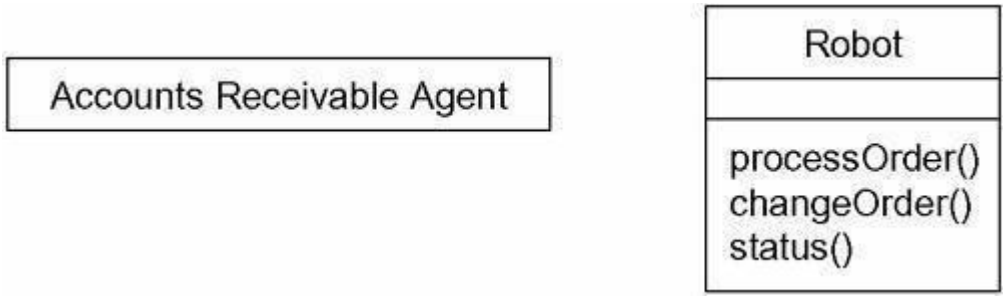
Modeling Nonsoftware Things

To model nonsoftware things,

- Model the thing you are abstracting as a class.
- If you want to distinguish these things from the UML's defined building blocks, create a new building block by using stereotypes to specify these new semantics and to give a distinctive visual cue.
- If the thing you are modeling is some kind of hardware that itself contains software, consider modeling it as a kind of node, as well, so that you can further expand on its structure.

As Figure shows, it's perfectly normal to abstract humans (like **AccountsReceivableAgent**) and hardware (like **Robot**) as classes, because each representsa set of objects with a common structure and a common behavior.

Figure Modeling Nonsoftware Things



Modeling Primitive Types

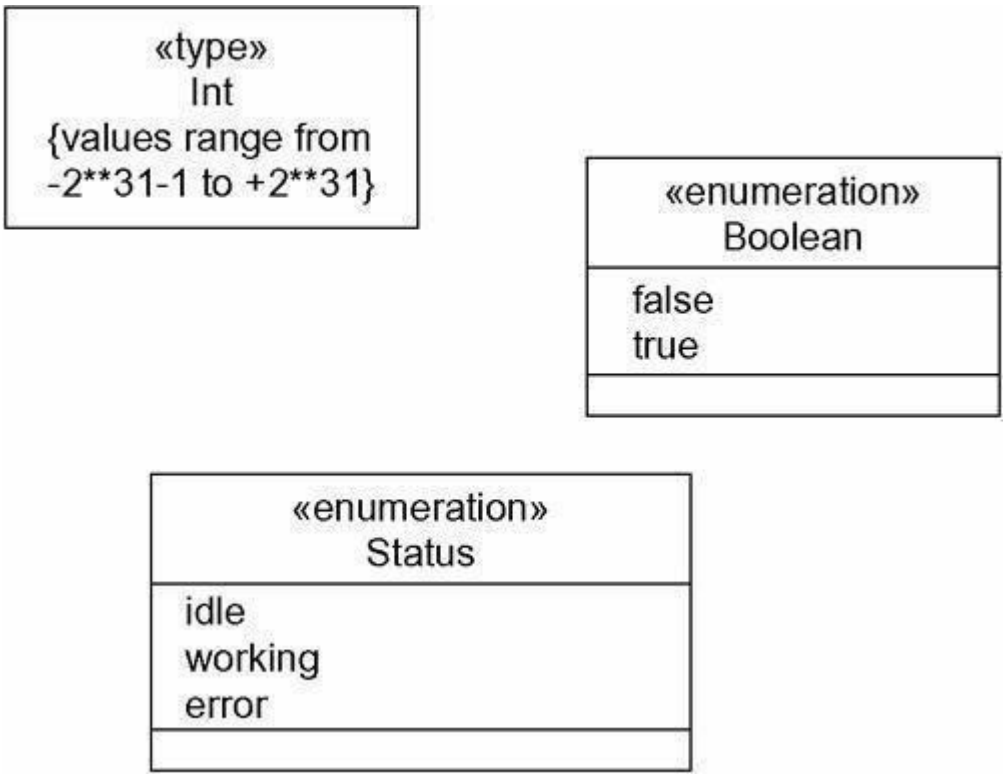
At the other extreme, the things you model may be drawn directly from the programming language you are using to implement a solution. Typically, these abstractions involve primitive types, such as integers, characters, strings, and even enumeration types, that you might create yourself.

To model primitive types,

- Model the thing you are abstracting as a type or an enumeration, which is rendered using class notation with the appropriate stereotype.
- If you need to specify the range of values associated with this type, use constraints.

As Figure shows, these things can be modeled in the UML as types or enumerations, which are rendered just like classes but are explicitly marked via stereotypes. Things like integers (represented by the class **Int**) are modeled as types, and you can explicitly indicate the range of values these things can take on by using a constraint. Similarly, enumeration types, such as **Boolean** and **Status**, can be modeled as enumerations, with their individual values provided asattributes.

Figure Modeling Primitive Types



Relationships

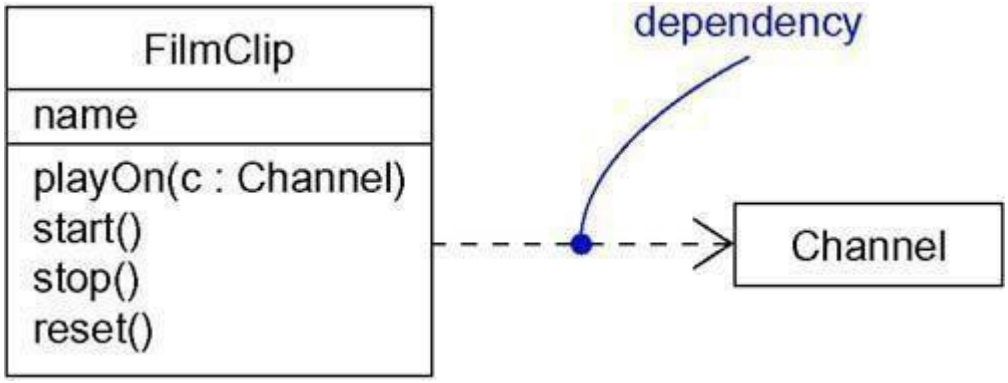
Terms and Concepts

A *relationship* is a connection among things. In object-oriented modeling, the three most important relationships are dependencies, generalizations, and associations. Graphically, a relationship is rendered as a path, with different kinds of lines used to distinguish the kinds of relationships.

Dependency

A *dependency* is a using relationship that states that a change in specification of one thing (for example, class **Event**) may affect another thing that uses it (for example, class**Window**), but notnecessarily the reverse. Graphically, a dependency is rendered as a dashed directed line, directed to the thing being depended on. Use dependencies when you want to show one thing using another.

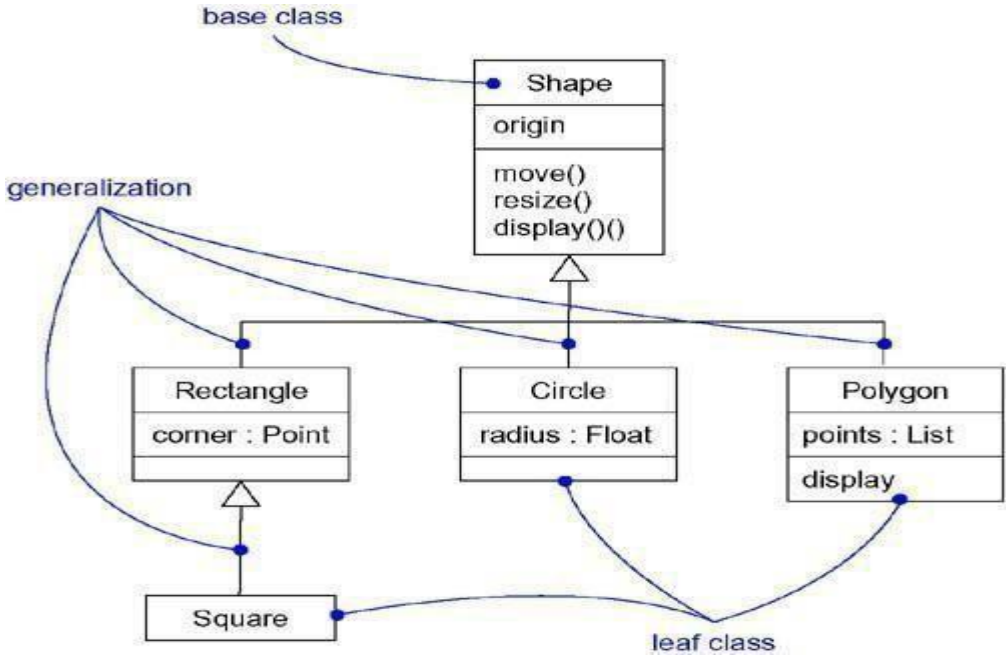
Figure Dependencies



Generalization

A *generalization* is a relationship between a general thing (called the superclass or parent)and a more specific kind of that thing (called the subclass or child). Generalization is sometimes called an "is-a-kind-of" relationship: one thing (like the class **BayWindow**) is-a-kind-of a more general thing (for example, the class **Window**). Generalization means that objects of the child may be used anywhere the parent may appear, but not the reverse. In other words, generalization means that the child is substitutable for the parent. A child inherits the properties of its parents, especially their attributes and operations. Often• but not always• the child has attributes and operations in addition to those found in its parents. An operation of a child that has the same signature as an operation in a parent overrides the operation of the parent; this is known as polymorphism. Graphically, generalization is rendered as a solid directed line with a large open arrowhead, pointing to the parent, as shown in Figure. Use generalizations when you want to show parent/child relationships.

Figure Generalization



A class may have zero, one, or more parents. A class that has no parents and one or more children is called a root class or a base class. A class that has no children is called a leaf class. A class that has exactly one parent is said to use single inheritance; a class with more than one parent is said to use multiple inheritance.

Association

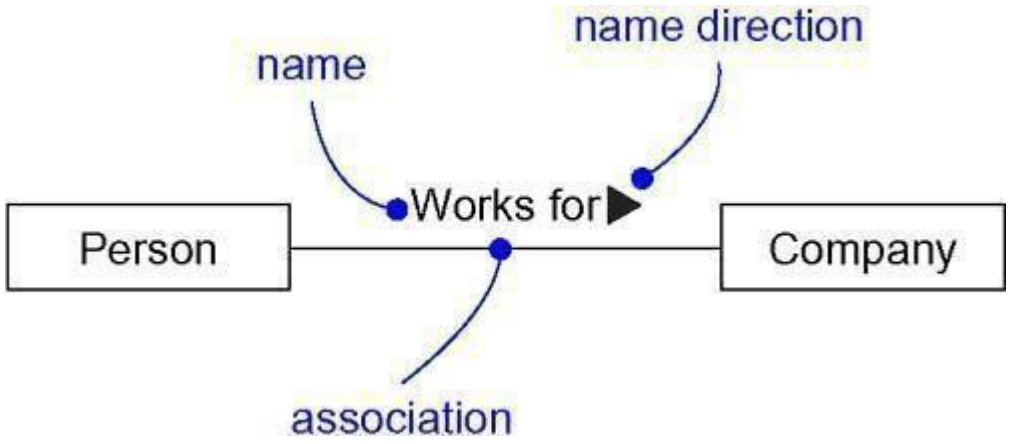
An *association* is a structural relationship that specifies that objects of one thing are connected to objects of another. Given an association connecting two classes, you can navigate from an objectof one class to an object of the other class, and vice versa. It's quite legal to have both ends of an association circle back to the same class. This means that, given an object of the class, you can link to other objects of the same class. An association that connects exactly two classes is called a binary association. Although it's not as common, you can have associations that connect more than two classes; these are called n-ary associations. Graphically, an association is rendered as a solid line connecting the same or different classes. Use associations when you want to show structural relationships.

Beyond this basic form, there are four adornments that apply to associations.

Name

An association can have a name, and you use that name to describe the nature of the relationship. So that there is no ambiguity about its meaning, you can give a direction to the name by providing a direction triangle that points in the direction you intend to read the name, as shown in Figure.

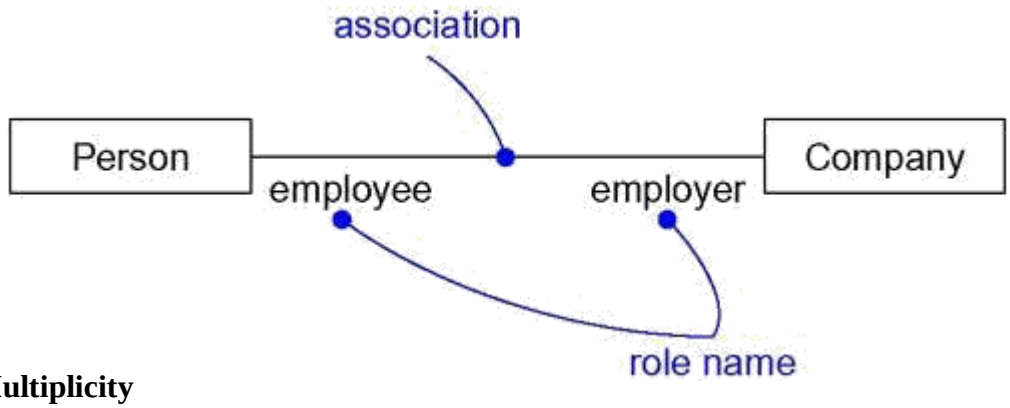
Figure Association Names



Role

When a class participates in an association, it has a specific role that it plays in that relationship; a role is just the face the class at the near end of the association presents to the class at the other end of the association. You can explicitly name the role a class plays in an association. In Figure, a**Person**playing the role of**employee**is associated with a**Company**playing therole of **employer.**

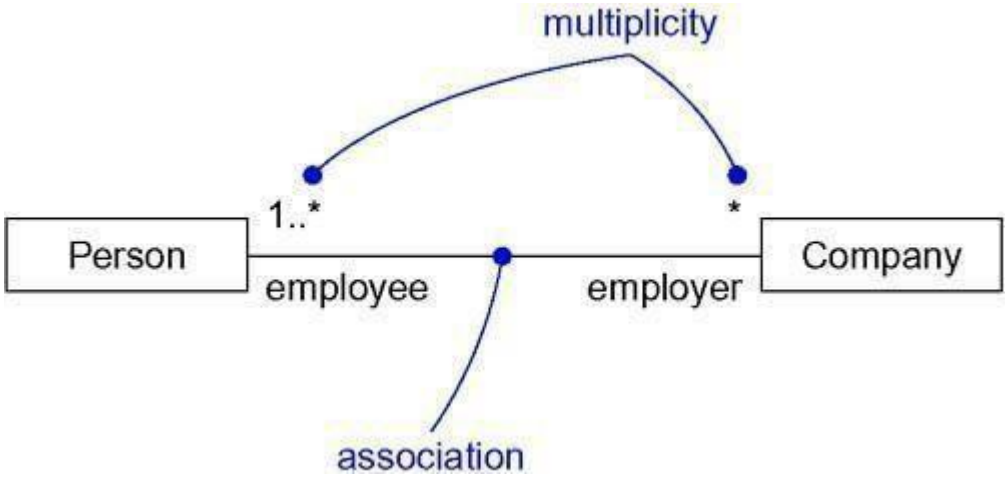
Figure Roles



Multiplicity

An association represents a structural relationship among objects. In many modeling situations, it's important for you to state how many objects may be connected across an instance of an association. This "how many" is called the multiplicity of an association's role, and is written as an expression that evaluates to a range of values or an explicit value as in Figure . When you state a multiplicity at one end of an association, you are specifying that, for each object of the class at the opposite end, there must be that many objects at the near end. You can show a multiplicity of exactly one (**1**), zero or one (**0..1**), many (**0..***), or one or more (**1..***). You can even state an exact number (for example, **3**).

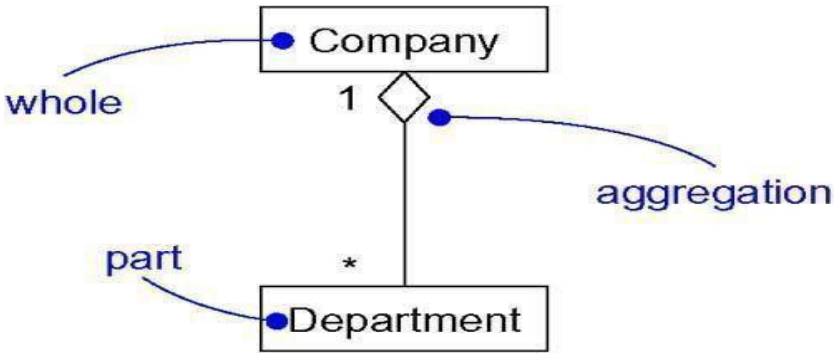
Figure Multiplicity



Aggregation

A plain association between two classes represents a structural relationship between peers, meaning that both classes are conceptually at the same level, no one more important than the other. Sometimes, you will want to model a "whole/part" relationship, in which one class represents a larger thing (the "whole"), which consists of smaller things (the "parts"). This kind of relationship is called aggregation, which represents a "has-a" relationship, meaning that an object of the whole has objects of the part. Aggregation is really just a special kind of association and is specified by adorning a plain association with an open diamond at the whole end, as shown in Figure .

Figure Aggregation



Other Features

Plain, unadorned dependencies, generalizations, and associations with names, multiplicities, and roles are the most common features you'll need when creating abstractions. In fact, for most of the models you build, the basic form of these three relationships will be all you need to convey the most important semantics of your relationships.

Common Modeling Techniques

Modeling Simple Dependencies

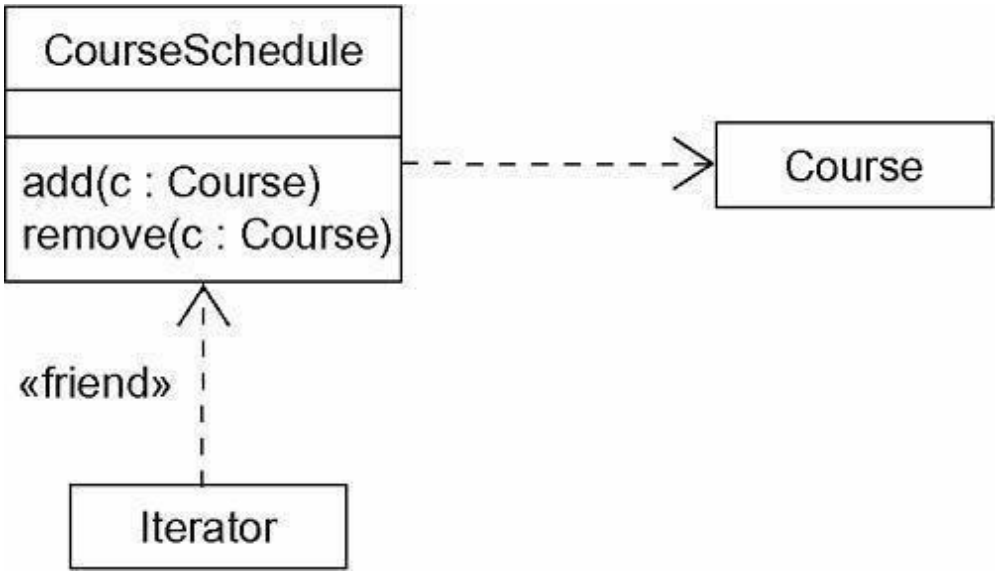
The most common kind of dependency relationship is the connection between a class that only uses another class as a parameter to an operation.

To model this using relationship,

- Create a dependency pointing from the class with the operation to the class used as a parameter in the operation.

For example, Figure shows a set of classes drawn from a system that manages the assignment of students and instructors to courses in a university. This figure shows a dependency from **CourseSchedule** to **Course**, because **Course** is used in both the **add** and **remove** operations of **CourseSchedule.**

Figure Dependency Relationships



Modeling Single Inheritance

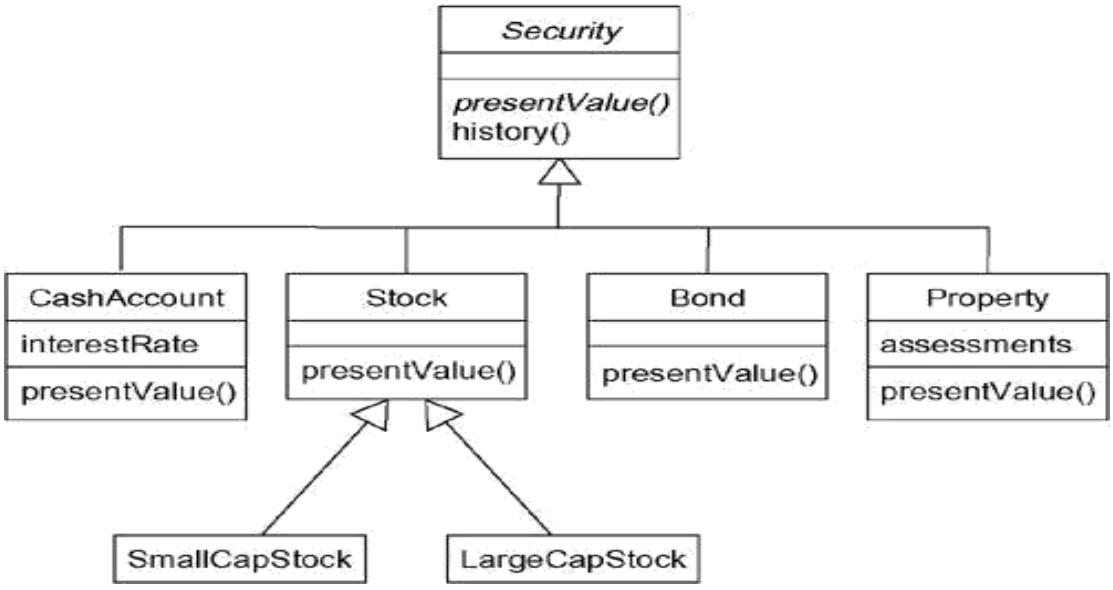
In modeling the vocabulary of your system, you will often run across classes that are structurally or behaviorally similar to others. You could model each of these as distinct and unrelated abstractions. A better way would be to extract any common structural and behavioral features and place them in more-general classes from which the specialized ones inherit.

To model inheritance relationships,

- Given a set of classes, look for responsibilities, attributes, and operations that are common to two or more classes.
- Elevate these common responsibilities, attributes, and operations to a more general class. If necessary, create a new class to which you can assign these elements (but be careful about introducing too many levels).
- Specify that the more-specific classes inherit from the more-general class by placing a generalization relationship that is drawn from each specialized class to its more-general parent.

For example, Figure shows a set of classes drawn from a trading application. You will find a generalization relationship from four classes• **CashAccount**, **Stock**, **Bond**, and **Property**• to the more-general class named **Security. Security** is the parent, and **CashAccount**, **Stock**, **Bond**, and **Property** are all children. Each of these specialized children is a kind of **Security.** You'll notice that **Security** includes two operations: **presentValue** and **history.** Because **Security** is their parent, **CashAccount**, **Stock**, **Bond**, and **Property** all inherit these twooperations, and for that matter, any other attributes and operations of **Security** that may be elided in this figure.

Figure Inheritance Relationships



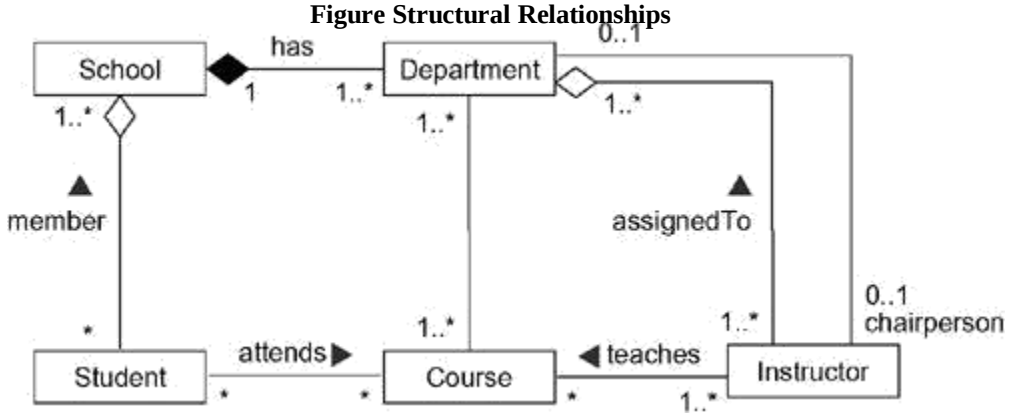
Modeling Structural Relationships

When you model with dependencies or generalization relationships, you are modeling classes that represent different levels of importance or different levels of abstraction. Given a dependencybetween two classes, one class depends on another but the other class has no knowledge of the one. Given a generalization relationship between two classes, the child inherits from its parent but the parent has no specific knowledge of its children. In short, dependency and generalization relationships areone-sided.

To model structural relationships,

- For each pair of classes, if you need to navigate from objects of one to objects of another, specify an association between the two. This is a data-driven view of associations.
- For each pair of classes, if objects of one class need to interact with objects of the other class other than as parameters to an operation, specify an association between the two. This is more of a behavior-driven view of associations.
- For each of these associations, specify a multiplicity (especially when the multiplicity is not *****, which is the default), as well as role names (especially if it helps to explain the model).
- If one of the classes in an association is structurally or organizationally a whole compared with the classes at the other end that look like parts, mark this as an aggregation by adorning the association at the end near the whole.

Figure Structural Relationships



The relationships between **School** and the classes **Student** and **Department** are a bit different. Here you'll see aggregation relationships. A school has zero or more students, each student may be a registered member of one or more schools, a school has one or more departments, each department belongs to exactly one school. You could leave off the aggregation adornments and use plain associations, but by specifying that **School** is a whole and that **Student** and **Department** are some of its parts, you make clear which one is organizationally superior to the other. Thus, schools are somewhat defined by the students and departments they have. Similarly, students and departments don't really stand alone outside the school to which they belong. Rather, they get some of their identity from theirschool.

You'll also see that there are two associations between **Department** and **Instructor.** One of these associations specifies that every instructor is assigned to one or more departments and that each department has one or more instructors. This is modeled as an aggregation because organizationally,

departments are at a higher level in the school's structure than are instructors. The other association specifies that for every department, there is exactly one instructor who is the department chair. The way this model is specified, an instructor can be the chair of no more than one department and some instructors are not chairs of any department.

Common Mechanisms

Terms and Concepts

A *note* is a graphical symbol for rendering constraints or 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.

A *stereotype* is an extension of the vocabulary of the UML, allowing you to create new kinds of building blocks similar to existing ones but specific to your problem. Graphically, a stereotype is rendered as a name enclosed by guillemets and placed above the name of another element. As an option, the stereotyped element may be rendered by using a new icon associated with that stereotype.

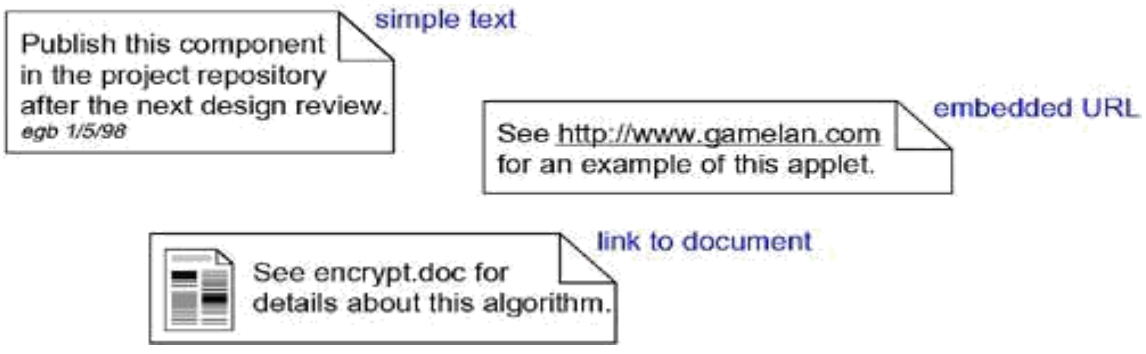
A *tagged value* is an extension of the properties of a UML element, allowing you to create new information in that element's specification. Graphically, a tagged value is rendered as a string enclosed by brackets and placed below the name of another element.

A *constraint* is an extension of the semantics of a UML element, allowing you to add new rules or to modify existing ones. Graphically, a constraint is rendered as a string enclosed by brackets and placed near the associated element or connected to that element or elements by dependency relationships. As an alternative, you can render a constraint in a note.

Notes

A note that renders a comment has no semantic impact, meaning that its contents do not alter the meaning of the model to which it is attached. This is why notes are used to specify things like requirements, observations, reviews, and explanations, in addition to rendering constraints.

Figure Notes

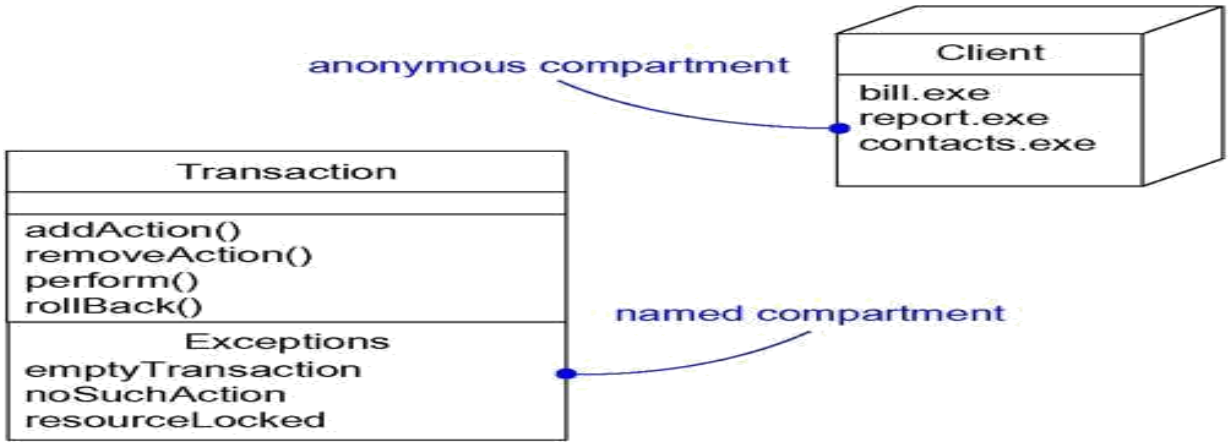


Other Adornments

Adornments are textual or graphical items that are added to an element's basic notation and are used to visualize details from the element's specification. For example, the basic notation for an association is a line, but this may be adorned with such details as the role and multiplicity of each end. In using the UML, the general rule to follow is this: Start with the basic notation for each element and then add other adornments only as they are necessary to convey specific information that is important to your model.

Most adornments are rendered by placing text near the element of interest or by adding a graphic symbol to the basic notation. However, sometimes you'll want to adorn an element with more detail than can be accommodated by simple text or graphics. In the case of such things as classes, components, and nodes, you can add an extra compartment below the usual compartments to provide this information, as Figureshows.

Figure Extra Compartments

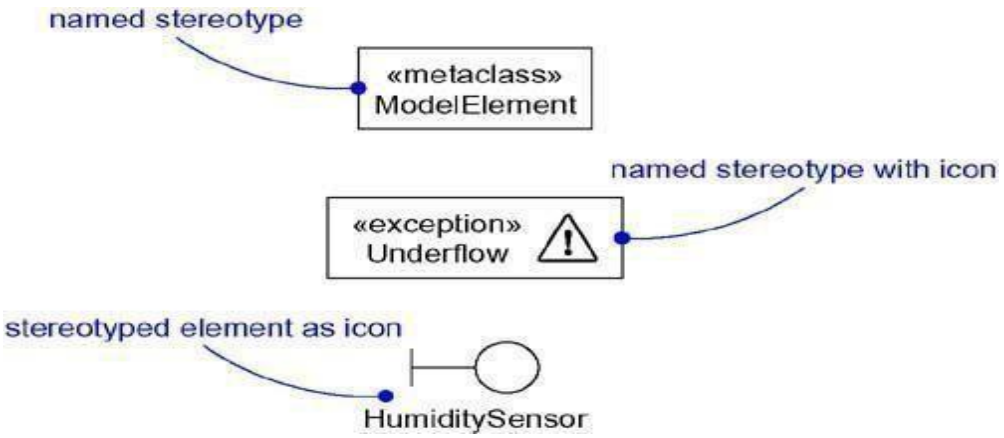


Stereotypes

The UML provides a language for structural things, behavioral things, grouping things, and notational things.

A stereotype is not the same as a parent class in a parent/child generalization relationship. Rather, you can think of a stereotype as a metatype, because each one creates the equivalent of a new class in the UML's metamodel. For example, if you are modeling a business process, you'll want to introduce things like workers, documents, and policies.
In its simplest form, a stereotype is rendered as a name enclosed by guillemets (for example, **»name**) and placed above the name of another element. As a visual cue, you may define anicon for the stereotype and render that icon to the right of the name (if you are using the basic notation for the element) or use that icon as the basic symbol for the stereotyped item. All three of these approaches are illustrated in Figure.

Figure Stereotypes

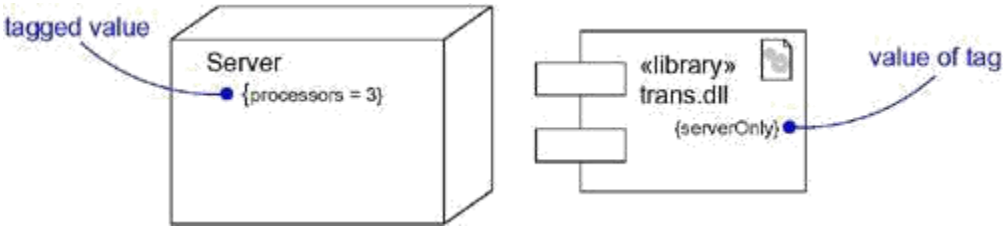


Tagged Values

Every thing in the UML has its own set of properties: classes have names, attributes, and operations; associations have names and two or more ends (each with its own properties); and so on. With stereotypes, you can add new things to the UML; with tagged values, you can add new properties.

You can define tags for existing elements of the UML, or you can define tags that apply to individual stereotypes so that everything with that stereotype has that tagged value. A tagged value is not the same as a class attribute. Rather, you can think of a tagged value as metadata because its value applies to the element itself, not its instances. For example, as Figure shows, you might want to specify the number of processors installed on each kind of node in a deployment diagram, or you might want to require that every component be stereotyped as a library if it is intended to be deployed on a client or a server.

Figure Tagged Values

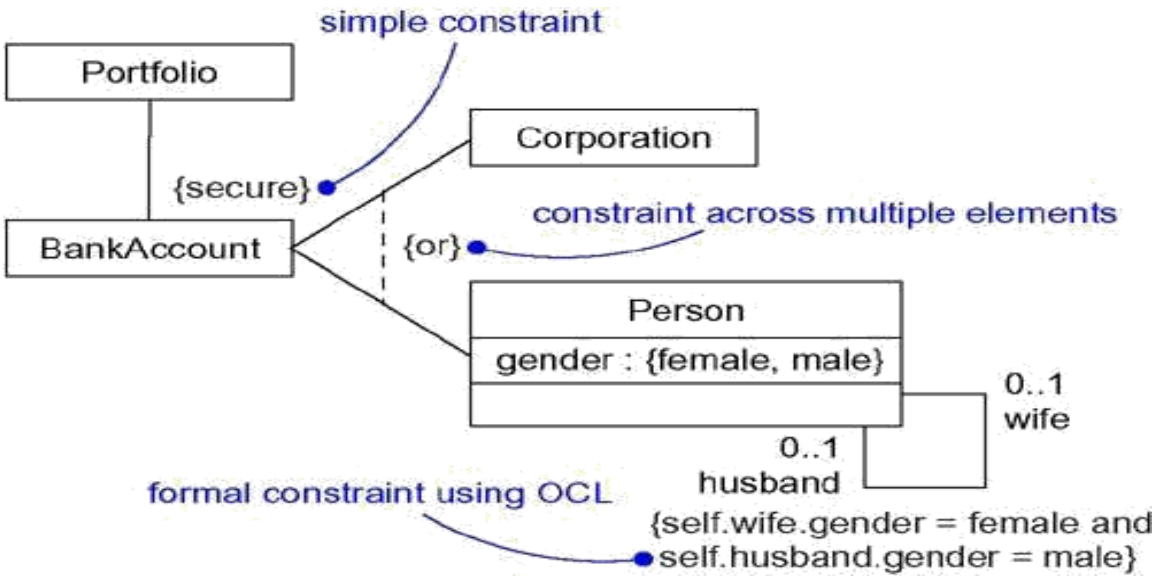


Constraints

Everything in the UML has its own semantics. Generalization implies the Liskov substitution principle, and multiple associations connected to one class denote distinct relationships. With constraints, you can add new semantics or change existing rules. A constraint specifies conditions that must be held true for the model to be well-formed.

For example, as Figure shows, you might want to specify that, across a given association, communication is encrypted. Similarly, you might want to specify that among a set of associations, only one is manifest at a time.

Figure Constraint



A constraint is rendered as a string enclosed by brackets and placed near the associated element. This notation is also used as an adornment to the basic notation of an element to visualize parts of an element's specification that have no graphical cue. For example, some properties of associations (order and changeability) are rendered using constraint notation.

Standard Elements

The UML defines a number of standard stereotypes for classifiers, components, relationships, and other modeling elements. There is one standard stereotype, mainly of interest to tool builders, that lets you model stereotypes themselves.

Stereotype	Specifies that the classifier is a stereotype that may be applied to other elements
documentation	Specifies a comment, description, or explanation of the element to which it attached

The UML also specifies one standard tagged value that applies to all modeling elements.
You'll use this tagged value when you want to attach a comment directly to the specification of an element, such as a class.

Common Modeling Techniques

Modeling Comments

To model a comment

To model a comment, Put your comment as text in a note and place it adjacent to the element to

which it refers. You can show a more explicit relationship by connecting a note to its elements using a dependency relationship.

- Remember that you can hide or make visible the elements of your model as you see fit. This means that you don't have to make your comments visible everywhere the elements to which it is attached are visible. Rather, expose your comments in your diagrams only insofar as you need to communicate that information in that context
- If your comment is lengthy or involves something richer than plain text, consider putting your comment in an external document and linking or embedding that document in a note attached to your model
- As your model evolves, keep those comments that record significant decisions that cannot be inferred from the model itself, and• unless they are of historic interest• discard the others

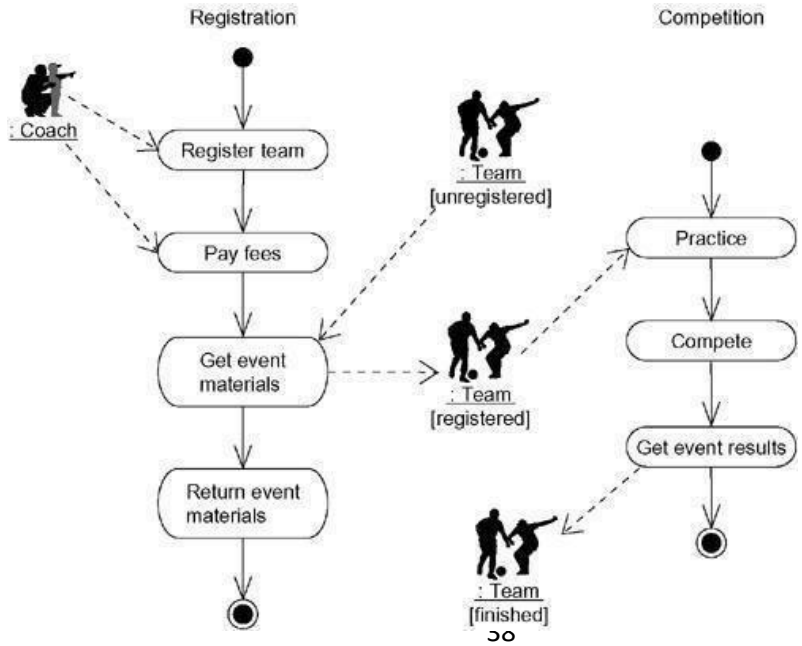
For example, Figure shows a model that's a work in progress of a class hierarchy, showing some requirements that shape the model as well as some notes from a design review.

Modeling New Building Blocks

To model new building blocks,

- Make sure there's not already a way to express what you want by using basic UML. If you have a common modeling problem, chance are there's already some standard stereotype that will do what you want.
- If you're convinced there's no other way to express these semantics, identify the primitive thing in the UML that's most like what you want to model (for example, class, interface, component, node, association, and so on) and define a new stereotype for that thing. Remember that you can define hierarchies of stereotypes so that you can have general kinds of stereotypes along with their specializations (but as with any hierarchy, use this sparingly).
- Specify the common properties and semantics that go beyond the basic element being stereotyped by defining a set of tagged values and constraints for the stereotype
- If you want these stereotype elements to have a distinctive visual cue, define a new icon for the stereotype

Figure Modeling New Building Blocks.



Modeling New Properties

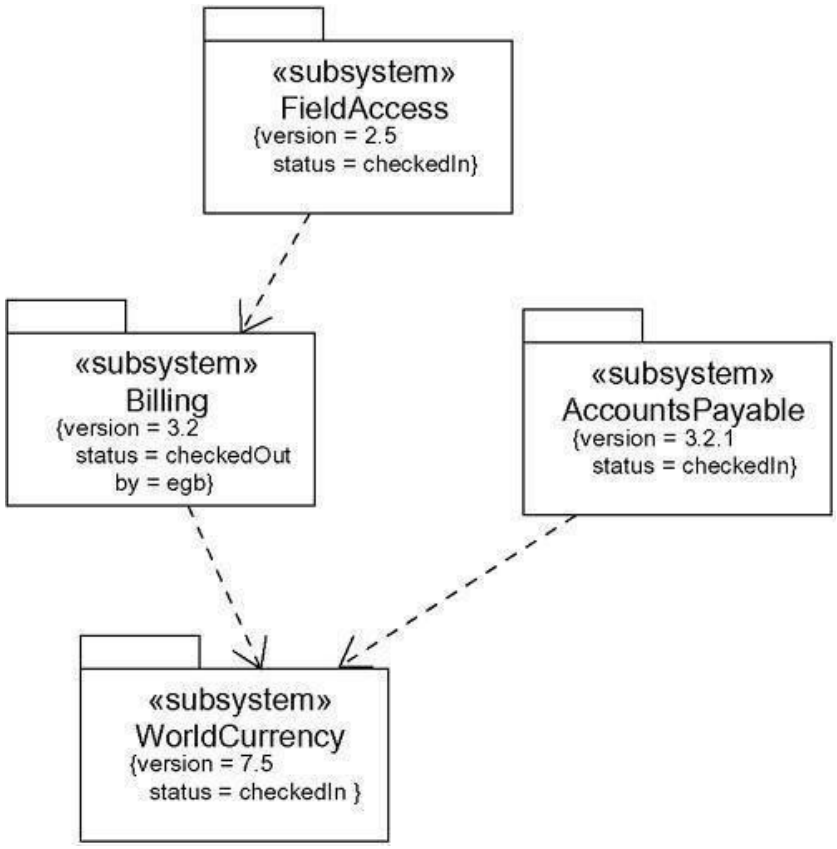
To model new properties,

- First, make sure there's not already a way to express what you want by using basic UML. If you have a common modeling problem, chances are that there's already some standard tagged value that will do what you want.
- If you're convinced there's no other way to express these semantics, add this new property to an individual element or a stereotype. The rules of generalization apply• tagged values defined for one kind of element apply to its children.
-

For example, suppose you want to tie the models you create to your project's configuration management system. Among other things, this means keeping track of the version number, current check in/check out status, and perhaps even the creation and modification dates of each subsystem. Because this is process-specific information, it is not a basic part of the UML, although you can add this information as tagged values. Furthermore, this information is not just a class attribute either. A subsystem's version number is part of its metadata, not part of the model.

Figure shows four subsystems, each of which has been extended to include its versionnumber and status. In the case of the **Billing** subsystem, one other tagged value is shown• the person who has currently checked out the subsystem.

Figure Modeling New Properties



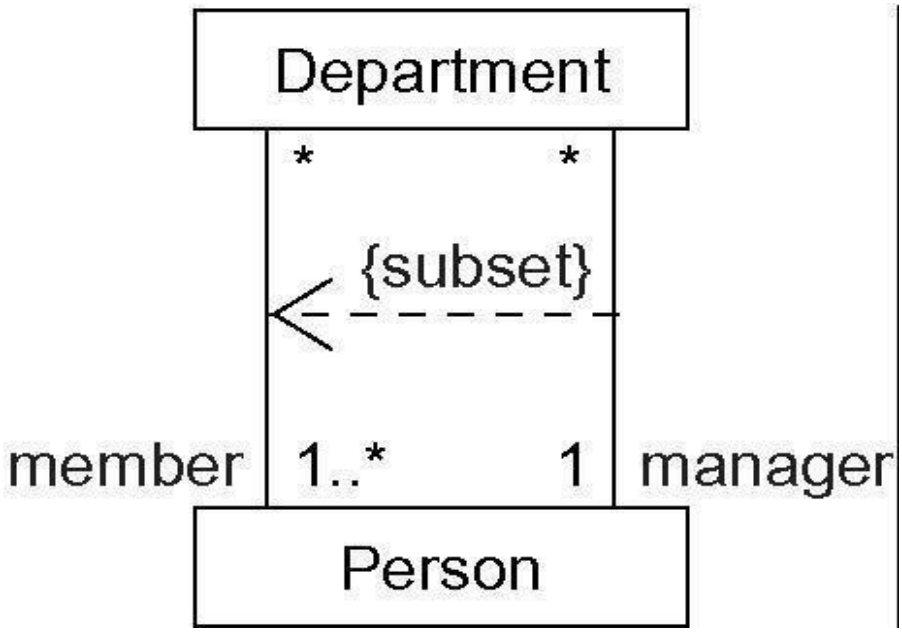
Modeling New Semantics

To model new semantics,

- First, make sure there's not already a way to express what you want by using basic UML. If you have a common modeling problem, chances are that there's already some standard constraint that will do what you want.
- If you're convinced there's no other way to express these semantics, write your new semantics as text in a constraint and place it adjacent to the element to which it refers. You can show a more explicit relationship by connecting a constraint to its elements using a dependency relationship.
- If you need to specify your semantics more precisely and formally, write your new semantics using OCL.

For example, Figure models a small part of a corporate human resources system.

Figure Modeling New Semantics



This diagram shows that each person may be a member of zero or more departments and that each department must have at least one person as a member. This diagram goes on to indicate that each department must have exactly one person as a manager and every person may be the manager of zero or more departments. All of these semantics can be expressed using simple uml. However, to assert that a manager must also be a member of the department is something that cuts across multiple associations and cannot be expressed using simple uml. To state this invariant, you have to write a constraint that shows the manager as a subset of the members of the department, connecting the two associations and the constraint by a dependency from the subset to the superset.

Diagrams

A *system* is a collection of subsystems organized to accomplish a purpose and described by a set of models, possibly from different viewpoints.
A *subsystem* is a grouping of elements, of which some constitute a specification of the behavior offered by the other contained elements.
A *model* is a semantically closed abstraction of a system, meaning that it represents a complete and self-consistent simplification of reality, created in order to better understand the system. In the context of architecture, a *view* is a projection into the organization and structure of a system's model, focused on one aspect of that system. A *diagram* is the graphical presentation of a set of elements, most often rendered as a connected graph of vertices (things) and arcs (relationships).

In modeling real systems, no matter what the problem domain, you'll find yourself creating the same kinds of diagrams, because they represent common views into common models. Typically, you'll view the static parts of a system using one of the four following diagrams.

1. Class diagram
2. Object diagram
3. Component diagram
4. Deployment diagram

You'll often use five additional diagrams to view the dynamic parts of a system.

1. Use case diagram
2. Sequence diagram
3. Collaboration diagram
4. Statechart diagram
5. Activity diagram

The UML defines these nine kinds of diagrams.

Structural Diagrams

The UML's structural diagrams are roughly organized around the major groups of things you'll find when modeling a system.

1	Class diagram	Class, interfaces and collaborations
2	Objects diagram	Objects
3	Component diagram	Components
4	Development diagram	Nodes

Class Diagram

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

Object Diagram

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

Component Diagram

A *component diagram* shows a set of components and their relationships. You use component diagrams to illustrate the static implementation view of a system. Component diagrams are related to class diagrams in that a component typically maps to one or more classes, interfaces, or collaborations.

Deployment Diagram

A *deployment diagram* shows a set of nodes and their relationships. You use deployment diagrams to illustrate the static deployment view of an architecture. Deployment diagrams are related to component diagrams in that a node typically encloses one or more components.

Behavioral Diagrams

The UML's five behavioral diagrams are used to visualize, specify, construct, and document the dynamic aspects of a system.

The UML's behavioral diagrams are roughly organized around the major ways you can model the dynamics of a system.

1	Use case diagram	Organizes the behaviors of the system
2	Sequence diagram	Focused on the time ordering of messages
3	Collaboration diagram	Focused on the structural organization of objects that send and receive messages
4	State chart diagram	Focused on the changing state of a system driven by events
5	Activity diagram	Focused on the flow of control from activity to activity

Use Case Diagram

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

Interaction diagram is the collective name given to sequence diagrams and collaboration diagrams. All sequence diagrams and collaborations are interaction diagrams, and an interaction diagram is either a sequence diagram or a collaboration diagram.

Sequence Diagram
A *sequence diagram* is an interaction diagram that emphasizes the time ordering of messages. A sequence diagram shows a set of objects and the messages sent and received by those objects. The objects are typically named or anonymous instances of classes, but may also represent instances of other things, such as collaborations, components, and nodes. You use sequence diagrams to illustrate the dynamic view of a system.

Collaboration Diagram

A *collaboration diagram* is an interaction diagram that emphasizes the structural organization of the objects that send and receive messages. A collaboration diagram shows a set of objects, links among those objects, and messages sent and received by those objects. The objects are typically named or anonymous instances of classes, but may also represent instances of other things, such as collaborations, components, and nodes. You use collaboration diagrams to illustrate the dynamic view of a system.

State Chart Diagram

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

Activity Diagram

An *activity diagram* shows the flow from activity to activity within a system. An activity shows a set of activities, the sequential or branching flow from activity to activity, and objects that act and are acted upon. You use activity diagrams to illustrate the dynamic view of a system. Activity diagrams are especially important in modeling the function of a system. Activity diagrams emphasize the flow of control among objects.

Common Modeling Techniques

Modeling Different Views of a System

To model a system from different views,

- Decide which views you need to best express the architecture of your system and to expose the technical risks to your project. The five views of an architecture described earlier are a good starting point.
- For each of these views, decide which artifacts you need to create to capture the essential details of that view. For the most part, these artifacts will consist of various UML diagrams.

- As part of your process planning, decide which of these diagrams you'll want to put under some sort of formal or semi-formal control. These are the diagrams for which you'll want to schedule reviews and to preserve as documentation for the project.
- Allow room for diagrams that are thrown away. Such transitory diagrams are still useful for exploring the implications of your decisions and for experimenting with changes.

For example, if you are modeling a simple monolithic application that runs on a single machine, you might need only the following handful of diagrams.

Use case view	Use case diagrams
Design view	Class diagrams (for structural modeling) Interaction diagrams (for behavioral modeling)
Process view	None required
Implementation view	None required
Deployment view	None required

Finally, if you are modeling a complex, distributed system, you'll need to employ the full range of the UML's diagrams in order to express the architecture of your system and the technical risks to your project, as in the following.

Use case view	Use case diagrams Activity diagrams (for behavioral modeling)
Design view	Class diagrams (for structural modeling) Interaction diagrams (for behavioral modeling) statechart diagram (for behavioral modeling)
Process view	Class diagrams (for structural modeling) Interaction diagrams (for behavioral modeling)
Implementation view	Component diagram
Deployment view	Component diagram

Modeling Different Levels of Abstraction

To model a system at different levels of abstraction by presenting diagrams with different levels of detail,

- Consider the needs of your readers, and start with a given model.
- If your reader is using the model to construct an implementation, she'll need diagrams that are at a lower level of abstraction, which means that they'll need to reveal a lot of detail. If she is using the model to present a conceptual model to an end user, she'll need diagrams that are at a higher level of abstraction, which means that they'll hide a lot of detail.
- Depending on where you land in this spectrum of low-to-high levels of abstraction, create a diagram at the right level of abstraction by hiding or revealing the following four categories of things from your model:

    1. **Building blocks and relationships:**

        Hide those that are not relevant to the intent of your diagram or the needs of your reader.

    2. **Adornments:**

        Reveal only the adornments of these building blocks and relationships that are essential to understanding your intent.

3. **Flow:**

   In the context of behavioral diagrams, expand only those messages or transitions that are essential to understanding your intent.

4. **Stereotypes:**

   In the context of stereotypes used to classify lists of things, such as attributes and operations, reveal only those stereotyped items that are essential to understanding your intent.

The main advantage of this approach is that you are always modeling from a common semantic repository. The main disadvantage of this approach is that changes from diagrams at one level of abstraction may make obsolete diagrams at a different level of abstraction.

To model a system at different levels of abstraction by creating models at different levels of abstraction,

- Consider the needs of your readers and decide on the level of abstraction that each should view, forming a separate model for each level.
- In general, populate your models that are at a high level of abstraction with simple abstractions and your models that are at a low level of abstraction with detailed abstractions. Establish trace dependencies among the related elements of different models.
- In practice, if you follow the five views of an architecture, there are four common situations you'll encounter when modeling a system at different levels of abstraction:

1. **Use cases and their realization:**

   Use cases in a use case model will trace to collaborations in a design model.

2. **Collaborations and their realization:**

   Collaborations will trace to a society of classes that work together to carry out the collaboration.

3. **Components and their design:**

   Components in an implementation model will trace to the elements in a design model.

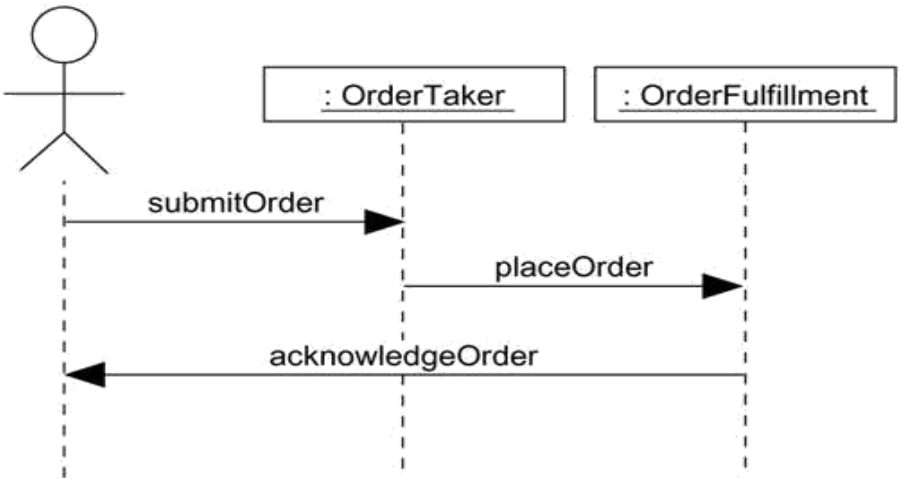
4. **Nodes and their components:**

   Nodes in a deployment model will trace to components in an implementation model.

The main advantage of the approach is that diagrams at different levels of abstraction remain more loosely coupled. This means that changes in one model will have less direct effect on other models. The main disadvantage of this approach is that you must spend resources to keep these models and their diagrams synchronized. This is especially true when your models parallel different phases of the software development life cycle, such as when you decide to maintain an analysis model separate from a design model.

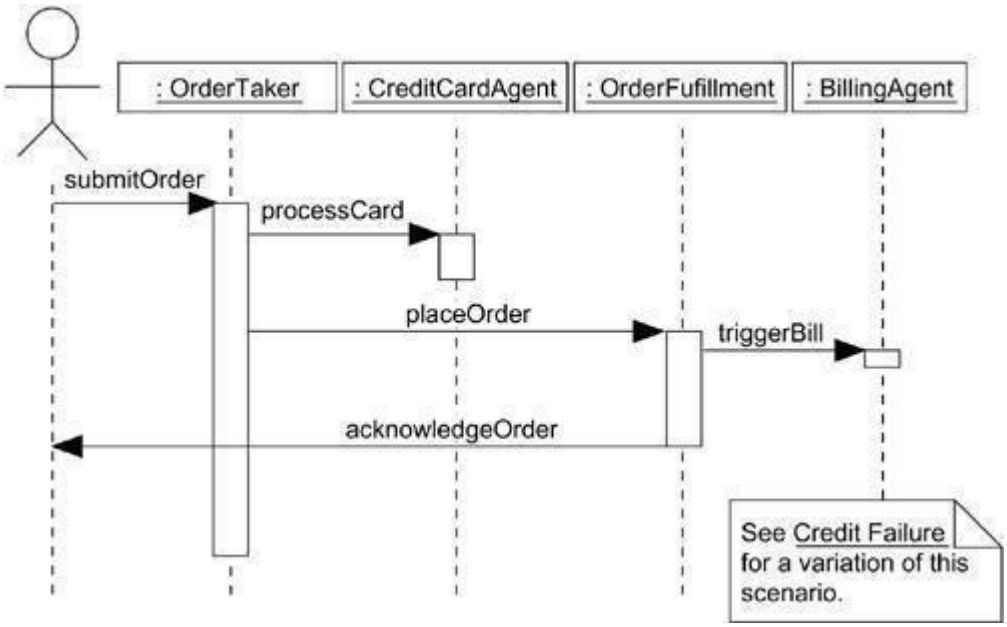
For example, suppose you are modeling a system for Web commerce• one of the main use cases of such a system would be for placing an order. If you're an analyst or an end user, you'd probably create some interaction diagrams at a high level of abstraction that show the action of placing an order, as in Figure.

Figure Interaction Diagram at a High Level of Abstraction



On the other hand, a programmer responsible for implementing this scenario would have to build on this diagram, expanding certain messages and adding other players in this interaction, as in Figure .

Figure Interaction at a Low Level of Abstraction



Both of these diagrams work against the same model, but at different levels of detail. It's reasonable to have many diagrams such as these, especially if your tools make it easy to navigate from one diagram to another.

Modeling Complex Views

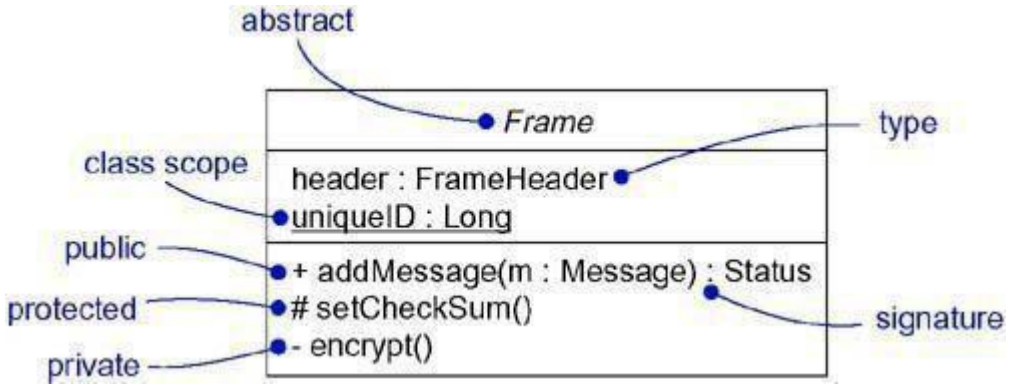
To model complex views,

- First, convince yourself there's no meaningful way to present this information at a higher level of abstraction, perhaps eliding some parts of the diagram and retaining the detail in other parts.
- If you've hidden as much detail as you can and your diagram is still complex, consider grouping some of the elements in packages or in higher level collaborations, then render only those packages or collaborations in your diagram.

- If your diagram is still complex, use notes and color as visual cues to draw the reader's attention to the points you want to make.
- If your diagram is still complex, print it in its entirety and hang it on a convenient large wall. You lose the interactivity an online version of the diagram brings, but you can step back from the diagram and study it for common patterns.

Classifiers (and especially classes) have a number of advanced features beyond the simpler properties of attributes and operations described in the previous section: You can model multiplicity, visibility, signatures, polymorphism, and other characteristics. In the UML, you can model the semantics of a class so that you can state its meaning to whatever degree of formality you like.

Figure Advanced Classes



Advanced classes

Terms and Concepts

A *classifier* is a mechanism that describes structural and behavioral features. Classifiers include classes, interfaces, datatypes, signals, components, nodes, use cases, and subsystems.

Classifiers

When you model, you'll discover abstractions that represent things in the real world and things in your solution. For example, if you are building a Web-based ordering system, the vocabulary of your project will likely include a **Customer** class (representing people who order products) and a**Transaction** class (an implementation artifact, representing an atomic action). In the deployedsystem, you might have a **Pricing** component, with instances living on every client node. Each of these abstractions will have instances; separating the essence and the instance of the things in your world is an important part of modeling.

The most important kind of classifier in the UML is the class. A class is a description of a set of objects that share the same attributes, operations, relationships, and semantics. Classes are not the only kind of classifier, however. The UML provides a number of other kinds of classifiers to help you model.

Interface	A collection of operations that are used to specify a service of a class or a component

Datatype	A type whose values have no identity, including primitive built-in types (such as numbers and strings), as well as enumeration types (such as Boolean)
Signal	The specification of an asynchronous stimulus communicated between instances
Component re	A physical and replaceable part of a system that conforms to and provides the alization of a set of interfaces
Node	A physical element that exists at run time and that represents a computational resource, generally having at least some memory and often processing capability
Use case	A description of a set of a sequence of actions, including variants, that a system performs that yields an observable result of value to a particular actor
Subsystem	A grouping of elements of which some constitute a specification of the behavior offered by the other contained elements

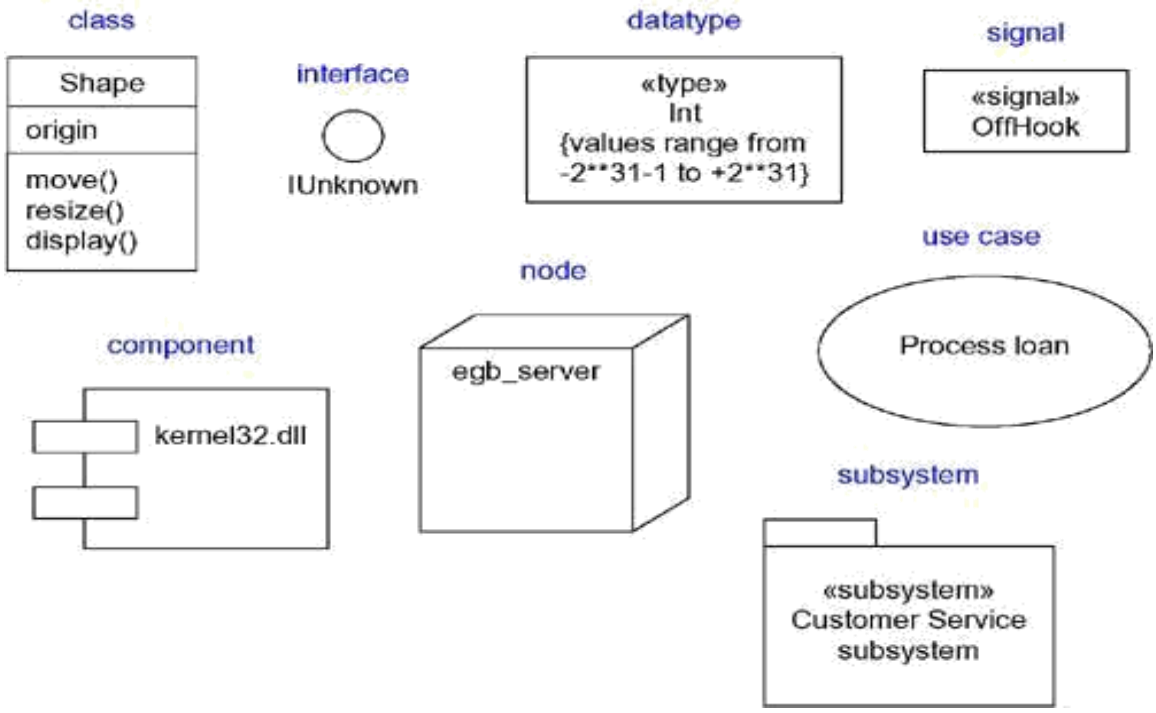
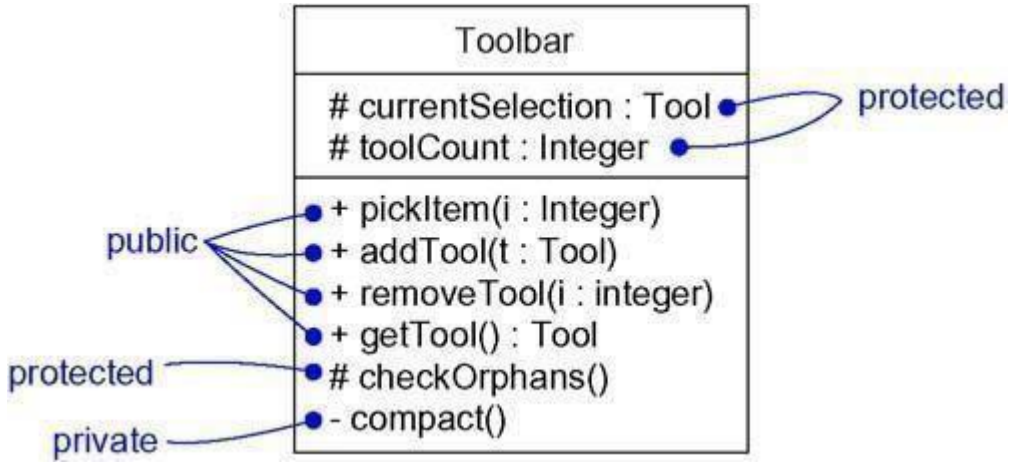


Figure Classifiers

Visibility
1.public
2.private
3.protected

Figure Visibility



When you specify the visibility of a classifier's features, you generally want to hide all its implementation details and expose only those features that are necessary to carry out the responsibilities of the abstraction. That's the very basis of information hiding, which is essential to building solid, resilient systems. If you don't explicitly adorn a feature with a visibility symbol, you can usually assume that it is public.

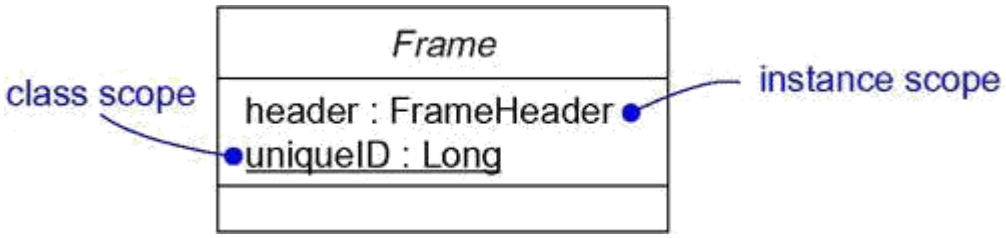
Scope

Another important detail you can specify for a classifier's attributes and operations is its owner scope. The owner scope of a feature specifies whether the feature appears in each instance of the classifier or whether there is just a single instance of the feature for all instances of the classifier. In the UML, you can specify two kinds of owner scope.

1. **instance**	Each instance of the classifier holds its own value for the feature.
2. **classifier**	There is just one value of the feature for all instances of the classifier.

As Figure (a simplification of the first figure) shows, a feature that is classifier scoped is rendered by underlining the feature's name. No adornment means that the feature is instance scoped.

Figure Owner Scope

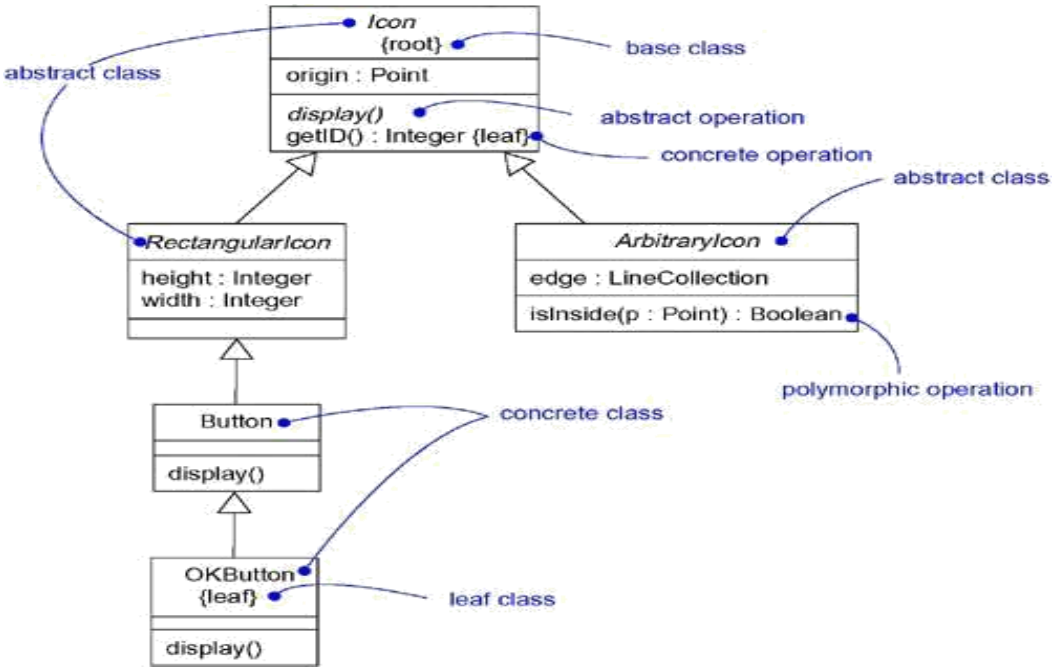


In general, most features of the classifiers you model will be instance scoped. The most common use of classifier scoped features is for private attributes that must be shared among a set of instances (and with the guarantee that no other instances have access to that attribute), such as for generating unique IDs among all instances of a given classifier, and for operations that create instances of the class.

Abstract, Root, Leaf, and Polymorphic Elements

You use generalization relationships to model a lattice of classes, with more-generalized abstractions at the top of the hierarchy and more-specific ones at the bottom. Within these hierarchies, it's common to specify that certain classes are abstract• meaning that they may not have any direct instances. In the UML, you specify that a class is abstract by writing its name in italics. For example, as Figure shows, ***Icon, RectangularIcon,*** and ***ArbitraryIcon*** are all abstract classes. By contrast, a concrete class (such as **Button** and **OKButton**) is one that may have direct instances.

Figure Abstract and Concrete Classes and Operations



Whenever you use a class, you'll probably want to inherit features from other, more-general, classes, and have other, more-specific, classes inherit features from it. These are the normal semantics you get from classes in the UML. However, you can also specify that a class may have no children. Such an element is called a leaf class and is specified in the UML by writing the property **leaf**below the class's name. For example, in the figure,**OKButton**is a leaf class, so itmay have no children.

Less common but still useful is the ability to specify that a class may have no parents. Such an element is called a root class, and is specified in the UML by writing the property **root** below the class's name. For example, in the figure, ***Icon*** is a root class. Especially when you have multiple, independent inheri-tance lattices, it's useful to designate the head of each hierarchy in this manner.

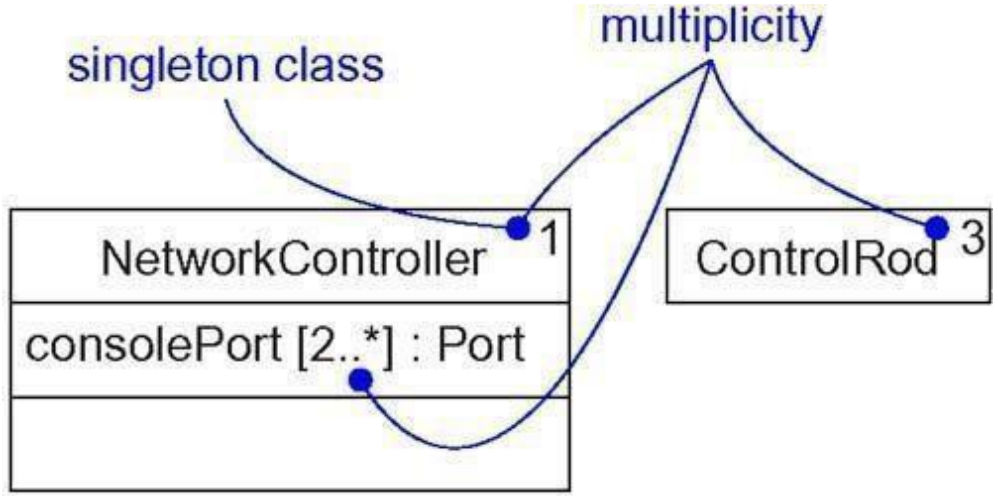
Operations have similar properties. Typically, an operation is polymorphic, which means that, in a hierarchy of classes, you can specify operations with the same signature at different points in the hierarchy. Ones in the child classes override the behavior of ones in the parent classes. When a message is dispatched at run time, the operation in the hierarchy that is invoked is chosen polymorphically• that is, a match is determined at run time according to the type of the object. For example, **display** and **isInside** are both polymorphic operations. Furthermore, the operation ***Icon::display()*** is abstract, meaning that it is incomplete and requires a child to supply an implementation of the operation. In the UML, you specify an abstract operation by writing its name in italics, just as you do for a class. By contrast, **Icon::getID()** is a leaf operation, so designated by the property **leaf.** This means that the operation is not polymorphic and may not be overridden.

Multiplicity

Whenever you use a class, it's reasonable to assume that there may be any number of instances of that class (unless, of course, it is an abstract class and so it may not have any direct instances, although there may be any number of instances of its concrete children). Sometimes, though, you'll want to restrict the number of instances a class may have. Most often, you'll want to specify zero instances (in which case, the class is a utility class that exposes only class scoped attributes and operations), one instance (a singleton class), a specific number of instances, or many instances (the default case).

The number of instances a class may have is called its multiplicity. Multiplicity is a specification of the range of allowable cardinalities an entity may assume. In the UML, you can specify the multiplicity of a class by writing a multiplicity expression in the upper-right corner of the class icon. For example, in Figure , **NetworkController** is a singleton class. Similarly, there are exactly three instances of the class **ControlRod** in the system.

Figure Multiplicity



Multiplicity applies to attributes, as well. You can specify the multiplicity of an attribute by writing a suitable expression in brackets just after the attribute name. For example, in the figure, there are two or more **consolePort** instances in the instance of **NetworkController.**

Attributes

At the most abstract level, when you model a class's structural features (that is, its attributes), you simply write each attribute's name. That's usually enough information for the average reader to understand the intent of your model.

In its full form, the syntax of an attribute in the UML is

[visibility] name [multiplicity] [: type] [= initial-value]
[{property-string}]

For example, the following are all legal attribute declarations:

origin	Name only
+ origin	Visibility and name
origin : Point	Name and type
head : *Item	Name and complex type
name [0..1] : String	Name, multiplicity, and type
origin : Point = (0,0)	Name, type, and initial value
id : Integer {frozen}	Name and property

There are three defined properties that you can use with attributes.

1. **changeable**	There are no restrictions on modifying the attribute's value.
2. **addOnly**	For attributes with a multiplicity greater than one, additional values may be added, but once created, a value may not be removed or altered.
3. **frozen**	The attribute's value may not be changed after the object is initialized.

Unless otherwise specified, attributes are always **changeable.** You'll mainly want to use **frozen** when modeling constant or write-once attributes.

Operations

At the most abstract level, when you model a class's behavioral features (that is, its operations and its signals), you will simply write each operation's name. That's usually enough information for the average reader to understand the intent of your model. As the previous sections have described, however, you can also specify the visibility and scope of each operation. There's still more: You can also specify the parameters, return type, concurrency semantics, and other properties of each operation. Collectively, the name of an operation plus its parameters (including its return type, if any) is called the operation's signature.

In its full form, the syntax of an operation in the UML is

[visibility] name [(parameter-list)] [: return-type]
[{property-string}]

For example, the following are all legal operation declarations:

display	Name only
+ display	Visibility and name
set(n : Name, s : String)	Name and parameters
getID() : Integer	Name and return type
restart() {guarded}	Name and property

In an operation's signature, you may provide zero or more parameters, each of which follows the syntax

[direction] name : type [= default-value]

Direction may be any of the following values:

in	An input parameter; may not be modified
out	An output parameter; may be modified to communicate information to the caller
inout	An input parameter; may be modified

In addition to the **leaf** property described earlier, there are four defined properties that you can use with operations.

1. **isQuery**	Execution of the operation leaves the state of the system unchanged. In other words, the operation is a pure function that has no side effects.
2. **sequential**	Callers must coordinate outside the object so that only one flow is in the object at a time. In the presence of multiple flows of control, the semantics and integrity of the object cannot be guaranteed.
3. **guarded**	The semantics and integrity of the object is guaranteed in the presence of multiple flows of control by sequentializing all calls to all of the object's guarded operations. In effect, exactly one operation at a time can be invoked on the object, reducing this to sequential semantics.
4. **concurrent**	The semantics and integrity of the object is guaranteed in the presence of multiple flows of control by treating the operation as atomic. Multiple calls from concurrent flows of control may occur simultaneously to one object on any concurrent operation, and all may proceed concurrently with correct semantics;

	concurrent operations must be designed so that they perform correctly in the case of a concurrent sequential or guarded operation on the same object.

The last three properties (**sequential, guarded, concurrent**) address the concurrency semantics of an operation, properties that are relevant only in the presence of active objects, processes, or threads.

Template Classes

A template is a parameterized element. In such languages as C++ and Ada, you can write template classes, each of which defines a family of classes (you can also write template functions, each of which defines a family of functions). A template includes slots for classes, objects, and values, and these slots serve as the template's parameters. You can't use a template directly; you have to instantiate it first. Instantiation involves binding these formal template parameters to actual ones. For a template class, the result is a concrete class that can be used just like any ordinary class.

For example, the following C++ code fragment declares a parameterized **Map** class.

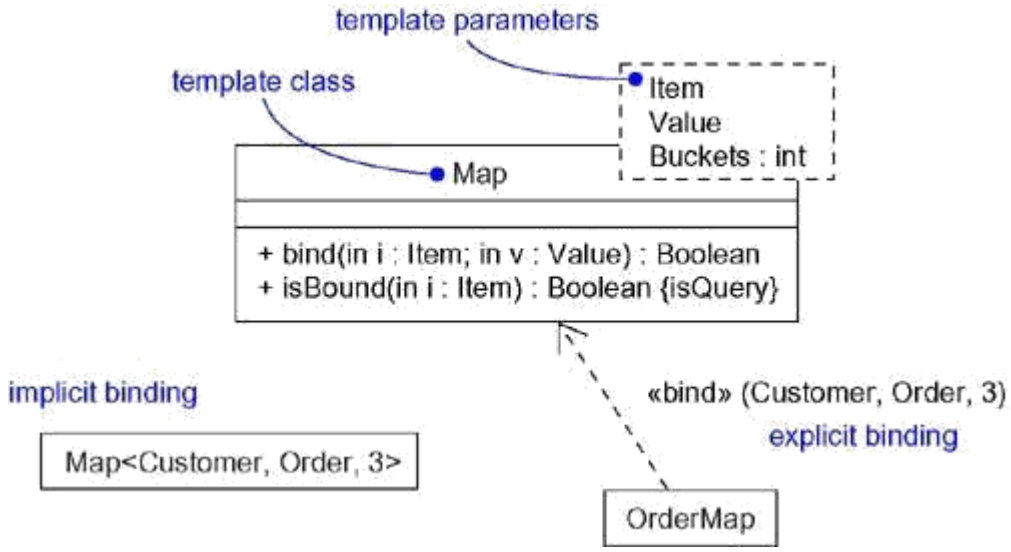
```
template<class Item, class Value, int Buckets> class Map {
public:
   virtual Boolean bind(const Item&, const Value&); virtual Boolean
   isBound(const Item&) const;
   ...
};
```

You might then instantiate this template to map **Customer** objects to **Order** objects.

```
m : Map<Customer, Order, 3>;
```

You can model template classes in the UML as well. As Figure shows, you render a template class just as you do an ordinary class, but with an additional dashed box in the upper-right corner of the class icon, which lists the template parameters.

Figure Template Classes



As the figure goes on to show, you can model the instantiation of a template class in two ways. First, you can do so implicitly, by declaring a class whose name provides the binding. Second, you can do so explicitly, by using a dependency stereotyped as **bind,** which specifies that the source instantiates the target template using the actual parameters.

Standard Elements

All of the UML's extensibility mechanisms apply to classes. Most often, you'll use tagged values to extend class properties (such as specifying the version of a class) and stereotypes to specify new kinds of components (such as model- specific components).

The UML defines four standard stereotypes that apply to classes.

1. **metaclass**	Specifies a classifier whose objects are all classes
2. **powertype**	Specifies a classifier whose objects are the children of a given parent
3. **stereotype**	Specifies that the classifier is a stereotype that may be applied to other elements
4. **utility**	Specifies a class whose attributes and operations are all class scoped

Common Modeling Techniques

Modeling the Semantics of a Class

The most common purpose for which you'll use classes is to model abstractions that are drawn from the problem you are trying to solve or from the technology you are using to implement a solution to that problem. Once you've identified those abstractions, the next thing you'll need to do is specify their semantics.

To model the semantics of a class, choose among the following possibilities, arranged from informal to formal.

- Specify the responsibilities of the class. A responsibility is a contract or obligation of a type or class and is rendered in a note (stereotyped as **responsibility**) attached to the class, or in an extra compartment in the class icon.
- Specify the semantics of the class as a whole using structured text, rendered in a note (stereotyped as **semantics**) attached to the class.
- Specify the body of each method using structured text or a programming language, rendered in a note attached to the operation by a dependency relationship.
- Specify the pre- and postconditions of each operation, plus the invariants of the class as a whole, using structured text. These elements are rendered in notes (stereotyped as **precondition, postcondition,** and **invariant**) attached to the operation or class by a dependency relationship.
- Specify a state machine for the class. A state machine is a behavior that specifies the sequences of states an object goes through during its lifetime in response to events, together with its responses to those events.
- Specify a collaboration that represents the class. A collaboration 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. A collaboration has a structural part, as well as a dynamic part, so you can use collaborations to specify all dimensions of a class's semantics.

- Specify the pre- and postconditions of each operation, plus the invariants of the class as a whole, using a formal language such as OCL.

Advanced Relationships

Terms and Concepts

A *relationship* is a connection among things. In object-oriented modeling, the four most important relationships are dependencies, generalizations, associattions, and realizations. Graphically, a relationship is rendered as a path, with different kinds of lines used to distinguish the different relationships.

Dependency

A *dependency* is a using relationship, specifying that a change in the specification of one thing (for example, class **SetTopController**) may affect another thing that uses it (for example, class **ChannelIterator**), but not necessarily the reverse. Graphically, a dependency is rendered as a dashed line, directed to the thing that is depended on. Apply dependencies when you want to show one thing using another.

A plain, unadorned dependency relationship is sufficient for most of the using relationships you'll encounter. However, if you want to specify a shade of meaning, the UML defines a number of stereotypes that may be applied to dependency relationships. There are 17 such stereotypes, all of which can be organized into six groups.

First, there are eight stereotypes that apply to dependency relationships among classes and objects in class diagrams.

1. **bind**	Specifies that the source instantiates the target template using the given actual **bind** parameters

You'll use **bind** when you want to model the details of template classes. For example, the relationship between a template container class and an instantiation of that class would be modeled as a **bind** dependency. **Bind** includes a list of actual arguments that map to the formal arguments of the template.

2. **derive**	Specifies that the source may be computed from the target

You'll use **derive** when you want to model the relationship between two attributes or two associations, one of which is concrete and the other is conceptual. For example, a **Person** class might have the attribute **BirthDate** (which is concrete), as well as the attribute **Age** (which can be derived from **BirthDate,** so is not separately manifest in the class). You'd show the relationship between **Age** and **BirthDate** by using a **derive** dependency, showing **Age** derived from **BirthDate.**

3. **friend**	Specifies that the source is given special visibility into the target

You'll use **friend** when you want to model relationships such as found with C++ friend classes.

4. **instanceOf**	Specifies that the source object is an instance of the target classifier
5. **instantiate**	Specifies that the source creates instances of the target

These last two stereotypes let you model class/object relationships explicitly. You'll use **instanceOf** when you want to model the relationship between a class and an object in thesame diagram, or between a class and its metaclass. You'll use **instantiate** when you want to specify which element creates objects of

another.

6. **powertype** whos	Specifies that the target is a powertype of the source; a powertype is a classifier objects are all the children of a given parent

You'll use **powertype** when you want to model classes that cover other classes, such as you'll find when modeling databases.

7. **refine** Speci	ies that the source is at a finer degree of abstraction than the target

You'll use **refine** when you want to model classes that are essentially the same but at different levels of abstraction. For example, during analysis, you might encounter a **Customer** class which, during design, you refine into a more detailed **Customer** class, complete with its implementation.

8. **use** pub	Specifies that the semantics of the source element depends on the semantics of the lic part of the target

You'll apply **use** when you want to explicitly mark a dependency as a using relationship, in contrast to the shades of dependencies other stereotypes provide.

Continuing, there are two stereotypes that apply to dependency relationships among packages.

1. target packa	Specifies that the source package is granted the right to reference the elements of the **access** e
2. **import**	A kind of access that specifies that the public contents of the target package enter the flat namespace of the source, as if they had been declared in the source

You'll use **access** and **import** when you want to model the relationships among packages. Between two peer packages, the elements in one cannot reference the elements in the other unless there's an explicit **access** or **import** dependency. For example, suppose a target package **T** contains the class **C.** If you specify an access dependency from **S**to **T**, then the elements of **S** can reference **C**, using the fully qualified name **T::C.** If you specify an import dependency from **S** to **T,** then the elements of **S** can reference **C** using just its simple name.

Two stereotypes apply to dependency relationships among use cases:

1. **extend** Spe	ifies that the target use case extends the behavior of the source
2. **include** use ca	Specifies that the source use case explicitly incorporates the behavior of another se at a location specified by the source

You'll use **extend** and **include** (and simple generalization) when you want to decompose use cases into reusable parts.

You'll encounter three stereotypes when modeling interactions among objects.

1. **become**	Specifies that the target is the same object as the source but at a later point in time and with possibly different values, state, or roles
2. **call**	Specifies that the source operation invokes the target operation
3. **copy**	Specifies that the target object is an exact, but independent, copy of the source

You'll use **become** and **copy** when you want to show the role, state, or attribute value of one object at different points in time or space. You'll use **call** when you want to model the calling dependencies among operations.

One stereotype you'll encounter in the context of state machines is

?**send**	Specifies that the source operation sends the target event

You'll use **send** when you want to model an operation (such as found in the action associated with a state transition) dispatching a given event to a target object (which in turn might have an associated state machine). The **send** dependency in effect lets you tie independent state machines together.

Finally, one stereotype that you'll encounter in the context of organizing the elements of your system into subsystems and models is

?**trace**	Specifies that the target is an historical ancestor of the source

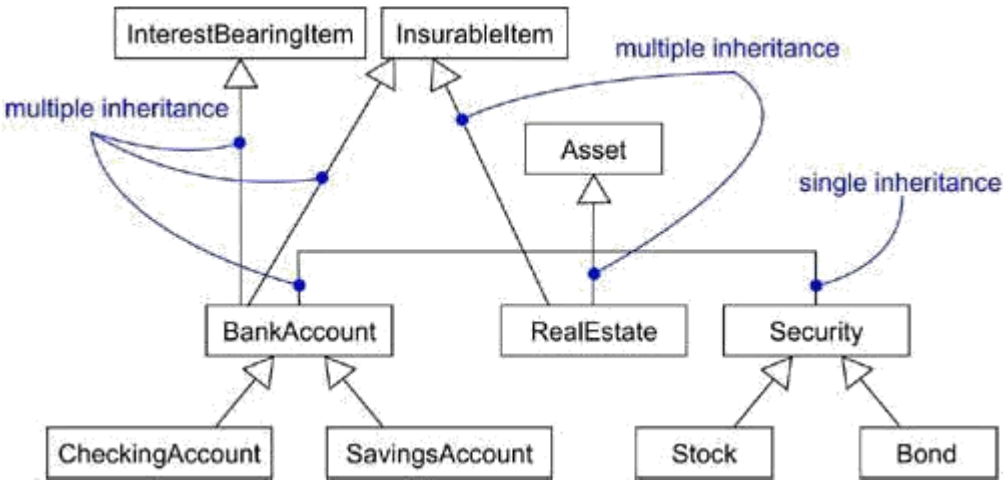
You'll use **trace** when you want to model the relationships among elements in different models. For example, in the context of a system's architecture, a use case in a use case model(representing a functional requirement) might trace to a package in the corresponding design model (representing the artifacts that realize that use case).

Generalization

A *generalization* is a relationship between a general thing (called the superclass or parent) and a more specific kind of that thing (called the subclass or child). For example, you might encounter the general class **Window** with its more specific kind, **MultiPaneWindow.** With a generalization relationship from the child to the parent, the child (**MultiPaneWindow**) will inherit all the structure and behavior of the parent (**Window**). The child may even add new structure and behavior, or it may modify the behavior of the parent. In a generalization relationship, instances of the child may be used anywhere instances of the parent apply• meaning that the child is substitutable for the parent.

Most of the time, you'll find single inheritance sufficient. A class that has exactly one parent is said to use single inheritance. There are times, however, when multiple inheritance is better, and you can model those relationships, as well, in the UML. For example, Figure shows a set of classes drawn from a financial services application. You see the class **Asset** with three children: **BankAccount, RealEstate,** and **Security.** Two of these children (**BankAccount** and **Security**) have their own children. For example, **Stock** and **Bond** are both children of**Security.**

Figure Multiple Inheritance



Two of these children (**BankAccount**and**RealEstate**) inherit from multiple parents.**RealEstate**, for example, is a kind of **Asset,** as well as a kind of **InsurableItem,** and**BankAccount** is a kind of **Asset,** as well as a kind of **InterestBearingItem** and an **InsurableItem.**

Parents, such as **InterestBearingItem** and **InsurableItem,** are called mixins because they don't stand alone but, rather, are intended to be mixed in with other parents (such as **Asset**) to form children from these various bits of structure and behavior.

A plain, unadorned generalization relationship is sufficient for most of the inheritance relationships you'll encounter. However, if you want to specify a shade of meaning, the UML defines one stereotype and four constraints that may be applied to generalization relationships.

First, there is the one stereotype.

?**implementation** Specifie	that the child inherits the implementation of the parent but does not make public nor support its interfaces, thereby violating substitutability

You'll use **implementation** when you want to model private inheritance, such as found in C++.

Next, there are four standard constraints that apply to generalization relationships.

1. **complete** Specifies	that all children in the generalization have been specified in the model (although some may be elided in the diagram) and that no additional children are permitted
2. **incomplete**	Specifies that not all children in the generalization have been specified (even if some are elided) and that additional children are permitted

Unless otherwise stated, you can assume that any diagram shows only a partial view of an inheritance lattice and so is elided. However, elision is different from the completeness of a model. Specifically, you'll use the **complete** constraint when you want to show explicitly that you've fully specified a hierarchy in the model (although no one diagram may show that hierarchy); you'll use **incomplete** to show explicitly that you have not stated the full specification of the hierarchy in the model (although one diagram may show everything in the model).

3. **disjoint**	Specifies that objects of the parent may have no more than one of the children as a type

4. **overlapping**	Specifies that objects of the parent may have more than one of the children as a type

These two constraints apply only in the context of multiple inheritance. You'll use **disjoint** and **overlapping** when you want to distinguish between static classification (**disjoint**) anddynamic classification (**overlapping**).

Association

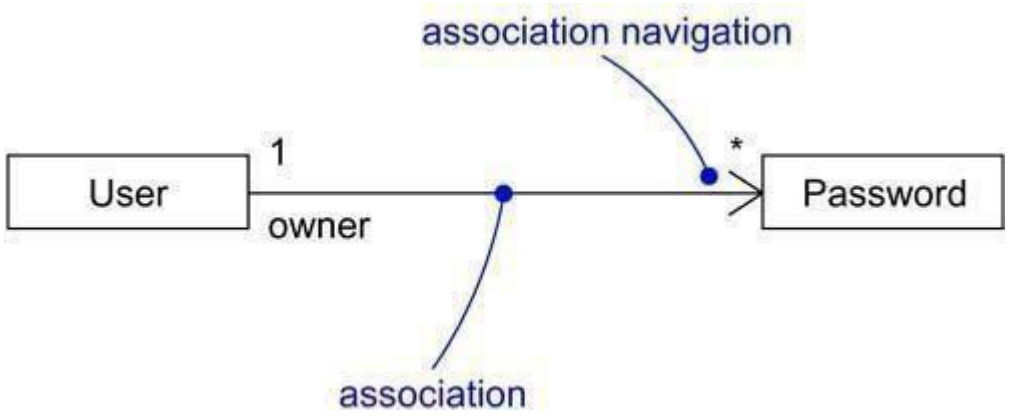
An *association* is a structural relationship, specifying that objects of one thing are connected to objects of another. For example, a **Library** class might have a one-to-many association to a **Book** class, indicating that each **Book** instance is owned by one **Library** instance. Furthermore,given a **Book**, you can find its owning **Library**, and given a **Library,** you can navigate to all its **Book** s. Graphically, an association is rendered as a solid line connecting the same or different classes. You use associations when you want to show structural relationships.

There are four basic adornments that apply to an association: a name, the role at each end of the association, the multiplicity at each end of the association, and aggregation. For advanced uses, there are a number of other properties you can use to model subtle details, such as navigation, qualification, and various flavors of aggregation.

Navigation

Given a plain, unadorned association between two classes, such as **Book** and **Library,** it's possible to navigate from objects of one kind to objects of the other kind. Unless otherwise specified, navigation across an association is bidirectional. However, there are some circumstances in which you'll want to limit navigation to just one direction. For example, asFigure 10-3shows, when modeling the services of an operating system, you'll find anassociation between **User** and **Password** objects. Given a **User,** you'll want to be able to find the corresponding **Password** objects; but given a **Password,** you don't want to be able to identify the corresponding **User.** You can explicitly represent the direction of navigation by adorning an association with an arrowhead pointing to the direction of traversal.

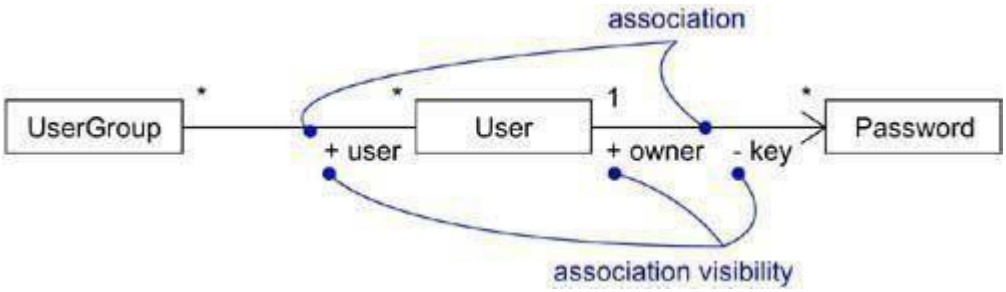
Figure Navigation



Visibility

Given an association between two classes, objects of one class can see and navigate to objects of the other, unless otherwise restricted by an explicit statement of navigation. However, there are circumstances in which you'll want to limit the visibility across that association relative to objects outside the association. For example, as Figure shows, there is an association between **UserGroup** and **User** and another between **User** and **Password.** Given a **User** object, it'spossible to identify its corresponding **Password** objects. However, a **Password** is private to a **User,** so it shouldn't be accessible from the outside (unless, of course, the **User** explicitlyexposes access to the **Password,** perhaps through some public operation). Therefore, as the figure shows, given a **UserGroup** object, you can navigate to its **User** objects (and vice versa), but you cannot in turn see the **User** object's **Password** objects; they are private to the **User.** In the UML, you can specify three levels of visibility for an association end, just as you can for a class's features by appending a visibility symbol to a role name. Unless otherwise noted, the visibility of a role is public. Private visibility indicates that objects at that end are not accessible to any objects outside the association; protected visibility indicates that objects at that end are not accessible to any objects outside the association, except for children of the other end.

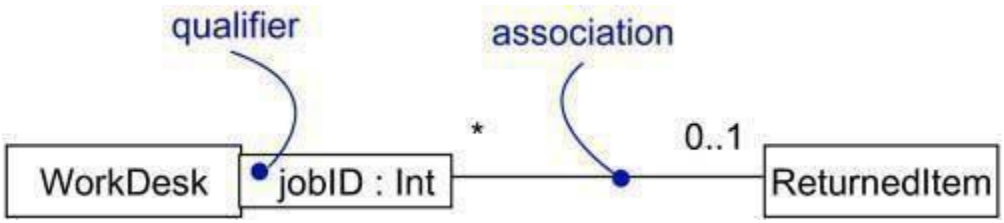
Figure Visibility



Qualification

In the context of an association, one of the most common modeling idioms you'll encounter is the problem of lookup. Given an object at one end of an association, how do you identify an object or set of objects at the other end? For example, consider the problem of modeling a work desk at a manufacturing site at which returned items are processed to be fixed. As Figure 10-5 shows, you'd model an association between two classes, **WorkDesk** and **ReturnedItem.** In the context of the **WorkDesk,** you'd have a **jobId** that would identify a particular **ReturnedItem.** In that sense, **jobId** is an attribute of the association. It's not a feature of **ReturnedItem** because items really have no knowledge of things like repairs or jobs. Then, given an object of **WorkDesk** and given a particular value for **jobId,** you can navigate to zero or one objects of **ReturnedItem.** In the UML, you'd model this idiom using a qualifier, which is an associationattribute whose values partition the set of objects related to an object across an association. You render a qualifier as a small rectangle attached to the end of an association, placing the attributes in the rectangle, as the figure shows. The source object, together with the values of the qualifier's attributes, yield a target object (if the target multiplicity is at most one) or a set of objects (if the target multiplicity is many).

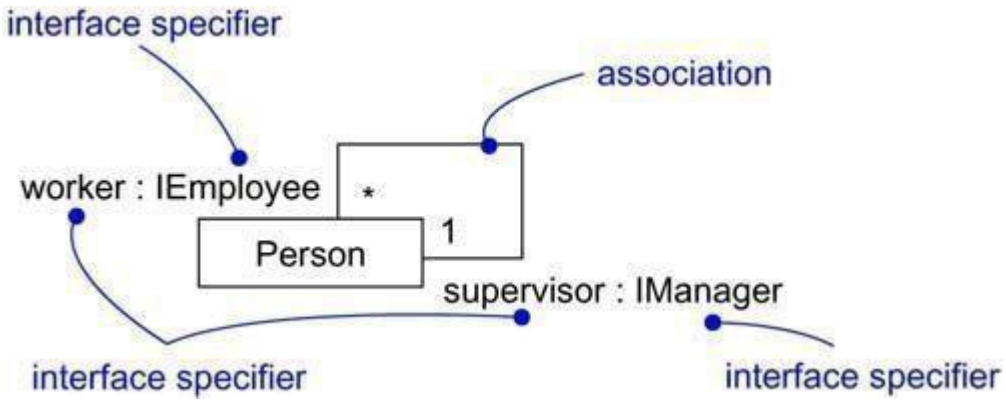
Figure Qualification



Interface Specifier

An interface is a collection of operations that are used to specify a service of a class or a component; every class may realize many interfaces. Collectively, the interfaces realized by a class represent a complete specification of the behavior of that class. However, in the context of an association with another target class, a source class may choose to present only part of its face to the world. For example, in the vocabulary of a human resources system, a **Person**class may realize many interfaces: **IManager, IEmployee, IOfficer,** and so on. As Figure shows, you can model the relationship between a supervisor and her workers with a one-to-many association, explicitly labeling the roles of this association as **supervisor** and **worker.** In the context of this association, a **Person** in the role of **supervisor** presents only the **IManager** face to the **worker;** a **Person** in the role of **worker** presents only the **IEmployee** face to the **supervisor.** As the figure shows, you can explicitly show the type of role using the syntax **rolename : iname,** where **iname** is some interface of the other classifier.

Figure Interface Specifiers



Composition

Aggregation turns out to be a simple concept with some fairly deep semantics. Simple aggregation is entirely conceptual and does nothing more than distinguish a "whole" from a "part." Simple aggregation does not change the meaning of navigation across the association between the whole and its parts, nor does it link the lifetimes of the whole and its parts.

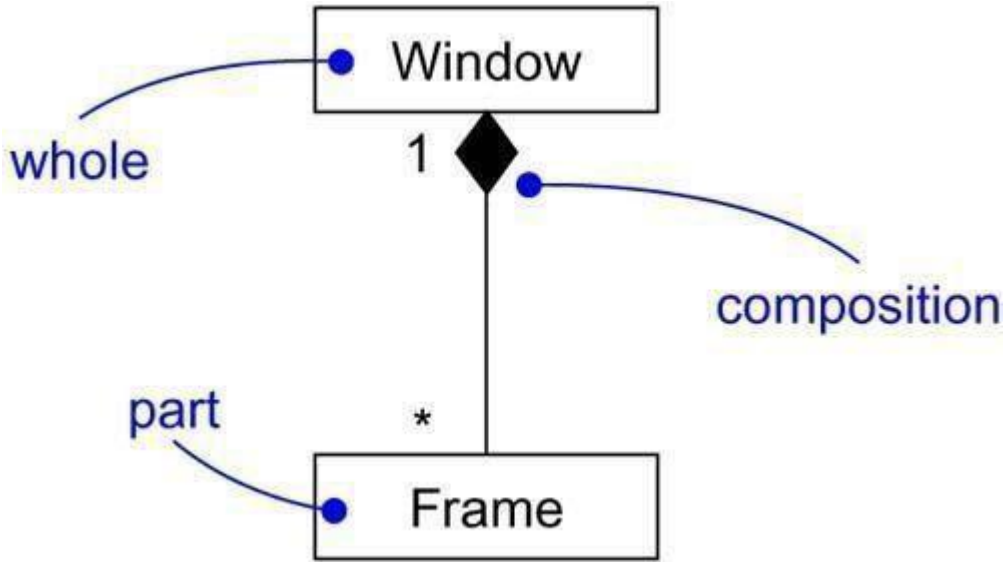
However, there is a variation of simple aggregation• composition• that does add some important semantics. Composition is a form of aggregation, with strong ownership and coincident lifetime as part of the whole. Parts with non-fixed multiplicity may be created after the composite itself, but once created they live and die with it. Such parts can also be explicitly removed before the death of the composite.

This means that, in a composite aggregation, an object may be a part of only one composite at a time. For example, in a windowing system, a **Frame** belongs to exactly one **Window.** This is in contrast to simple aggregation, in which a part may be shared by several wholes. For example, in the model of a house, a **Wall** may be a part of one or more **Room** objects.

In addition, in a composite aggregation, the whole is responsible for the disposition of its parts, which means that the composite must manage the creation and destruction of its parts. For example, when you create a **Frame** in a windowing system, you must attach it to an enclosing **Window.** Similarly, when you destroy the **Window,** the **Window** object must in turn destroy its **Frame** parts.

As Figure shows, composition is reallyjust a special kind of association and is specified byadorning a plain association with a filled diamond at the whole end.

Figure Composition

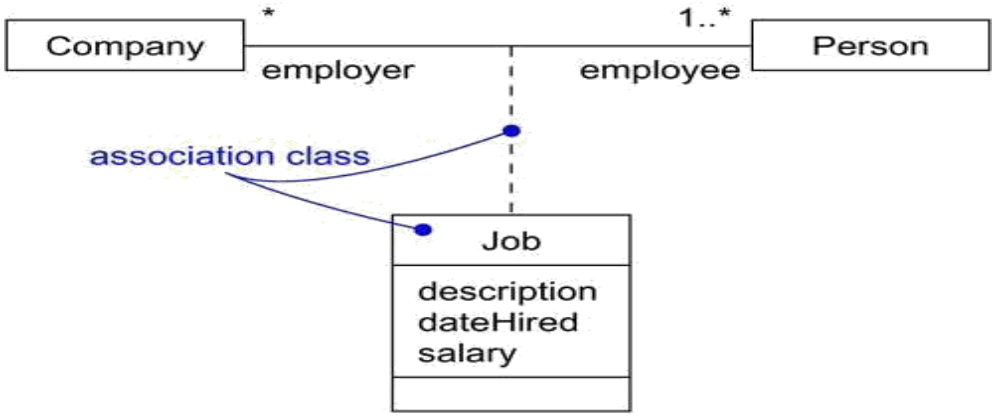


Association Classes

In an association between two classes, the association itself might have properties. For example, in an employer/employee relationship between a **Company** and a **Person,** there is a **Job** that represents the properties of that relationship that apply to exactly one pairing of the **Person** and **Company.** It wouldn't be appropriate to model this situation with a **Company** to **Job** associationtogether with a **Job** to **Person** association. That wouldn't tie a specific instance of the **Job** to the specific pairing of **Company** and **Person.**

In the UML, you'd model this as an association class, which is a modeling element that has both association and class properties. An association class can be seen as an association that also has class properties, or as a class that also has association properties. You render an association class as a class symbol attached by a dashed line to an association as in Figure.

Figure Association Classes



Constraints

These simple and advanced properties of associations are sufficient for most of the structural relationships you'll encounter. However, if you want to specify a shade of meaning, the UML defines five constraints that may be applied to association relationships.

First, you can distinguish if the association is real or conceptual.

1. **implicit** Specifi	s that the relationship is not manifest but, rather, is only conceptual

For example, if you have an association between two base classes, you can specify that same association between two children of those base classes (because they inherit the relationships of the parent classes). You'd mark it as **implicit,** because it's not manifest separately but, rather, is implicit from the relationship that exists between the parent classes.

Second, you can specify that the objects at one end of an association (with a multiplicity greater than one) are ordered or unordered.

2. **ordered** Specifies that the set of objects at one end of an association are in an explicit order	

For example, in a **User**/**Password** association, the **Password**s associated with the **User** might be kept in a least-recently used order, and would be marked as **ordered.**

Next, there are three properties, defined using constraint notation, that relate to the changeability of the instances of an association.

Finally, there are three defined constraints that relate to the changeability of the instances of an association.

3. **changeable**	Links between objects may be added, removed, and changed freely
4. **addOnly**	New links may be added from an object on the opposite end of the association
5. **frozen**	A link, once added from an object on the opposite end of the association, may not be modified or deleted

Finally, there is one constraint for managing related sets of associations:

6. Specifies that, over a set of associations, exactly one is manfest for each associated **xor** object

Realization

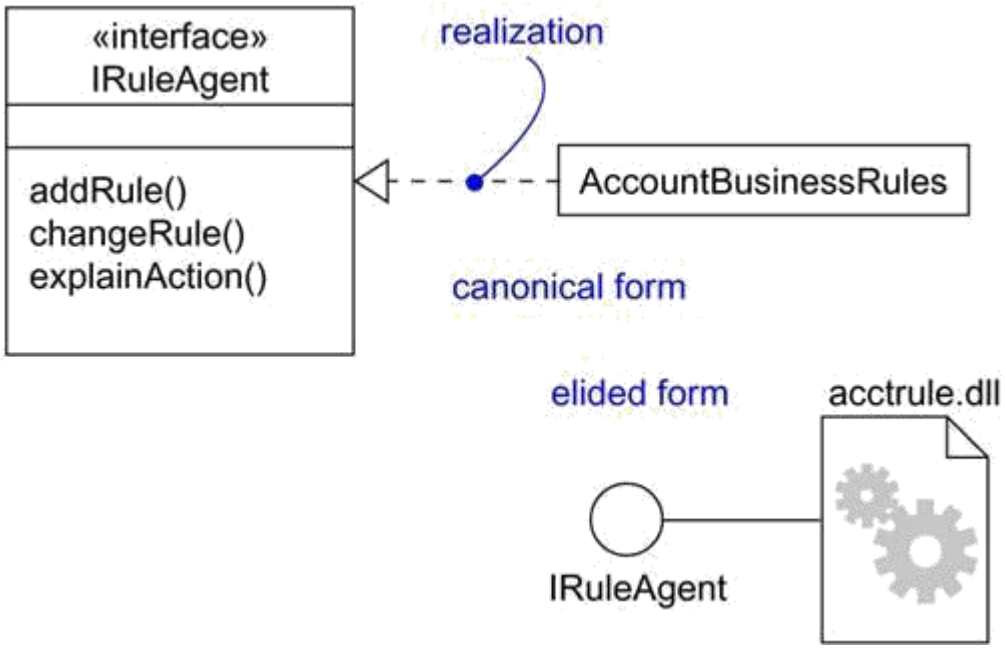
A *realization*is a semantic relationship between classifiers in which one classifier specifies a contract that another classifier guarantees to carry out. Graphically, a realization is rendered as a dashed directed line with a large open arrowhead pointing to the classifier that specifies the contract.

Realization is sufficiently different from dependency, generalization, and association relationships that it is treated as a separate kind of relationship. Semantically, realization is somewhat of a cross between dependency and generalization, and its notation is a combination of the notation for dependency and generalization. You'll use realization in two circumstances: in the context of interfaces and in the context of collaborations.

Most of the time, you'll use realization to specify the relationship between an interface and the class or component that provides an operation or service for it. An interface is a collection of operations that are used to specify a service of a class or a component. Therefore, an interface specifies a contract that a class or component must carry out. An interface may be realized by many such classes or components, and a class or component may realize many interfaces. Perhaps the most interesting thing about

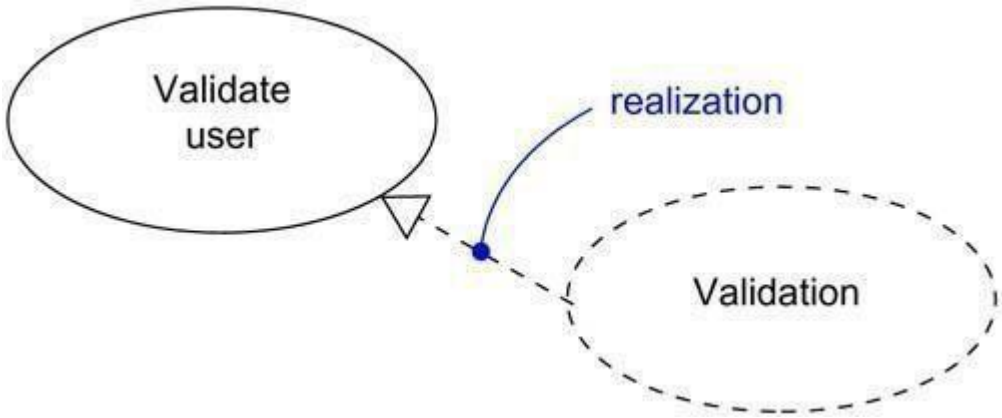
interfaces is that they let you separate the specification of a contract (the interface itself) from its implementation (by a class or a component). Furthermore, interfaces span the logical and physical parts of a system's architecture. For example, as Figure 10-9 shows, a class (such as **AccountBusinessRules** in an order entry system) in a system's design view might realize a given interface (such as **IRuleAgent**). That same interface (**IRuleAgent**) might also be realized by a component (such as **acctrule.dll**) in the system's implementation view. Note that you can represent realization in two ways: in the canonical form (using the **interface** stereotype and the dashed directed line with a large open arrowhead) and in an elided form (using the interface lollipop notation).

Figure Realization of an Interface



You'll also use realization to specify the relationship between a use case and the collaboration that realizes that use case, as Figure shows. In this circumstance, you'll almost always use the canonical form of realization.

Figure Realization of a Use Case



Modeling Webs of Relationships

When you model these webs of relationships,

- Don't begin in isolation. Apply use cases and scenarios to drive your discovery of the relationships among a set of abstractions.
- In general, start by modeling the structural relationships that are present. These reflect the static view of the system and are therefore fairly tangible.
- Next, identify opportunities for generalization/specialization relationships; use multiple inheritance sparingly.
- Only after completing the preceding steps should you look for dependencies; they generally represent more-subtle forms of semantic connection.
- For each kind of relationship, start with its basic form and apply advanced features only as absolutely necessary to express your intent.
- Remember that it is both undesirable and unnecessary to model all relationships among a set of abstractions in a single diagram or view. Rather, build up your system's relationships by considering different views on the system. Highlight interesting sets of relationships in individual diagrams.

Interfaces, Types, and Roles

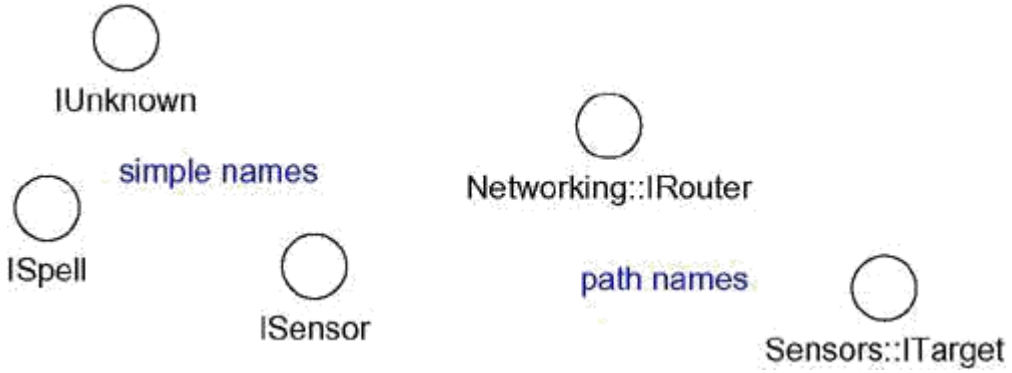
Terms and Concepts

An *interface* is a collection of operations that are used to specify a service of a class or a component. A *type* is a stereotype of a class used to specify a domain of objects, together with the operations (but not the methods) applicable to the object. A *role* is the behavior of an entity participating in a particular context. Graphically, an interface is rendered as a circle; in its expanded form, an interface may be rendered as a stereotyped class in order to expose its operations and other properties.

Names

Every interface must have a name that distinguishes it from other interfaces. A *name* is a textual string. That name alone is known as a *simple name;* a *path name* is the interface name prefixed by the name of the package in which that interface lives. An interface may be drawn showing only its name, as in Figure

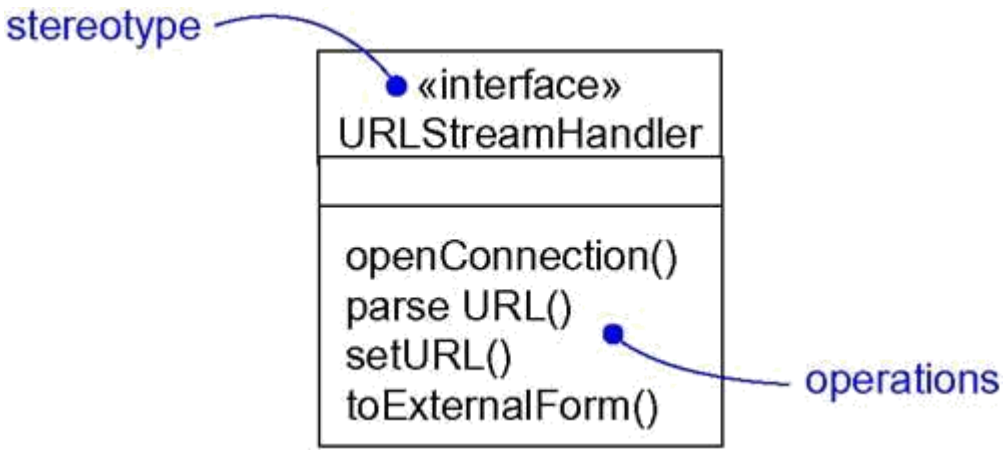
Figure Simple and Path Names



Operations

An interface is a named collection of operations used to specify a service of a class or of a component. Unlike classes or types, interfaces do not specify any structure (so they may not include any attributes), nor do they specify any implementation (so they may not include any methods, which provide the implementation of an operation). Like a class, an interface may have any number of operations. These operations may be adorned with visibility properties, concurrency properties, stereotypes, tagged values, and constraints.

Figure Operations



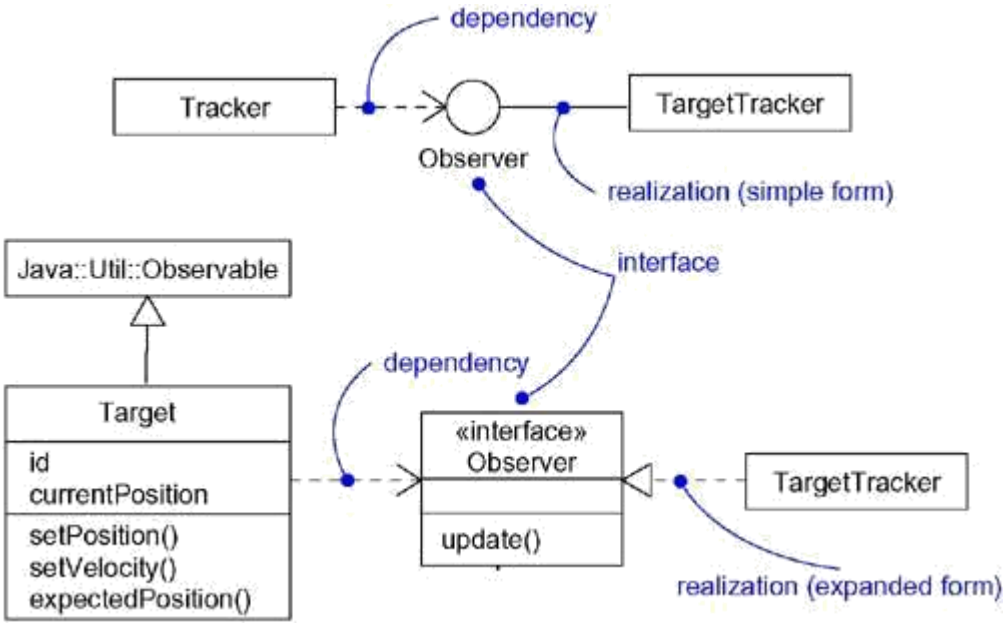
Relationships

Like a class, an interface may participate in generalization, association, and dependency relationships. In addition, an interface may participate in realization relationships. Realization is a semantic relationship between two classifiers in which one classifier specifies a contract that another classifier guarantees to carry out.

An interface specifies a contract for a class or a component without dictating its implementation. A class or component may realize many interfaces. In so doing, it commits to carry out all these contracts faithfully, which means that it provides a set of methods that properly implement the operations defined in the interface. Similarly, a class or a component may depend on many interfaces. In so doing, it expects that these contracts will be honored by some set of components that realize them. This is why we say that an interface represents a seam in a system. An interface specifies a contract, and the client and the supplier on each side of the contract may change independently, as long as each fulfills its obligations to the contract.

As Figure illustrates, you can show that an element realizes an interface in two ways. First, you can use the simple form in which the interface and its realization relationship are rendered as a lollipop sticking off to one side of a class or component. This form is useful when you simply want to expose the seams in your system. However, the limitation of this style is that you can't directly visualize the operations or signals provided by the interface. Second, you can use the expanded form in which you render an interface as a stereotyped class, which allows you to visualize its operations and other properties, and then draw a realization relationship from the classifier or component to the interface. In the UML, a realization relationship is rendered as a dashed directed line with a large open arrowhead pointing to the interface. This notation is a cross between generalization and dependency.

Figure Realizations



In both cases, you attach the class or component that builds on an interface with a dependency relationship from the element to the interface.

Understanding an Interface

When you are handed an interface, the first thing you'll see is a set of operations that specify a service of a class or a component. Look a little deeper and you'll see the full signature of those operations, along with any of their special properties, such as visibility, scope, and concurrency semantics.

These properties are important, but for complex interfaces, they aren't enough to help you understand the semantics of the service they represent, much less know how to use those operations properly. In the absence of any other information, you'd have to dive into some abstraction that realizes the interface to figure out what each operation does and how those operations are meant to work together. However, that defeats the purpose of an interface, which is to provide a clear separation of concerns in a system.
In the UML, you can supply much more information to an interface in order to make it understandable and approachable. First, you may attach pre- and postconditions to each operation and invariants to the class or component as a whole. By doing this, a client who needs to use an interface will be able to understand what the interface does and how to use it, without having to dive into an implementation. If you need to be rigorous, you can use the UML's OCL to formally specify the semantics. Second, you can attach a state machine to the interface. You can use this state machine to specify the legal partial ordering of an interface's operations. Third, you can attach collaborations to the interface. You can use collaborations to specify the expected behavior of the interface through a series of interaction diagrams.

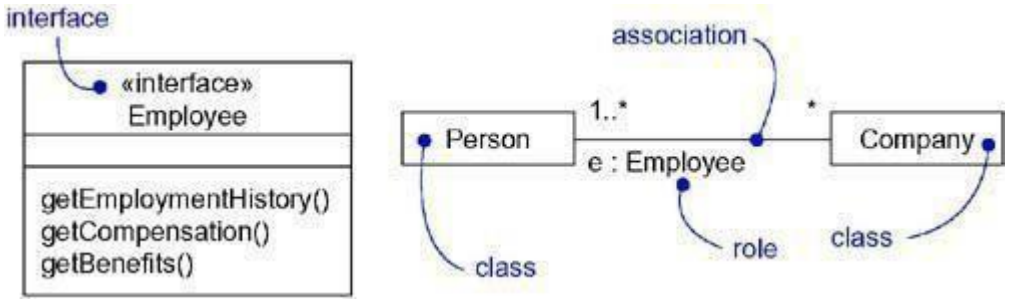
Types and Roles

A class may realize many interfaces. An instance of that class must therefore support all those interfaces, because an interface defines a contract, and any abstraction that conforms to that interface must, by definition, faithfully carry out that contract. Nonetheless, in a given context, an instance may present only one or more of its interfaces as relevant. In that case, each interface represents a role that the object plays. A role names a behavior of an entity participating in a particular context. Stated another way, a role is the face that an abstraction presents to the world.

For example, consider an instance of the class **Person.** Depending on the context, that **Person** instance may play the role of **Mother, Comforter, PayerOfBills, Employee, Customer,Manager, Pilot, Singer,** and so on. When an object plays a particular role, it presents aface to the world, and clients that interact with it expect a certain behavior depending on the role that it plays at the time. For example, an instance of **Person** in the role of **Manager** would present a different set of properties than if the instance were playing the role of **Mother.**

In the UML, you can specify a role an abstraction presents to another abstraction by adorning the name of an association end with a specific interface. For example, Figure shows the interface **Employee,** whose definition includes three operations. There exists an association between the classes **Person** and **Company** in which context **Person** plays the role **e,** whose type is **Employee.** In a different association, the **Person** might present an entirely different face to the world. With this explicit type, the role the **Person** plays is more than just a name meaningful to the human reader of this diagram. In the UML, this means that the **Person** presents the role of **Employee** to the **Company,** and in that context, only the properties specified by **Employee** are visible and relevant to the **Company.**

Figure Roles



A class diagram like this one is useful for modeling the static binding of an abstraction to its interface. You can model the dynamic binding of an abstraction to its interface by using the **become** stereotype in an interaction diagram, showing an object changing from one role toanother.

If you want to formally model the semantics of an abstraction and its conformance to a specific interface, you'll want to use the defined stereotype **type. Type** is a stereotype of class, and you use it to specify a domain of objects, together with the operations (but not the methods) applicable to the objects of that type. The concept of type is closely related to that of interface, except that a type's definition may include attributes while an interface may not. If you want to show that an abstraction is statically typed, you'll want to use **implementationClass,**a stereotype of class that specifies a class whose instances are statically typed (unlike **Person** in the example above) and that defines the physical data structure and methods of an object as implemented in traditional programming languages.

Common Modeling Techniques

Modeling the Seams in a Systemeling the Seams in a System

The most common purpose for which you'll use interfaces is to model the seams in a system composed of software components, such as COM+ or Java Beans. You'll reuse some components from other systems or buy off the shelf; you will create others from scratch. In any case, you'll need to write glue code that weaves these components together. That requires you to understand the interfaces provided and relied on by each component.

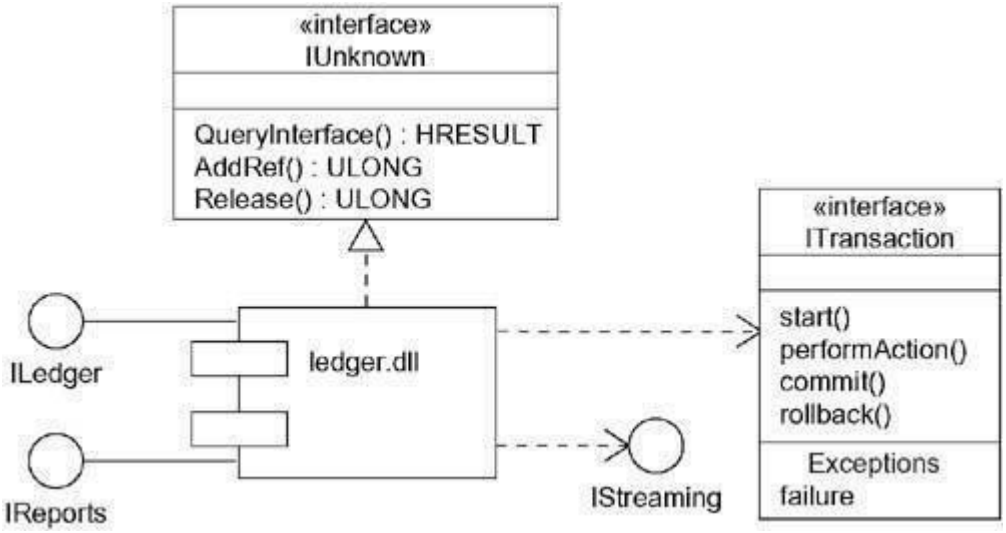
Identifying the seams in a system involves identifying clear lines of demarcation in your architecture. On either side of those lines, you'll find components that may change independently, without affecting the components on the other side, as long as the components on both sides conform to the contract specified by that interface.

To model the seams in a system,

- Within the collection of classes and components in your system, draw a line around those that tend to be tightly coupled relative to other sets of classes and components.
- Refine your grouping by considering the impact of change. Classes or components that tend to change together should be grouped together as collaborations.
- Consider the operations and the signals that cross these boundaries, from instances of one set of classes or components to instances of other sets of classes and components.
- Package logically related sets of these operations and signals as interfaces.
- For each such collaboration in your system, identify the interfaces it relies on (imports) and those it provides to others (exports). You model the importing of interfaces by dependency relationships, and you model the exporting of interfaces by realization relationships.
- For each such interface in your system, document its dynamics by using pre- and postconditions for each operation, and use cases and state machines for the interface as a whole.

For example, Figure 11-6 shows the seams surrounding a component (the library **ledger.dll**) drawn from a financial system. This component realizes three interfaces: **IUnknown, ILedger,** and **IReports.** In this diagram, **IUnknown** is shown in its expanded form; the other two are shown in their simple form, as lollipops. These three interfaces are realized by **ledger.dll** and are exported to other components for them to build on.

Figure Modeling the Seams in a System



As this diagram also shows, **ledger.dll**imports two interfaces,**IStreaming**and**ITransaction,** the latter of which is shown in its expanded form. These two interfaces arerequired by the **ledger.dll** component for its proper operation. Therefore, in a running system, you must supply components that realize these two interfaces. By identifying interfaces such as **ITransaction,** you've effectively decoupled the components on either side of the interface,permitting you to employ any component that conforms to that interface.

Interfaces such as **ITransaction** are more than just a pile of operations. This particular interface makes some assumptions about the order in which its operations should be called. Although not shown here, you could attach use cases to this interface and enumerate the common ways you'd use it.

Modeling Static and Dynamic Typeseling Static and Dynamic Types

Most object -oriented programming languages are statically typed, which means that the type of an object is bound at the time the object is created. Even so, that object will likely play different roles over time. This means that clients that use that object interact with the object through different sets of interfaces, representing interesting, possibly overlapping, sets of operations.

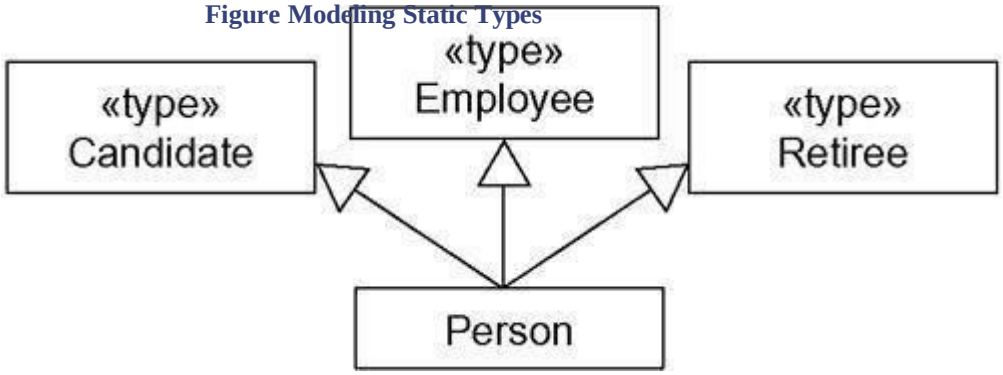
Modeling the static nature of an object can be visualized in a class diagram. However, when you are modeling things like business objects, which naturally change their roles throughout a workflow, it's sometimes useful to explicitly model the dynamic nature of that object's type. In these circumstances, an object can gain and lose types during its life.

To model a dynamic type,

- Specify the different possible types of that object by rendering each type as a class stereotyped as **type** (if the abstraction requires structure and behavior) or as **interface** (if the abstraction requires only behavior).
- Model all the roles the class of the object may take on at any point in time. You can do so in two ways:
    1. First, in a class diagram, explicitly type each role that the class plays in its association with other classes. Doing this specifies the face instances of that class put on in the context of the associated object.
    2. Second, also in a class diagram, specify the class-to-type relationships using generalization.
- In an interaction diagram, properly render each instance of the dynamically typed class. Display the role of the instance in brackets below the object's name.
- To show the change in role of an object, render the object once for each role it plays in the interaction, and connect these objects with a message stereotyped as **become.**

For example, Figure shows the roles that instances of the class**Person**might play in thecontext of a human resources system.

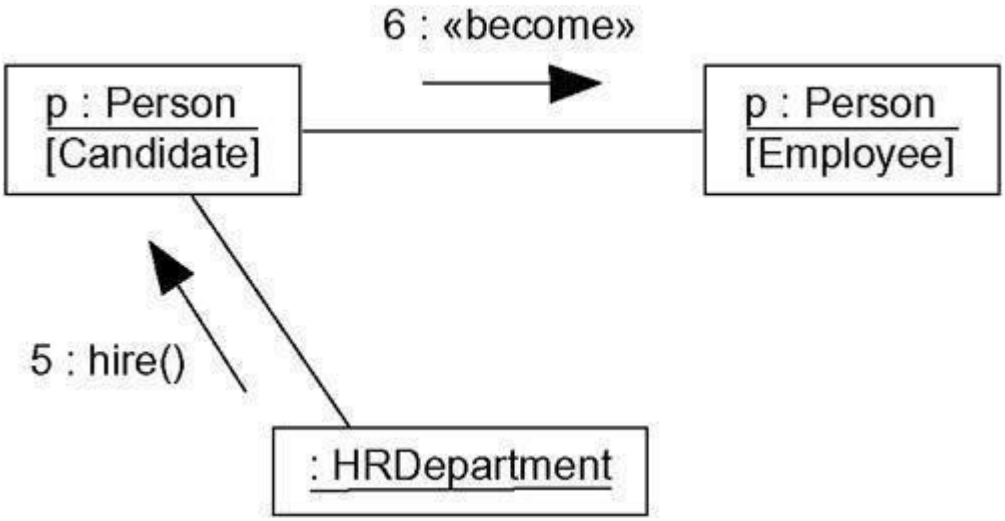
Figure Modeling Static Types



This diagram specifies that instances of the **Person** class may be any of the three types• namely, **Candidate, Employee,** or **Retiree.**

Figure shows the dynamic nature of a person's type. In this fragment of an interactiondiagram, **p** (the **Person** object) changes its role from **Candidate** to **Employee.**

Figure Modeling Dynamic Types



Packages

Terms and Concepts

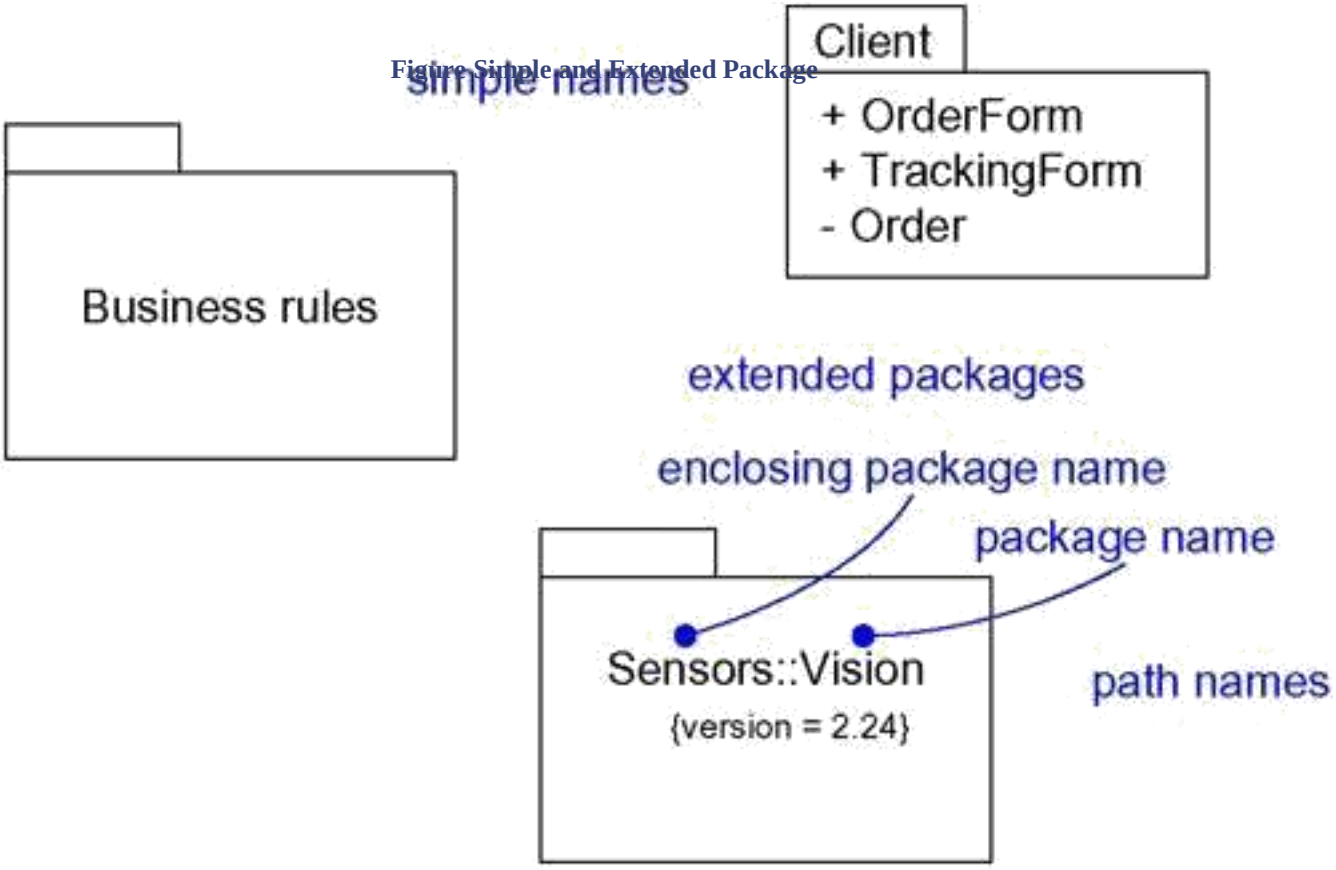
A *package* is a general- purpose mechanism for organizing elements into groups. Graphically, a package is rendered as a tabbed folder.

Names

A package name must be unique within its enclosing package.

Every package must have a name that distinguishes it from other packages. A *name* is a textual string. That name alone is known as a *simple name;* a *path name* is the package name prefixed by the name of the package in which that package lives, if any. A package is typically drawn showing only its name, as in Figure. Just as with classes, you may draw packages adorned with tagged values or with additional compartments to expose their details.

Figure Simple and Extended Package



Owned Elements

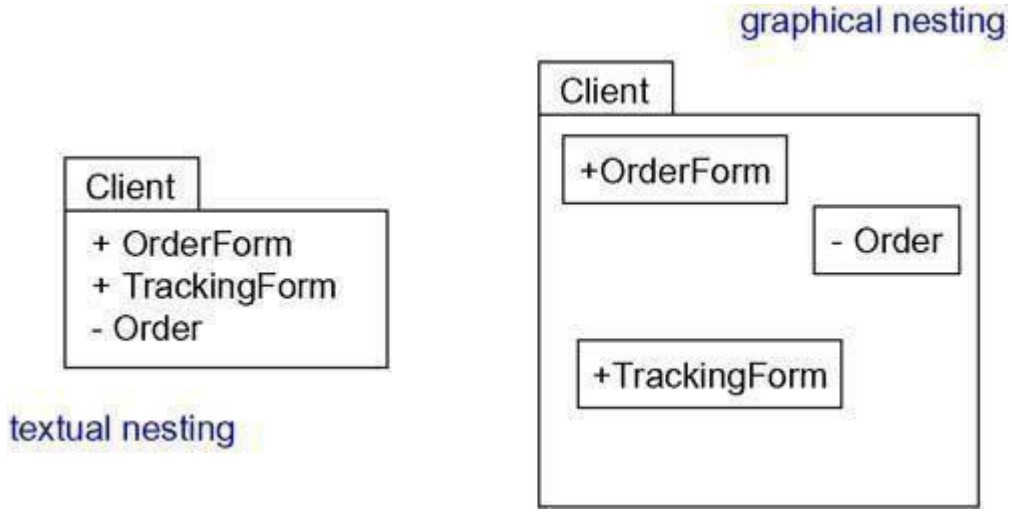
A package may own other elements, including classes, interfaces, components, nodes, collaborations, use cases, diagrams, and even other packages. Owning is a composite relationship, which means that the element is declared in the package. If the package is destroyed, the element is destroyed. Every element is uniquely owned by exactly one package.

A package forms a namespace, which means that elements of the same kind must be named uniquely within the context of its enclosing package. For example, you can't have two classes named **Queue** owned by the same package, but you can have a class named **Queue** in package **P1** and another (and different) class named **Queue** in package **P2.** The classes **P1::Queue** and **P2::Queue** are, in fact, different classes and can be distinguished by their path names. Differentkinds of elements may have the same name.
Elements of different kinds may have the same name within a package. Thus, you can have a class named **Timer,**as well as a component named**Timer,**within the same package. Inpractice, however, to avoid confusion, it's best to name elements uniquely for all kinds within a package.

Packages may own other packages. This means that it's possible to decompose your models hierarchically. For example, you might have a class named **Camera** that lives in the package **Vision** that in turn lives in the package **Sensors.** The full name of this class is **Sensors::Vision::Camera.** In practice, it's best to avoid deeply nested packages. Two tothree levels of nesting is about the limit that's manageable. More than nesting, you'll use importing to organize your packages.

As Figure shows, you can explicitly show the contents of a package either textually or graphically. Note that when you show these owned elements, you place the name of the package in the tab. In practice, you typically won't want to show the contents of packages this way. Instead, you'll use tools to zoom into the contents of a package.

Figure Owned Elements



Visibility

You can control the visibility of the elements owned by a package just as you can control the visibility of the attributes and operations owned by a class. Typically, an element owned by a package is public, which means that it is visible to the contents of any package that imports the element's enclosing package. Conversely, protected elements can only be seen by children, and private elements cannot be seen outside the package in which they are declared. In Figure ,**OrderForm**is a public part of the package**Client,**and**Order**is a private part. A packagethat imports **Client** can see **OrderForm,** but it cannot see **Order.** As viewed from the outside, the fully qualified name of **OrderForm** would be **Client::OrderForm.**

You specify the visibility of an element owned by a package by prefixing the element's name with an appropriate visibility symbol. Public elements are rendered by prefixing their name with a + symbol, as for **OrderForm** in Figure. Collectively, the public parts of a package constitute the package's interface.

Just as with classes, you can designate an element as protected or private, rendered by prefixing the element's name with a # symbol and a - symbol, respectively. Protected elements are visible only to packages that inherit from another package; private elements are not visible outside the package at all.

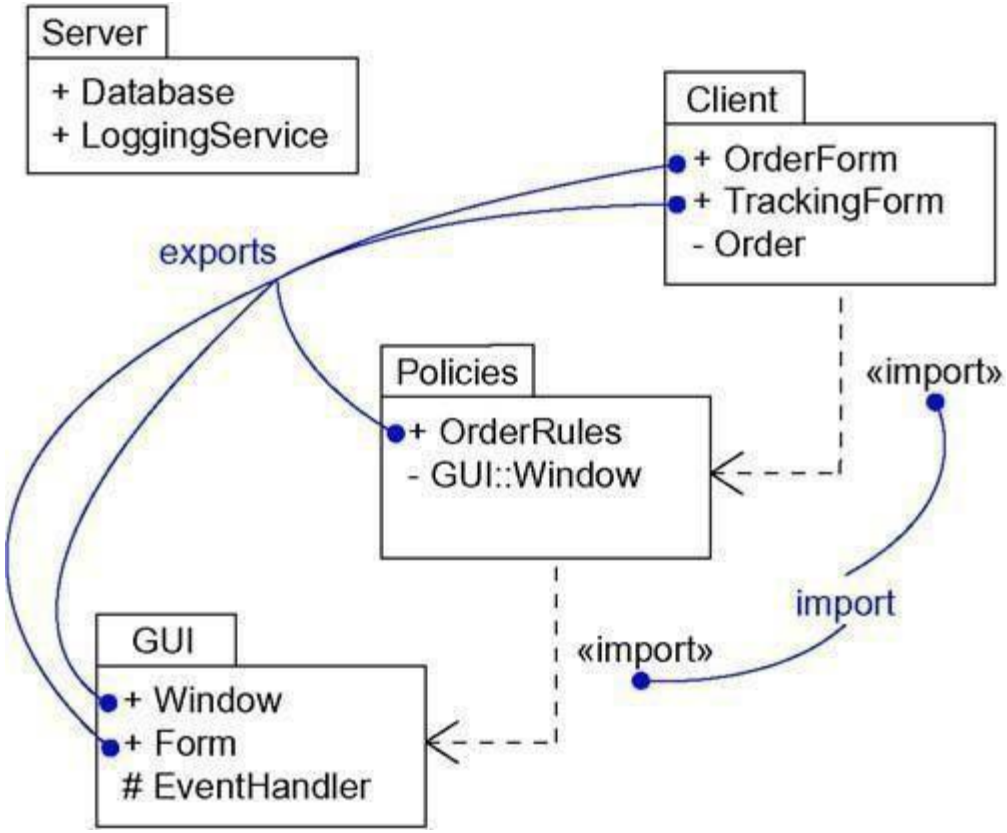
Importing and Exporting

Suppose you have two classes named **A** and **B** sitting side by side. Because they are peers, **A** can see **B** and **B** can see **A,** so both can depend on the other. Just two classes makes for a trivial system, so you really don't need any kind of packaging.

So suppose that instead you put **A** in one package and **B** in another package, both packages sitting side by side. Suppose also that **A** and **B** are both declared as public parts of their respective packages. This is a very different situation. Although **A** and **B** are both public, neither can access the other because their enclosing packages form an opaque wall. However, if **A**'s package imports **B**'s package, **A** can now see **B,** although **B** cannot see **A.** Importing grants a one-way permission for the elements in one package to access the elements in another package. In the UML, you model an import relationship as a dependency adorned with the stereotype **import.** By packaging your abstractions into meaningful chunks and then controlling theiraccess by importing, you can control the complexity of large numbers ofabstractions.

The public parts of a package are called its exports. For example, in Figure, the package **GUI** exports two classes, **Window** and **Form. EventHandler** is not exported by **GUI**; **EventHandler** is a protected part of the package.

Figure Importing and Exporting



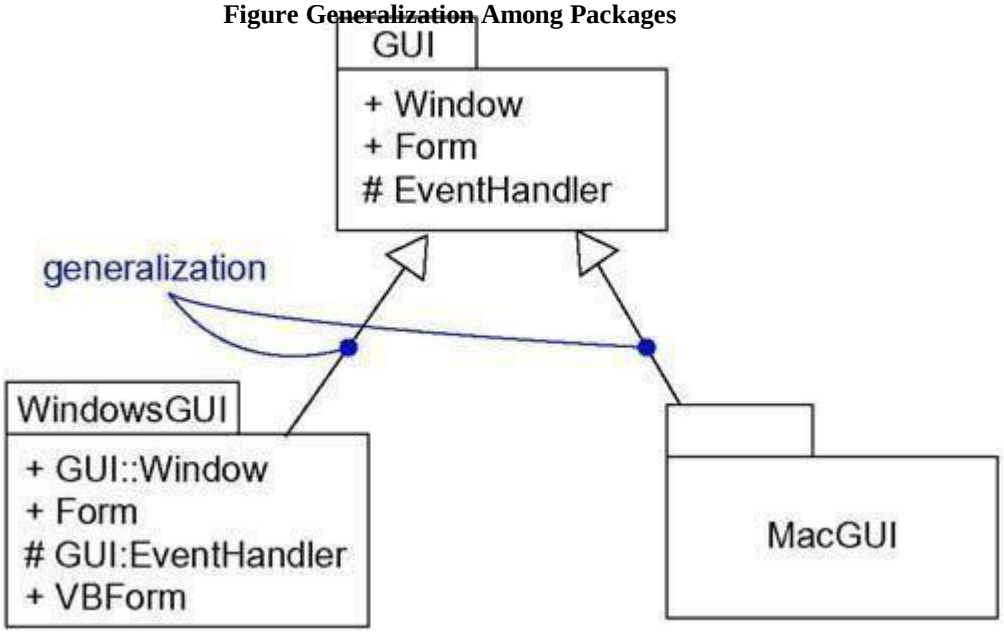
The parts that one package exports are visible only to the contents of those packages that explicitly import the package. In this example, **Policies** explicitly imports the package **GUI.GUI::Window** and **GUI::Form** are therefore made visible to the contents of the package **Policies.** However, **GUI::EventHandler** is not visible because it is protected. Because thepackage **Server** doesn't import **GUI,** the contents of **Server** don't have permission to access any of the contents of **GUI.** Similarly, the contents of **GUI** don't have permission to access any of the contents of **Server.**Import and access dependencies are not transitive. In this example, **Client**imports**Policies**and **Policies** imports **GUI,** but **Client** does not by implication import **GUI.** Therefore, the contents of **Client** have access to the exports of **Policies,** but they do not have access to the exports of **GUI.** To gain access, **Client** would have to import **GUI** explicitly.

Generalization

There are two kinds of relationships you can have between packages: import and access dependencies, used to import into one package elements exported from another, and generalizations, used to specify families of packages.

Generalization among packages is very much like generalization among classes. For example, in Figure , the package**GUI**is shown to export two classes (**Window**and**Form**) and oneprotected class (**EventHandler**). Two packages specialize the more general package **GUI**: **WindowsGUI** and **MacGUI.** These specialized packages inherit the public and protectedelements of the more general package. But, just as in class inheritance, packages can replace more general elements and add new ones. For example, the package **WindowsGUI** inherits from **GUI,** so it includes the classes **GUI::Window** and **GUI::EventHandler.** In addition, **WindowsGUI** overrides one class (**Form**) and adds a new one (**VBForm**).

Figure Generalization Among Packages



Packages involved in generalization relationships follow the same principle of substitutability as do classes. A specialized package (such as **WindowsGUI**) can be used anywhere a more general package (such as **GUI**) is used.

Standard Elements

All of the UML's extensibility mechanisms apply to packages. Most often, you'll use tagged values to add new package properties (such as specifying the author of a package) and stereotypes to specify new kinds of packages (such as packages that encapsulate operating system services).

The UML defines five standard stereotypes that apply to packages.

1. **facade**	Specifies a package that is only a view on some other package
2. **framework**	Specifies a package consisting mainly of patterns
3. **stub**	Specifies a package that serves as a proxy for the public contents of another

	package
4. **subsystem**	Specifies a package representing an independent part of the entire system being modeled
5. **system**	Specifies a package representing the entire system being modeled

The UML does not specify icons for any of these stereotypes. In addition to these five package stereotypes, you'll also use dependencies designated using the standard stereotype import.

Most of these standard elements are discussed elsewhere, except for facade and stub. These two stereotypes help you to manage very large models. You use facades to provide elided views on otherwise complex packages. For example, your system might contain the package **BusinessModel.** You might want to expose a subset of its elements to one set of users (toshow only those elements associated with customers), and another subset to a different set of users (to show only those elements associated with products). To do so, you would define facades, which import (and never own) only a subset of the elements in another package. You use stubs when you have tools that split apart a system into packages that you manipulate at different times and potentially in different places. For example, if you have a development team working in two locations, the team at one site would provide a stub for the packages the other team required. This strategy lets the teams work independently without disturbing each other's work, with the stub packages capturing the seams in the system that must be managed carefully.

Common Modeling Techniques

Modeling Groups of Elements

The most common purpose for which you'll use packages is to organize modeling elements into groups that you can name and manipulate as a set. If you are developing a trivial application, you won't need packages at all. All your abstractions will fit nicely into one package. For every other system, however, you'll find that many of your system's classes, interfaces, components, nodes, and even diagrams tend to naturally fall into groups. You model these groups as packages.

There is one important distinction between classes and packages: Classes are abstractions of things found in your problem or solution; packages are mechanisms you use to organize the things in your model. Packages have no identity (meaning that you can't have instances of packages, so they are invisible in the running system); classes do have identity (classes have instances, which are elements of a running system).

Most of the time, you'll use packages to group the same basic kind of elements. For example, you might separate all the classes and their corresponding relationships from your system's design view into a series of packages, using the UML's import dependencies to control access among these packages. You might organize all the components in your system's implementation view in a similar fashion.

You can also use packages to group different kinds of elements. For example, for a system being developed by a geographically distributed team, you might use packages as your unit of configuration management, putting in them all the classes and diagrams that each team can check in and check out separately. In fact, it's common to use packages to group modeling elements and their associated diagrams.

To model groups of elements,

- Scan the modeling elements in a particular architectural view and look for clumps defined by elements that are conceptually or semantically close to one another.
- Surround each of these clumps in a package.

- For each package, distinguish which elements should be accessible outside the package. Mark them public, and all others protected or private. When in doubt, hide the element.
- Explicitly connect packages that build on others via import dependencies.
- In the case of families of packages, connect specialized packages to their more general part via generalizations.

For example, Figure shows a set of packages that organize the classes in an information system's design view into a classic three-tier architecture. The elements in the package **UserServices** provide the visual interface for presenting information and gathering data. Theelements in the package **Data Services** maintain, access, and update data. The elements in the package **Business Services** bridge the elements in the other two packages and encompass all the classes and other elements that manage requests from the user to execute a business task, including business rules that dictate the policies for manipulating data.

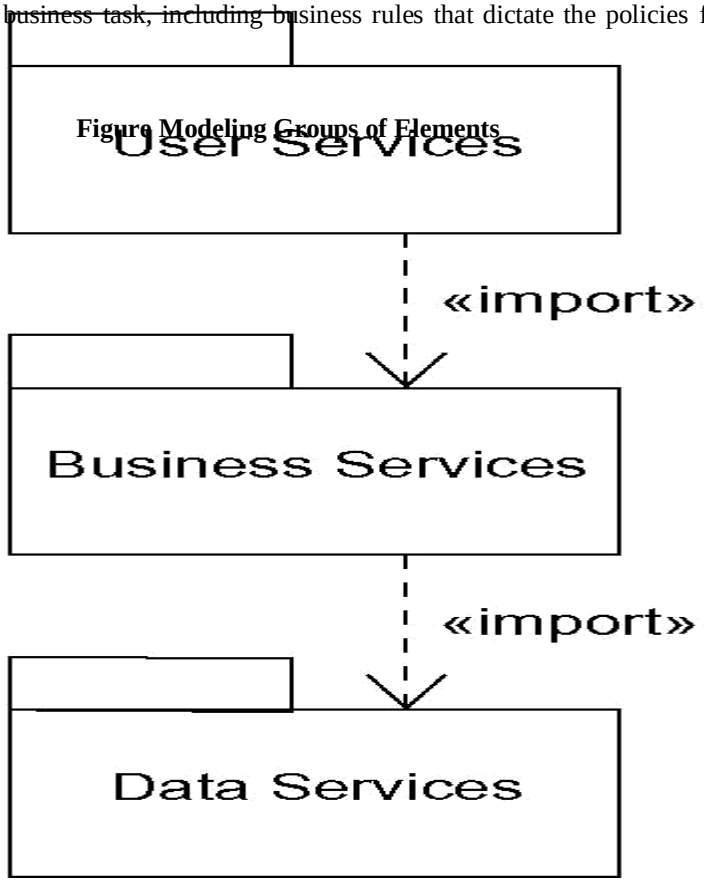


Figure Modeling Groups of Elements

In a trivial system, you could lump all your abstractions into one package. However, by organizing your classes and other elements of the system's design view into three packages, you not only make your model more understandable, but you can control access to the elements of your model by hiding some and exporting others.

Modeling Architectural Views

Using packages to group related elements is important; you can't develop complex models without doing so. This approach works well for organizing related elements, such as classes, interfaces, components, nodes, and diagrams. As you consider the different views of a software system's architecture, you need even larger chunks. You can use packages to model the views of an architecture.

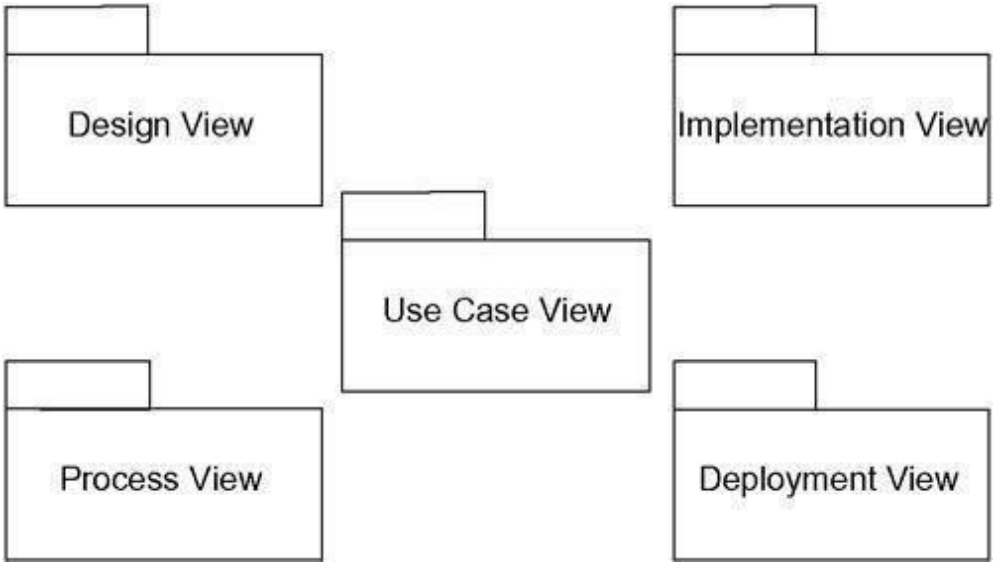
Remember that a view is a projection into the organization and structure of a system, focused on a particular aspect of that system. This definition has two implications. First, you can decompose a system into almost orthogonal packages, each of which addresses a set of architecturally significant decisions. For example, you might have a design view, a process view, an implementation view, a deployment view, and a use case view. Second, these packages own all the abstractions germane to that view. For example, all the components in your model would belong to the package that represents the implementation view.

To model architectural views,

- Identify the set of architectural views that are significant in the context of your problem. In practice, this typically includes a design view, a process view, an implementation view, a deployment view, and a use case view.
- Place the elements (and diagrams) that are necessary and sufficient to visualize, specify, construct, and document the semantics of each view into the appropriate package.
- As necessary, further group these elements into their own packages.
- There will typically be dependencies across the elements in different views. So, in general, let each view at the top of a system be open to all others at that level.

For example, Figure illustrates a canonical top-level decomposition that's appropriate foreven the most complex system you might encounter.

Figure Modeling Architectural Views



Class Diagrams

Terms and Concepts

A *class diagram* is a diagram that shows a set of classes, interfaces, and collaborations and their relationships. Graphically, a class diagram is a collection of vertices and arcs.

Common Properties

A class diagram is just a special kind of diagram and shares the same common properties as do all other diagrams• a name and graphical content that are a projection into a model. What distinguishes a class diagram from all other kinds of diagrams is its particular content.

Contents

Class diagrams commonly contain the following things:

- Classes
- Interfaces
- Collaborations
- Dependency, generalization, and association relationships

Like all other diagrams, class diagrams may contain notes and constraints.

Class diagrams may also contain packages or subsystems, both of which are used to group elements of your model into larger chunks. Sometimes, you'll want to place instances in your class diagrams, as well, especially when you want to visualize the (possibly dynamic) type of an instance.

Common Uses

You use class diagrams to model the static design view of a system. This view primarily supports the functional requirements of a system• the services the system should provide to its end users.

When you model the static design view of a system, you'll typically use class diagrams in one of three ways.

1. To model the vocabulary of a system

Modeling the vocabulary of a system involves making a decision about which abstractions are a part of the system under consideration and which fall outside its boundaries. You use class diagrams to specify these abstractions and their responsibilities.

2. To model simple collaborations

A collaboration is a society of classes, interfaces, and other elements that work together to provide some cooperative behavior that's bigger than the sum of all the elements. For example, when you're modeling the semantics of a transaction in a distributed system, you can't just stare at a single class to understand what's going on. Rather, these semantics are carried out by a set of classes that work together. You use class diagrams to visualize and specify this set of classes and their relationships.

3. To model a logical database schema

Think of a schema as the blueprint for the conceptual design of a database. In many domains, you'll want to store persistent information in a relational database or in an object-oriented database. You can model schemas for these databases using class diagrams.

Common Modeling Techniques

Modeling Simple Collaborations

No class stands alone. Rather, each works in collaboration with others to carry out some semantics greater than each individual. Therefore, in addition to capturing the vocabulary of your system, you'll also need to turn your attention to visualizing, specifying, constructing, and documenting the various ways these things in your vocabulary work together. You use class diagrams to represent such collaborations.

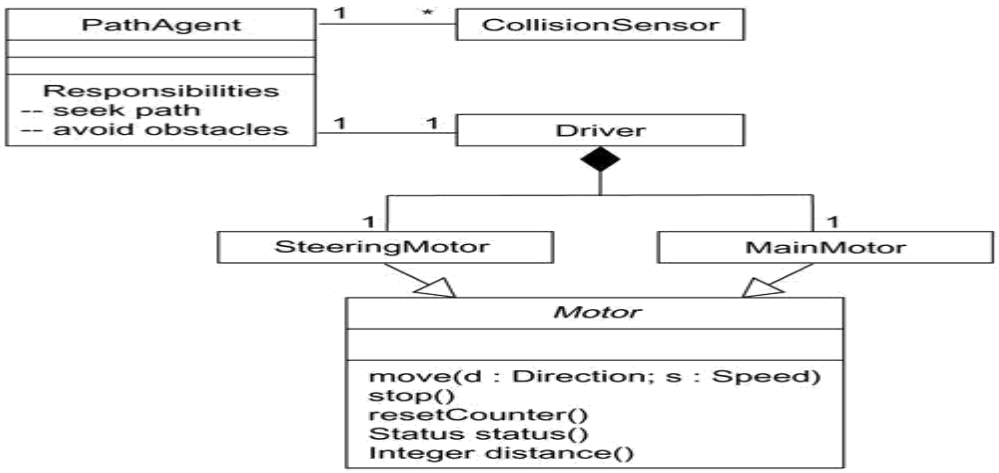
When you create a class diagram, you just model a part of the things and relationships that make up your system's design view. For this reason, each class diagram should focus on one collaboration at a time.

To model a collaboration,

- Identify the mechanism you'd like to model. A mechanism represents some function or behavior of the part of the system you are modeling that results from the interaction of a society of classes, interfaces, and other things.
- For each mechanism, identify the classes, interfaces, and other collaborations that participate in this collaboration. Identify the relationships among these things, as well.
- Use scenarios to walk through these things. Along the way, you'll discover parts of your model that were missing and parts that were just plain semantically wrong.
- Be sure to populate these elements with their contents. For classes, start with getting a good balance of responsibilities. Then, over time, turn these into concrete attributes and operations.

For example, Figure shows a set of classes drawn from the implementation of an autonomous robot. The figure focuses on the classes involved in the mechanism for moving the robot along a path. You'll find one abstract class (**Motor**) with two concrete children, **SteeringMotor**and**MainMotor.** Both of these classes inherit the five operations of theirparent, **Motor.** The two classes are, in turn, shown as parts of another class, **Driver.** The class **PathAgent** has a one-to-one association to **Driver** and a one-to-many association to **CollisionSensor.** No attributes or operations are shown for**PathAgent,** although itsresponsibilities are given.

Figure Modeling Simple Collaborations



There are many more classes involved in this system, but this diagram focuses only on those abstractions that are directly involved in moving the robot. You'll see some of these same classes in other diagrams. For example, although not shown here, the class **PathAgent**collaborateswith at least two other classes (**Environment** and **GoalAgent**) in a higher-level mechanism for managing the conflicting goals the robot might have at a given moment. Similarly, also not shown here, the classes **CollisionSensor** and **Driver** (and its parts) collaborate with another class (**FaultAgent**) in a mechanism responsible for continuously checking the robot's hardware for errors. By focusing on each of these collaborations in different diagrams, you provide an understandable view of the system from several angles.

Modeling a Logical Database Schema

Many of the systems you'll model will have persistent objects, which means that they can be stored in a database for later retrieval. Most often, you'll use a relational database, an object-oriented database, or a hybrid object/relational database for persistent storage. The UML is well-suited to modeling logical database schemas, as well as physical databases themselves.

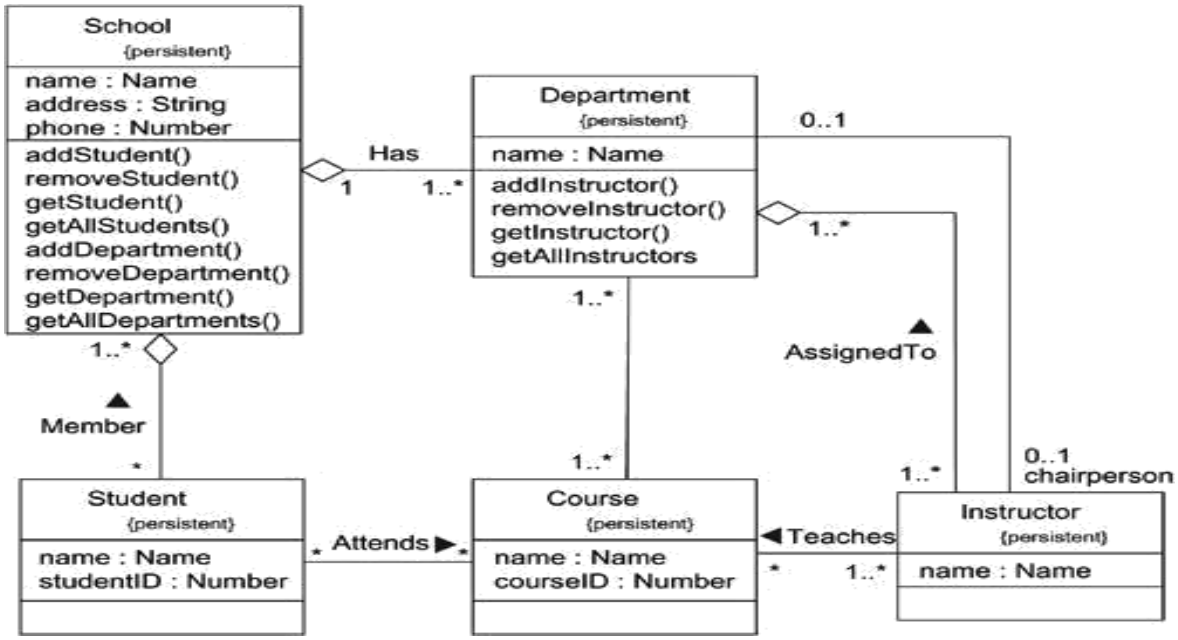
The UML's class diagrams are a superset of entity-relationship (E-R) diagrams, a common modeling tool for logical database design. Whereas classical E-R diagrams focus only on data,class diagrams go a step further by permitting the modeling of behavior, as well. In the physical database, these logical operations are generally turned into triggers or stored procedures.

To model a schema,

- Identify those classes in your model whose state must transcend the lifetime of their applications.
- Create a class diagram that contains these classes and mark them as persistent (a standard tagged value). You can define your own set of tagged values to address database-specific details.
- Expand the structural details of these classes. In general, this means specifying the details of their attributes and focusing on the associations and their cardinalities that structure these classes.
- Watch for common patterns that complicate physical database design, such as cyclic associations, one-to-one associations, and n-ary associations. Where necessary, create intermediate abstractions to simplify your logical structure.
- Consider also the behavior of these classes by expanding operations that are important for data access and data integrity. In general, to provide a better separation of concerns, business rules concerned with the manipulation of sets of these objects should be encapsulated in a layer above these persistent classes.
- Where possible, use tools to help you transform your logical design into a physical design.

Figure shows a set of classes drawn from an information system for a school. This figureexpands upon an earlier class diagram, and you'll see the details of these classes revealed to a level sufficient to construct a physical database. Starting at the bottom-left of this diagram, you will find the classes named **Student, Course,** and **Instructor.** There's an association between **Student** and **Course,** specifying that students attend courses. Furthermore, every student may attend any number of courses and every course may have any number of students.

Figure Modeling a Schema



All six of these classes are marked as persistent, indicating that their instances are intended to live in a database or some other form of persistent store. This diagram also exposes the attributes of all six of these classes. Notice that all the attributes are primitive types. When you are modeling a schema, you'll generally want to model the relationship to any nonprimitive types using an explicit aggregation rather than an attribute.

Two of these classes (**School** and **Department**) expose several operations for manipulating their parts. These operations are included because they are important to maintain data integrity (adding or removing a **Department,** for example, will have some rippling effects). There are many other operations that you might consider for these and the other classes, such as querying the prerequisites of a course before assigning a student. These are more business rules than they are operations for database integrity and so are best placed at a higher level of abstraction than this schema.

Forward and Reverse Engineering

Modeling is important, but you have to remember that the primary product of a development team is software, not diagrams. Of course, the reason you create models is to predictably deliver at the right time the right software that satisfies the evolving goals of its users and the business. For this reason, it's important that the models you create and the implementations you deploy map to one another and do so in a way that minimizes or even eliminates the cost of keeping your models and your implementation in sync with one another.

For some uses of the UML, the models you create will never map to code. For example, if you are modeling a business process using activity diagrams, many of the activities you model will involve people, not computers. In other cases, you'll want to model systems whose parts are, from your level of abstraction, just a piece of hardware (although at another level of abstraction, it's a good bet that this hardware contains an embedded computer and software).

In most cases, though, the models you create will map to code. The UML does not specify a particular mapping to any object-oriented programming language, but the UML was designed with such mappings

in mind. This is especially true for class diagrams, whose contents have a clear mapping to all the industrial-strength object-oriented languages, such as Java, C++, Smalltalk, Eiffel, Ada, ObjectPascal, and Forte. The UML was also designed to map to a variety of commercial object-based languages, such as Visual Basic.

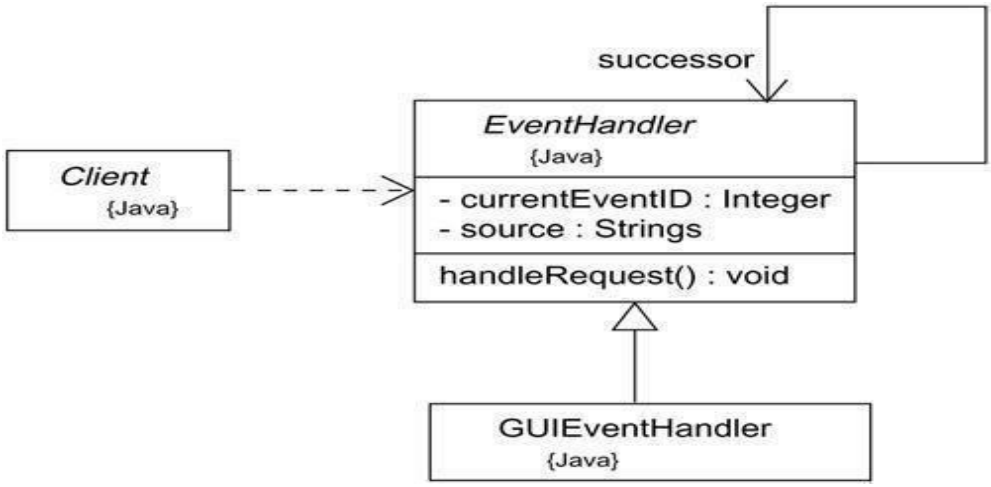
Forward engineering is the process of transforming a model into code through a mapping to animplementation language. Forward engineering results in a loss of information, because models written in the UML are semantically richer than any current object-oriented programming language. In fact, this is a major reason why you need models in addition to code. Structural features, such as collaborations, and behavioral features, such as interactions, can be visualized clearly in the UML, but not so clearly from raw code.

To forward engineer a class diagram,

- Identify the rules for mapping to your implementation language or languages of choice. This is something you'll want to do for your project or your organization as a whole.
- Depending on the semantics of the languages you choose, you may have to constrain your use of certain UML features. For example, the UML permits you to model multiple inheritance, but Smalltalk permits only single inheritance. You can either choose to prohibit developers from modeling with multiple inheritance (which makes your models language-dependent) or develop idioms that transform these richer features into the implementation language (which makes the mapping more complex).
- Use tagged values to specify your target language. You can do this at the level of individual classes if you need precise control. You can also do so at a higher level, such as with collaborations or packages.
- Use tools to forward engineer your models.

Figure illustrates a simple class diagram specifying an instantiation of the chain ofresponsibility pattern. This particular instantiation involves three classes: **Client,EventHandler,** and**GUIEventHandler. Client** and**EventHandler**are shown as abstractclasses, whereas **GUIEventHandler** is concrete. **EventHandler** has the usual operation expected of this pattern (**handleRequest**), although two private attributes have been added for this instantiation.

Figure Forward Engineering



All of these classes specify a mapping to Java, as noted in their tagged value. Forward engineering the classes in this diagram to Java is straightforward, using a tool. Forward engineering the class **EventHandler** yields the following code.

```
public abstract class EventHandler {

    EventHandler successor;
    private Integer currentEventID; private
    String source;

    EventHandler() {}
    public void handleRequest() {}

}
```

Reverse engineering is the process of transforming code into a model through a mapping from aspecific implementation language. Reverse engineering results in a flood of information, some of which is at a lower level of detail than you'll need to build useful models. At the same time, reverse engineering is incomplete. There is a loss of information when forward engineering models into code, and so you can't completely recreate a model from code unless your tools encode information in the source comments that goes beyond the semantics of the implementation language.

To reverse engineer a class diagram,

- Identify the rules for mapping from your implementation language or languages of choice. This is something you'll want to do for your project or your organization as a whole.
- Using a tool, point to the code you'd like to reverse engineer. Use your tool to generate a new model or modify an existing one that was previously forward engineered.

Instances

Terms and Concepts

An *instance* is a concrete manifestation of an abstraction to which a set of operations can be applied and which has a state that stores the effects of the operations. *Instance* and *object* are largely synonymous. Graphically, an instance is rendered by underlining its name.

Abstractions and Instances

Instances don't stand alone; they are almost always tied to an abstraction. Most instances you'll model with the UML will be instances of classes (and these things are called objects), although you can have instances of other things, such as components, nodes, use cases, and associations. In the UML, an instance is easily distinguishable from an abstraction. To indicate an instance, you underline its name.

In a general sense, an object is something that takes up space in the real or conceptual world, and you can do things to it. For example, an instance of a node is typically a computer that physically sits in a room; an instance of a component takes up some space on the file system; an instance of a customer record consumes some amount of physical memory. Similarly, an instance of a flight envelope for an aircraft is something you can manipulate mathematically.

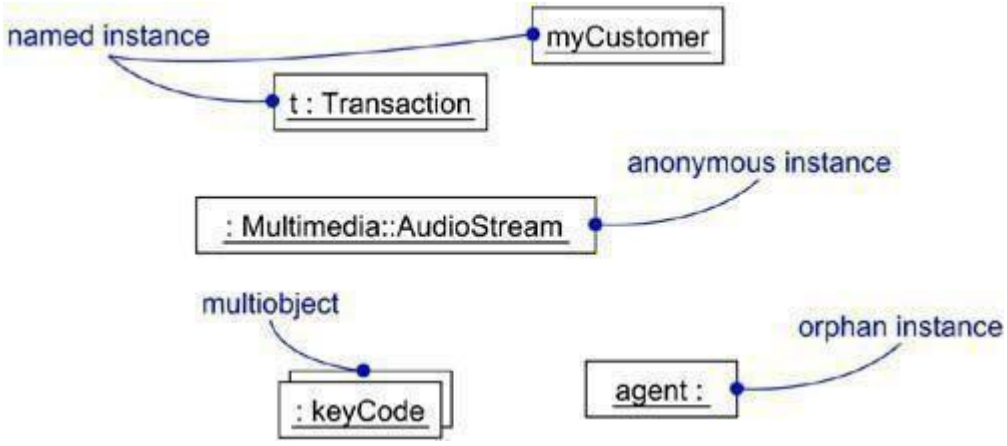
You can use the UML to model these physical instances, but you can also model things that are not so concrete. For example, an abstract class, by definition, may not have any direct instances.

However, you can model indirect instances of abstract classes in order to show the use of a prototypical instance of that abstract class. Literally, no such object might exist. But pragmatically, this instance lets you name one of any potential instances of concrete children of that abstract class. This same touch applies to interfaces. By their very definition, interfaces may not have any direct instances, but you can model a prototypical instance of an interface, representing one of any potential instances of concrete classes that realize that interface.

When you model instances, you'll place them in object diagrams (if you want to visualize their structural details) or in interaction and activity diagrams (if you want to visualize their participation in dynamic situations). Although typically not necessary, you can place objects in class diagrams if you want to explicitly show the relationship of an object to its abstraction.

The classifier of an instance is usually static. For example, once you create an instance of a class, its class won't change during the lifetime of that object. In some modeling situations and in some programming languages, however, it is possible to change the abstraction of an instance. For example, a **Caterpillar** object might become a **Butterfly** object. It's the same object, but of a different abstraction.

Figure Named, Anonymous, Multiple, and Orphan Instances



Names

Every instance must have a name that distinguishes it from other instances within its context. Typically, an object lives within the context of an operation, a component, or a node. A *name* is a textual string, such as **t**and**myCustomer**inFigure. That name alone is known as a*simple name.* The abstraction of the instance may be a simple name (such as**Transaction**) orit may be a *path name* (such as **Multimedia::AudioStream**) which is the abstraction's name prefixed by the name of the package in which that abstraction lives.

When you explicitly name an object, you are really giving it a name (such as **t**) that's usable by a human. You can also simply name an object (such as **aCustomer**) and elide its abstraction if it's obvious in the

given context. In many cases, however, the real name of an object is known only to the computer on which that object lives. In such cases, you can render an anonymous object (such as **: Multimedia::AudioStream**). Each occurrence of an anonymous object is considered distinct from all other occurrences. If you don't even know the object's associated abstraction, you must at least give it an explicit name (such as **agent :**).

Especially when you are modeling large collections of objects, it's clumsy to render the collection itself plus its individual instances. Instead, you can model multiobjects (such as **:keyCode**) as in Figure, representing a collection of anonymous objects.

Operations

Not only is an object something that usually takes up space in the real world, it is also something you can do things to. The operations you can perform on an object are declared in the object's abstraction. For example, if the class **Transaction** defines the operation **commit,** then given the instance **t : Transaction,** you can write expressions such as **t.commit().** The execution of this expression means that **t** (the object) is operated on by **commit** (the operation). Depending on the inheritance lattice associated with **Transaction,** this operation may or may not be invoked polymorphically.

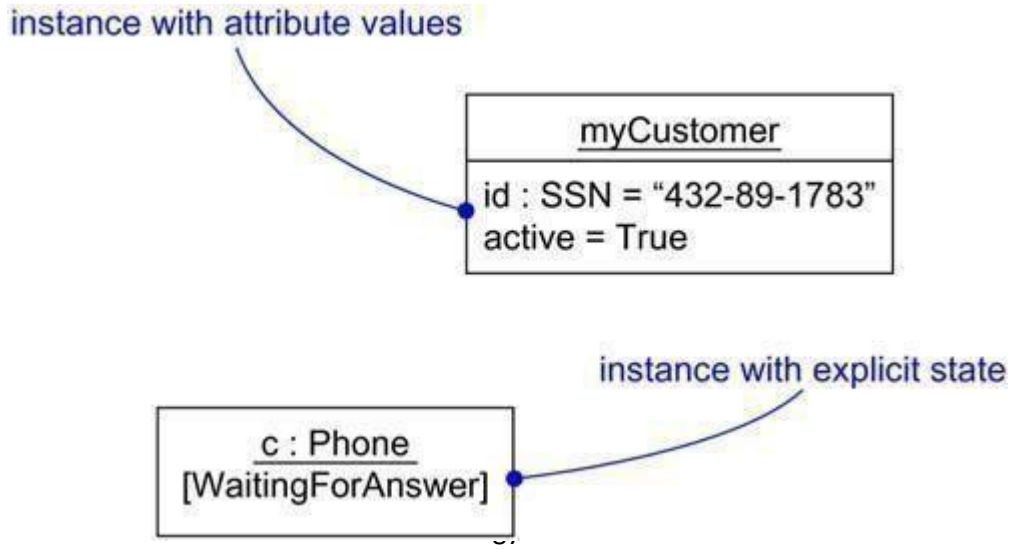
State

An object also has state, which in this sense encompasses all the (usually static) properties of the object plus the current (usually dynamic) values of each of these properties. These properties include the attributes of the object, as well as all its aggregate parts. An object's state is therefore dynamic. So when you visualize its state, you are really specifying the value of its state at a given moment in time and space. It's possible to show the changing state of an object by showing itmultiple times in the same interaction diagram, but with each occurrence representing a different state.

When you operate on an object, you typically change its state; when you query an object, you don't change its state. For example, when you make an airline reservation (represented by the object **r : Reservation**), you might set the value of one of its attributes (for example,**price= 395.75**). If you change your reservation, perhaps by adding a new leg to your itinerary, thenits state might change (for example, **price = 1024.86**).

As Figure shows, you can use the UML to render the value of an object's attributes. For example, **myCustomer** is shown with the attribute **id** having the value "**432-89-1783.**" In this case, **id**'s type (**SSN**) is shown explicitly, although it can be elided (as for **active = True**), because its type can be found in the declaration of **id** in **myCustomer**'s associated class.

Figure Object State

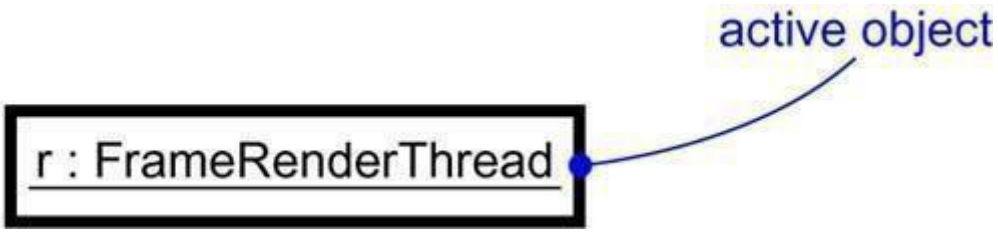


You can associate a state machine with a class, which is especially useful when modeling event-driven systems or when modeling the lifetime of a class. In these cases, you can also show the state of this machine for a given object at a given time. For example, as Figure shows, the object **c** (an instance of the class **Phone**) is indicated in the state **WaitingForAnswer,** a named state defined in the state machine for **Phone.**

Other Features

Processes and threads are an important element of a system's process view, so the UML provides a visual cue to distinguish elements that are active (those that are part of a process or thread and represent a root of a flow of control) from those that are passive. You can declare active classes that reify a process or thread, and in turn you can distinguish an instance of an active class, as in Figure.

Figure 13-4 Active Objects

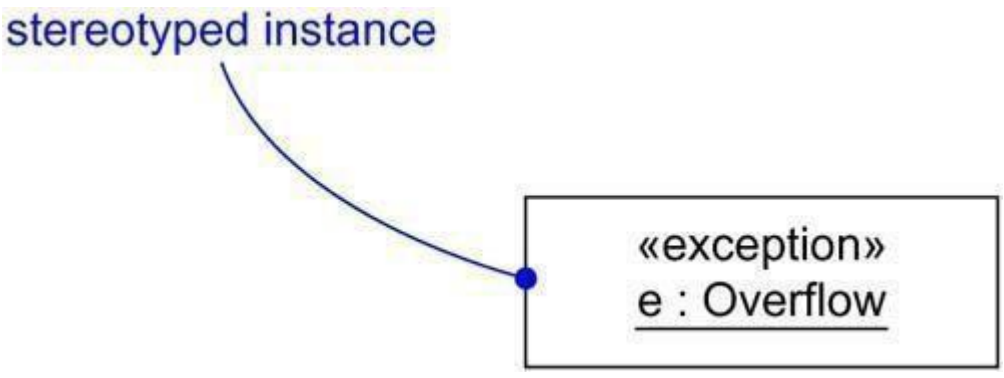


There are two other elements in the UML that may have instances. The first is a link. A link is a semantic connection among objects. An instance of an association is therefore a link. A link is rendered as a line, just like an association, but it can be distinguished from an association because links only connect objects. The second is a class-scoped attribute and operation. A class-scoped feature is in effect an object in the class that is shared by all instances of the class.

Standard Elements

All of the UML's extensibility mechanisms apply to objects. Usually, however, you don't stereotype an instance directly, nor do you give it its own tagged values. Instead, an object's stereotype and tagged values derive from the stereotype and tagged values of its associated abstraction. For example, as Figure shows, you can explicitly indicate an object's stereotype, as well as its abstraction.

Figure Stereotyped Objects



The UML defines two standard stereotypes that apply to the dependency relationships among objects and among classes:

1. **instanceOf**	Specifies that the client object is an instance of the supplier classifier
2. **instantiate**	Specifies that the client class creates instances of the supplier class

There are also two stereotypes related to objects that apply to messages and transitions:

1. **become**	Specifies that the client is the same object as the supplier, but at a later time and with possibly different values, state, or roles
2. **copy**	Specifies that the client object is an exact but independent copy of the supplier

The UML defines a standard constraint that applies to objects:

transient	Specifies that an instance of the role is created during execution of the enclosing interaction but is destroyed before completion of execution

Common Modeling Techniques

Modeling Concrete Instances

When you model concrete instances, you are in effect visualizing things that live in the real world. You can't exactly see an instance of a **Customer** class, for example, unless that customer is standing beside you; in a debugger, you might be able to see a representation of that object, however.One of the things for which you'll use objects is to model concrete instances that exist in the real world. For example, if you want to model the topology of your organization's network, you'll use deployment diagrams containing instances of nodes. Similarly, if you want to model the components that live on the physical nodes in this network, you'll use component diagrams containing instances of the components. Finally, suppose you have a debugger connected to your running system; it can present the structural relationships among instances by rendering an object diagram.

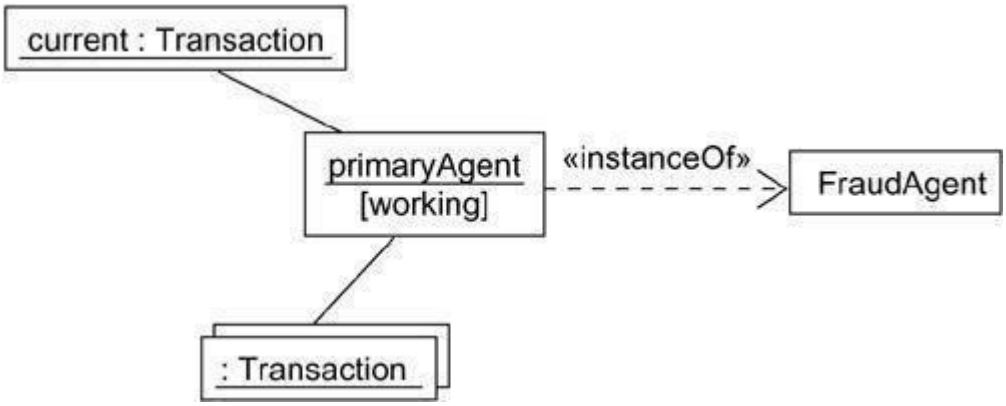
To model concrete instances,

- Identify those instances necessary and sufficient to visualize, specify, construct, or document the problem you are modeling.
- Render these objects in the UML as instances. Where possible, give each object a name. Ifthere is no meaningful name for the object, render it as an anonymous object.
- Expose the stereotype, tagged values, and attributes (with their values) of each instance necessary and sufficient to model your problem.
- Render these instances and their relationships in an object diagram or other diagram appropriate to the kind of the instance.

For example, Figure 13-6shows an object diagram drawn from the execution of a credit cardvalidation system, perhaps as seen by a debugger that's probing the running system. There is one multiobject, containing anonymous instances of the class **Transaction.** There are also two explicitly named objects•

primaryAgent and **current•** both of which expose their class, although in different ways. The diagram also explicitly shows the current state of the object **primaryAgent.**

Figure Modeling Concrete Instances



Note also the use of the dependency stereotyped as **instanceOf,** indicating the class of **primaryAgent.** Typically, you'll want to explicitly show these class/object relationships only ifyou also intend to show relationships with other classes.

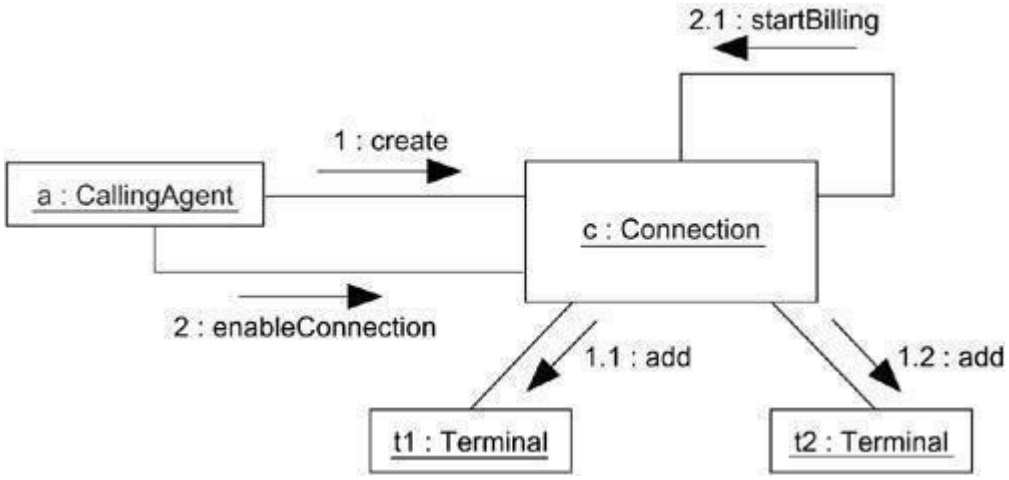
Modeling Prototypical Instances

Perhaps the most important thing for which you'll use instances is to model the dynamic interactions among objects. When you model such interactions, you are generally not modeling concrete instances that exist in the real world. Instead, you are modeling conceptual objects that are essentially proxies or stand-ins for objects that will eventually act that way in the real world. These are prototypical objects and, therefore, are roles to which concrete instances conform. For example, if you want to model the ways objects in a windowing application react to a mouse event, you'd draw an interaction diagram containing prototypical instances of windows, events, and handlers.
To model prototypical instances,

- Identify those prototypical instances necessary and sufficient to visualize, specify, construct, or document the problem you are modeling.
- Render these objects in the UML as instances. Where possible, give each object a name. If there is no meaningful name for the object, render it as an anonymous object.
- Expose the properties of each instance necessary and sufficient to model your problem.
- Render these instances and their relationships in an interaction diagram or an activity diagram.

Figure shows an interaction diagram illustrating a partial scenario for initiating a phone callin the context of a switch. There are four prototypical objects: **a** (a **CallingAgent**), **c** (a **Connection**), and **t1** and **t2** (both instances of **Terminal**). All four of these objects areprototypical; all represent conceptual proxies for concrete objects that may exist in the real world.

Figure Modeling Prototypical Instances



Object Diagrams

Terms and Concepts

An *object diagram* is a diagram that shows a set of objects and their relationships at a point in time. Graphically, an object diagram is a collection of vertices and arcs.

Common Properties

An object diagram is a special kind of diagram and shares the same common properties as all other diagrams• that is, a name and graphical contents that are a projection into a model. What distinguishes an object diagram from all other kinds of diagrams is its particular content.

Contents

Object diagrams commonly contain

- Objects
- Links

Like all other diagrams, object diagrams may contain notes and constraints.

Object diagrams may also contain packages or subsystems, both of which are used to group elements of your model into larger chunks. Sometimes, you'll want to place classes in your object diagrams, as well, especially when you want to visualize the classes behind each instance.

Common Uses

You use object diagrams to model the static design view or static process view of a system just as you do with class diagrams, but from the perspective of real or prototypical instances. This view primarily supports the functional requirements of a system• that is, the services the system should provide to its end users. Object diagrams let you model static data structures.

When you model the static design view or static process view of a system, you typically use object

diagrams in one way:

- To model object structures

Modeling object structures involves taking a snapshot of the objects in a system at a given moment in time. An object diagram represents one static frame in the dynamic storyboard represented by an interaction diagram. You use object diagrams to visualize, specify, construct, and document the existence of certain instances in your system, together with their relationships to one another.

Common Modeling Techniques

Modeling Object Structures

When you construct a class diagram, a component diagram, or a deployment diagram, what you are really doing is capturing a set of abstractions that are interesting to you as a group and, in that context, exposing their semantics and their relationships to other abstractions in the group. These diagrams show only potentiality. If class **A** has a one-to-many association to class **B,** then for one instance of **A** there might be five instances of **B;** for another instance of **A** there might be only one instance of **B.** Furthermore, at a given moment in time, that instance of **A,** along with the related instances of **B,** will each have certain values for their attributes and state machines.

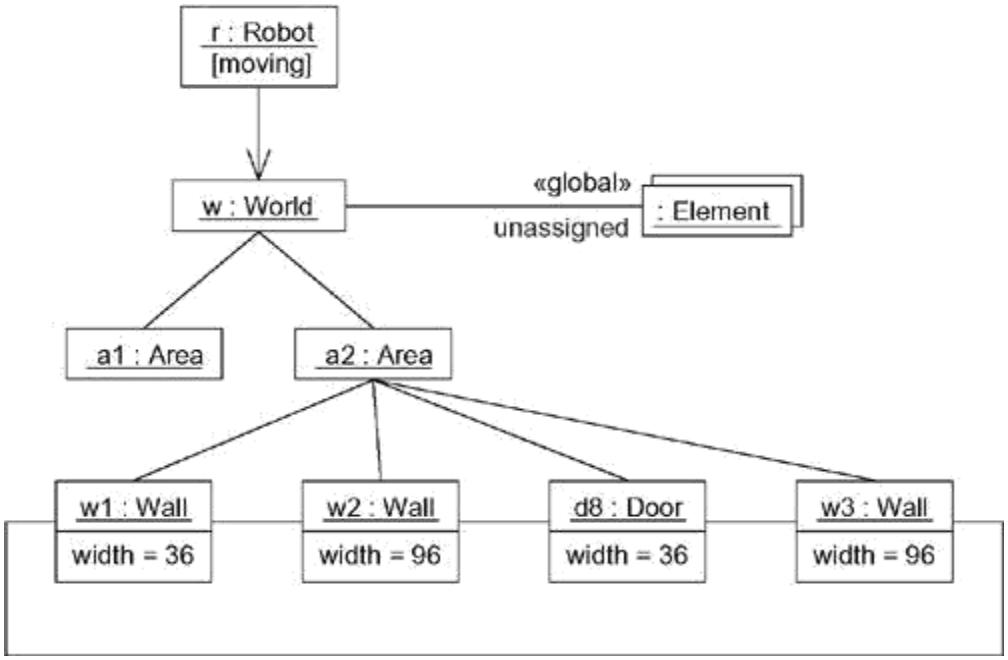
If you freeze a running system or just imagine a moment of time in a modeled system, you'll find a set of objects, each in a specific state, and each in a particular relationship to other objects. You can use object diagrams to visualize, specify, construct, and document the structure of these objects. Object diagrams are especially useful for modeling complex data structures.
When you model your system's design view, a set of class diagrams can be used to completely specify the semantics of your abstractions and their relationships. With object diagrams, however, you cannot completely specify the object structure of your system. For an individual class, there may be a multitude of possible instances, and for a set of classes in relationship to one another, there may be many times more possible configurations of these objects. Therefore, when you use object diagrams, you can only meaningfully expose interesting sets of concrete or prototypical objects. This is what it means to model an object structure• an object diagram shows one set of objects in relation to one another at one moment in time.

To model an object structure,

- Identify the mechanism you'd like to model. A mechanism represents some function or behavior of the part of the system you are modeling that results from the interaction of a society of classes, interfaces, and other things.
- For each mechanism, identify the classes, interfaces, and other elementsthat participate in this collaboration; identify the relationships among these things, as well.
- Consider one scenario that walks through this mechanism. Freeze that scenario at a moment in time, and render each object that participates in the mechanism.
- Expose the state and attribute values of each such object, as necessary, to understand the scenario.
- Similarly, expose the links among these objects, representing instances of associationsamong them.

For example, Figure shows a set of objects drawn from the implementation of an autonomous robot. This figure focuses on some of the objects involved in the mechanism used by the robot to calculate a model of the world in which it moves. There are many more objects involved in a running system, but this diagram focuses on only those abstractions that are directly involved in creating this world view.

Figure Modeling Object Structures



As this figure indicates, one object represents the robot itself (**r,**an instance of**Robot**), and**r**iscurrently in the state marked **moving.** This object has a link to **w,** an instance of **World,** which represents an abstraction of the robot's world model. This object has a link to a multiobject that consists of instances of **Element,** which represent entities that the robot has identified but not yet assigned in its world view. These elements are marked as part of the robot's global state.

At this moment in time, **w** is linked to two instances of **Area.** One of them (**a2**) is shown with its own links to three **Wall** and one **Door** object. Each of these walls is marked with its current width, and each is shown linked to its neighboring walls. As this object diagram suggests, the robot has recognized this enclosed area, which has walls on three sides and a door on the fourth.

Forward and Reverse Engineering

Forward engineering (the creation of code from a model) an object diagram is theoretically possible but pragmatically of limited value. In an object- oriented system, instances are things that are created and destroyed by the application during run time. Therefore, you can't exactly instantiate these objects from the outside.

Although this is true of most typical object diagrams (which contain instances of classes), it's not true of object diagrams containing instances of components and of nodes. Both of these are special cases of component diagrams and deployment diagrams, respectively, and are discussed elsewhere. In these cases, component instances and node instances are things that live outside the running system and are amenable to some degree of forward engineering.

Reverse engineering (the creation of a model from code) an object diagram is a very different thing. In fact, while you are debugging your system, this is something that you or your tools will do all the time. For example, if you are chasing down a dangling link, you'll want to literally or mentally draw an object diagram of the affected objects to see where, at a given moment in time, an object's state or its relationship to other objects is broken.

To reverse engineer an object diagram,

- Chose the target you want to reverse engineer. Typically, you'll set your context inside an operation or relative to an instance of one particular class.
- Using a tool or simply walking through a scenario, stop execution at a certain moment in time.
- Identify the set of interesting objects that collaborate in that context and render them in an object diagram.
- As necessary to understand their semantics, expose these object's states.
- As necessaryto understand their semantics, identify the links that exist among these objects.
- If your diagram ends up overly complicated, prune it by eliminating objects that are not germane to the questions about the scenario you need answered. If your diagram is too simplistic, expand the neighbors of certain interesting objects and expose each object's state more deeply.

Unit-3
Basic Behavioral Modeling-I

Syllabus : Interactions, Interaction diagrams, Use cases, Use case Diagrams, Activity diagrams

Interactions

Terms and Concepts

An *interaction* is a behavior that comprises a set of messages exchanged among a set of objects within a context to accomplish a purpose. A *message* is a specification of a communication between objects that conveys information with the expectation that activity will ensue.

Context

You may find an interaction wherever objects are linked to one another. You'll find interactions in the collaboration of objects that exist in the context of your system or subsystem. You will also find interactions in the context of an operation. Finally, you'll find interactions in the context of a class.

Most often, you'll find interactions in the collaboration of objects that exist in the context of your system or subsystem as a whole. For example, in a system for Web commerce, you'll find objects on the client (such as instances of the classes **BookOrder** and **OrderForm**) interacting with one another. You'll also find objects on the client (again, such as instances of **BookOrder**) interacting with objects on the server (such as instances of **BackOrderManager**). These interactions therefore not only involve localized collaborations of objects (such as the interactions surrounding **OrderForm**), but they may also cut across many conceptual levels of your system (such as theinteractions surrounding **BackOrderManager**).

You'll also find interactions among objects in the implementation of an operation. The parameters of an operation, any variables local to the operation, and any objects global to the operation (but still visible to the operation) may interact with one another to carry out the algorithm of that operation's implementation. For example, invoking the operation **moveToPosition(p :Position)** defined for a class in a mobile robot will involve the interaction of a parameter (**p**), anobject global to the operation (such as the object **currentPosition**), and possibly several local objects (such as local variables used by the operation to calculate intermediate points in a path to the new position).

Finally, you will find interactions in the context of a class. You can use interactions to visualize, specify, construct, and document the semantics of a class. For example, to understand the meaning of a class **RayTraceAgent,** you might create interactions that show how the attributes of that class collaborate with one another (and with objects global to instances of the class and with parameters defined in the class's operations).

Objects and Roles

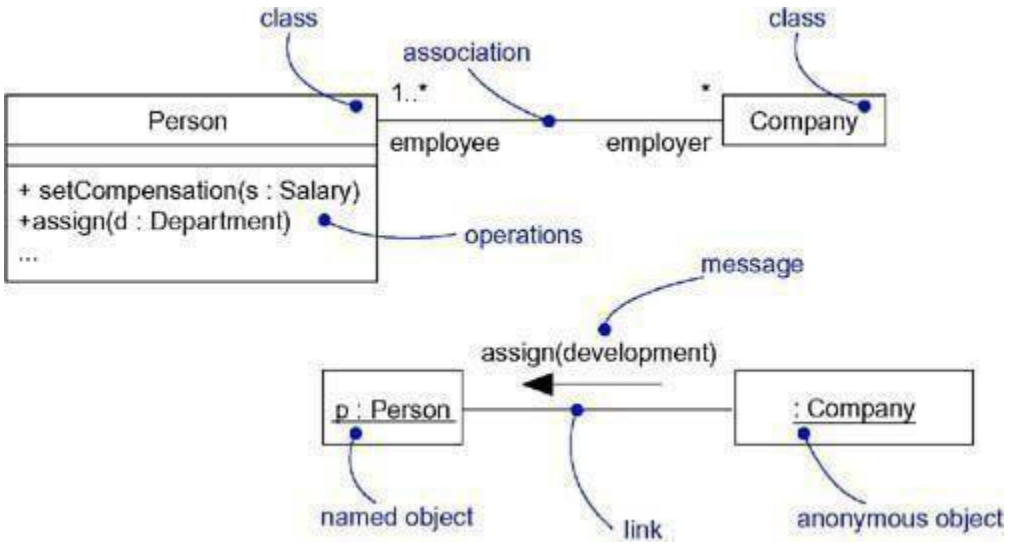
The objects that participate in an interaction are either concrete things or prototypical things. As a concrete thing, an object represents something in the real world. For example, **p,** an instance of the class **Person,** might denote a particular human. Alternately, as a prototypical thing, **p** might represent any instance of **Person.**
In the context of an interaction, you may find instances of classes, components, nodes, and use cases. Although abstract classes and interfaces, by definition, may not have any direct instances, you may find instances of these things in an interaction. Such instances do not represent direct instances of the abstract class or of the interface, but may represent, respectively, indirect (or prototypical) instances of any concrete children of the abstract class of some concrete class that realizes that interface.

You can think of an object diagram as a representation of the static aspect of an interaction, setting the stage for the interaction by specifying all the objects that work together. An interaction goes further by introducing a dynamic sequence of messages that may pass along the links that connect these objects

Links

A link is a semantic connection among objects. In general, a link is an instance of an association. As Figure shows, wherever a class has an association to another class, there may be a link between the instances of the two classes; wherever there is a link between two objects, one object can send a message to the other object.

Figure Links and Associations



A link specifies a path along which one object can dispatch a message to another (or the same) object. Most of the time, it is sufficient to specify that such a path exists. If you need to be more precise about how that path exists, you can adorn the appropriate end of the link with any of the following standard stereotypes.

association	Specifies that the corresponding object is visible by association
self	Specifies that the corresponding object is visible because it is the dispatcher of the operation
global	Specifies that the corresponding object is visible because it is in an enclosing scope
local	Specifies that the corresponding object is visible because it is in a local scope
parameter	Specifies that the corresponding object is visible because it is a parameter

Messages

Call	Invokes an operation on an object; an object may send a message to itself, resulting in the local invocation of an operation
Return	Returns a value to the caller
Send	Sends a signal to an object
Create	Creates an object
Destroy	Destroys an object; an object may commit suicide by destroying itself

Suppose you have a set of objects and a set of links that connect those objects. If that's all you have, then you have a completely static model that can be represented by an object diagram. Object diagrams model the state of a society of objects at a given moment in time and are useful when you want to visualize, specify, construct, or document a static object structure.

Suppose you want to model the changing state of a society of objects over a period of time. Think of it as taking a motion picture of a set of objects, each frame representing a successive moment in time. If these objects are not totally idle, you'll see objects passing messages to other objects, sending events, and invoking operations. In addition, at each frame, you can explicitly visualize the current state and role of individual instances.

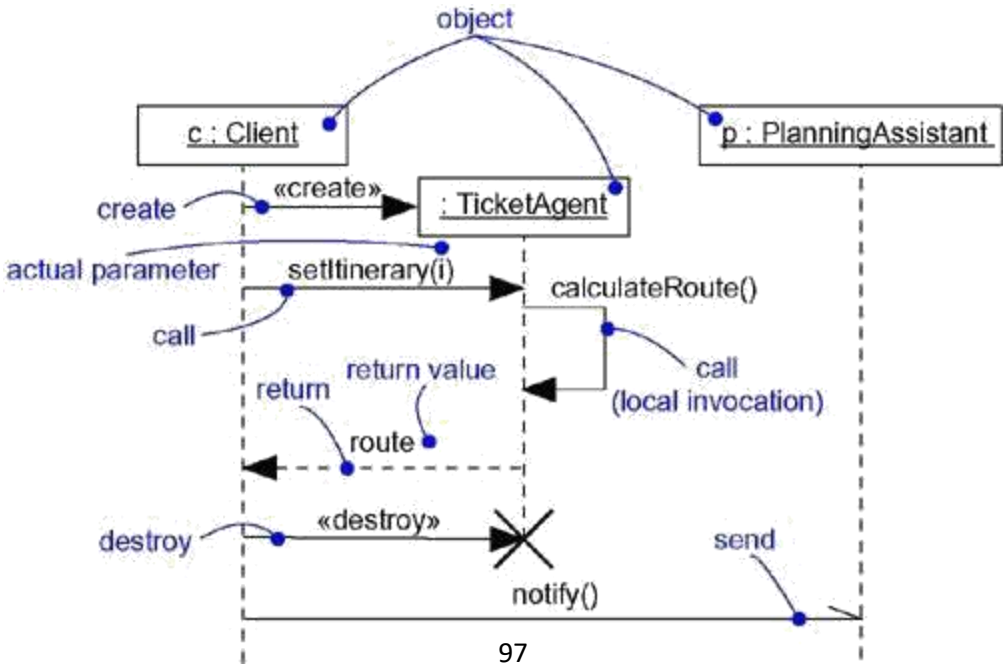
A message is the specification of a communication among objects that conveys information with the expectation that activity will ensue. The receipt of a message instance may be considered an instance of an event.

When you pass a message, the action that results is an executable statement that forms an abstraction of a computational procedure. An action may result in a change in state.

In the UML, you can model several kinds of actions.

The UML provides a visual distinction among these kinds of messages, as Figure shows.

Figure Messages



The most common kind of message you'll model is the call, in which one object invokes an operation of another (or the same) object. An object can't just call any random operation. If an object, such as **c** in the example above, calls the operation **setItinerary** on an instance of the class **TicketAgent,** the operation **setItinerary** must not only be defined for the class **TicketAgent**(that is, it must be declared in the class**TicketAgent**or one of its parents), itmust also be visible to the caller **c.**

When an object calls an operation or sends a signal to another object, you can provide actual parameters to the message. Similarly, when an object returns control to another object, you can model the return value, as well.

Sequencing

When an object passes a message to another object (in effect, delegating some action to the receiver), the receiving object might in turn send a message to another object, which might send a message to yet a different object, and so on. This stream of messages forms a sequence. Any sequence must have a beginning; the start of every sequence is rooted in some process or thread. Furthermore, any sequence will continue as long as the process or thread that owns it lives. A nonstop system, such as you might find in real time device control, will continue to execute as long as the node it runs on isup.

Each process and thread within a system defines a distinct flow of control, and within each flow, messages are ordered in sequence by time. To better visualize the sequence of a message, you can explicitly model the order of the message relative to the start of the sequence by prefixing the message with a sequence number set apart by a colon separator.

Most commonly, you can specify a procedural or nested flow of control, rendered using a filled solid arrowhead, as Figure shows. In this case, the message **findAt**is specified as the first message nested in the second message of the sequence (**2.1**).

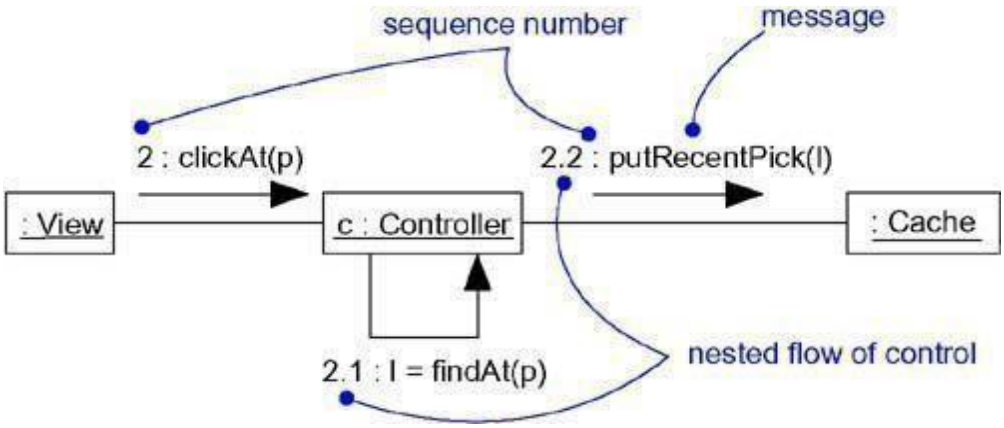


Figure Procedural Sequence

Less common but also possible, as Figure shows, you can specify a flat flow of control, rendered using a stick arrowhead, to model the nonprocedural progression of control from step to step. In this case, the message **assertCall** is specified as the second message in the sequence.

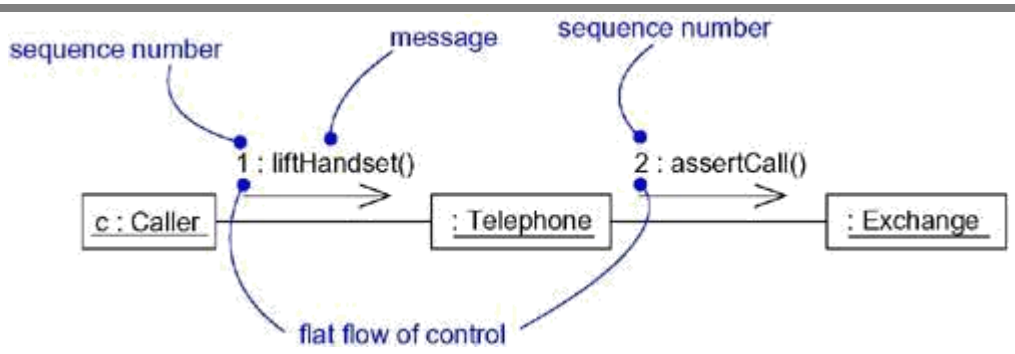


Figure Flat Sequenc

When you are modeling interactions that involve multiple flows of control, it's especially important to identify the process or thread that sent a particular message. In the UML, you can distinguish one flow of control from another by prefixing a message's sequence number with the name of the process or thread that sits at the root of the sequence. For example, the expression

D5 :ejectHatch(3)

specifies that the operation **ejectHatch** is dispatched (with the actual argument **3**) as the fifth message in the sequence rooted by the process or thread named **D.**

Not only can you show the actual arguments sent along with an operation or a signal in the context of an interaction, you can show the return values of a function as well. As the following expression shows, the value **p** is returned from the operation **find,** dispatched with the actual parameter **"Rachelle".** This is a nested sequence, dispatched as the second message nested in the third message nested in the first message of the sequence. In the same diagram, **p** can then be used as an actual parameter in other messages.

Creation, Modification, and Destruction

Most of the time, the objects you show participating in an interaction exist for the entire duration of the interaction. However, in some interactions, objects may be created (specified by a **create** message) and destroyed (specified by a **destroy** message). The same is true of links: the relationships among objects may come and go. To specify if an object or link enters and/or leaves during an interaction, you can attach one of the following constraints to the element:

new	Specifies that the instance or link is created during execution of the enclosing interaction
destroyed the en	Specifies that the instance or link is destroyed prior to completion of execution of losing interaction
transient interac	Specifies that the instance or link is created during execution of the enclosing tion but is destroyed before completion of execution

During an interaction, an object typically changes the values of its attributes, its state, or its roles. You can represent the modification of an object by replicating the object in the interaction (with possibly different attribute values, state, or roles). On a sequence diagram, you'd place each variant of the object on the same lifeline. In an interaction diagram, you'd connect each variant with a **become** message.

Representation

When you model an interaction, you typically include both objects (each one playing a specific role) and messages (each one representing the communication between objects, with some resulting action).

You can visualize those objects and messages involved in an interaction in two ways: by emphasizing the time ordering of its messages, and by emphasizing the structural organization of the objects that send and

receive messages. In the UML, the first kind of representation is called a sequence diagram; the second kind of representation is called a collaboration diagram. Both sequence diagrams and collaboration diagrams are kinds of interaction diagrams.

Sequence diagrams and collaboration diagrams are largely isomorphic, meaning that you can take one and transform it into the other without loss of information. There are some visual differences, however. First, sequence diagrams permit you to model the lifeline of an object. An object's lifeline represents the existence of the object at a particular time, possibly covering the object's creation and destruction. Second, collaboration diagrams permit you to model the structural links that may exist among the objects in an interaction.

Common Modeling Techniques

Modeling a Flow of Contro

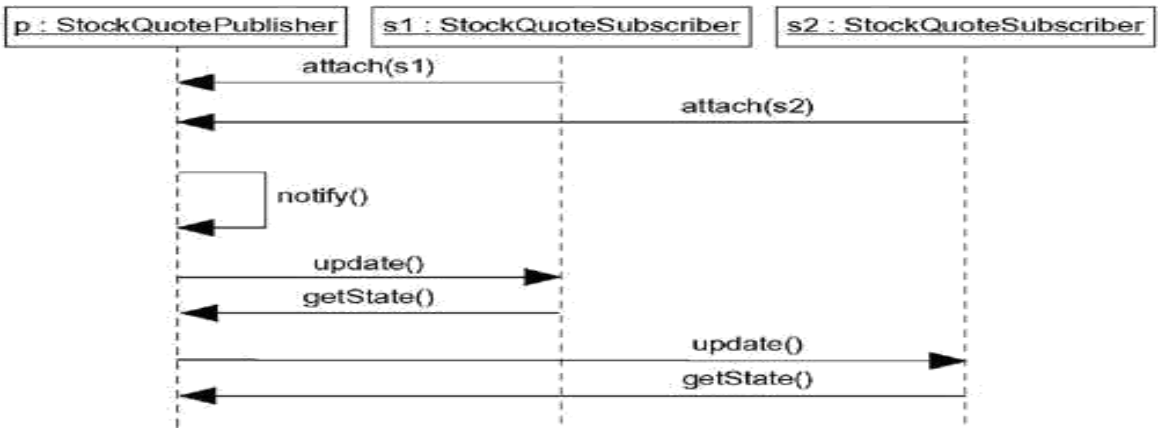
When you model an interaction, you essentially build a storyboard of the actions that take place among a set of objects. Techniques such as CRC cards are particularly useful in helping you to discover and think about such interactions.

To model a flow of control,

- Set the context for the interaction, whether it is the system as a whole, a class, or an individual operation.
- Set the stage for the interaction by identifying which objects play a role; set their initial properties, including their attribute values, state, and role.
- If your model emphasizes the structural organization of these objects, identify the links that connect them, relevant to the paths of communication that take place in this interaction. Specify the nature of the links using the UML's standard stereotypes and constraints, as necessary.
- In time order, specify the messages that pass from object to object. As necessary, distinguish the different kinds of messages; include parameters and return values to convey the necessary detail of this interaction.
- Also to convey the necessary detail of this interaction, adorn each object at every moment in time with its state and role.

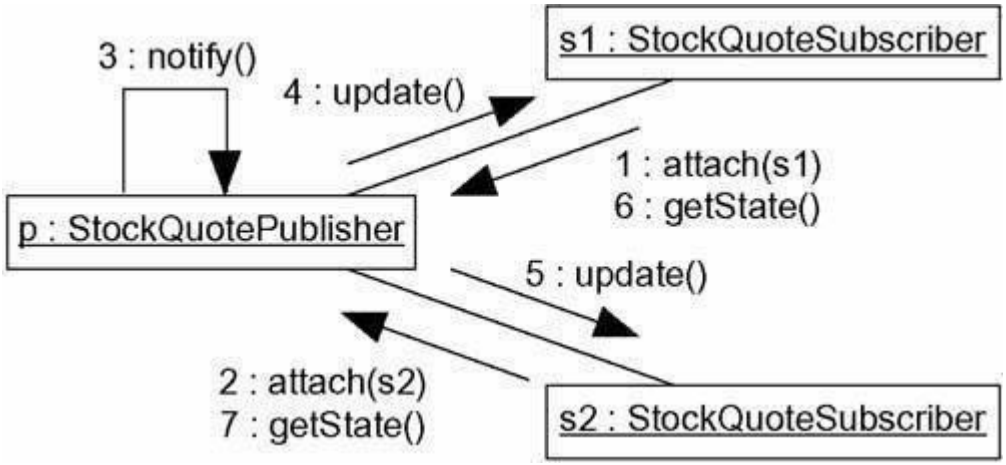
For example, Figure shows a set of objects that interact in the context of a publish and subscribe mechanism (an instance of the observer design pattern). This figure includes three objects: **p** (a **StockQuotePublisher**), **s1,** and **s2** (both instances of **StockQuoteSubscriber**). This figure is an example of a sequence diagram, which emphasizesthe time order of messages.

Figure Flow of Control by Time



Figureis semantically equivalent to the previous one, but it is drawn as a collaborationdiagram, which emphasizes the structural organization of the objects. This figure shows the same flow of control, but it also provides a visualization of the links among these objects.

Figure Flow of Control by Organization



Interaction Diagrams

Terms and Concepts

An *interaction diagram* shows an interaction, consisting of a set of objects and their relationships, including the messages that may be dispatched among them. A *sequence diagram* is an interaction diagram that emphasizes the time ordering of messages. Graphically, a sequence diagram is a table that shows objects arranged along the X axis and messages, ordered in increasing time, along the Y axis. A *collaboration diagram* is an interaction diagram that emphasizes the structural organization of the objects that send and receive messages. Graphically, a collaboration diagram is a collection of vertices and arcs.

Common Properties

An interaction diagram is just a special kind of diagram and shares the same common properties as do all other diagrams• a name and graphical contents that are a projection into a model. What distinguishes an interaction diagram from all other kinds of diagrams is its particular content.

Contents

Interaction diagrams commonly contain

- Objects
- Links
- Messages

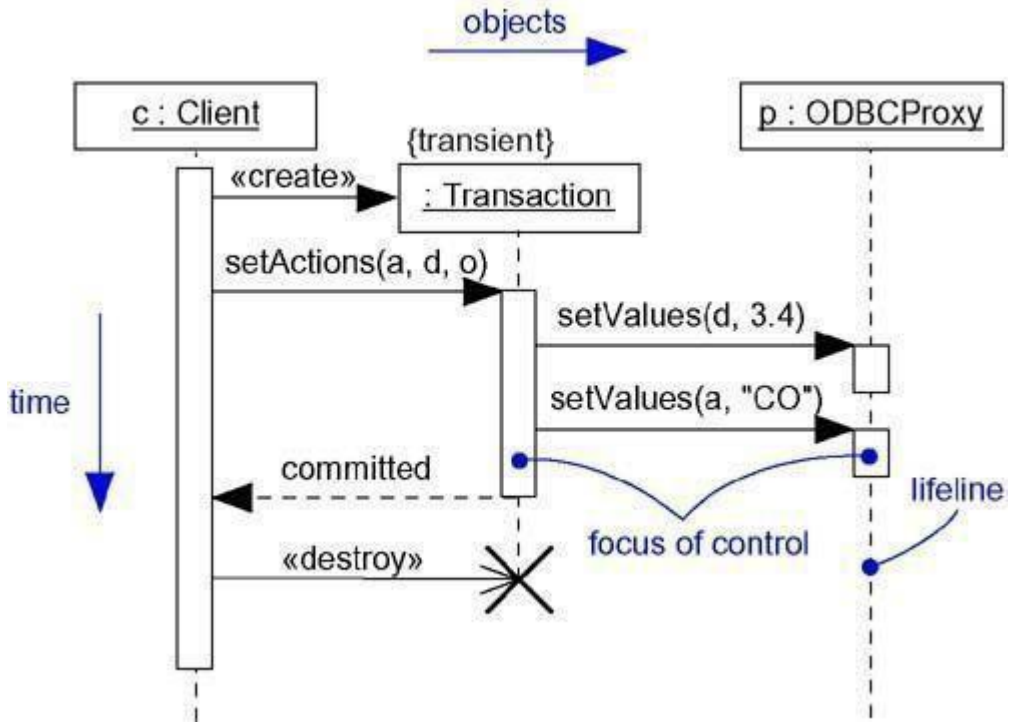
Like all other diagrams, interaction diagrams may contain notes and constraints.

Sequence Diagrams

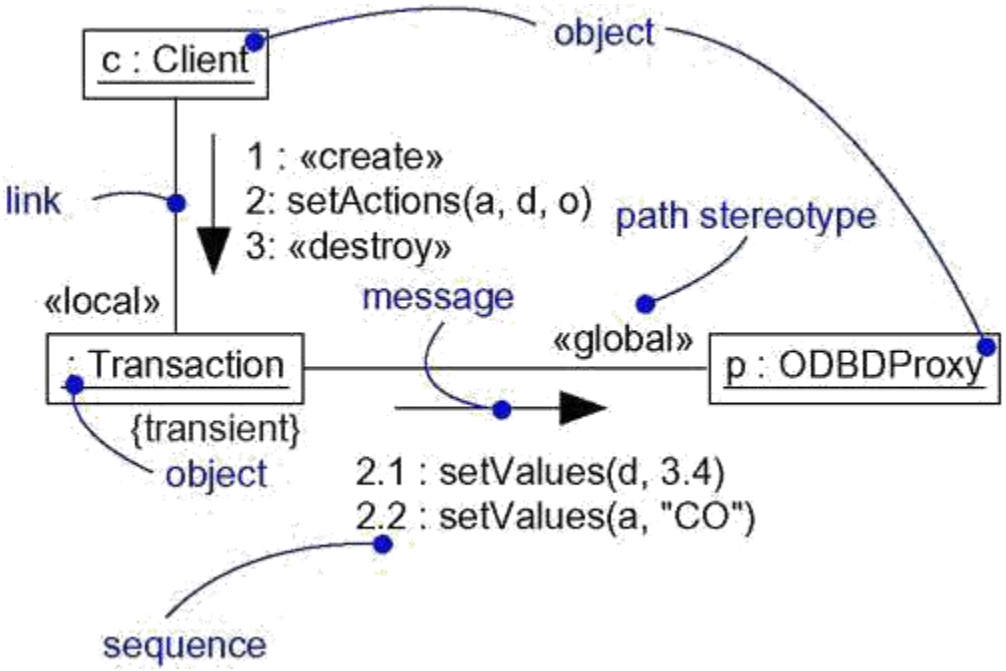
A sequence diagram emphasizes the time ordering of messages. As Figure shows, you form a sequence diagram by first placing the objects that participate in the interaction at the top of your diagram, across the

X axis. Typically, you place the object that initiates the interaction at the left, and increasingly more subordinate objects to the right. Next, you place the messages that these objects send and receive along the Y axis, in order of increasing time from top to bottom. This gives the reader a clear visual cue to the flow of control over time.

Figure Sequence Diagram



Sequence diagrams have two features that distinguish them from collaboration diagrams.



First, there is the object lifeline. An object lifeline is the vertical dashed line that represents the existence of an object over a period of time. Most objects that appear in an interaction diagram will be in existence for the duration of the interaction, so these objects are all aligned at the top of the diagram, with their lifelines drawn from the top of the diagram to the bottom. Objects may be created during the interaction. Their lifelines start with the receipt of the message stereotyped as **create.** Objects may be destroyed during the interaction. Their lifelines end with the receipt ofthe message stereotyped as **destroy** (and are given the visual cue of a large **X**, marking the end of their lives).

Second, there is the focus of control. The focus of control is a tall, thin rectangle that shows the period of time during which an object is performing an action, either directly or through a subordinate procedure. The top of the rectangle is aligned with the start of the action; the bottom is aligned with its completion (and can be marked by a return message). You can show the nesting of a focus of control (caused by recursion, a call to a self- operation, or by a callback from another object) by stacking another focus of control slightly to the right of its parent (and can do so to an arbitrary depth). If you want to be especially precise about where the focus of control lies, you can also shade the region of the rectangle during which the object's method is actually computing (and control has not passed to another object).

Collaboration Diagrams

A collaboration diagram emphasizes the organization of the objects that participate in an interaction. As Figure shows, you form a collaboration diagram by first placing the objects that participate in the interaction as the vertices in a graph. Next, you render the links that connect these objects as the arcs of this graph. Finally, you adorn these links with the messages that objects send and receive. This gives the reader a clear visual cue to the flow of control in the context of the structural organization of objects that collaborate.

Figure Collboration Diagram

Collaboration diagrams have two features that distinguish them from sequence diagrams.

First, there is the path. To indicate how one object is linked to another, you can attach a path stereotype to the far end of a link (such as **»local**, indicating that the designated object is localto the sender). Typically, you will only need to render the path of the link explicitly for **local**, **parameter**, **global**, and **self** (but not **association**) paths.

Second, there is the sequence number. To indicate the time order of a message, you prefix the message with a number (starting with the message numbered **1**), increasing monotonically for each new message in the flow of control (**2**, **3**, and so on). To show nesting, you use Dewey decimal numbering (**1** is the first message; **1.1** is the first message nested in message **1**; **1.2** is the second message nested in message **1** ; and so on). You can show nesting to an arbitrary depth. Note also that, along the same link, you can show many messages (possibly being sent from different directions), and each will have a unique sequence number.

Most of the time, you'll model straight, sequential flows of control. However, you can also model more-complex flows, involving iteration and branching. An iteration represents a repeated sequence of messages. To model an iteration, you prefix the sequence number of a message with an iteration expression such as ***[i := 1..n]** (or just ***** if you want to indicate iteration but don't want to specify its details). An iteration indicates that the message (and any nested messages) will be repeated in accordance with the given expression. Similarly, a condition represents a message whose execution is contingent on the evaluation of a Boolean condition. To model a condition, you prefix the sequence number of a message with a condition clause, such as **[x > 0].** The alternate paths of a branch will have the same sequence number, but each path must be uniquely distinguishable by a nonoverlapping condition.

For both iteration and branching, the UML does not prescribe the format of the expression inside the brackets; you can use pseudocode or the syntax of a specific programming language.

Semantic Equivalence

Because they both derive from the same information in the UML's metamodel, sequence diagrams and collaboration diagrams are semantically equivalent. As a result, you can take a diagram in one form and convert it to the other without any loss of information, as you can see in the previous two figures, which are semantically equivalent. However, this does not mean that both diagrams will explicitly visualize the same information. For example, in the previous two figures, the collaboration diagram shows how the objects are linked (note the **»local** and **»global** stereotypes), whereas the corresponding sequence diagram does not. Similarly, thesequence diagram shows message return (note the return value **committed**), but the corresponding collaboration diagram does not. In both cases, the two diagrams share the same underlying model, but each may render some things the other does not.

Common Uses

You use interaction diagrams to model the dynamic aspects of a system. These dynamic aspects may involve the interaction of any kind of instance in any view of a system's architecture, including instances of classes (including active classes), interfaces, components, and nodes.

When you use an interaction diagram to model some dynamic aspect of a system, you do so in the context of the system as a whole, a subsystem, an operation, or a class. You can also attach interaction diagrams to use cases (to model a scenario) and to collaborations (to model the dynamic aspects of a society of objects).

When you model the dynamic aspects of a system, you typically use interaction diagrams in two ways.

1. To model flows of control by time ordering

Here you'll use sequence diagrams. Modeling a flow of control by time ordering emphasizes the passing of messages as they unfold over time, which is a particularly useful way to visualize dynamic behavior in the context of a use case scenario. Sequence diagrams do a better job of visualizing simple iteration and branching than do collaboration diagrams.

2. To model flows of control by organization

Here you'll use collaboration diagrams. Modeling a flow of control by organization emphasizes the structural relationships among the instances in the interaction, along which messages may be

passed. Collaboration diagrams do a better job of visualizing complex iteration and branching and of visualizing multiple concurrent flows of control than do sequence diagrams.

Common Modeling Techniques

Modeling Flows of Control by Time Ordering

Consider the objects that live in the context of a system, subsystem, operation or class. Consider also the objects and roles that participate in a use case or collaboration. To model a flow of control that winds through these objects and roles, you use an interaction diagram; to emphasize the passing of messages as they unfold over time, you use a sequence diagram, a kind of interaction diagram.

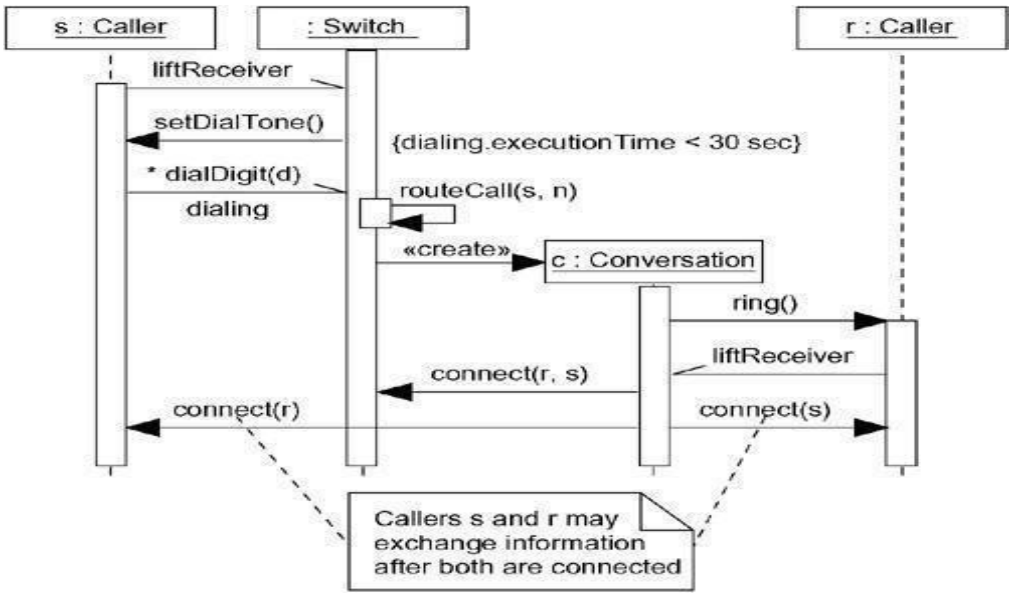
To model a flow of control by time ordering,

- Set the context for the interaction, whether it is a system, subsystem, operation, or class, or one scenario of a use case or collaboration.
- Set the stage for the interaction by identifying which objects play a role in the interaction. Lay them out on the sequence diagram from left to right, placing the more important objects to the left and their neighboring objects to the right.
- Set the lifeline for each object. In most cases, objects will persist through the entire interaction. For those objects that are created and destroyed during the interaction, set their lifelines, as appropriate, and explicitly indicate their birth and death with appropriately stereotyped messages.
- Starting with the message that initiates this interaction, lay out each subsequent message from top to bottom between the lifelines, showing each message's properties (such as its parameters), as necessary to explain the semantics of the interaction.
- If you need to visualize the nesting of messages or the points in time when actual computation is taking place, adorn each object's lifeline with its focus of control.
- If you need to specify time or space constraints, adorn each message with a timing mark and attach suitable time or space constraints.
- If you need to specify this flow of control more formally, attach pre- and postconditions to each message.

A single sequence diagram can show only one flow of control (although you can show simple variations by using the UML's notation for iteration and branching). Typically, you'll have a number of interaction diagrams, some of which are primary and others that show alternative paths or exceptional conditions. You can use packages to organize these collections of sequence diagrams, giving each diagram a suitable name to distinguish it from its siblings.

For example, Figure shows a sequence diagram that specifies the flow of control involvedin initiating a simple, two-party phone call. At this level of abstraction, there are four objects involved: two **Caller**s (**s** and **r**), an unnamed telephone **Switch**, and **c**, the reification of the **Conversation** between the two parties. The sequence begins with one **Caller** (**s**) dispatchinga signal (**liftReceiver**) to the **Switch** object. In turn, the **Switch** calls **setDialTone** on the **Caller**, and the **Caller** iterates on the message**dialDigit.** Note that this message has atiming mark (**dialing**) that is used in a timing constraint (its **executionTime** must be less than 30 seconds). This diagram does not indicate what happens if this time constraint is violated. For that you could include a branch or a completely separate sequence diagram. The **Switch** object then calls itself with the message **routeCall.** It then creates a **Conversation** object (**c**), to which it delegates the rest of the work. Although not shown in this interaction, **c** would have the additional responsibility of being a party in the switch's billing mechanism (which would be expressed in another interaction diagram). The **Conversation** object (**c**) rings the **Caller** (**r**), who asynchronously sends the message **liftReceiver.** The **Conversation** object then tells the **Switch** to **connect** the call, then tells both **Caller** objects to **connect**, after which they may exchange information, as indicated by the attached note.

Figure Modeling Flows of Control by Time Ordering



An interaction diagram can begin or end at any point of a sequence. A complete trace of the flow of control would be incredibly complex, so it's reasonable to break up parts of a larger flow into separate diagrams.

Modeling Flows of Control by Organization

Consider the objects that live in the context of a system, subsystem, operation, or class. Consider also the objects and roles that participate in a use case or collaboration. To model a flow of control that winds through these objects and roles, you use an interaction diagram; to show the passing of messages in the context of that structure, you use a collaboration diagram, a kind of interaction diagram.

To model a flow of control by organization,

- Set the context for the interaction, whether it is a system, subsystem, operation, or class, or one scenario of a use case or collaboration.
- Set the stage for the interaction by identifying which objects play a role in the interaction. Lay them out on the collaboration diagram as vertices in a graph, placing the more important objects in the center of the diagram and their neighboring objects to the outside.
- Set the initial properties of each of these objects. If the attribute values, tagged values, state, or role of any object changes in significant ways over the duration of the interaction, place a duplicate object on the diagram, update it with these new values, and connect them by a message stereotyped as **become** or **copy** (with a suitable sequence number).
- Specify the links among these objects, along which messages may pass.
    1. Lay out the association links first; these are the most important ones, because they represent structural connections.
    2. Lay out other links next, and adorn them with suitable path stereotypes (such as **global** and **local**) to explicitly specify how these objects are related to oneanother.
- Starting with the message that initiates this interaction, attach each subsequent message to the appropriate link, setting its sequence number, as appropriate. Show nesting by using Dewey

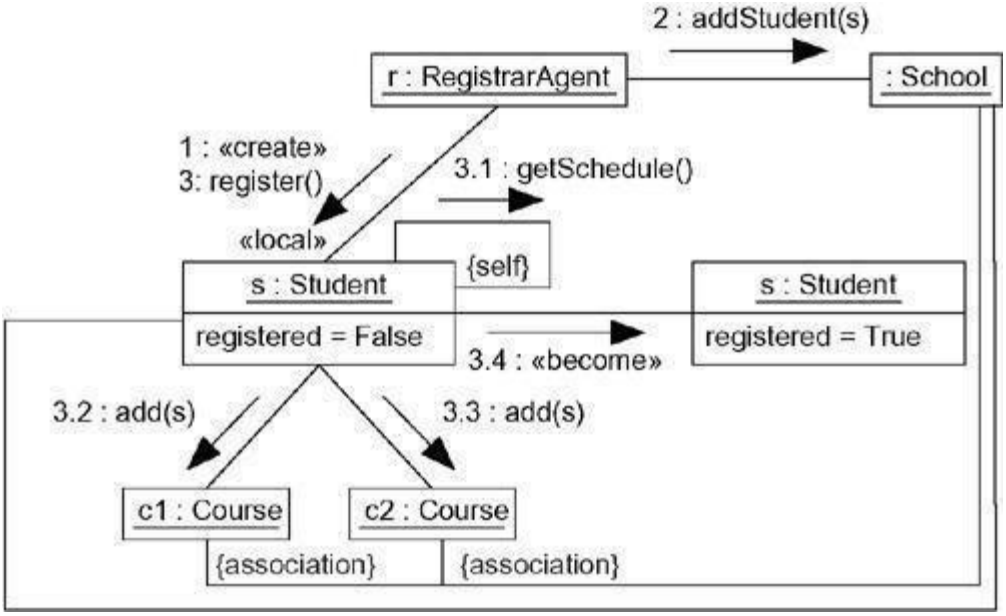
decimal numbering.

- If you need to specify time or space constraints, adorn each message with a timing mark and attach suitable time or space constraints.
- If you need to specify this flow of control more formally, attach pre- and postconditions to each message.

As with sequence diagrams, a single collaboration diagram can show only one flow of control (although you can show simple variations by using the UML's notation for interaction and branching). Typically, you'll have a number of such interaction diagrams, some of which are primary and others that show alternative paths or exceptional conditions. You can use packages to organize these collections of collaboration diagrams, giving each diagram a suitable name to distinguish it from its siblings.

For example, Figure shows a collaboration diagram that specifies the flow of control involved in registering a new student at a school, with an emphasis on the structural relationships among these objects. You see five objects: a **RegistrarAgent** (**r**), a **Student** (**s**), two **Course** objects (**c1** and **c2**), and an unnamed **School** object. The flow of control is numbered explicitly. Action begins with the **RegistrarAgent** creating a **Student** object, adding the student to the school (the message **addStudent**), then telling the **Student** object to register itself. The **Student** object then invokes**getSchedule**on itself, presumably obtaining the **Course** objectsfor which it must register. The **Student** object then adds itself to each **Course** object. The flow ends with **s** rendered again, showing that it has an updated value for its **registered** attribute.

Figure Modeling Flows of Control by Organization



Note that this diagram shows a link between the **School** object and the two **Course** objects, plus another link between the **School** object and the **Student** object, although no messages are shown along these paths. These links help explain how the **Student** object can see the two **Course** objects to which it adds itself.**s**, **c1**, and **c2** are linked to the **School** via association, so **s** can find **c1** and **c2** during its call to**getSchedule**(which might return a collection of **Course** objects), indirectly through the **School** object.

Forward and Reverse Engineering

Forward engineering (the creation of code from a model) is possible for both sequence andcollaboration diagrams, especially if the context of the diagram is an operation. For example, using the previous collaboration diagram, a reasonably clever forward engineering tool could generate the following Java code for the operation **register**, attached to the **Student** class.

```
public void register() { CourseCollection c =
    getSchedule(); for (inti = 0; i<c.size(); i++)
        c.item(i).add(this);
    this.registered = true;
}
```

"Reasonably clever" means the tool would have to realize that **getSchedule** returns a **CourseCollection**object, which it could determine by looking at the operation's signature. Bywalking across the contents of this object using a standard iteration idiom (which the tool could know about implicitly), the code could then generalize to any number of course offerings.

Reverse engineering (the creation of a model from code) is also possible for both sequence andcollaboration diagrams, especially if the context of the code is the body of an operation. Segments of the previous diagram could have been produced by a tool from a prototypical execution of the **register** operation.

When you are modeling interactions that involve multiple flows of control, it's especially important to identify the process or thread that sent a particular message. In the UML, you can distinguish one flow of control from another by prefixing a message's sequence number with the name of the process or thread that sits at the root of the sequence. For example, the expression

D5 :ejectHatch(3)

specifies that the operation **ejectHatch** is dispatched (with the actual argument **3**) as the fifth message in the sequence rooted by the process or thread named **D.**

Not only can you show the actual arguments sent along with an operation or a signal in the context of an interaction, you can show the return values of a function as well. As the following expression shows, the value **p** is returned from the operation **find,** dispatched with the actual parameter **"Rachelle".** This is a nested sequence, dispatched as the second message nested in the third message nested in the first message of the sequence. In the same diagram, **p** can then be used as an actual parameter in other messages.

Creation, Modification, and Destruction

Most of the time, the objects you show participating in an interaction exist for the entire duration of the interaction. However, in some interactions, objects may be created (specified by a **create** message) and destroyed (specified by a **destroy** message). The same is true of links: the relationships among objects may come and go. To specify if an object or link enters and/or leaves during an interaction, you can attach one of the following constraints to the element:

new	Specifies that the instance or link is created during execution of the enclosing interaction
destroyed	Specifies that the instance or link is destroyed prior to completion of execution of the enclosing interaction
transient	Specifies that the instance or link is created during execution of the enclosing interaction but is destroyed before completion of execution

During an interaction, an object typically changes the values of its attributes, its state, or its roles. You can represent the modification of an object by replicating the object in the interaction (with possibly different attribute values, state, or roles). On a sequence diagram, you'd place each variant of the object on the same lifeline. In an interaction diagram, you'd connect each variant with a **become** message.

Representation

When you model an interaction, you typically include both objects (each one playing a specific role) and messages (each one representing the communication between objects, with some resulting action).

You can visualize those objects and messages involved in an interaction in two ways: by emphasizing the time ordering of its messages, and by emphasizing the structural organization of the objects that send and receive messages. In the UML, the first kind of representation is called a sequence diagram; the second kind of representation is called a collaboration diagram. Both sequence diagrams and collaboration diagrams are kinds of interaction diagrams.

Sequence diagrams and collaboration diagrams are largely isomorphic, meaning that you can take one and transform it into the other without loss of information. There are some visual differences, however. First, sequence diagrams permit you to model the lifeline of an object. An object's lifeline represents the existence of the object at a particular time, possibly covering the object's creation and destruction. Second, collaboration diagrams permit you to model the structural links that may exist among the objects in an interaction.

Common Modeling Techniques

Modeling a Flow of Control

The most common purpose for which you'll use interactions is to model the flow of control that characterizes the behavior of a system as a whole, including use cases, patterns, mechanisms, and frameworks, or the behavior of a class or an individual operation. Whereas classes, interfaces, components, nodes, and their relationships model the static aspects of your system, interactions model its dynamic aspects.

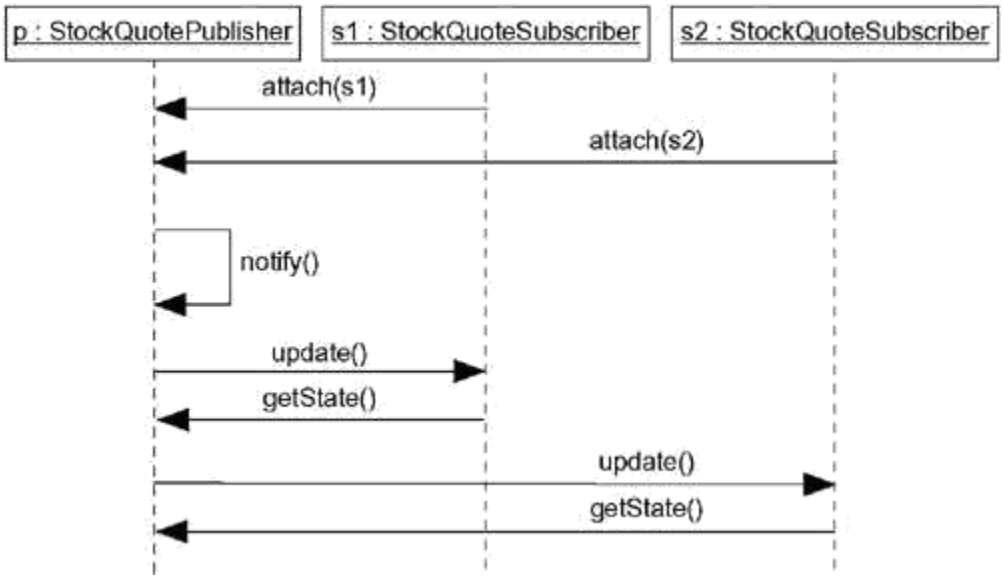
When you model an interaction, you essentially build a storyboard of the actions that take place among a set of objects. Techniques such as CRC cards are particularly useful in helping you to discover and think about such interactions.
To model a flow of control,

- Set the context for the interaction, whether it is the system as a whole, a class, or an individual operation.
- Set the stage for the interaction by identifying which objects play a role; set their initial properties, including their attribute values, state, and role.
- If your model emphasizes the structural organization of these objects, identify the links that connect them, relevant to the paths of communication that take place in this interaction. Specify the nature of the links using the UML's standard stereotypes and constraints, as necessary.
- In time order, specify the messages that pass from object to object. As necessary, distinguish the different kinds of messages; include parameters and return values to convey the necessary detail of this interaction.
- Also to convey the necessary detail of this interaction, adorn each object at every

moment in time with its state and role.

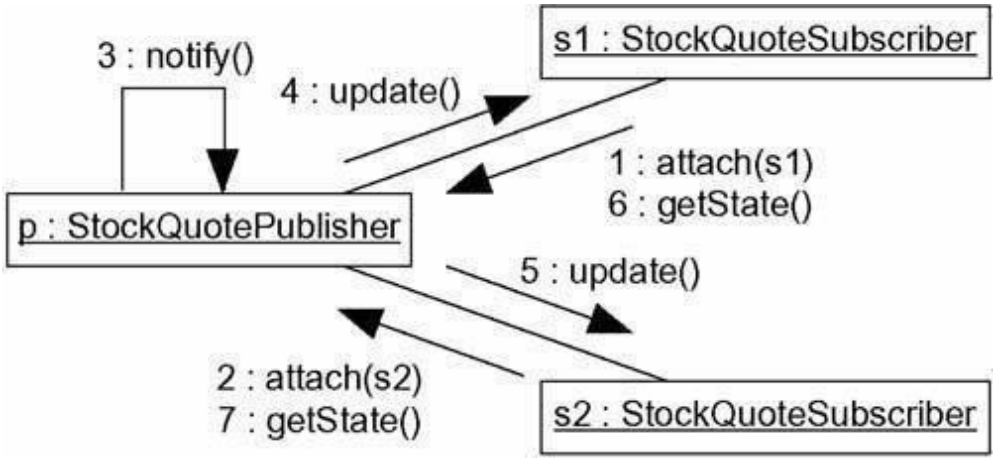
For example, Figure shows a set of objects that interact in the context of a publish and subscribe mechanism (an instance of the observer design pattern). This figure includes three objects: **p** (a **StockQuotePublisher**), **s1,** and **s2** (both instances of **StockQuoteSubscriber**). This figure is an example of a sequence diagram, which emphasizesthe time order of messages.

Figure Flow of Control by Time



Figureis semantically equivalent to the previous one, but it is drawn as a collaborationdiagram, which emphasizes the structural organization of the objects. This figure shows the same flow of control, but it also provides a visualization of the links among these objects.

Figure Flow of Control by Organization



Interaction Diagrams

Terms and Concepts

An *interaction diagram* shows an interaction, consisting of a set of objects and their relationships, including the messages that may be dispatched among them. A *sequence diagram* is an interaction diagram that emphasizes the time ordering of messages. Graphically, a sequence diagram is a table that shows objects arranged along the X axis and messages, ordered in increasing time, along the Y axis. A *collaboration diagram* is an interaction diagram that emphasizes the structural organization of the objects that send and receive messages. Graphically, a collaboration diagram is a collection of vertices and arcs.

Common Properties

An interaction diagram is just a special kind of diagram and shares the same common properties as do all other diagrams• a name and graphical contents that are a projection into a model. What distinguishes an interaction diagram from all other kinds of diagrams is its particular content.

Contents

Interaction diagrams commonly contain

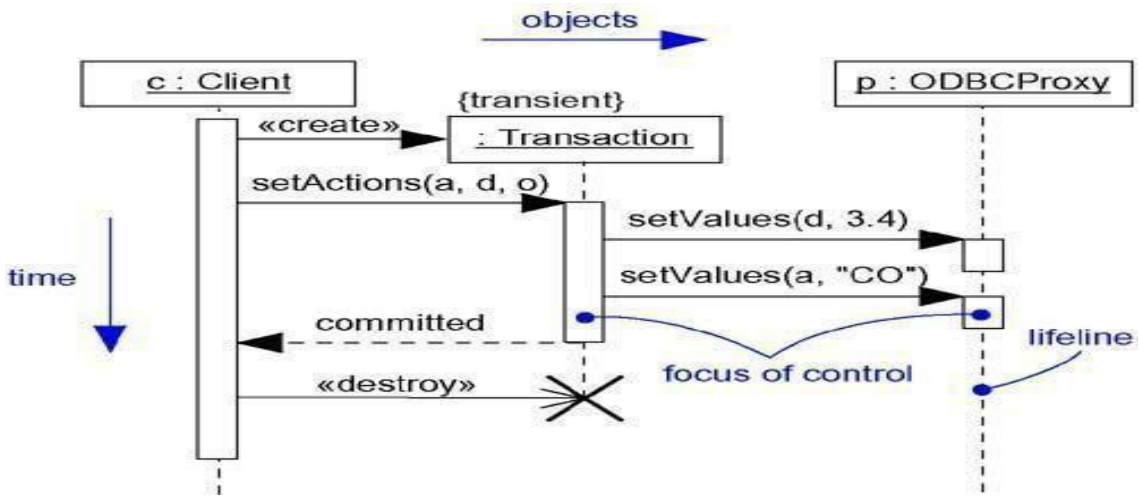
- Objects
- Links
- Messages

Like all other diagrams, interaction diagrams may contain notes and constraints.

Sequence Diagrams

A sequence diagram emphasizes the time ordering of messages. As Figure shows, you form a sequence diagram by first placing the objects that participate in the interaction at the top of your diagram, across the X axis. Typically, you place the object that initiates the interaction at the left, and increasingly more subordinate objects to the right. Next, you place the messages that these objects send and receive along the Y axis, in order of increasing time from top to bottom. This gives the reader a clear visual cue to the flow of control over time.

Figure Sequence Diagram



sequence diagrams have two features that distinguish them from collaboration diagrams.

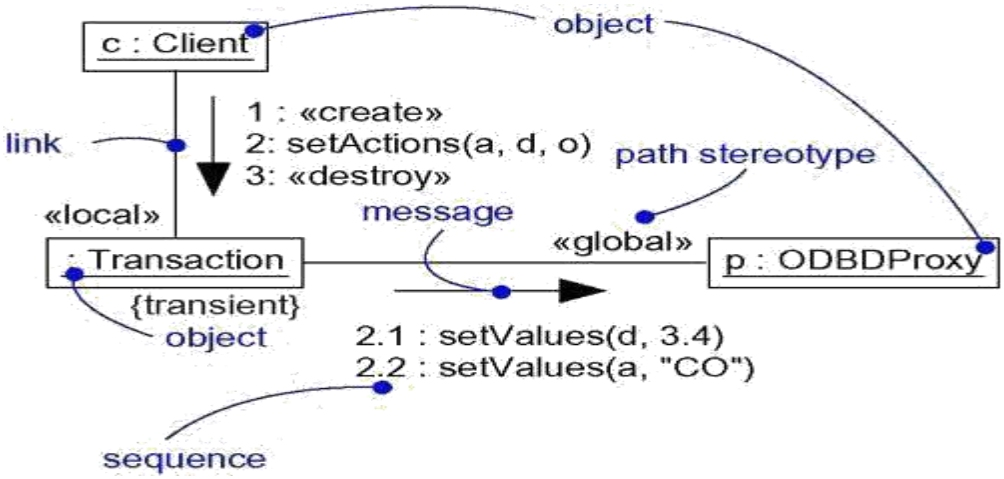
First, there is the object lifeline. An object lifeline is the vertical dashed line that represents the existence of an object over a period of time. Most objects that appear in an interaction diagram will be in existence for the duration of the interaction, so these objects are all aligned at the top of the diagram, with their lifelines drawn from the top of the diagram to the bottom. Objects may be created during the interaction. Their lifelines start with the receipt of the message stereotyped as **create.** Objects may be destroyed during the interaction. Their lifelines end with the receipt ofthe message stereotyped as **destroy** (and are given the visual cue of a large **X**, marking the end of their lives).

Second, there is the focus of control. The focus of control is a tall, thin rectangle that shows the period of time during which an object is performing an action, either directly or through a subordinate procedure. The top of the rectangle is aligned with the start of the action; the bottom is aligned with its completion (and can be marked by a return message). You can show the nesting of a focus of control (caused by recursion, a call to a self- operation, or by a callback from another object) by stacking another focus of control slightly to the right of its parent (and can do so to an arbitrary depth). If you want to be especially precise about where the focus of control lies, you can also shade the region of the rectangle during which the object's method is actually computing (and control has not passed to another object).

Collaboration Diagrams

A collaboration diagram emphasizes the organization of the objects that participate in an interaction. As Figure shows, you form a collaboration diagram by first placing the objects that participate in the interaction as the vertices in a graph. Next, you render the links that connect these objects as the arcs of this graph. Finally, you adorn these links with the messages that objects send and receive. This gives the reader a clear visual cue to the flow of control in the context of the structural organization of objects that collaborate.

Figure Collaboration Diagram



collaboration diagrams have two features that distinguish them from sequence diagrams.

First, there is the path. To indicate how one object is linked to another, you can attach a path stereotype to the far end of a link (such as **»local**, indicating that the designated object is localto the sender). Typically, you will only need to render the path of the link explicitly for **local**, **parameter**, **global**, and **self** (but not **association**) paths.

Second, there is the sequence number. To indicate the time order of a message, you prefix the message with a number (starting with the message numbered **1**), increasing monotonically for each new message in

the flow of control (**2**, **3**, and so on). To show nesting, you use Dewey decimal numbering (**1** is the first message; **1.1** is the first message nested in message **1**; **1.2** is the second message nested in message **1** ; and so on). You can show nesting to an arbitrary depth. Note also that, along the same link, you can show many messages (possibly being sent from different directions), and each will have a unique sequence number.

Most of the time, you'll model straight, sequential flows of control. However, you can also model more-complex flows, involving iteration and branching. An iteration represents a repeated sequence of messages. To model an iteration, you prefix the sequence number of a message with an iteration expression such as ***[i := 1..n]** (or just ***** if you want to indicate iteration but don't want to specify its details). An iteration indicates that the message (and any nested messages) will be repeated in accordance with the given expression. Similarly, a condition represents a message whose execution is contingent on the evaluation of a Boolean condition. To model a condition, you prefix the sequence number of a message with a condition clause, such as **[x > 0].** The alternate paths of a branch will have the same sequence number, but each path must be uniquely distinguishable by a nonoverlapping condition.

For both iteration and branching, the UML does not prescribe the format of the expression inside the brackets; you can use pseudocode or the syntax of a specific programming language.

Semantic Equivalence

Because they both derive from the same information in the UML's metamodel, sequence diagrams and collaboration diagrams are semantically equivalent. As a result, you can take a diagram in one form and convert it to the other without any loss of information, as you can see in the previous two figures, which are semantically equivalent. However, this does not mean that both diagrams will explicitly visualize the same information. For example, in the previous two figures, the collaboration diagram shows how the objects are linked (note the **»local** and **»global** stereotypes), whereas the corresponding sequence diagram does not. Similarly, thesequence diagram shows message return (note the return value **committed**), but the corresponding collaboration diagram does not. In both cases, the two diagrams share the same underlying model, but each may render some things the other does not.

Common Uses

You use interaction diagrams to model the dynamic aspects of a system. These dynamic aspects may involve the interaction of any kind of instance in any view of a system's architecture, including instances of classes (including active classes), interfaces, components, and nodes.

When you use an interaction diagram to model some dynamic aspect of a system, you do so in the context of the system as a whole, a subsystem, an operation, or a class. You can also attach interaction diagrams to use cases (to model a scenario) and to collaborations (to model the dynamic aspects of a society of objects).

When you model the dynamic aspects of a system, you typically use interaction diagrams in two ways.

1. To model flows of control by time ordering

Here you'll use sequence diagrams. Modeling a flow of control by time ordering emphasizes the passing of messages as they unfold over time, which is a particularly useful way to visualize dynamic behavior in the context of a use case scenario. Sequence diagrams do a better job of visualizing simple iteration and branching than do collaboration diagrams.

2. To model flows of control by organization

Here you'll use collaboration diagrams. Modeling a flow of control by organization emphasizes the structural relationships among the instaes in the interaction, along which messages may be assed. Collaboration diagrams do a better job of visualizing complex iteration and branching and of visualizing multiple concurrent flows of control than do sequence diagrams.

Common Modeling Techniques

Modeling Flows of Control by Time Ordering

Consider the objects that live in the context of a system, subsystem, operation or class. Consider also the objects and roles that participate in a use case or collaboration. To model a flow of control that winds through these objects and roles, you use an interaction diagram; to emphasize the passing of messages as they unfold over time, you use a sequence diagram, a kind of interaction diagram.

To model a flow of control by time ordering,

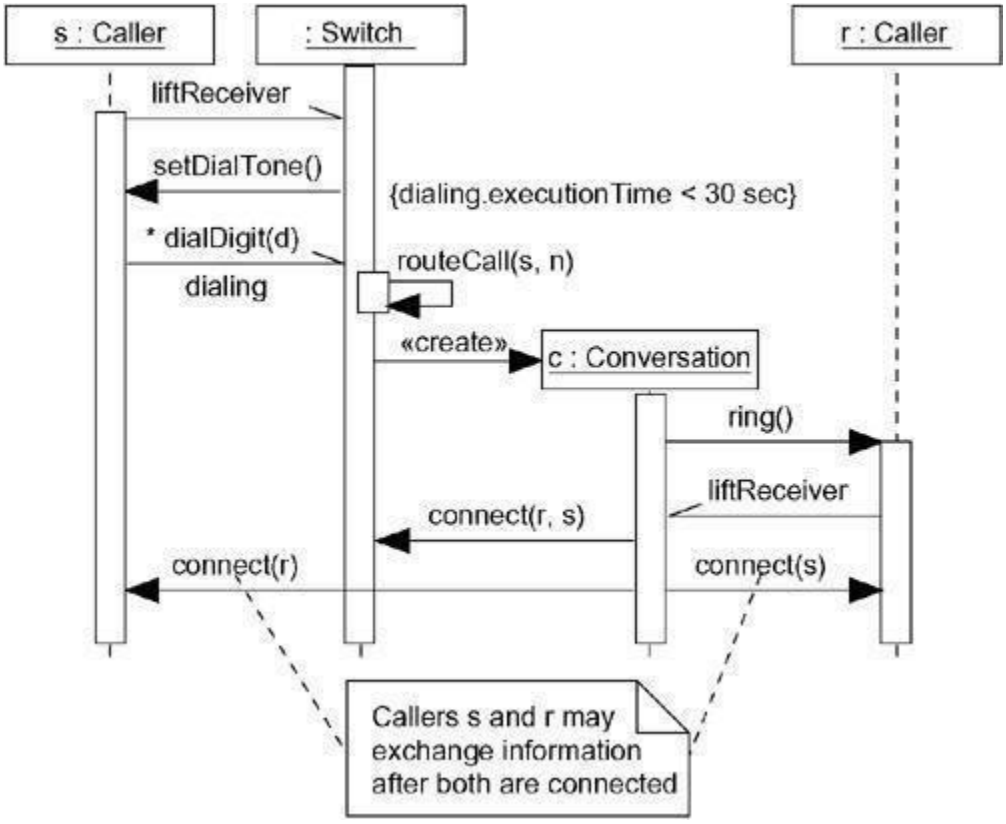
- Set the context for the interaction, whether it is a system, subsystem, operation, or class, or one scenario of a use case or collaboration.
- Set the stage for the interaction by identifying which objects play a role in the interaction. Lay them out on the sequence diagram from left to right, placing the more important objects to the left and their neighboring objects to the right.
- Set the lifeline for each object. In most cases, objects will persist through the entire interaction. For those objects that are created and destroyed during the interaction, set their lifelines, as appropriate, and explicitly indicate their birth and death with appropriately stereotyped messages.
- Starting with the message that initiates this interaction, lay out each subsequent message from top to bottom between the lifelines, showing each message's properties (such as its parameters), as necessary to explain the semantics of the interaction.
- If you need to visualize the nesting of messages or the points in time when actual computation is taking place, adorn each object's lifeline with its focus of control.
- If you need to specify time or space constraints, adorn each message with a timing mark and attach suitable time or space constraints.
- If you need to specify this flow of control more formally, attach pre- and postconditions to each message.

A single sequence diagram can show only one flow of control (although you can show simple variations by using the UML's notation for iteration and branching). Typically, you'll have a number of interaction diagrams, some of which are primary and others that show alternative paths or exceptional conditions. You can use packages to organize these collections of sequence diagrams, giving each diagram a suitable name to distinguish it from its siblings.

For example, Figure shows a sequence diagram that specifies the fslow of control involvedin initiating a simple, two-party phone call. At this level of abstraction, there are four objects involved: two **Caller**s (**s** and **r**), an unnamed telephone **Switch**, and **c**, the reification of the **Conversation** between the two parties. The sequence begins with one **Caller** (**s**) dispatchinga signal (**liftReceiver**) to the **Switch** object. In turn, the **Switch** calls **setDialTone** on the **Caller**, and the **Caller** iterates on the message**dialDigit.** Note that this message has atiming mark (**dialing**) that is used in a timing constraint (its **executionTime** must be less than 30 seconds). This diagram does not indicate what happens if this time constraint is violated. For that you could include a branch or a completely separate sequence diagram. The **Switch** object then calls itself with the message **routeCall.** It then creates a **Conversation** object (**c**), to which it delegates the rest of the work. Although not shown in this interaction, **c** would have the additional responsibility of being a party in the switch's billing mechanism (which would be expressed in another interaction diagram). The **Conversation** object (**c**) rings the **Caller** (**r**), who asynchronously sends the message **liftReceiver.** The **Conversation** object then tells the **Switch** to **connect** the call, then tells both **Caller** objects to **connect**,

after which they may exchange information, as indicated by the attached note.

Figure Modeling Flows of Control by Time Ordering



An interaction diagram can begin or end at any point of a sequence. A complete trace of the flow of control would be incredibly complex, so it's reasonable to break up parts of a larger flow into separate diagrams.

Modeling Flows of Control by Organization

Consider the objects that live in the context of a system, subsystem, operation, or class. Consider also the objects and roles that participate in a use case or collaboration. To model a flow of control that winds through these objects and roles, you use an interaction diagram; to show the passing of messages in the context of that structure, you use a collaboration diagram, a kind of interaction diagram.

To model a flow of control by organization,

- Set the context for the interaction, whether it is a system, subsystem, operation, or class, or one scenario of a use case or collaboration.
- Set the stage for the interaction by identifying which objects play a role in the interaction. Lay them out on the collaboration diagram as vertices in a graph, placing the more important objects in the center of the diagram and their neighboring objects to the outside.
- Set the initial properties of each of these objects. If the attribute values, tagged values, state, or role of any object changes in significant ways over the duration of the interaction, place a duplicate object on the diagram, update it with these new values, and connect them by a message

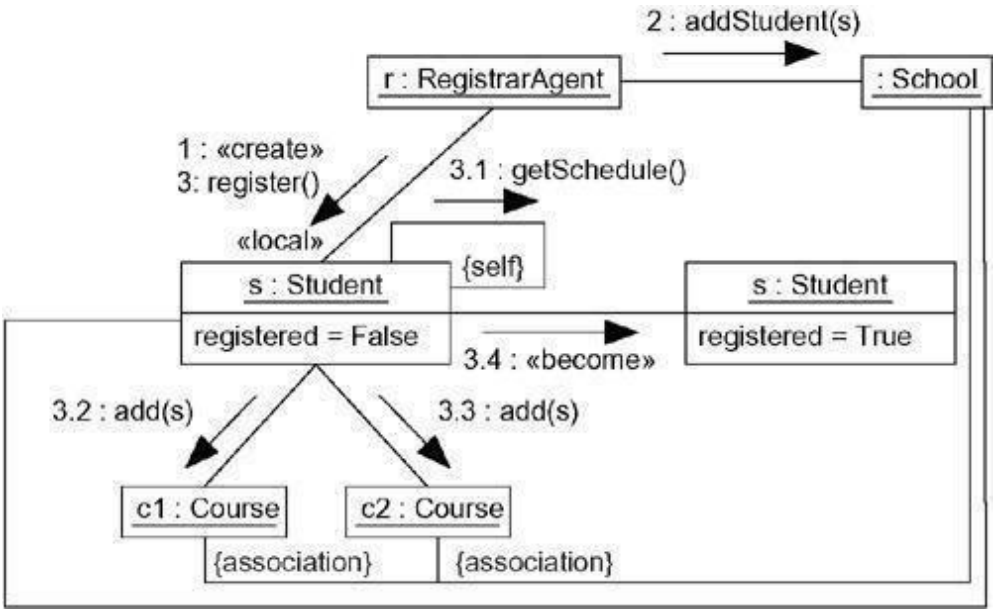
stereotyped as **become** or **copy** (with a suitable sequence number).

- Specify the links among these objects, along which messages may pass.
    1. Lay out the association links first; these are the most important ones, because they represent structural connections.
    2. Lay out other links next, and adorn them with suitable path stereotypes (such as **global** and **local**) to explicitly specify how these objects are related to oneanother.
- Starting with the message that initiates this interaction, attach each subsequent message to the appropriate link, setting its sequence number, as appropriate. Show nesting by using Dewey decimal numbering.
- If you need to specify time or space constraints, adorn each message with a timing mark and attach suitable time or space constraints.
- If you need to specify this flow of control more formally, attach pre- and postconditions to each message.

As with sequence diagrams, a single collaboration diagram can show only one flow of control (although you can show simple variations by using the UML's notation for interaction and branching). Typically, you'll have a number of such interaction diagrams, some of which are primary and others that show alternative paths or exceptional conditions. You can use packages to organize these collections of collaboration diagrams, giving each diagram a suitable name to distinguish it from its siblings.

For example, Figure shows a collaboration diagram that specifies the flow of control involved in registering a new student at a school, with an emphasis on the structural relationships among these objects. You see five objects: a **RegistrarAgent** (**r**), a **Student** (**s**), two **Course** objects (**c1** and **c2**), and an unnamed **School** object. The flow of control is numbered explicitly. Action begins with the **RegistrarAgent** creating a **Student** object, adding the student to the school (the message **addStudent**), then telling the **Student** object to register itself. The **Student** object then invokes**getSchedule**on itself, presumably obtaining the **Course** objectsfor which it must register. The **Student** object then adds itself to each **Course** object. The flow ends with **s** rendered again, showing that it has an updated value for its **registered** attribute.

Figure Modeling Flows of Control by Organization



Note that this diagram shows a link between the **School** object and the two **Course** objects, plus another link between the **School** object and the **Student** object, although no messages are shown along these paths. These links help explain how the **Student** object can see the two **Course** objects to which it adds itself.**s**, **c1**, and **c2** are linked to the **School** via association, so **s** can find **c1** and **c2** during its call to**getSchedule**(which might return a collection of **Course** objects), indirectly through the **School** object.

Forward and Reverse Engineering

Forward engineering (the creation of code from a model) is possible for both sequence andcollaboration diagrams, especially if the context of the diagram is an operation. For example, using the previous collaboration diagram, a reasonably clever forward engineering tool could generate the following Java code for the operation **register**, attached to the **Student** class.

```
public void register() { CourseCollection c =
    getSchedule(); for (inti = 0; i<c.size(); i++)
        c.item(i).add(this);
    this.registered = true;
}
```

"Reasonably clever" means the tool would have to realize that **getSchedule** returns a **CourseCollection**object, which it could determine by looking at the operation's signature. Bywalking across the contents of this object using a standard iteration idiom (which the tool could know about implicitly), the code could then generalize to any number of course offerings.

Reverse engineering (the creation of a model from code) is also possible for both sequence andcollaboration diagrams, especially if the context of the code is the body of an operation. Segments of the previous diagram could have been produced by a tool from a prototypical execution of the **register** operation.

Use Cases

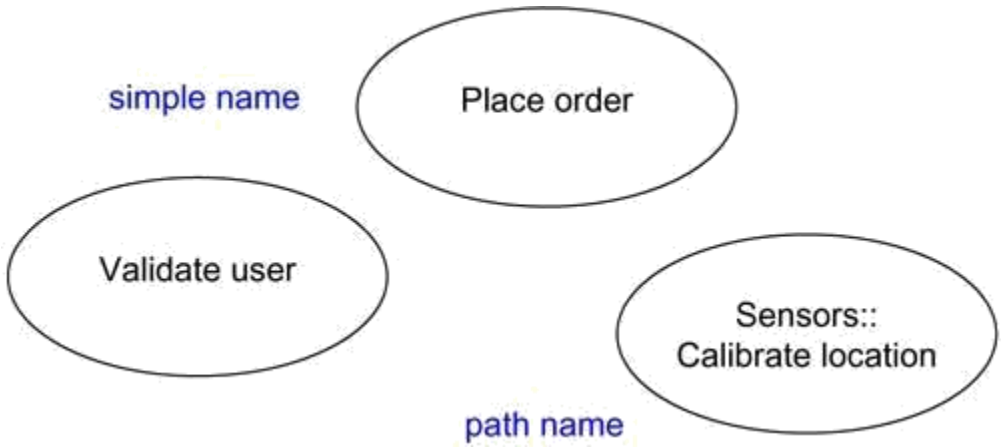
Terms and Concepts

A *use case* is a description of a set of sequences of actions, including variants, that a system performs to yield an observable result of value to an actor. Graphically, a use case is rendered as an ellipse.

Names

Every use case must have a name that distinguishes it from other use cases. A *name* is a textual string. That name alone is known as a *simple name;* a *path name* is the use case name prefixed by the name of the package in which that use case lives. A use case is typically drawn showing only its name, as in Figure.

Figure Simple and Path Names



Note

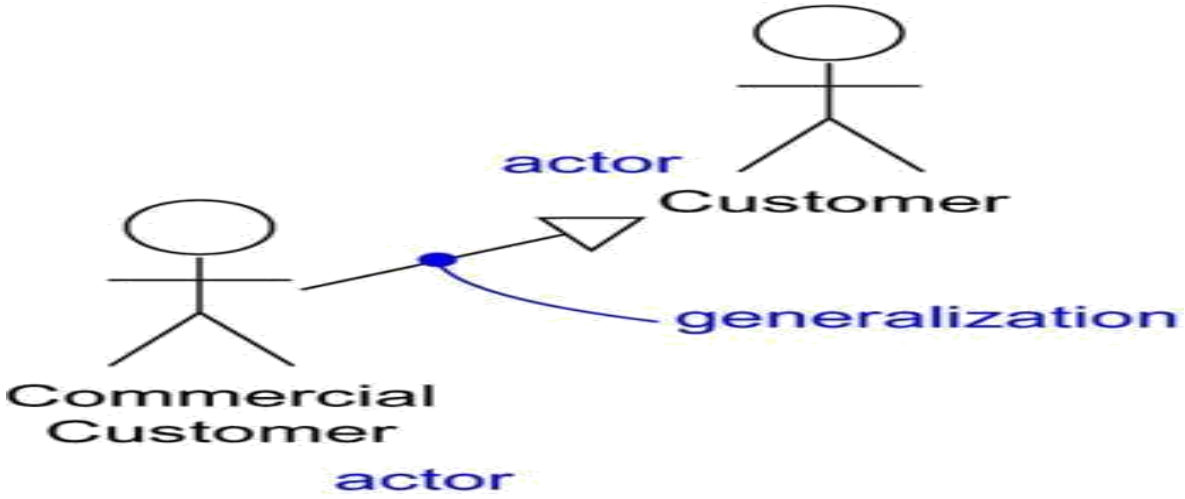
A use case name may be text consisting of any number of letters, numbers, and most punctuation marks (except for marks such as the colon, which is used to separate a class name and the name of its enclosing package) and may continue over several lines. In practice, use case names are short active verb phrases naming some behavior found in the vocabulary of the system you are modeling.

Use Cases and Actors

An actor represents a coherent set of roles that users of use cases play when interacting with these use cases. Typically, an actor represents a role that a human, a hardware device, or even another system plays with a system. For example, if you work for a bank, you might be a **LoanOfficer.** If you do your personal banking there, as well, you'll also play the role of **Customer.** An instance of an actor, therefore, represents an individual interacting with thesystem in a specific way. Although you'll use actors in your models, actors are not actually part of the system. They live outside the system.

As Figure indicates, actors are rendered as stick figures. You can define general kinds of actors (such as **Customer**) and specialize them (such as **CommercialCustomer**) using generalization relationships.

Figure Actors



Actors may be connected to use cases only by association. An association between an actor and a use case indicates that the actor and the use case communicate with one another, each one possibly sending and receiving messages.

Use Cases and Flow of Events

A use case describes *what* a system (or a subsystem, class, or interface) does but it does not specify *how* it does it. When you model, it's important that you keep clear the separation of concerns between this outside and inside view.

You can specify the behavior of a use case by describing a flow of events in text clearly enough for an outsider to understand it easily. When you write this flow of events, you should include how and when the use case starts and ends, when the use case interacts with the actors and what objects are exchanged, and the basic flow and alternative flows of the behavior.

For example, in the context of an ATM system, you might describe the use case **ValidateUser** in the following way:

> **Main flow of events:**
>
> The use case starts when the system prompts the *Customer* for a PIN number. The *Customer* can now enter a PIN number via the keypad. The *Customer* commits the entry by pressing the Enter button. The system then checks this PIN number to see if it is valid. If the PIN number is valid, the system acknowledges the entry, thus ending the use case.
>
> **Exceptional flow of events:**
>
> The *Customer* can cancel a transaction at any time by pressing the Cancel button, thus restarting the use case. No changes are made to the *Customer*'s account.
>
> **Exceptional flow of events:**

The *Customer* can clear a PIN number anytime before committing it and reenter a new PIN number.

Exceptional flow of events:

If the *Customer* enters an invalid PIN number, the use case restarts. If this happens three times in a row, the system cancels the entire transaction, preventing the *Customer* from interacting with the ATM for 60 seconds.

Use Cases and Scenarios

Typically, you'll first describe the flow of events for a use case in text. As you refine your understanding of your system's requirements, however, you'll want to also use interaction diagrams to specify these flows graphically. Typically, you'll use one sequence diagram to specify a use case's main flow, and variations of that diagram to specify a use case's exceptional flows.

It is desirable to separate main versus alternative flows because a use case describes a set of sequences, not just a single sequence, and it would be impossible to express all the details of an interesting use case in just one sequence. For example, in a human resources system, you might find the use case **Hire employee.** This general business function might have many possible variations. You might hire a person from another company (the most common scenario); you might transfer a person from one division to another (common in international companies); or you might hire a foreign national (which involves its own special rules). Each of these variants can be expressed in a different sequence.

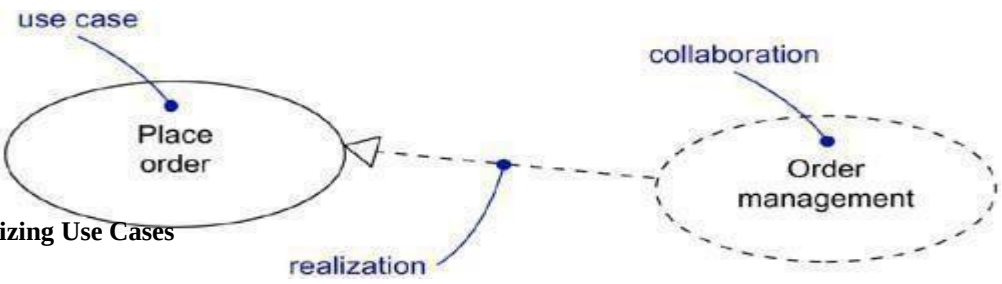
This one use case (**Hire employee**) actually describes a set of sequences in which each sequence in the set represents one possible flow through all these variations. Each sequence is called a scenario. A scenario is a specific sequence of actions that illustrates behavior. Scenarios are to use cases as instances are to classes, meaning that a scenario is basically one instance of a use case.

Use Cases and Collaborations

A use case captures the intended behavior of the system (or subsystem, class, or interface) you are developing, without having to specify how that behavior is implemented. That's an important separation because the analysis of a system (which specifies behavior) should, as much as possible, not be influenced by implementation issues (which specify how that behavior is to be carried out). Ultimately, however, you have to implement your use cases, and you do so by creating a society of classes and other elements that work together to implement the behavior of this use case. This society of elements, including both its static and dynamic structure, is modeled in the UML as a collaboration.

As Figure shows, you can explicitly specify the realization of a use case by a collaboration. Most of the time, though, a given use case is realized by exactly one collaboration, so you will not need to model this relationship explicitly.

Figure Use Cases and Collaborations



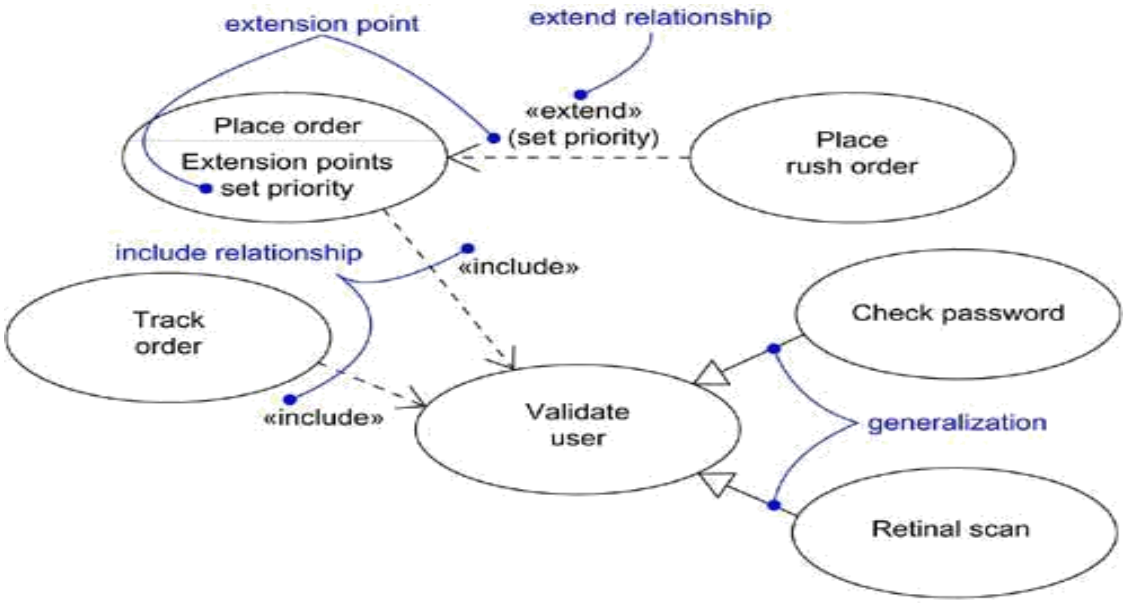
Organizing Use Cases

You can organize use cases by grouping them in packages in the same manner in which you can organize classes.

You can also organize use cases by specifying generalization, include, and extend relationships among them. You apply these relationships in order to factor common behavior (by pulling such behavior from other use cases that it includes) and in order to factor variants (by pushing such behavior into other use cases that extend it).

Generalization among use cases is just like generalization among classes. Here it means that the child use case inherits the behavior and meaning of the parent use case; the child may add to or override the behavior of its parent; and the child may be substituted any place the parent appears (both the parent and the child may have concrete instances). For example, in a banking system, you might have the use case **Validate User**, which is responsible for verifying the identify of the user. You might then have two specialized children of this use case (**Check password** and **Retinal scan**), both of which behave just like **Validate User** and may be applied anywhere **Validate User** appears, yet both of which add their own behavior (the former by checking atextual password, the latter by checking the unique retina patterns of the user). As shown in Figure , generalization among use cases is rendered as a solid directed line with a largeopen arrowhead, just like generalization among classes.

Figure Generalization, Include, and Extend



In include relationship between use cases means that the base use case explicitly incorporates the behavior of another use case at a location specified in the base. The included use case never stands alone, but is only instantiated as part of some larger base that includes it. You can think of include as the base use case pulling behavior from the supplier use case.

You use an include relationship to avoid describing the same flow of events several times, by putting the common behavior in a use case of its own (the use case that is included by a base use case). The include relationship is essentially an example of delegation• you take a set of responsibilities of the system and capture it in one place (the included use case), then let all other parts of the system (other use cases) include the new aggregation of responsibilities whenever they need to use that functionality.

You render an include relationship as a dependency, stereotyped as **include.** To specify the location in a flow of events in which the base use case includes the behavior of another, you simply write **include** followed by the name of the use case you want to include, as in the following flow for **Track order.**

> **Main flow of events:**
>
> Obtain and verify the order number. **include** (**Validate user**). For each part in the order, query its status, then report back to the user.

An extend relationship between use cases means that the base use case implicitly incorporates the behavior of another use case at a location specified indirectly by the extending use case. The base use case may stand alone, but under certain conditions, its behavior may be extended by the behavior of another use case. This base use case may be extended only at certain points called, not surprisingly, its extension points. You can think of extend as the extension use case pushing behavior to the base use case You use an extend relationship to model the part of a use case the user may see as optional system behavior. In this way, you separate optional behavior from mandatory behavior. You may also use an extend relationship to model a separate subflow that is executed only under given conditions. Finally, you may use an extend relationship to model several flows that may be inserted at a certain point, governed by explicit interaction with an actor.

You render an extend relationship as a dependency, stereotyped as **extend.** You may list the extension points of the base use case in an extra compartment. These extension points are just labels that may appear in the flow of the base use case. For example, the flow for **Place order** might read as follows:

> **Main flow of events:**
>
> **include**(**Validate user**). Collect the user's order items. **(set priority).**
> Submit the order for processing.

In this example, **set priority** is an extension point. A use case may have more than one extension point (which may appear more than once), and these are always matched by name. Under normal circumstances, this base use case will execute without regard for the priority of the order. If, on the other hand, this is an instance of a priority order, the flow for this base case will carry out as above. But at the extension point **(set priority)**, the behavior of the extending use case (**Place rush order**) will be performed, then the flow will resume. If there are multiple extension points, the extending use case will simply fold in its flows in order.

Other Features

Use cases are classifiers, so they may have attributes and operations that you may render just as for classes. You can think of these attributes as the objects inside the use case that you need to describe its outside behavior. Similarly, you can think of these operations as the actions of the system you need to describe a flow of events. These objects and operations may be used in your interaction diagrams to specify the behavior of the use case.As classifiers, you can also attach state machines to use cases. You can use state machines as yet another way to describe the behavior represented by a use case.

Common Modeling Techniques

Modeling the Behavior of an Element

The most common thing for which you'll apply use cases is to model the behavior of an element, whether it is the system as a whole, a subsystem, or a class. When you model the behavior of these things, it's important that you focus on what that element does, not how it does it.

Applying use cases to elements in this way is important for three reasons. First, by modeling the behavior

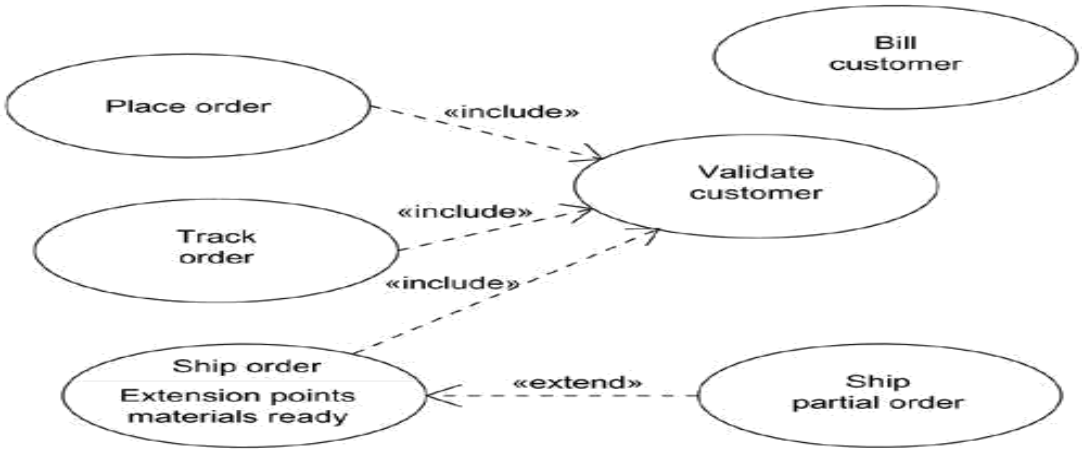
of an element with use cases, you provide a way for domain experts to specify its outside view to a degree sufficient for developers to construct its inside view. Use cases provide a forum for your domain experts, end users, and developers to communicate to one another. Second, use cases provide a way for developers to approach an element and understand it. A system, subsystem, or class may be complex and full of operations and other parts. By specifying an element's use cases, you help users of these elements to approach them in a direct way, according to how they are likely to use them. In the absence of such use cases, users have to discover on their own how to use those elements. Use cases let the author of an element communicate his or her intent about how that element should be used. Third, use cases serve as the basis for testing each element as it evolves during development. By continuously testing each element against its use cases, you continuously validate its implementation. Not only do these use cases provide a source of regression tests, but every time you throw a new use case at an element, you are forced to reconsider your implementation to ensure that this element is resilient to change. If it is not, you must fix your architecture appropriately.

To model the behavior of an element,

- Identify the actors that interact with the element. Candidate actors include groups that require certain behavior to perform their tasks or that are needed directly or indirectly to perform the element's functions.
- Organize actors by identifying general and more specialized roles.
- For each actor, consider the primary ways in which that actor interacts with the element. Consider also interactions that change the state of the element or its environment or that involve a response to some event.
- Consider also the exceptional ways in which each actor interacts with the element.
- Organize these behaviors as use cases, applying include and extend relationships to factor common behavior and distinguish exceptional behavior.

For example, a retail system will interact with customers who place and track orders. In turn, the system will ship orders and bill the customer. As Figure shows, you can model the behavior of such a system by declaring these behaviors as use cases (**Place order**, **Trackorder**, **Ship order**, and **Bill customer**). Common behavior can be factored out (**Validate custome**r) and variants (**Ship partial order**) can be distinguished, as well. For each ofthese use cases, you would include a specification of the behavior, either by text, state machine, or interactions.

Figure Modeling the Behavior of an Element



Use Case Diagrams

Terms and Concepts

A *use case diagram* is a diagram that shows a set of use cases and actors and their relationships.

Common Properties

A use case diagram is just a special kind of diagram and shares the same common properties as do all other diagrams• a name and graphical contents that are a projection into a model. What distinguishes a use case diagram from all other kinds of diagrams is its particular content.

Contents

Use case diagrams commonly contain

- Use cases
- Actors
- Dependency, generalization, and association relationships

Like all other diagrams, use case diagrams may contain notes and constraints.
Use case diagrams may also contain packages, which are used to group elements of your model into larger chunks. Occasionally, you'll want to place instances of use cases in your diagrams, as well, especially when you want to visualize a specific executing system.

Common Uses
You apply use case diagrams to model the static use case view of a system. This view primarily supports the behavior of a system• the outwardly visible services that the system provides in the context of its environment.

When you model the static use case view of a system, you'll typically apply use case diagrams in one of two ways.

1. To model the context of a system

Modeling the context of a system involves drawing a line around the whole system and asserting which actors lie outside the system and interact with it.Here, you'll apply use case diagrams to specify the actors and the meaning of their roles.

2. To model the requirements of a system

Modeling the requirements of a system involves specifying what that system should do (from a point of view of outside the system), independent of how that system should do it. Here, you'll apply use case diagrams to specify the desired behavior of the system. In this manner, a use case diagram lets you view the whole system as a black box; you can see what's outside the system and you can see how that system reacts to the things outside, but you can't see how that system works on the inside.

Common Modeling Techniques

Modeling the Context of a System

Given a system• any system• some things will live inside the system, some things will live outside it. For example, in a credit card validation system, you'll find such things as accounts, transactions, and fraud detection agents inside the system. Similarly, you'll find such things as credit card customers and retail institutions outside the system. The things that live inside the system are responsible for carrying out the behavior that those on the outside expect the system to provide. All those things on the outside that

interact with the system constitute the system's context. This context defines the environment in which that system lives.

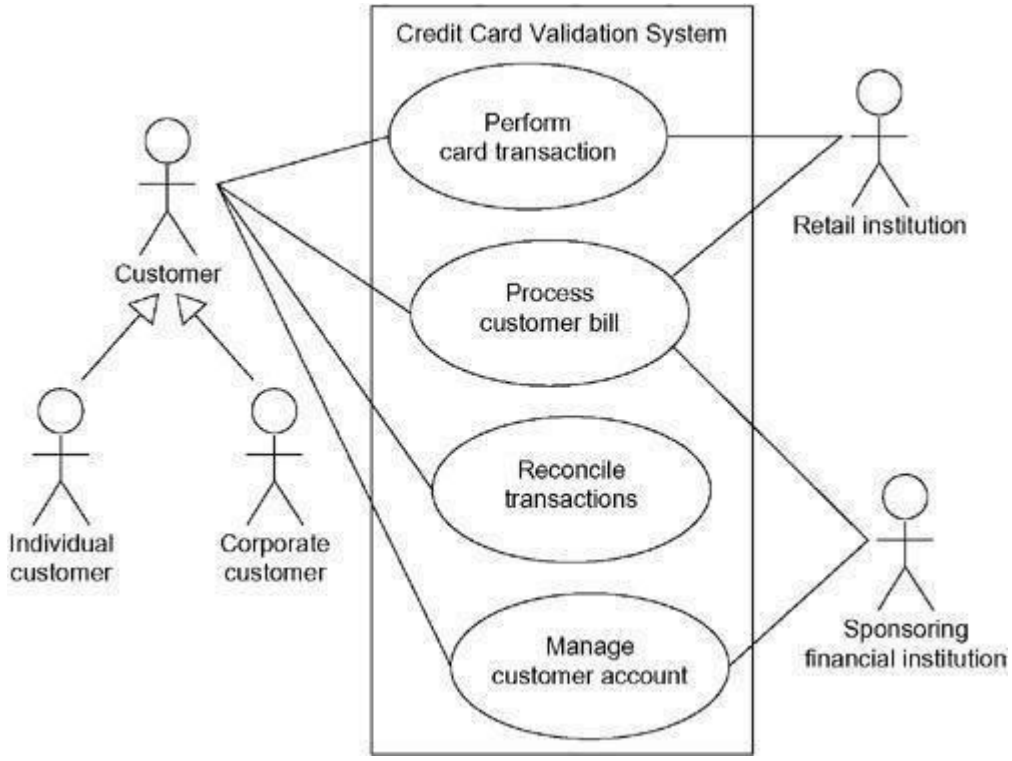
In the UML, you can model the context of a system with a use case diagram, emphasizing the actors that surround the system. Deciding what to include as an actor is important because in doing so you specify a class of things that interact with the system. Deciding what not to include as an actor is equally, if not more, important because that constrains the system's environment to include only those actors that are necessary in the life of the system.

To model the context of a system,

- Identify the actors that surround the system by considering which groups require help from the system to perform their tasks; which groups are needed to execute the system's functions; which groups interact with external hardware or other software systems; and which groups perform secondary functions for administration and maintenance.
- Organize actors that are similar to one another in a generalization/specialization hierarchy.
- Where it aids understandability, provide a stereotype for each such actor.
- Populate a use case diagram with these actors and specify the paths of communication from each actor to the system's use cases.

For example, Figure shows the context of a credit card validation system, with anemphasis on the actors that surround the system. You'll find **Customers,** of which there are two kinds (**Individual customer** and **Corporate customer**). These actors are the roles that humans play when interacting with the system. In this context, there are also actors that represent other institutions, such as **Retail institution** (with which a **Customer** performs a card transaction to buy an item or a service) and **Sponsoring financial institution** (which serves as the clearinghouse for the credit card account). In the real world, these latter two actors are likely software-intensive systems themselves.

Figure Modeling the Context of a System



This same technique applies to modeling the context of a subsystem. A system at one level of abstraction is often a subsystem of a larger system at a higher level of abstraction. Modeling the context of a subsystem is therefore useful when you are building systems of interconnected systems.

Modeling the Requirements of a System

A requirement is a design feature, property, or behavior of a system. When you state a system's requirements, you are asserting a contract, established between those things that lie outside the system and the system itself, which declares what you expect that system to do. For the most part, you don't care how the system does it, you just care *that* it does it. A well-behaved system will carry out all its requirements faithfully, predictably, and reliably. When you build a system, it's important to start with agreement about what that system should do, although you will certainly evolve your understanding of those requirements as you iteratively and incrementally implementthe system. Similarly, when you are handed a system to use, knowing how it behaves is essential to using it properly.

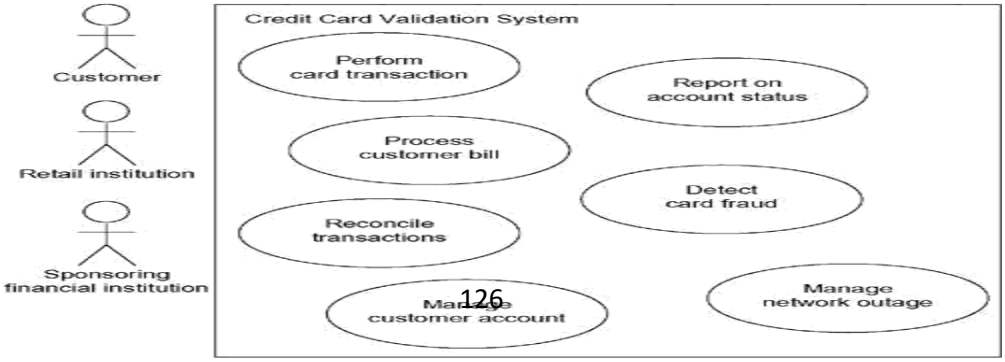
Requirements can be expressed in various forms, from unstructured text to expressions in a formal language, and everything in between. Most, if not all, of a system's functional requirements can be expressed as use cases, and the UML's use case diagrams are essential for managing these requirements.

To model the requirements of a system,

- Establish the context of the system by identifying the actors that surround it.
- For each actor, consider the behavior that each expects or requires the system to provide.
- Name these common behaviors as use cases.
- Factor common behavior into new use cases that are used by others; factor variant behavior into new use cases that extend more main line flows.
- Model these use cases, actors, and their relationships in a use case diagram.
- Adorn these use cases with notes that assert nonfunctional requirements; you may have to attach some of these to the whole system.

Figure expands on the previous use case diagram. Although it elides the relationshipsamong the actors and the use cases, it adds additional use cases that are somewhat invisible to the average customer, yet are essential behaviors of the system. This diagram is valuable because it offers a common starting place for end users, domain experts, and developers to visualize, specify, construct, and document their decisions about the functional requirements of this system. For example, **Detect card fraud** is a behavior important to both the **Retailinstitution** and the **Sponsoring financial institution.** Similarly, **Report on account status** is another behavior required of the system by the various institutions in itscontext.

Figure Modeling the Requirements of a System



The requirement modeled by the use case **Manage network outage**is a bit different from allthe others because it represents a secondary behavior of the system necessary for its reliable and continuous operation.

This same technique applies to modeling the requirements of a subsystem.

Forward and Reverse Engineering

Most of the UML's other diagrams, including class, component, and statechart diagrams, are clear candidates for forward and reverse engineering because each has an analog in the executable system. Use case diagrams are a bit different in that they reflect rather than specify the implementation of a system, subsystem, or class. Use cases describe how an element behaves, not how that behavior is implemented, so it cannot be directly forward or reverse engineered.

Forward engineering is the process of transforming a model into code through a mapping to animplementation language. A use case diagram can be forward engineered to form tests for the element to which it applies. Each use case in a use case diagram specifies a flow of events (and variants of those flows), and these flows specify how the element is expected to behave• that's something worthy of testing. A well-structured use case will even specify pre- and postconditions that can be used to define a test's initial state and its success criteria. For each use case in a use case diagram, you can create a test case that you can run every time you release a new version of that element, thereby confirming that it works as required before other elements rely on it.

To forward engineer a use case diagram,

- For each use case in the diagram, identify its flow of events and its exceptional flow of events.
- Depending on how deeply you choose to test, generate a test script for each flow, using the flow's preconditions as the test's initial state and its postconditions as its success criteria.
- As necessary, generate test scaffolding to represent each actor that interacts with the use case. Actors that push information to the element or are acted on by the element may either be simulated or substituted by its real-world equivalent.
- Use tools to run these tests each time you release the element to which the use case diagram applies.

Reverse engineering is the process of transforming code into a model through a mapping from aspecific implementation language. Automatically reverse engineering a use case diagram is pretty much beyond the state of the art, simply because there is a loss of information when moving from a specification of how an element behaves to how it is implemented. However, you can study an existing system and discern its intended behavior by hand, which you can then put in the form of a use case diagram. Indeed, this is pretty much what you have to do anytime you are handed an undocumented body of software. The UML's use case diagrams simply give you a standard and expressive language in which to state what you discover.

To reverse engineer a use case diagram,

- Identify each actor that interacts with the system.
- For each actor, consider the manner in which that actor interacts with thesystem, changes the state of the system or its environment, or responds to some event.
- Trace the flow of events in the executable system relative to each actor. Start with primary flows and only later consider alternative paths.
- Cluster related flows by declaring a corresponding use case. Consider modeling variants using

extend relationships, and consider modeling common flows by applying include relationships.

- Render these actors and use cases in a use case diagram, and establish their relationships.

Activity Diagrams

Terms and Concepts

An *activity diagram* shows the flow from activity to activity. An is an ongoing nonatomic execution within a state machine. Activities ultimately result in some *action*, which is made up of executable atomic computations that result in a change in state of the system or the return of a value. Actions encompass calling another operation, sending a signal, creating or destroying an object, or some pure computation, such as evaluating an expression. Graphically, an activity diagram is a collection of vertices and arcs.

Common Properties

An activity diagram is just a special kind of diagram and shares the same common properties as do all other diagrams• a name and graphical contents that are a projection into a model. What distinguishes an interaction diagram from all other kinds of diagrams is its content.

Contents

Activity diagrams commonly contain

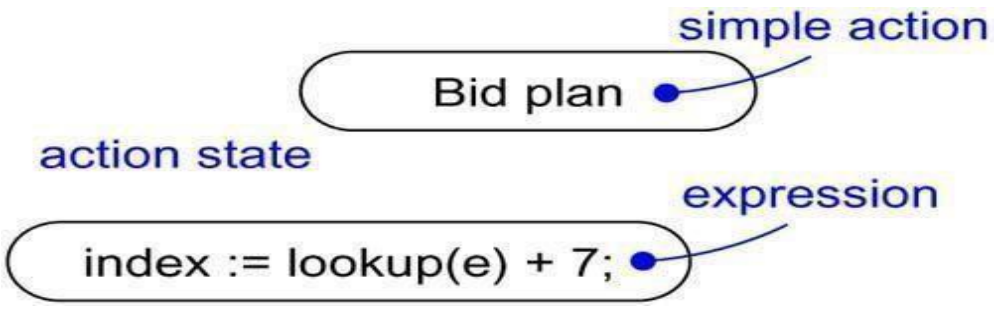
- Activity states and action states
- Transitions
- Objects

Like all other diagrams, activity diagrams may contain notes and constraints.

Action States and Activity States

In the flow of control modeled by an activity diagram, things happen. You might evaluate some expression that sets the value of an attribute or that returns some value. Alternately, you might call an operation on an object, send a signal to an object, or even create or destroy an object.These executable, atomic computations are called action states because they are states of the system, each representing the execution of an action. As Figure shows, you represent an action state using a lozenge shape (a symbol with horizontal top and bottom and convex sides). Inside that shape, you may write any expression.

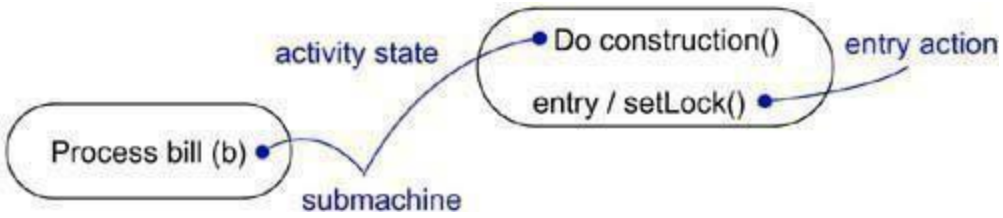
Figure Action States



Action states can't be decomposed. Furthermore, action states are atomic, meaning that events may occur, but the work of the action state is not interrupted. Finally, the work of an action state is generally considered to take insignificant execution time.

In contrast, activity states can be further decomposed, their activity being represented by other activity diagrams. Furthermore, activity states are not atomic, meaning that they may be interrupted and, in general, are considered to take some duration to complete. You can think of an action state as a special case of an activity state. An action state is an activity state that cannot be further decomposed. Similarly, you can think of an activity state as a composite, whose flow of control is made up of other activity states and action states. Zoom into the details of an activity state, and you'll find another activity diagram. As Figure shows, there's no notational distinction between action and activity states, except that an activity state may have additional parts, such as entry and exit actions (actions which are involved on entering and leaving the state, respectively) and submachine specifications.

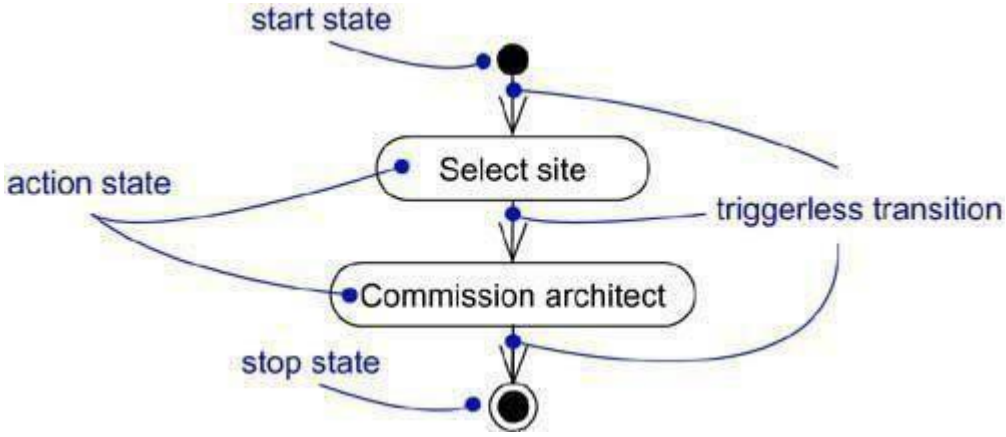
Figure Activity States



Transitions

When the action or activity of a state completes, flow of control passes immediately to the next action or activity state. You specify this flow by using transitions to show the path from one action or activity state to the next action or activity state. In the UML, you represent a transition as a simple directed line, as Figure shows.

Figure Triggerless Transitions

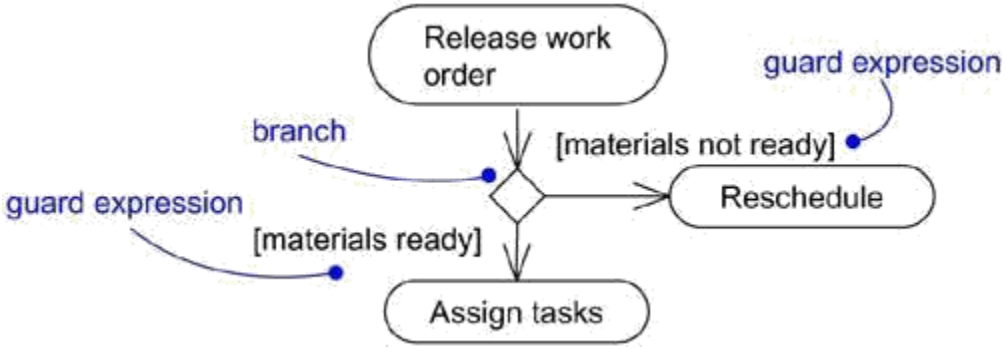


Indeed, a flow of control has to start and end someplace (unless, of course, it's an infinite flow, in which case it will have a beginning but no end). Therefore, as the figure shows, you may specify this initial state (a solid ball) and stop state (a solid ball inside a circle).

Branching

Simple, sequential transitions are common, but they aren't the only kind of path you'll need to model a flow of control. As in a flowchart, you can include a branch, which specifies alternate paths taken based on some Boolean expression. As Figure shows, you represent a branch as a diamond. A branch may have one incoming transition and two or more outgoing ones. On each outgoing transition, you place a Boolean expression, which is evaluated only once on entering the branch. Across all these outgoing transitions, guards should not overlap (otherwise, the flow of control would be ambiguous), but they should cover all possibilities (otherwise, the flow of control would freeze).

Figure Branching



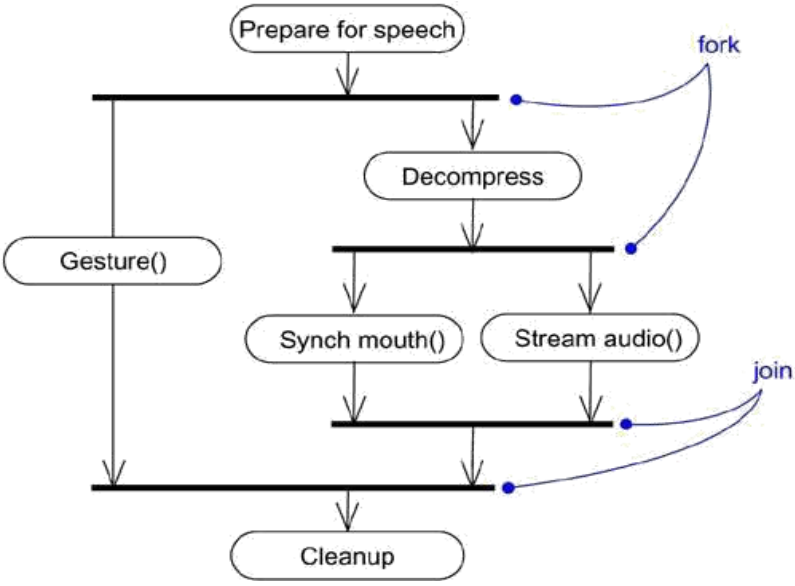
As a convenience, you can use the keyword **else**to mark one outgoing transition, representingthe path taken if no other guard expression evaluates to true.You can achieve the effect of iteration by using one action state that sets the value of an iterator, another action state that increments the iterator, and a branch that evaluates if the iteration is finished.

Forking and Joining

Simple and branching sequential transitions are the most common paths you'll find in activity diagrams. However• especially when you are modeling workflows of business processes• you might encounter flows that are concurrent. In the UML, you use a synchronization bar to specify the forking and joining of these parallel flows of control. A synchronization bar is rendered as a thick horizontal or vertical line.

For example, consider the concurrent flows involved in controlling an audio-animatronic device that mimics human speech and gestures. As Figure shows, a fork represents the splitting of a single flow of control into two or more concurrent flows of control. A fork may have one incoming transition and two or more outgoing transitions, each of which represents an independent flow of control. Below the fork, the activities associated with each of these paths continues in parallel. Conceptually, the activities of each of these flows are truly concurrent, although, in a running system, these flows may be either truly concurrent (in the case of a system deployed across multiple nodes) or sequential yet interleaved (in the case of a system deployed across one node), thus giving only the illusion of true concurrency.

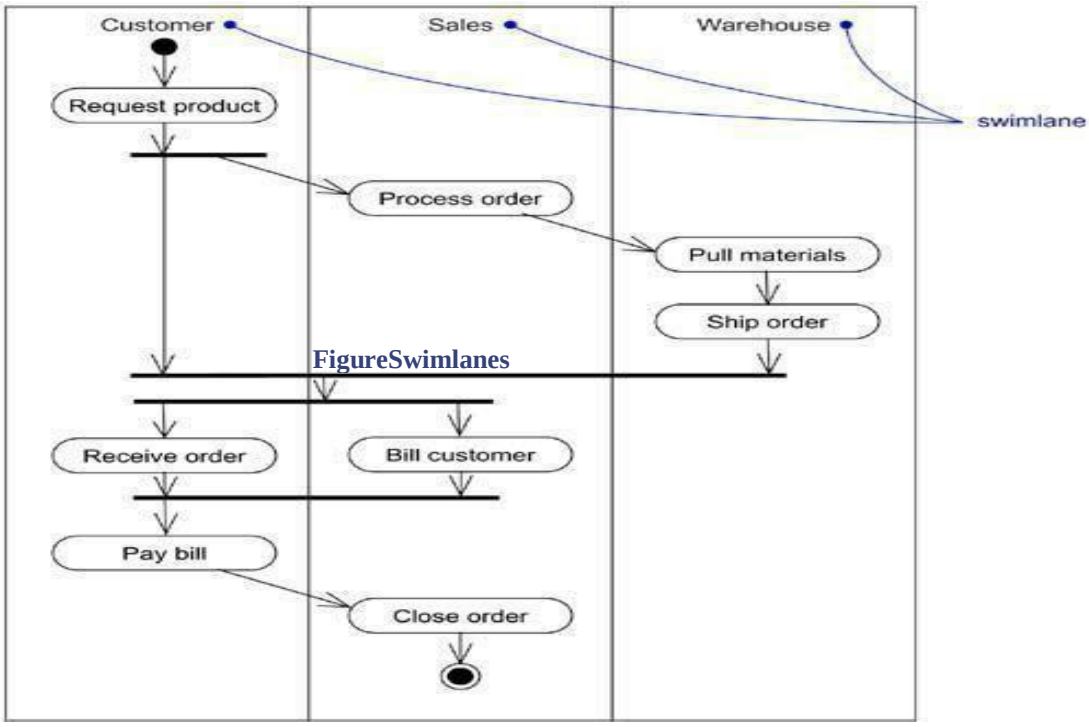
Figure Forking and Joining



As the figure also shows, a join represents the synchronization of two or more concurrent flows of control. A join may have two or more incoming transitions and one outgoing transition. Above the join, the activities associated with each of these paths continues in parallel. At the join, the concurrent flows synchronize, meaning that each waits until all incoming flows have reached the join, at which point one flow of control continues on below the join.

Swimlanes

You'll find it useful, especially when you are modeling workflows of business processes, to partition the activity states on an activity diagram into groups, each group representing the business organization responsible for those activities. In the UML, each group is called a swimlane because, visually, each group is divided from its neighbor by a vertical solid line, as shown in Figure . A swimlane specifies a locus of activities.



FigureSwimlanes

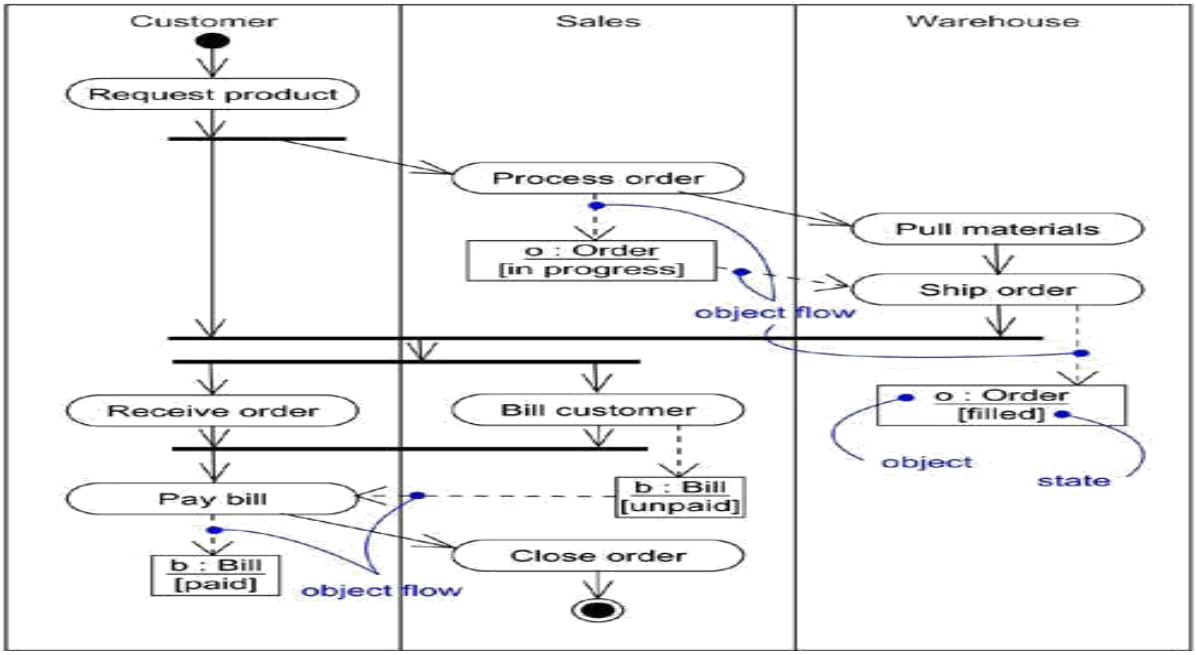
Each swimlane has a name unique within its diagram. A swimlane really has no deep semantics, except that it may represent some real-world entity. Each swimlane represents a high-level responsibility for part of the overall activity of an activity diagram, and each swimlane mayeventually be implemented by one or more classes. In an activity diagram partitioned into swimlanes, every activity belongs to exactly one swimlane, but transitions may cross lanes.

Object Flow

Objects may be involved in the flow of control associated with an activity diagram. For example, in the workflow of processing an order as in the previous figure, the vocabulary of your problem space will also include such classes as **Order** and **Bill.** Instances of these two classes will be produced by certain activities (**Process order** will create an **Order** object, for example); other activities may modify these objects (for example, **Ship order** will change the state of the **Order** object to **filled**).

As Figure shows, you can specify the things that are involved in an activity diagram by placing these objects in the diagram, connected using a dependency to the activity or transition that creates, destroys, or modifies them. This use of dependency relationships and objects is called an object flow because it represents the participation of an object in a flow of control.

Figure Object Flow



In addition to showing the flow of an object through an activity diagram, you can also show how its role, state and attribute values change. As shown in the figure, you represent the state of an object by naming its state in brackets below the object's name. Similarly, you can represent the value of an object's attributes by rendering them in a compartment below the object's name.

Common Uses

You use activity diagrams to model the dynamic aspects of a system. These dynamic aspects may involve the activity of any kind of abstraction in any view of a system's architecture, including classes (which includes active classes), interfaces, components, and nodes.

When you use an activity diagram to model some dynamic aspect of a system, you can do so in the context of virtually any modeling element. Typically, however, you'll use activity diagrams in the context of the system as a whole, a subsystem, an operation, or a class. You can also attach activity diagrams to use cases (to model a scenario) and to collaborations (to model the dynamic aspects of a society of objects).

When you model the dynamic aspects of a system, you'll typically use activity diagrams in two ways.

1. To model a workflow

Here you'll focus on activities as viewed by the actors that collaborate with the system. Workflows often lie on the fringe of software-intensive systems and are used to visualize, specify, construct, and document business processes that involve the system you are developing. In this use of activity diagrams, modeling object flow is particularly important.

2. To model an operation

Here you'll use activity diagrams as flowcharts, to model the details of a computation. In this use of activity diagrams, the modeling of branch, fork, and join states is particularly important. The context of an activity diagram used in this way involves the parameters of the operation and its local objects.

Common Modeling Techniques

Modeling a Workflow

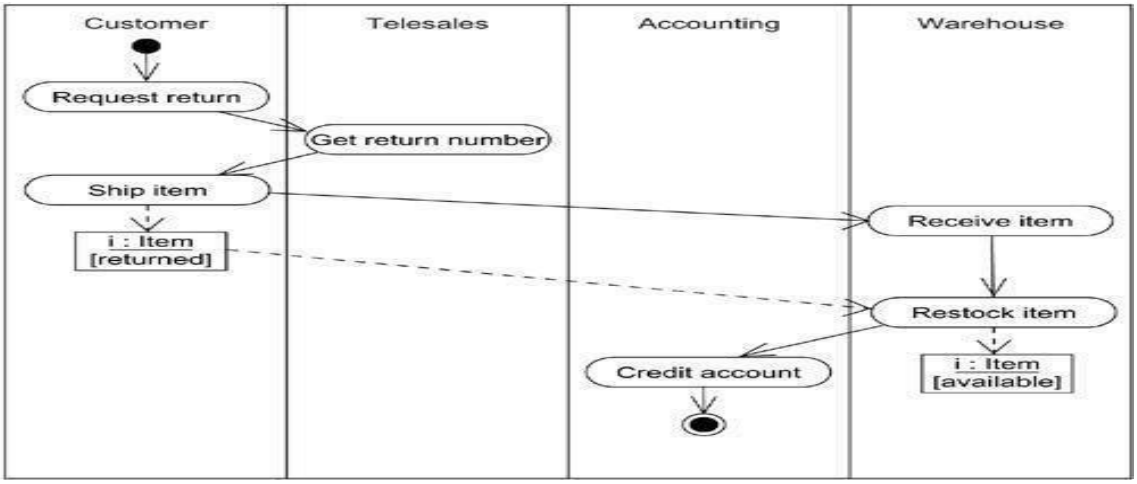
To model a workflow,

- Establish a focus for the workflow. For nontrivial systems, it's impossible to show all interesting workflows in one diagram.
- Select the business objects that have the high-level responsibilities for parts of the overall workflow. These may be real things from the vocabulary of the system, or they may be more abstract. In either case, create a swimlane for each important business object.
- Identify the preconditions of the workflow's initial state and the postconditions of the workflow's final state. This is important in helping you model the boundaries of the workflow.
- Beginning at the workflow's initial state, specify the activities and actions that take place over time and render them in the activity diagram as either activity states or actionstates.
- For complicated actions, or for sets of actions that appear multiple times, collapse these into activity states, and provide a separate activity diagram that expands on each.
- Render the transitions that connect these activity and action states. Start with the sequential flows in the workflow first, next consider branching, and only then consider forking and joining.
- If there are important objects that are involved in the workflow, render them in the activity diagram, as well. Show their changing values and state as necessary to communicate the intent of the object flow.

For example, Figure shows an activity diagram for a retail business, which specifies theworkflow involved when a customer returns an item from a mail order. Work starts with the**Customer** action **Request return** and then flows through **Telesales** (**Get return number**), back to the **Customer** (**Ship item**), then to the **Warehouse** (**Receive item** then **Restock item**), finally ending in **Accounting** (**Credit account**). As the diagramindicates,one significant object (**i**, an instance of **Item**) also flows

the process, changing from the **returned** to the **available** state.

Figure Modeling a Workflow



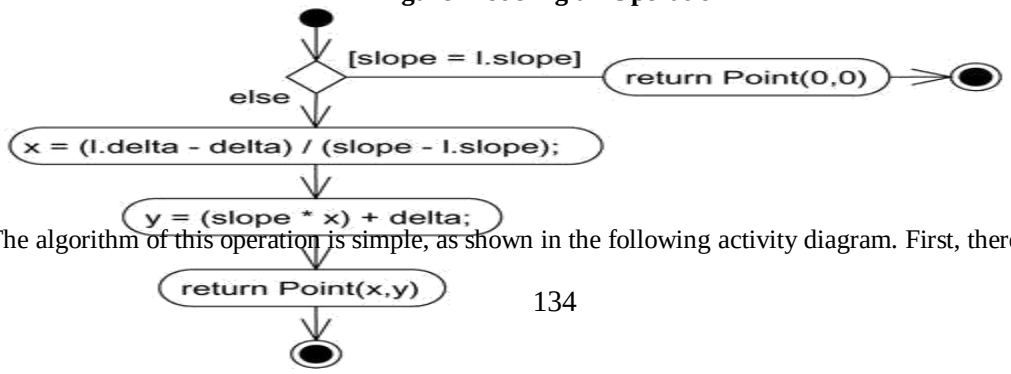
Modeling an Operation

To model an operation,

- Collect the abstractions that are involved in this operation. This includes the operation's parameters (including its return type, if any), the attributes of the enclosing class, and certain neighboring classes.
- Identify the preconditions at the operation's initial state and the postconditions at the operation's final state. Also identify any invariants of the enclosing class that must hold during the execution of the operation.
- Beginning at the operation's initial state, specify the activities and actions that take place over time and render them in the activity diagram as either activity states or actionstates.
- Use branching as necessary to specify conditional paths and iteration.
- Only if this operation is owned by an active class, use forking and joining as necessary to specify parallel flows of control.

For example, in the context of the class **Line**, Figure shows an activity diagram that specifies the algorithm of the operation **intersection**, whose signature includes one parameter (**l**, an **in** parameter of the class **Line**) and one return value (of the class **Point**). The class **Line** has two attributes of interest: **slope** (which holds the slope of the line) and **delta** (which holds the offset of the line relative to the origin).

Figure Modeling an Operation



The algorithm of this operation is simple, as shown in the following activity diagram. First, there's a guard

that tests whether the **slope** of the current line is the same as the **slope** of parameter **l.** If so, the lines do not intersect, and a **Point** at **(0,0)** is returned. Otherwise, the operationfirst calculates an **x** value for the point of intersection, then a **y** value; **x** and **y**are both objects local to the operation. Finally, a **Point** at **(x,y)** is returned.

Forward and Reverse Engineering

Forward engineering (the creation of code from a model) is possible for activity diagrams,especially if the context of the diagram is an operation. For example, using the previous activity diagram, a forward engineering tool could generate the following C++ code for the operation **intersection.**

```
Point Line::intersection (l : Line) {
    if (slope == l.slope) return Point(0,0);
    int x = (l.delta - delta) / (slope - l.slope); int y = (slope * x) + delta;
    return Point(x, y);
    }
```

There's a bit of cleverness here, involving the declaration of the two local variables. A less-sophisticated tool might have first declared the two variables and then set their values.

Reverse engineering (the creation of a model from code) is also possible for activity diagrams,especially if the context of the code is the body of an operation. In particular, the previous diagram could have been generated from the implementation of the class **Line.**

More interesting than the reverse engineering of a model from code is the animation of a model against the execution of a deployed system. For example, given the previous diagram, a tool could animate the action states in the diagram as they were dispatched in a running system. Even better, with this tool also under the control of a debugger, you could control the speed of execution, possibly setting breakpoints to stop the action at interesting points in time to examine the attribute values of individual objects.

UNIT – 4 : Advanced Behavioral Modeling

Syllabus :Events and signals,statemachines,processes and Threads ,time and space chart diagrams, Component, Deployment, Component Diagrams and Deployment diagrams

Events and Signals

Terms and Concepts

An *event* is the specification of a significant occurrence that has a location in time and space. In the context of state machines, an event is an occurrence of a stimulus that can trigger a state transition. A *signal* is a kind of event that represents the specification of an asynchronous stimulus communicated between instances.

Kinds of Events

Events may be external or internal. External events are those that pass between the system and its actors. For example, the pushing of a button and an interrupt from a collision sensor are both examples of external events. Internal events are those that pass among the objects that live inside the system. An overflow exception is an example of an internal event.

In the UML, you can model four kinds of events: signals, calls, the passing of time, and a change in state.

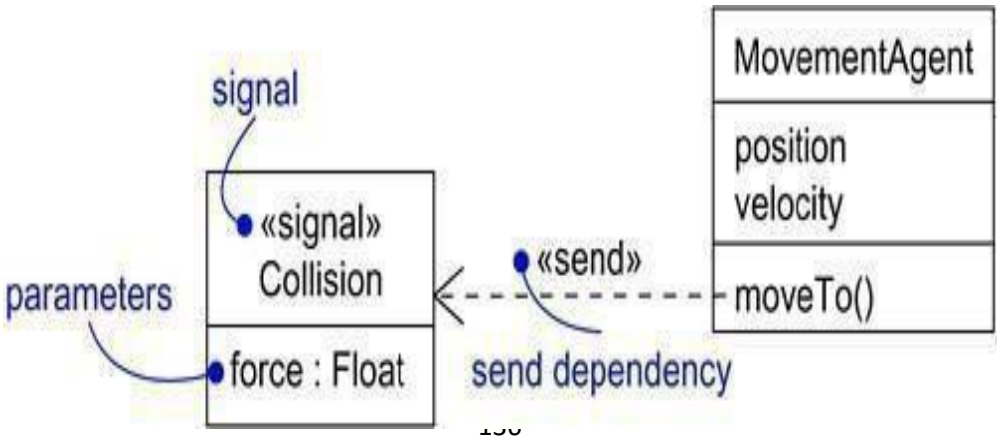
Signals

A signal represents a named object that is dispatched (thrown) asynchronously by one object and then received (caught) by another. Exceptions are supported by most contemporary programming languages and are the most common kind of internal signal that you will need to model.

Signals have a lot in common with plain classes. For example, signals may have instances, although you don't generally need to model them explicitly. Signals may also be involved in generalization relationships, permitting you to model hierarchies of events, some of which are general (for example, the signal **NetworkFailure**) and some of which are specific (for example, a specialization of **NetworkFailure** called **WarehouseServerFailure**). Also as for classes, signals may have attributes and operations.

A signal may be sent as the action of a state transition in a state machine or the sending of a message in an interaction. The execution of an operation can also send signals. In fact, when you model a class or an interface, an important part of specifying the behavior of that element is specifying the signals that its operations can send. In the UML, you model the relationship between an operation and the events that it can send by using a dependency relationship, stereotyped as **send.**

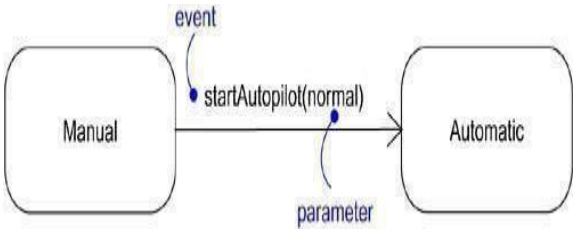
Figure Signals



Call Events

Just as a signal event represents the occurrence of a signal, a call event represents the dispatch of an operation. In both cases, the event may trigger a state transition in a state machine.Whereas a signal is an asynchronous event, a call event is, in general, synchronous. This means that when an object invokes an operation on another object that has a state machine, control passes from the sender to the receiver, the transition is triggered by the event, the operation is completed, the receiver transitions to a new state, and control returns to the sender.

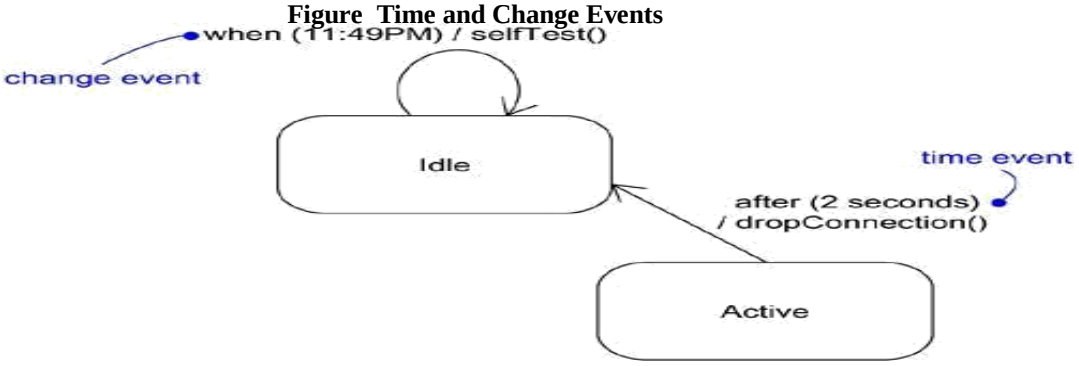
Figure Call Events



Time and Change Events

A time event is an event that represents the passage of time. As Figure shows, in the UML you model a time event by using the keyword **after** followed by some expression that evaluates to a period of time. Such expressions can be simple (for example, **after 2 seconds**) or complex (for example, **after 1 ms since exiting Idle**). Unless you specify it explicitly, the starting time of such an expression is the time since entering the current state.

Figure Time and Change Events



A change event is an event that represents a change in state or the satisfaction of some condition. As Figure shows, in the UML you model a change event by using the keyword**when** followed by some Boolean expression. You can use such expressions to mark an absolutetime (such as **when time = 11:59**) or for the continuous test of an expression (for example, **when altitude < 1000**).

Sending and Receiving Events

Signal events and call events involve at least two objects: the object that sends the signal or invokes the operation, and the object to which the event is directed. Because signals are asynchronous, and because asynchronous calls are themselves signals, the semantics of events interact with the semantics of active

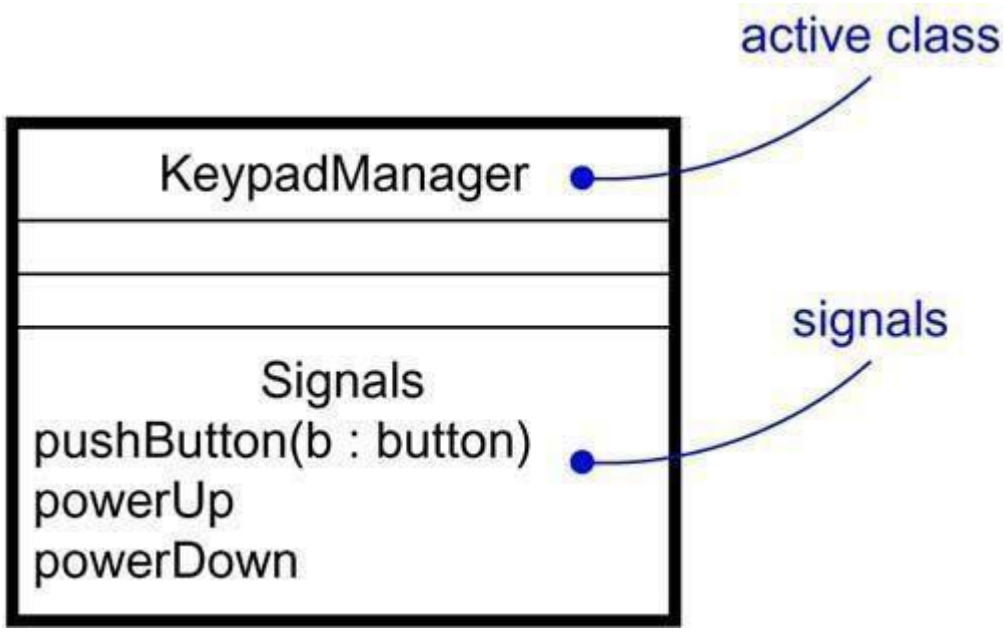
objects and passive objects.

Any instance of any class can send a signal to or invoke an operation of a receiving object. When an object sends a signal, the sender dispatches the signal and then continues along its flow of control, not waiting for any return from the receiver. For example, if an actor interacting with an ATM system sends the signal **pushButton,** the actor may continue along its way independent of the system to which the signal was sent. In contrast, when an object calls an operation, the sender dispatches the operation and then waits for the receiver. For example, in a trading system, an instance of the class **Trader** might invoke the operation **confirmTransaction** on some instance of the class **Trade,** thereby affecting the state of the **Trade** object. If this is a synchronous call, the **Trader** object will wait until the operation is finished.

Any instance of any class can receive a call event or a signal. If this is a synchronous call event, then the sender and the receiver are in a rendezvous for the duration of the operation. This means that the flow of control of the sender is put in lock step with the flow of control of the receiver until the activity of the operation is carried out. If this is a signal, then the sender and receiver do not rendezvous: the sender dispatches the signal but does not wait for a response from the receiver. In either case, this event may be lost (if no response to the event is specified), it may trigger the receiver's state machine (if there is one), or it may just invoke a normal method call.

In the UML, you model the call events that an object may receive as operations on the class of the object. In the UML, you model the named signals that an object may receive by naming them in an extra compartment of the class, as shown in Figure .

Figure Signals and Active Classes.



Common Modeling Techniques

Modeling a Family of Signals

In most event-driven systems, signal events are hierarchical. For example, an autonomous robot might distinguish between external signals, such as a **Collision,** and internal ones, such as a **HardwareFault.** External and internal signals need not be disjoint, however. Even within thesetwo broad classifications, you might find specializations. For example, **HardwareFault** signals might be further specialized as **BatteryFault** and **MovementFault.** Even these might be further specialized, such as **MotorStall,** a kind

of **MovementFault.**

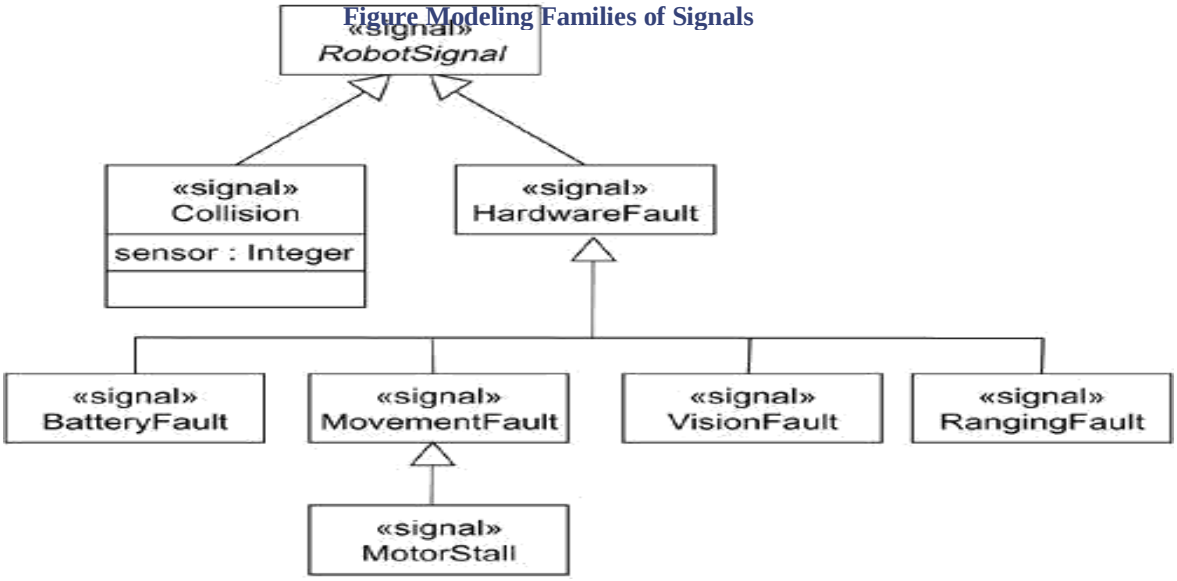
By modeling hierarchies of signals in this manner, you can specify polymorphic events. For example, consider a state machine with a transition triggered only by the receipt of a **MotorStall.** As a leaf signal in this hierarchy, the transition can be triggered only by thatsignal, so it is not polymorphic. In contrast, suppose you modeled the state machine with a transition triggered by the receipt of a **HardwareFault.** In this case, the transition is polymorphic and can be triggered by a **HardwareFault** or any of its specializations, including**BatteryFault, MovementFault,** and**MotorStall.**

To model a family of signals,

- Consider all the different kinds of signals to which a given set of active objects may respond.
- Look for the common kinds ofsignals and place them in a generalization/specialization hierarchy using inheritance. Elevate more general ones and lower more specialized ones.
- Look for the opportunity for polymorphism in the state machines of these active objects. Where you find polymorphism, adjust the hierarchy as necessary by introducing intermediate abstract signals.

Figure models a family of signals that may be handled by an autonomous robot. Note thatthe root signal (**RobotSignal**) is abstract, which means that there may be no direct instances. This signal has two immediate concrete specializations (**Collision** and **HardwareFault**), one of which (**HardwareFault**) is further specialized. Note that the **Collision** signal has one parameter.

Figure Modeling Families of Signals



Modeling Exceptions

An important part of visualizing, specifying, and documenting the behavior of a class or an interface is specifying the exceptions that its operations can raise. If you are handed a class or an interface, the operations you can invoke will be clear, but the exceptions that each operation may raise will not be clear unless you model them explicitly.

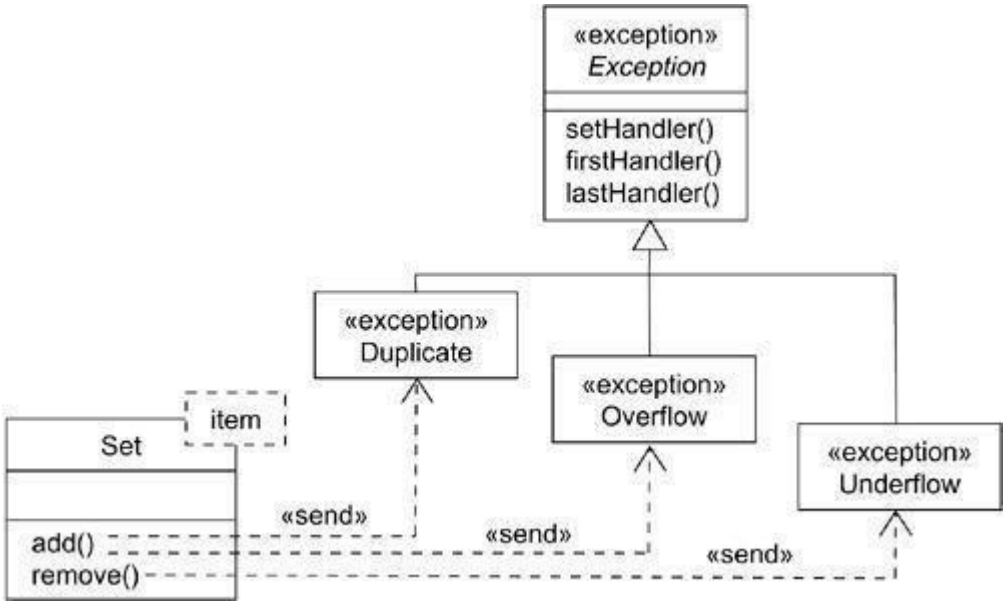
In the UML, exceptions are kinds of signals, which you model as stereotyped classes. Exceptions may be attached to specification operations. Modeling exceptions is somewhat the inverse of modeling a general family of signals. You model a family of signals primarily to specify the kinds of signals an active object may receive; you model exceptions primarily to specify the kinds of exceptions that an object may throw through its operations.

To model exceptions,

- For each class and interface, and for each operation of such elements, consider the exceptional conditions that may be raised.
- Arrange these exceptions in a hierarchy. Elevate general ones, lower specialized ones, and introduce intermediate exceptions, as necessary.
- For each operation, specify the exceptions that it may raise. You can do so explicitly (by showing **send** dependencies from an operation to its exceptions) or you can put this in the operation's specification.

Figure models a hierarchy of exceptions that may be raised by a standard library ofcontainer classes, such as the template class **Set.** This hierarchy is headed by the abstract signal **Exception** and includes three specialized exceptions: **Duplicate, Overflow,** and **Underflow.** As shown, the **add** operation raises **Duplicate** and **Overflow** exceptions, andthe **remove** operation raises only the **Underflow** exception. Alternatively, you could have put these dependencies in the background by naming them in each operation's specification. Either way, by knowing which exceptions each operation may send, you can create clients that use the **Set** class correctly.

Figure Modeling Exceptions



State Machines

Terms and Concepts

A *state machine* is a behavior that specifies the sequences of states an object goes through during its lifetime in response to events, together with its responses to those events. A *state* is a condition or situation during the life of an object during which it satisfies some condition, performs some activity, or waits for some event. An *event* is the specification of a significant occurrence that has a location in time and space. In the context of state machines, an event is an occurrence of a stimulus that can trigger a state transition. A *transition* is a relationship between two states indicating that an object in the first state will perform certain actions and enter the second state when a specified event occurs and specified conditions are satisfied. An *activity* is ongoing nonatomic execution within a state machine. An *action* is an executable atomic computation that results in a change in state of the model or the return of a value. Graphically, a state is rendered as a rectangle with rounded corners. A transition is rendered as a solid directed line.

Context

Every object has a lifetime. On creation, an object is born; on destruction, an object ceases to exist. In between, an object may act on other objects (by sending them messages), as well as be acted on (by being the target of a message). In many cases, these messages will be simple, synchronous operation calls. For example, an instance of the class **Customer** might invoke the operation **getAccountBalance** on an instance of the class **BankAccount.** Objects such as these don't need a state machine to specify their behavior because their current behavior does not depend on their past.

In other kinds of systems, you'll encounter objects that must respond to signals, which are asynchronous stimuli communicated between instances. For example, a cellular phone must respond to random phone calls (from other phones), keypad events (from the customer initiating a phone call), and to events from the network (when the phone moves from one call to another). Similarly, you'll encounter objects whose current behavior depends on their past behavior. For example, the behavior of an air-to-air missile guidance system will depend on its current state, such as **NotFlying** (it's not a good idea to launch a missile while it's attached to an aircraft that's still sitting on the ground) or **Searching** (you shouldn't arm the missile until you have a good idea what it's going to hit).

The behavior of objects that must respond to asynchronous stimulus or whose current behavior depends on their past is best specified by using a state machine. This encompasses instances of classes that can receive signals, including many active objects. In fact, an object that receives a signal but has no state machine will simply ignore that signal. You'll also use state machines to model the behavior of entire systems, especially reactive systems, which must respond to signals from actors outside the system.

States

A state is a condition or situation during the life of an object during which it satisfies some condition, performs some activity, or waits for some event. An object remains in a state for a finite amount of time. For example, a **Heater** in a home might be in any of four states: **Idle** (waiting for a command to start heating the house), **Activating** (its gas is on, but it's waiting to come up to temperature), **Active** (its gas and blower are both on), and **ShuttingDown** (its gas is off but its blower is on, flushing residual heat from the system).

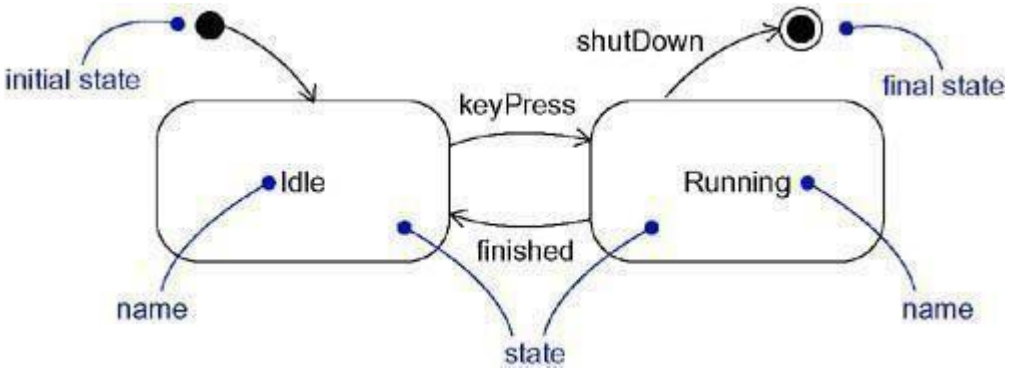
When an object's state machine is in a given state, the object is said to be in that state. For example, an instance of **Heater** might be **Idle** or perhaps **ShuttingDown.**

Astate has several parts.

1. Name	A textual string that distinguishes the state from other states; a state may be anonymous, meaning that it has no name
2. Entry/exit actions	Actions executed on entering and exiting the state, respectively
3. Internal transitions	Transitions that are handled without causing a change in state
4. Substates	The nested structure of a state, involving disjoint (sequentially active) or concurrent (concurrently active) substates
5. Deferred events	A list of events that are not handled in that state but, rather, are postponed and queued for handling by the object in another state

As Figure shows, you represent a state as a rectangle with rounded corners.

Figure States



Initial and Final States

As the figure shows, there are two special states that may be defined for an object's state machine. First, there's the initial state, which indicates the default starting place for the state machine or substate. An initial state is represented as a filled black circle. Second, there's the final state, which indicates that the execution of the state machine or the enclosing state has been completed. A final state is represented as a filled black circle surrounded by an unfilled circle.

> **Note**
>
> Initial and final states are really pseudostates. Neither may have the usual parts of a normal state, except for a name. A transition from an initial state to a final state may have the full complement of features, including a guard condition and action (but not a trigger event).

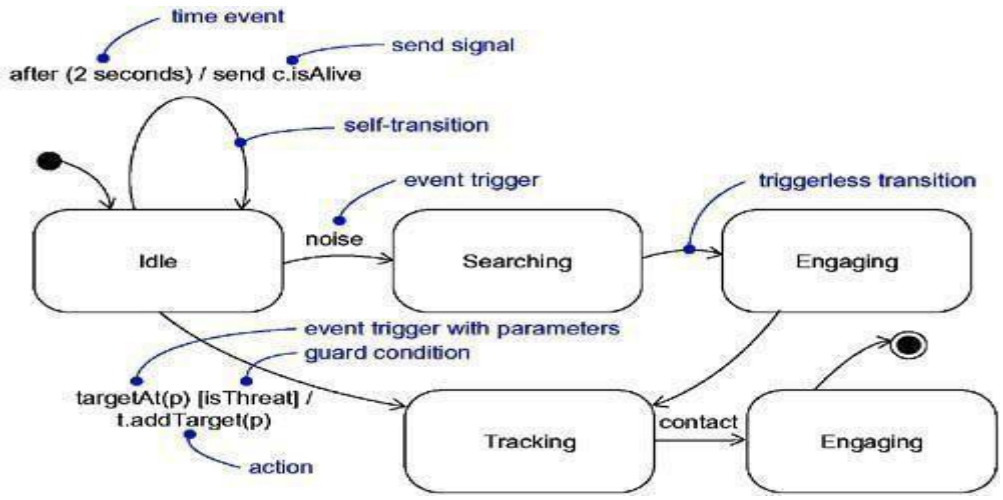
Transitions

A transition is a relationship between two states indicating that an object in the first state will perform

certain actions and enter the second state when a specified event occurs and specified conditions are satisfied. On such a change of state, the transition is said to fire. Until the transition fires, the object is said to be in the source state; after it fires, it is said to be in the target state. For example, a **Heater** might transition from the **Idle** to the **Activating** state when an event such as **tooCold** (with the parameter **desiredTemp**) occurs.

As Figure shows, a transition is rendered as a solid directed line from the source to the target state. A self-transition is a transition whose source and target states are the same.

Figure Transitions



Event Trigger

An event is the specification of a significant occurrence that has a location in time and space. In the context of state machines, an event is an occurrence of a stimulus that can trigger a state transition. As shown in the previous figure, events may include signals, calls, the passing of time,or a change in state. A signal or a call may have parameters whose values are available to the transition, including expressions for the guard condition and action.

It is also possible to have a triggerless transition, represented by a transition with no event trigger. A triggerless transition• also called a completion transition• is triggered implicitly when its source state has completed its activity.

Guard

As the previous figure shows, a guard condition is rendered as a Boolean expression enclosed in square brackets and placed after the trigger event. A guard condition is evaluated only after the trigger event for its transition occurs. Therefore, it's possible to have multiple transitions from the same source state and with the same event trigger, as long as those conditions don't overlap.

A guard condition is evaluated just once for each transition at the time the event occurs, but it may be evaluated again if the transition is retriggered. Within the Boolean expression, you can include conditions about the state of an object (for example, the expression **aHeater in Idle,** which evaluates True if the **Heater** object is currently in the **Idle** state).

Action

An action is an executable atomic computation. Actions may include operation calls (to the object that owns the state machine, as well as to other visible objects), the creation or destruction of another object, or the sending of a signal to an object. As the previous figure shows, there's a special notation for sending a signal• the signal name is prefixed with the keyword **send** as a visual cue.

Activities are discussed in a later section of this chapter; dependencies are discussed in An action is atomic, meaning that it cannot be interrupted by an event and therefore runs to completion. This is in contrast to an activity, which may be interrupted by other events.

Advanced States and Transitions

You can model a wide variety of behavior using only the basic features of states and transitions in the UML. Using these features, you'll end up with flat state machines, which means that your behavioral models will consist of nothing more than arcs (transitions) and vertices (states).

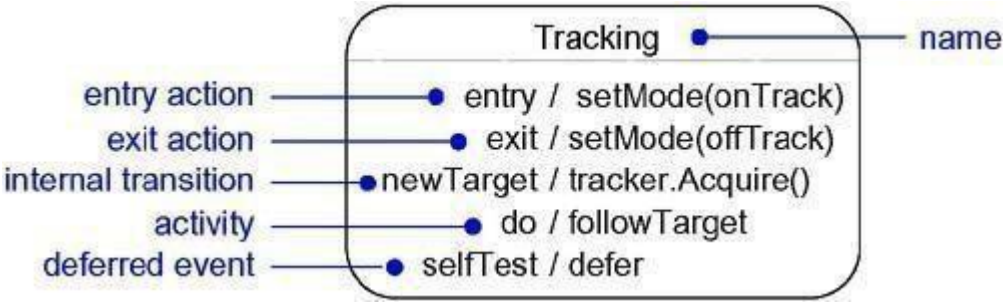
However, the UML's state machines have a number of advanced features that help you to manage complex behavioral models. These features often reduce the number of states and transitions you'll need, and they codify a number of common and somewhat complex idioms you'd otherwise encounter using flat state machines. Some of these advanced features include entry and exit actions, internal transitions, activities, and deferred events.

Entry and Exit Actions

In a number of modeling situations, you'll want to dispatch the same action whenever you enter a state, no matter which transition led you there. Similarly, when you leave a state, you'll want to dispatch the same action no matter which transition led you away. For example, in a missile guidance system, you might want to explicitly announce the system is **onTrack** whenever it's in the **Tracking** state, and **offTrack** whenever it's out of the state. Using flat state machines, you can achieve this effect by putting those actions on every entering and exiting transition, as appropriate. However, that's somewhat error prone; you have to remember to add these actions every time you add a new transition. Furthermore, modifying this action means that you have to touch every neighboring transition.

As Figure shows, the UML provides a shorthand for this idiom. In the symbol for the state, you can include an entry action (marked by the keyword event **entry**) and an exit action (marked by the keyword event **exit**), together with an appropriate action. Whenever you enter the state, its entry action is dispatched; whenever you leave the state, its exit action is dispatched.

Figure Advanced States and Transitions



Internal Transitions

Once inside a state, you'll encounter events you'll want to handle without leaving the state. These are called internal transitions, and they are subtly different from self-transitions. In a self-transition, such as you see in Figure , an event triggers the transition, you leave the state,an action (if any) is dispatched, and then you reenter the same state. Because this transition exits and then enters the state, a self-transition dispatches the state's exit action, then it dispatches the action of the self-transition, and finally, it

dispatches the state's entry action. However, suppose you want to handle the event but don't want to fire the state's entry and exit actions. Using flat state machines, you can achieve that effect, but you have to be diligent about remembering which of a state's transitions have these entry and exit actions and which do not.

As Figure shows, the UML provides a shorthand for this idiom, as well (for example, for the event **newTarget**). In the symbol for the state, you can include an internal transition (marked by an event). Whenever you are in the state and that event is triggered, the corresponding action is dispatched without leaving and then reentering the state. Therefore, the event is handled without dispatching the state's exit and then entry actions.

Activities

When an object is in a state, it generally sits idle, waiting for an event to occur. Sometimes, however, you may wish to model an ongoing activity. While in a state, the object does some work that will continue until it is interrupted by an event. For example, if an object is in the **Tracking** state, it might **followTarget** as long as it is in that state. As Figure shows, in the UML, you use the special **do** transition to specify the work that's to be done inside a state after the entry action is dispatched. The activity of a **do** transition might name another state machine (such as **followTarget**). You can also specify a sequence of actions•for example, do / op1(a); op2(b);op3(c). Actions are never interruptible, but sequences of actions are. In between each action (separated by the semicolon), events may be handled by the enclosing state, which results in transitioning out of the state.

Deferred Events

Consider a state such as **Tracking.** As illustrated in Figure, suppose there's only one transition leading out of this state, triggered by the event **contact.** While in the state **Tracking,** any events other than **contact** and other than those handled by its substates willbe lost. That means that the event may occur, but it will be postponed and no action will result because of the presence of that event. In every modeling situation, you'll want to recognize some events and ignore others. You include those you want to recognize as the event triggers of transitions; those you want to ignore you just leave out. However, in some modeling situations, you'll want to recognize some events but postpone a response to them until later. For example, while in the **Tracking** state, you may want to postpone a response to signals such as **selfTest,**perhaps sent by some maintenanceagent in thesystem.

In the UML, you can specify this behavior by using deferred events. A deferred event is a list of events whose occurrence in the state is postponed until a state in which the listed events are not deferred becomes active, at which time they occur and may trigger transitions as if they had just occurred. As you can see in the previous figure, you can specify a deferred event by listing the event with the special action **defer.** In this example, **selfTest** events may happen while in the **Tracking** state, but they are held until the object is in the **Engaging** state, at which time itappears as if they just occurred.

Substates

These advanced features of states and transitions solve a number of common state machine modeling problems. However, there's one more feature of the UML's state machines• substates• that does even more to help you simplify the modeling of complex behaviors. A substate is a state that's nested inside another one. For example, a **Heater** might be in the **Heating** state, but also while in the **Heating** state, there might be a nested state called **Activating.** In this case, it's proper to say that the object is both **Heating** and **Activating.**

A simple state is a state that has no substructure. A state that has substates• that is, nested states• is called a composite state. A composite state may contain either concurrent (orthogonal) or sequential (disjoint) substates. In the UML, you render a composite state just as you do a simple state, but with an option al graphic compartment that shows a nested state machine. Substates may be nested to any level.

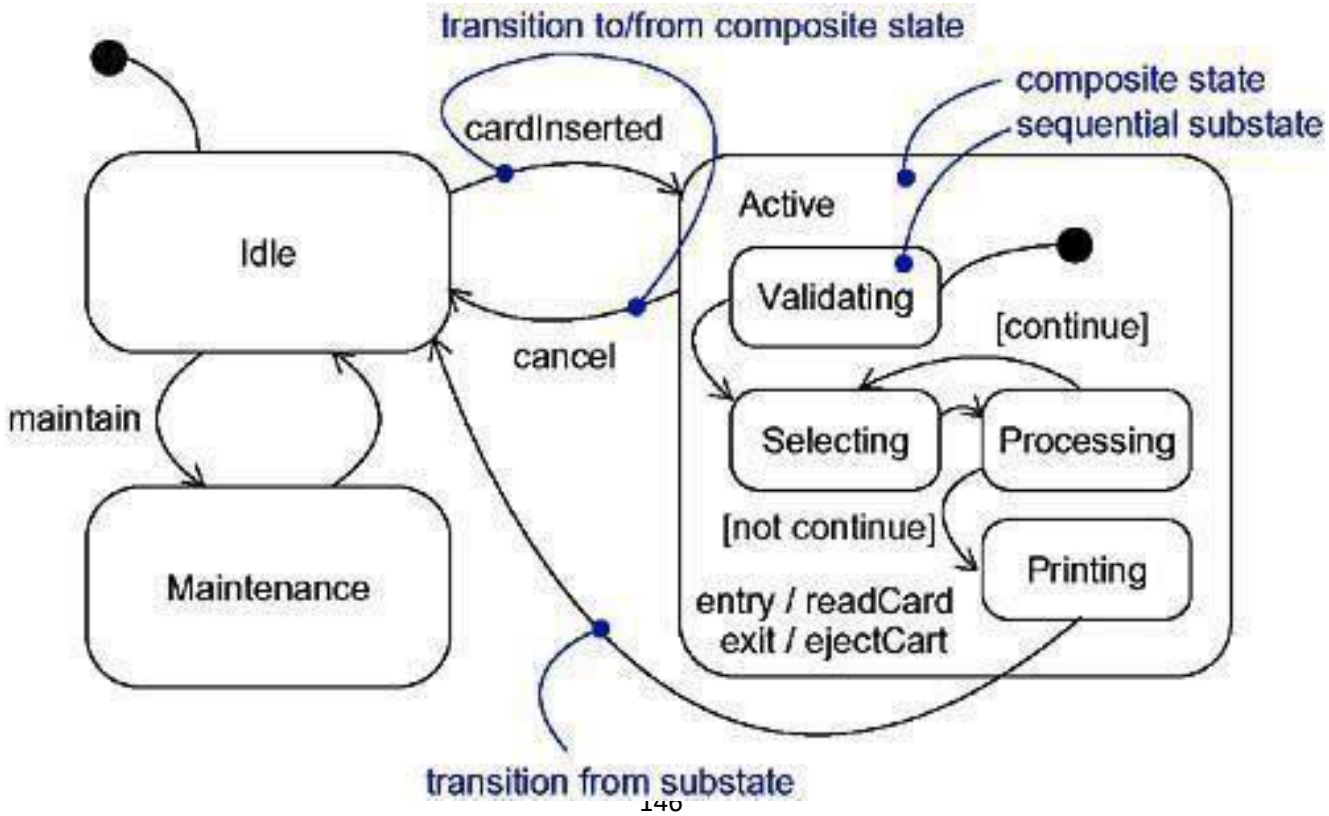
Sequential Substates

Consider the problem of modeling the behavior of an ATM. There are three basic states in which this system might be: **Idle** (waiting for customer interaction), **Active** (handling a customer's transaction), and **Maintenance** (perhaps having its cash store replenished). While **Active,** the behavior of the ATM follows a simple path: Validate the customer, select a transaction, process the transaction, and then print a receipt. After printing, the ATM returns to the **Idle** state. You might represent these stages of behavior as the states **Validating, Selecting,Processing,** and **Printing.** It would even be desirable to let the customer select andprocess multiple transactions after **Validating** the account and before **Printing** a finalreceipt.

The problem here is that, at any stage in this behavior, the customer might decide to cancel the transaction, returning the ATM to its **Idle** state. Using flat state machines, you can achieve that effect, but it's quite messy. Because the customer might cancel the transaction at any point, you'd have to include a suitable transition from every state in the **Active** sequence. That's messy because it's easy to forget to include these transitions in all the right places, and many suchinterrupting events means you end up with a multitude of transitions zeroing in on the same target state from various sources, but with the same event trigger, guard condition, and action.

Using sequential substates, there's a simpler way to model this problem, as Figure shows.Here, the **Active** state has a substructure, containing the substates**Validating,Selecting, Processing,** and **Printing.** The state of the ATM changes from **Idle** to**Active** when the customer enters a credit card in the machine. On entering the **Active** state,the entry action **readCard** is performed. Starting with the initial state of the substructure, control passes to the **Validating** state, then to the **Selecting** state, and then to the **Processing** state. After **Processing,** control may return to **Selecting** (if the customer has selected another transaction) or it may move on to **Printing.** After **Printing,** there's a triggerless transition back to the **Idle** state. Notice that the **Active** state has an exit action, which ejects the customer's credit card.

Figure SequentialSubstates



Notice also the transition from the **Active** state to the **Idle** state, triggered by the event **cancel.** In any substate of **Active,** the customer might cancel the transaction, and that returnsthe ATM to the **Idle** state (but only after ejecting the customer's credit card, which is the exit action dispatched on leaving the **Active** state, no matter what caused a transition out of that state). Without substates, you'd need a transition triggered by **cancel** on every substructure state.

Substates such as **Validating** and **Processing** are called sequential, or disjoint, substates. Given a set of disjoint substates in the context of an enclosing composite state, the object is said to be in the composite state and in only one of those substates (or the final state) at a time. Therefore, sequential substates partition the state space of the composite state into disjoint states.

From a source outside an enclosing composite state, a transition may target the composite state or it may target a substate. If its target is the composite state, the nested state machine must include an initial state, to which control passes after entering the composite state and after dispatching its entry action (if any). If its target is the nested state, control passes to the nested state, after dispatching the entry action (if any) of the composite state and then the entry action (if any) of the substate.

A transition leading out of a composite state may have as its source the composite state or a substate. In either case, control first leaves the nested state (and its exit action, if any, is dispatched), then it leaves the composite state (and its exit action, if any, is dispatched). A transition whose source is the composite state essentially cuts short (interrupts) the activity of the nested state machine.

History States

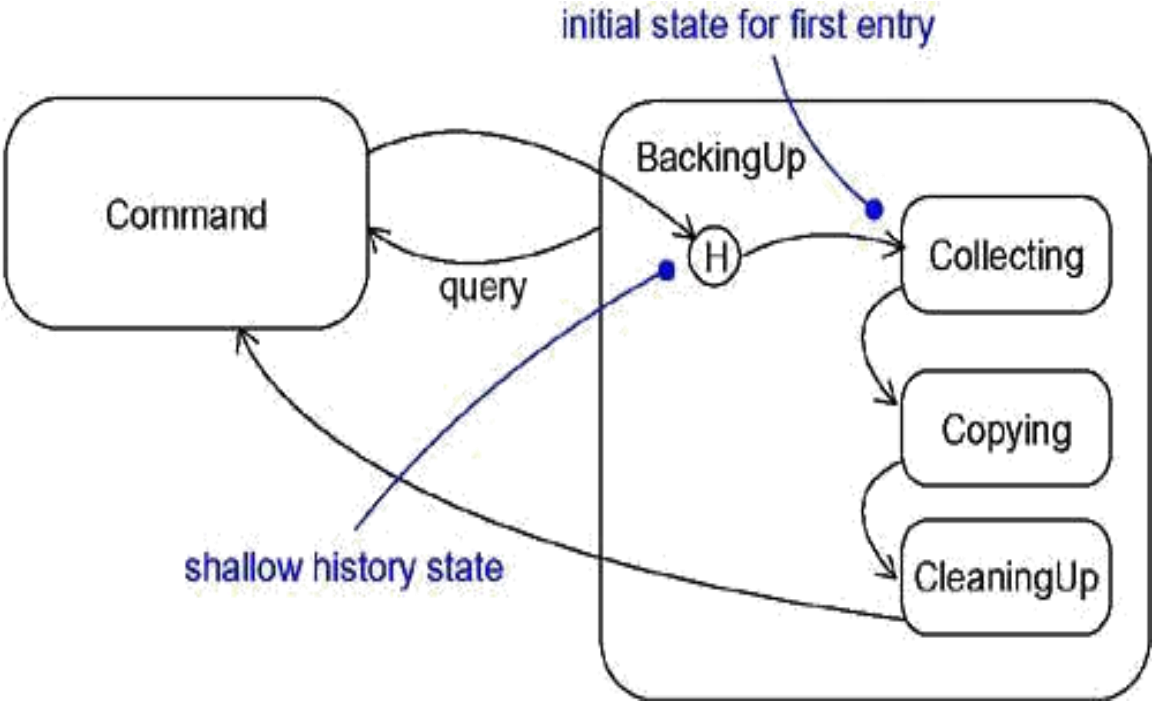
A state machine describes the dynamic aspects of an object whose current behavior depends on its past. A state machine in effect specifies the legal ordering of states an object may go through during its lifetime.

Unless otherwise specified, when a transition enters a composite state, the action of the nested state machine starts over again at its initial state (unless, of course, the transition targets a substate directly). However, there are times you'd like to model an object so that it remembers the last substate that was active prior to leaving the composite state. For example, in modeling the behavior of an agent that does an unattended backup of computers across a network, you'd like it to remember where it was in the process if it ever gets interrupted by, for example, a query from the operator.

Using flat state machines, you can model this, but it's messy. For each sequential substate, you'd need to have its exit action post a value to some variable local to the composite state. Then the initial state to this composite state would need a transition to every substate with a guard condition, querying the variable. In this way, leaving the composite state would cause the last substate to be remembered; entering the composite state would transition to the proper substate. That's messy because it requires you to remember to touch every substate and to set an appropriate exit action. It leaves you with a multitude of transitions fanning out from the same initial state to different target substates with very similar (but different) guard conditions.

In the UML, a simpler way to model this idiom is by using history states. A history state allows a composite state that contains sequential substates to remember the last substate that was active in it prior to the transition from the composite state. As Figure shows, you represent ashallow history state as a small circle containing the symbol **H.**

Figure History State



If you want a transition to activate the last substate, you show a transition from outside the composite state directly to the history state. The first time you enter a composite state, it has no history. This is the meaning of the single transition from the history state to a sequential substate such as **Collecting.**The target of this transition specifies the initial state of the nested statemachine the first time it is entered. Continuing, suppose that while in the **BackingUp** state and the **Copying** state, the **query** event is posted. Control leaves **Copying** and **BackingUp** (dispatching their exit actions as necessary) and returns to the **Command** state. When the action of **Command** completes, the triggerless transition returns to the history state of the composite state **BackingUp.** This time, because there is a history to the nested state machine, control passes back to the **Copying** state• thus bypassing the **Collecting** state• because **Copying** was the last substate active prior to the transition from **BackingUp.**

In either case, if a nested state machine reaches a final state, it loses its stored history and behaves as if it had not yet been entered for the first time.

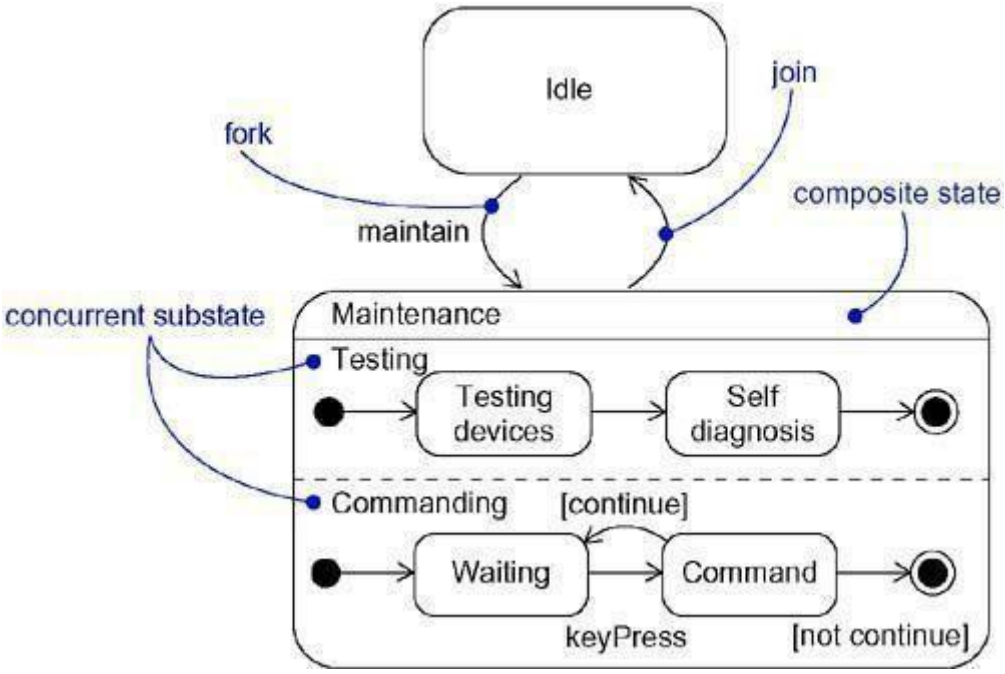
Concurrent Substates

Sequential substates are the most common kind of nested state machine you'll encounter. In certain modeling situations, however, you'll want to specify concurrent substates. These substates let you specify two or more state machines that execute in parallel in the context of the enclosing object.

For example, Figure shows an expansion of the **Maintenance** state from Figure .**Maintenance** is decomposed into two concurrent substates, **Testing** and **Commanding,** shown by nesting them in the **Maintenance** state but separating them from one another with a dashed line. Each of these concurrent

substates is further decomposed into sequential substates. When control passes from the **Idle** to the **Maintenance** state, control then forks to two concurrent flows• the enclosing object will be in the **Testing** state and the **Commanding** state. Furthermore, while in the **Commanding** state, the enclosing object will be in the **Waiting** or the **Command** state.

Figure Concurrent Substates



Execution of these two concurrent substates continues in parallel. Eventually, each nested state machine reaches its final state. If one concurrent substate reaches its final state before the other, control in that substate waits at its final state. When both nested state machines reach their final state, control from the two concurrent substates joins back into one flow.

Whenever there's a transition to a composite state decomposed into concurrent substates, control forks into as many concurrent flows as there are concurrent substates. Similarly, whenever there's a transition from a composite substate decomposed into concurrent substates, control joins back into one flow. This holds true in all cases. If all concurrent substates reach their final state, or if there is an explicit transition out of the enclosing composite state, control joins back into one flow.

Common Modeling Techniques

Modeling the Lifetime of an Object

The most common purpose for which you'll use state machines is to model the lifetime of an object, especially instances of classes, use cases, and the system as a whole. Whereas interactions model the behavior of a society of objects working together, a state machine models the behavior of a single object over its lifetime, such as you'll find with user interfaces, controllers, and devices.

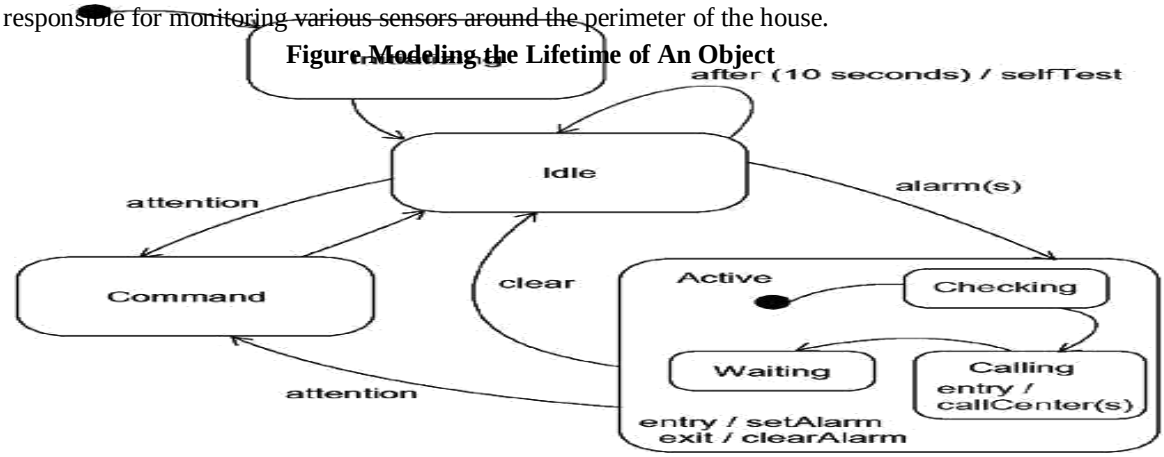
When you model the lifetime of an object, you essentially specify three things: the events to which the object can respond, the response to those events, and the impact of the past on current behavior. Modeling the lifetime of an object also involves deciding on the order in which the object can meaningfully respond to events, starting at the time of the object's creation and continuing until its destruction.

To model the lifetime of an object,

- Set the context for the state machine, whether it is a class, a use case, or the system as a whole.
    1. If the context is a class or a use case, collect the neighboring classes, including any parents of the class and any classes reachable by associations or dependences. These neighbors are candidate targets for actions and are candidates for including in guard conditions.
    2. if the context is the system as a whole, narrow your focus to one behavior of the system. Theoretically, every object in the system may be a participant in a model of the system's lifetime, and except for the most trivial systems, a complete model would be intractable.
- Establish the initial and final states for the object. To guide the rest of your model, possibly state the pre- and postconditions of the initial and final states, respectively.
- Decide on the events to which this object may respond. If already specified, you'll find these in the object's interfaces; if not already specified, you'll have to consider which objects may interact with the object in your context, and then which events they may possibly dispatch.
- Starting from the initial state to the final state, lay out the top-level states the object may be in. Connect these states with transitions triggered by the appropriate events. Continue by adding actions to these transitions.
- Identify any entry or exit actions (especially if you find that the idiom they cover is used in the state machine).
- Expand these states as necessary by using substates.
- Check that all events mentioned in the state machine match events expected by the interface of the object. Similarly, check that all events expected by the interface of the object are handled by the state machine. Finally, look to places where you explicitly want to ignore events.
- Check that all actions mentioned in the state machine are sustained by the relationships, methods, and operations of the enclosing object.
- Trace through the state machine, either manually or by using tools, to check it against expected sequences of events and their responses. Be especially diligent in looking for unreachable states and states in which the machine may get stuck.
- After rearranging your state machine, check it against expected sequences again to ensure that you have not changed the object's semantics.

For example, Figure shows the state machine for the controller in a home security system, which is responsible for monitoring ~~various sensors around the~~ perimeter of the house.

Figure Modeling the Lifetime of An Object



In the lifetime of this controller class, there are four main states: **Initializing**(the controller isstarting up), **Idle** (the controller is ready and waiting for alarms or commands from the user), **Command** (the controller is processing commands from the user), and **Active** (the controller isprocessing an alarm condition). When the controller object is first created, it moves first to the **Initializing** state and then unconditionally to the**Idle**state. The details of these two statesare not shown, other than the self-transition with the time event in the **Idle** state. This kind of time event is common in embedded systems, which often have a heartbeat timer that causes a periodic check of the system's health.

Control passes from the **Idle** state to the **Active** state on receipt of an **alarm** event (which includes the parameter **s,** identifying the sensor that was tripped). On entering the **Active** state, **setAlarm**is dispatched as the entry action, and control then passes first to the **Checking** state(validating the alarm), then to the **Calling** state (calling the alarm company to register the alarm), and finally to the **Waiting** state. The **Active** and **Waiting** states are exited only upon **clear**ing the alarm, or by the user signaling the controller for **attention,** presumably to issuea command.

Notice that there is no final state. That, too, is common in embedded systems, which are intended to run continuously.

Processes and thread

Terms and Concepts

An *active object* is an object that owns a process or thread and can initiate control activity. An *active class* is a class whose instances are active objects. A *process* is a heavyweight flow thatcan execute concurrently with other processes. A *thread* is a lightweight flow that can execute concurrently with other threads within the same process. Graphically, an active class is rendered as a rectangle with thick lines. Processes and threads are rendered as stereotyped active classes (and also appear as sequences in interaction diagrams).

Flow of Control

In a purely sequential system, there is one flow of control. This means that one thing, and one thing only, can take place at a time. When a sequential program starts, control is rooted at the beginning of the program and operations are dispatched one after another. Even if there are concurrent things happening among the actors outside the system, a sequential program will process only one event at a time, queuing or discarding any concurrent external events.This is why it's called a flow of control. If you trace the execution of a sequential program, you'll see the locus of execution flow from one statement to another, in sequential order. You might see actions that branch, loop, and jump about, and if there is any recursion or iteration, you see the flow circle back on itself. Nonetheless, in a sequential system, there would be a single flow of execution.

In a concurrent system, there is more than one flow of control• that is, more than one thing can take place at a time. In a concurrent system, there are multiple simultaneous flows of control, each rooted at the head of an independent process or a thread. If you take a snapshot of a concurrent system while it's running, you'll logically see multiple loci of execution.

In the UML, you use an active class to represent a process or thread that is the root of an independent flow of control and that is concurrent with all peer flows of control.

Classes and Events

Active classes are just classes, albeit ones with a very special property. An active class represents an independent flow of control, whereas a plain class embodies no such flow. In contrast to active classes, plain classes are implicitly called passive because they cannot independently initiate control activity.

You use active classes to model common families of processes or threads. In technical terms, this means that an active object• an instance of an active class• reifies (is a manifestation of) a process or thread. By modeling concurrent systems with active objects, you give a name to each independent flow of control. When an active object is created, the associated flow of control is started; when the active object is destroyed, the associated flow of control is terminated.

Active classes share the same properties as all other classes. Active classes may have instances. Active classes may have attributes and operations. Active classes may participate in dependency, generalization, and association (including aggregation) relationships. Active classes may use any of the UML's extensibility mechanisms, including stereotypes, tagged values, and constraints. Active classes may be the realization of interfaces. Active classes may be realized by collaborations, and the behavior of an active class may be specified by using state machines.

In your diagrams, active objects may appear wherever passive objects appear. You can model the collaboration of active and passive objects by using interaction diagrams (including sequence and collaboration diagrams). An active object may appear as the target of an event in a state machine.

Speaking of state machines, both passive and active objects may send and receive signal events and call events.

Standard Elements

All of the UML's extensibility mechanisms apply to active classes. Most often, you'll use tagged values to extend active class properties, such as specifying the scheduling policy of the active class.

The UML defines two standard stereotypes that apply to active classes.

1. **process**	Specifies a heavyweight flow that can execute concurrently with other processes
2. **thread**	Specifies a lightweight flow that can execute concurrently with other threads withinthe same process

The distinction between a process and a thread arises from the two different ways a flow of control may be managed by the operating system of the node on which the object resides.

A process is heavyweight, which means that it is a thing known to the operating system itself and runs in an independent address space. Under most operating systems, such as Windows and Unix, each program runs as a process in its own address space. In general, all processes on a node are peers of one another, contending for all the same resources accessible on the node. Processes are never nested inside one another. If the node has multiple processors, then true concurrency on that node is possible. If the node has only one processor, there is only the illusion of true concurrency, carried out by the underlying operating system.

A thread is lightweight. It may be known to the operating system itself. More often, it is hidden inside a heavier-weight process and runs inside the address space of the enclosing process. In Java, for example, a thread is a child of the class **Thread.** All the threads that live in the context of a process are peers of one another, contending for the same resources accessible inside the process. Threads are never nested inside one another. In general, there is only the illusion of true concurrency among threads because it is processes, not threads, that are scheduled by a node's operating system.

Communication

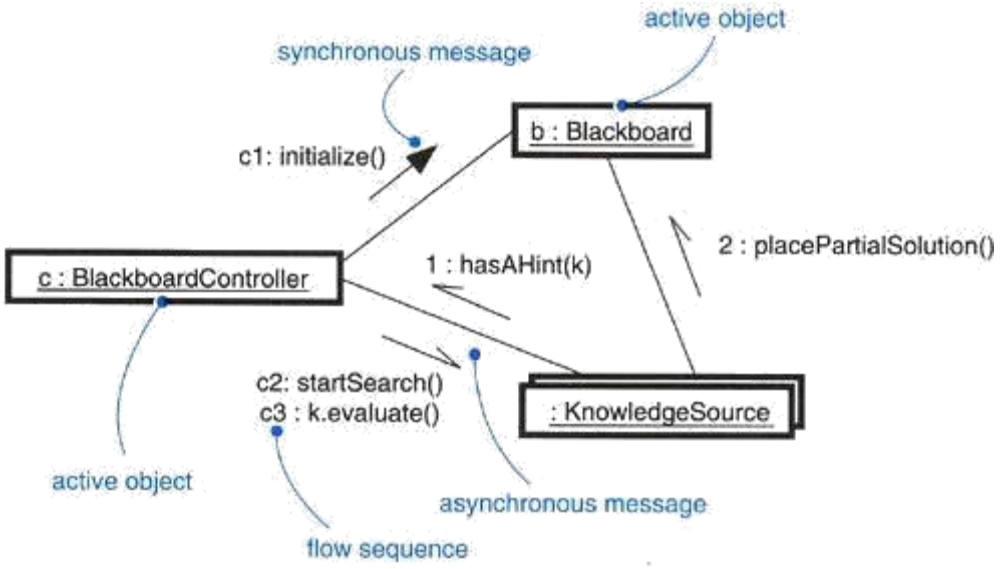
When objects collaborate with one another, they interact by passing messages from one to the other. In a system with both active and passive objects, there are four possible combinations of interaction that you must consider.

First, a message may be passed from one passive object to another. Assuming there is only one flow of control passing through these objects at a time, such an interaction is nothing more than the simple invocation of an operation.

Second, a message may be passed from one active object to another. When that happens, you have interprocess communication, and there are two possible styles of communication. First, one active object might synchronously call an operation of another. That kind of communication has rendezvous semantics, which means that the caller calls the operation; the caller waits for the receiver to accept the call; the operation is invoked; a return object (if any) is passed back to the caller; and then the two continue on their independent paths. For the duration of the call, the two flows of controls are in lock step. Second, one active object might asynchronously send a signal or call an operation of another object. That kind of communication has mailbox semantics, which means that the caller sends the signal or calls the operation and then continues on its independent way. In the meantime, the receiver accepts the signal or call whenever it is ready (with intervening events or calls queued) and continues on its way after it is done. This is called a mailbox because the two objects are not synchronized; rather, one object drops off a message for the other.

In the UML, you render a synchronous message as a full arrow and an asynchronous message as a half arrow, as in Figure .

Figure Communication



Third, a message may be passed from an active object to a passive object. A difficulty arises if more than one active object at a time passes their flow of control through one passive object. In that situation, you have to model the synchronization of these two flows very carefully, as discussed in the next section.

Fourth, a message may be passed from a passive object to an active one. At first glance, this may seem illegal, but if you remember that every flow of control is rooted in some active object, you'll understand

that a passive object passing a message to an active object has the same semantics as an active object passing a message to an active object.

Synchronization

Visualize for a moment the multiple flows of control that weave through a concurrent system. When a flow passes through an operation, we say that at a given moment, the locus of control is in the operation. If that operation is defined for some class, we can also say that at a given moment, the locus of control is in a specific instance of that class. You can have multiple flows of control in one operation (and therefore in one object), and you can have different flows of control in different operations (but still result in multiple flows of control in the one object).

The problem arises when more than one flow of control is in one object at the same time. If you are not careful, anything more than one flow will interfere with another, corrupting the state of the object. This is the classical problem of mutual exclusion. A failure to deal with it properly yields all sorts of race conditions and interference that cause concurrent systems to fail in mysterious and unrepeatable ways.

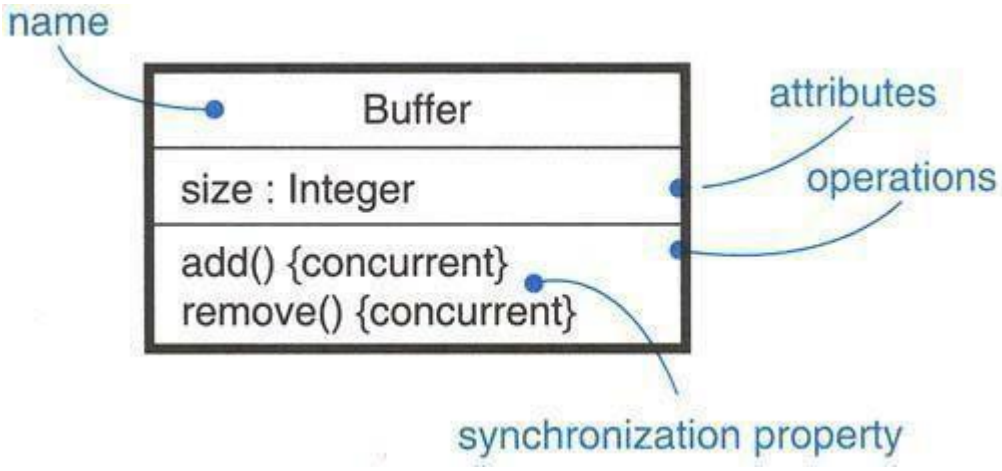
The key to solving this problem in object-oriented systems is by treating an object as a critical region. There are three alternatives to this approach, each of which involves attaching certain synchronization properties to the operations defined in a class. In the UML, you can model all three approaches.

1. Sequential	Callers must coordinate outside the object so that only one flow is in the object at a time. In the presence of multiple flows of control, the semantics and integrity of the object cannot be guaranteed.
2. Guarded	The semantics and integrity of the object is guaranteed in the presence of multiple flows of control by sequentializing all calls to all of the object's guarded operations. In effect, exactly one operation at a time can be invoked on the object, reducing this to sequential semantics.
3. Concurrent	The semantics and integrity of the object is guaranteed in the presence of multiple flows of control by treating the operation as atomic.

Some programming languages support these constructs directly. Java, for example, has the **synchronized** property, which is equivalent to the UML's **concurrent** property. In everylanguage that supports concurrency, you can build support for all these properties by constructing them out of semaphores.

As Figure shows, you can attach these properties to an operation, which you can render in the UML by using constraint notation.

Figure Synchronization



Note

It is possible to model variations of these synchronization primitives by using constraints. For example, you might modify the **concurrent** property by allowing multiple simultaneous readers but only a single writer.

Process Views

Active objects play an important role in visualizing, specifying, constructing, and documenting a system's process view. 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• that is, class diagrams, interaction diagrams, activity diagrams, and statechart diagrams, but with a focus on the active classes that represent these threads and processes.

Common Modeling Techniques

Modeling Multiple Flows of Control

Building a system that encompasses multiple flows of control is hard. Not only do you have to decide how best to divide work across concurrent active objects, but once you've done that, you also have to devise the right mechanisms for communication and synchronization among your system's active and passive objects to ensure that they behave properly in the presence of these multiple flows. For that reason, it helps to visualize the way these flows interact with one another. You can do that in the UML by applying class diagrams (to capture their static semantics) and interaction diagrams (to capture their dynamic semantics) containing active classes and objects.

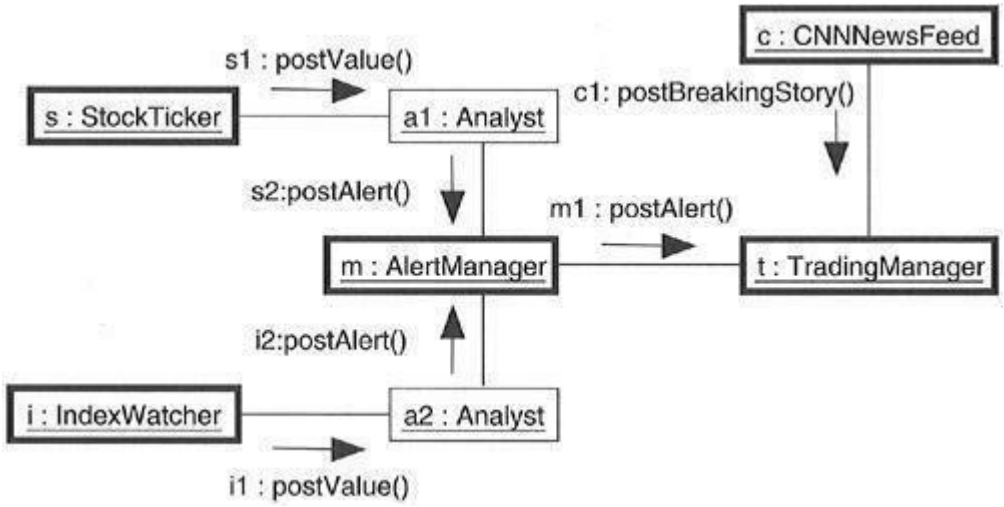
To model multiple flows of control,

- Identify the opportunities for concurrent action and reify each flow as an active class. Generalize common sets of active objects into an active class. Be careful not to over-engineer the process view of your system by introducing too much concurrency.

- Consider a balanced distribution of responsibilities among these active classes, then examine the other active and passive classes with which each collaborates statically. Ensure that each active class is both tightly cohesive and loosely coupled relative to these neighboring classes and that each has the right set of attributes, operations, and signals.
- Capture these static decisions in class diagrams, explicitly highlighting each active class.
- Consider how each group of classes collaborates with one another dynamically. Capture those decisions in interaction diagrams. Explicitly show active objects as the root of such flows. Identify each related sequence by identifying it with the name of the active object.
- Pay close attention to communication among active objects. Apply synchronous and asynchronous messaging, as appropriate.
- Pay close attention to synchronization among these active objects and the passive objects with which they collaborate. Apply sequential, guarded, or concurrent operation semantics, as appropriate.

For example, Figure shows part of the process view of a trading system. You'll find three objects that push information into the system concurrently: a **StockTicker,** an **IndexWatcher,** and a**CNNNewsFeed**(named **s, i,** and **c,** respectively). Two of these objects (**s**and**i)**communicate with their own**Analyst**instances (**a1**and**a2**). At least as far as thismodel goes, the **Analyst** can be designed under the simplifying assumption that only one flow of control will be active in its instances at a time. Both **Analyst** instances, however, communicate simultaneously with an **AlertManager** (named **m**). Therefore, **m** must be designed to preserve its semantics in the presence of multiple flows. Both **m** and **c** communicate simultaneously with **t**, a **TradingManager.** Each flow is given a sequence number that is distinguished by the flow of control that owns it.

Figure Modeling Flows of Control



Note

Interaction diagrams such as these are useful in helping you to visualize where two flows of control might cross paths and, therefore, where you must pay particular attention to the problems of communication and synchronization. Tools are permitted to offer even more distinct visual cues, such as by coloring each flow in a distinct way.

In diagrams such as this, it's also common to attach corresponding state machines, with orthogonal states showing the detailed behavior of each active object.

Modeling Interprocess Communication

As part of incorporating multiple flows of control in your system, you also have to consider the mechanisms by which objects that live in separate flows communicate with one another. Across threads (which live in the same address space), objects may communicate via signals or call events, the latter of which may exhibit either asynchronous or synchronous semantics. Across processes (which live in separate address spaces), you usually have to use different mechanisms.

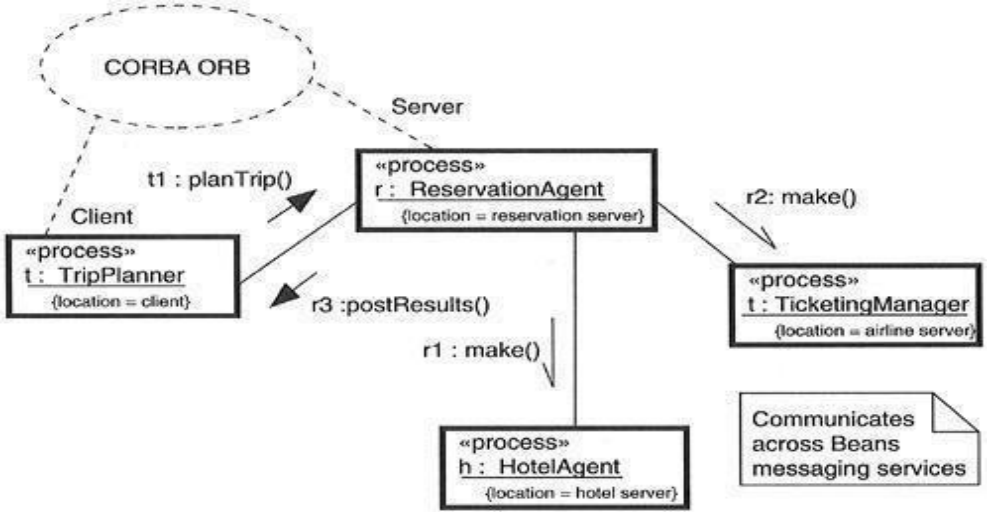
The problem of interprocess communication is compounded by the fact that, in distributed systems, processes may live on separate nodes. Classically, there are two approaches tointerprocess communication: message passing and remote procedure calls. In the UML, you still model these as asynchronous or synchronous events, respectively. But because these are no longer simple in-process calls, you need to adorn your designs with further information.

To model interprocess communication,

- Model the multiple flows of control.
- Consider which of these active objects represent processes and which represent threads. Distinguish them using the appropriate stereotype.
- Model messaging using asynchronous communication; model remote procedure calls using synchronous communication.
- Informally specify the underlying mechanism for communication by using notes, or more formally by using collaborations.

Figure shows a distributed reservation system with processes spread across four nodes.Each object is marked using the **process** stereotype. Each object is also marked with a **location** tagged value, specifying its physical location. Communication among the**ReservationAgent, TicketingManager,** and**HotelAgent**is asynchronous. Modeled witha note, communication is described as building on a Java Beans messaging service. Communication between the **TripPlanner** and the **ReservationSystem** is synchronous. The semantics of their interaction is found in the collaboration named **CORBA ORB.** The
TripPlanneracts as a **client,** and the**ReservationAgent**acts as a **server.** By zoominginto the collaboration, you'll find the details of how this server and client collaborate.

Figure Modeling Interprocess Communication



Time and Space

Terms and Concepts

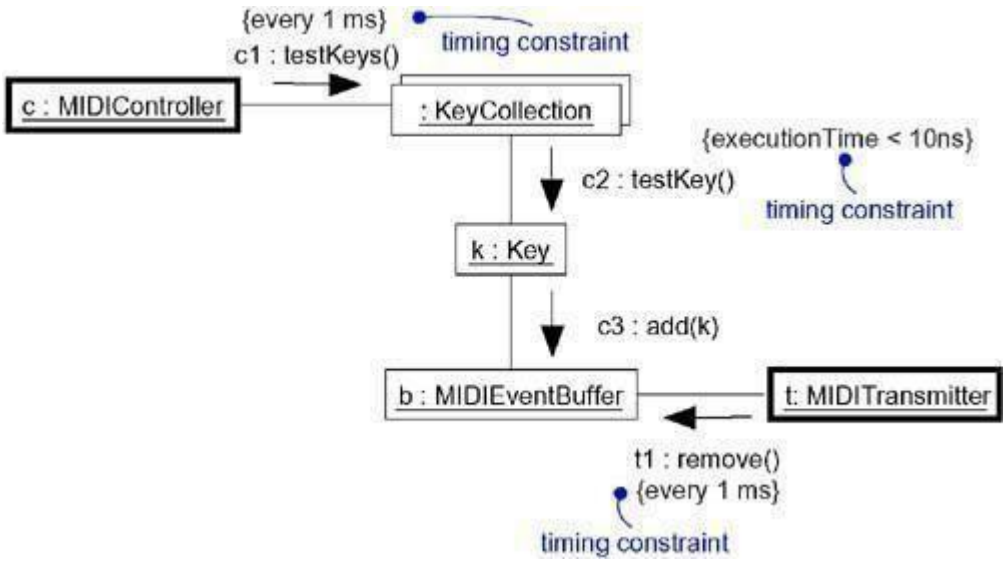
A *timing mark* is a denotation for the time at which an event occurs. Graphically, a timing mark is formed as an expression from the name given to the message (which is typically different from the name of the action dispatched by the message). A *time expression* is an expression that evaluates to an absolute or relative value of time. A *timing constraint* is a semantic statement about the relative or absolute value of time. Graphically, a timing constraint is rendered as for any constraint• that is, a string enclosed by brackets and generally connected to an element by a dependency relationship. *Location* is the placement of a component on a node. Graphically, location is rendered as a tagged value• that is, a string enclosed by brackets and placed below an element's name, or as the nesting of components inside nodes.

Time

Real time systems are, by their very name, time-critical systems. Events may happen at regular or irregular times; the response to an event must happen at predictable absolute times or at predictable times relative to the event itself.

The passing of messages represents the dynamic aspect of any system, so when you model the time-critical nature of a system with the UML, you can give a name to each message in an interaction to be used as a timing mark. Messages in an interaction are usually not given names. They are mainly rendered with the name of an event, such as a signal or a call. As a result, you can't use the event name to write an expression because the same event may trigger different messages. If the designated message is ambiguous, use the explicit name of the message in a timing mark to designate the message you want to mention in a time expression. A timing mark is nothing more than an expression formed from the name of a message in an interaction. Given a message name, you can refer to any of three functions of that message• that is, **startTime**,**stopTime**, and**executionTime.** You can then use these functions to specify arbitrarilycomplex time expressions, perhaps even using weights or offsets that are either constants or variables (as long as those variables can be bound at execution time). Finally, as shown in Figure , you can place these time expressions in a timing constraint to specify the timingbehavior of the system. As constraints, you can render them by placing them adjacent to the appropriate message, or you can explicitly attach them using dependency relationships.

Figure Time

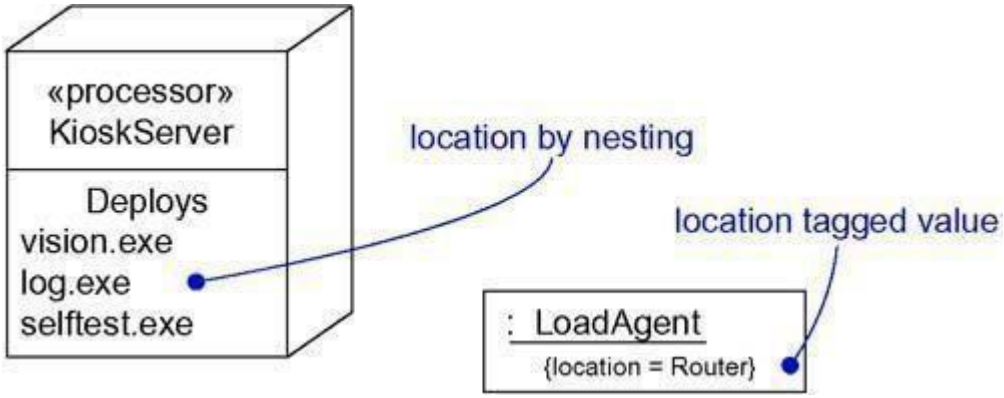


Location

Distributed systems, by their nature, encompass components that are physically scattered among the nodes of a system. For many systems, components are fixed in place at the time they are loaded on the system; in other systems, components may migrate from node to node.

In the UML, you model the deployment view of a system by using deployment diagrams that represent the topology of the processors and devices on which your system executes. Components such as executables, libraries, and tables reside on these nodes. Each instance of a node will own instances of certain components, and each instance of a component will be owned by exactly one instance of a node (although instances of the same kind of component may be spread across different nodes). For example, as Figure shows, the executable component **vision.exe** may reside on the node named**KioskServe**r.

Figure Location



Instances of plain classes may reside on a node, as well. For example, as Figure shows, an instance of the class **LoadAgent** lives on the node named **Router.**

As the figure illustrates, you can model the location of an element in two ways in the UML. First, as shown for the **KioskServer**, you can physically nest the element (textually or graphically) in a extra compartment in its enclosing node. Second, as shown for the **LoadAgent**, you can use the defined tagged value **location** to designate the node on which the class instance resides.
You'll typically use the first form when it's important for you to give a visual cue in your diagrams about the spatial separation and grouping of components. Similarly, you'll use the second form when modeling the location of an element is important, but secondary, to the diagram at hand, such as when you want to visualize the passing of messages among instances.

Common Modeling Techniques

Modeling Timing Constraints

Modeling the absolute time of an event, modeling the relative time between events, and modeling the time it takes to carry out an action are the three primary time-critical properties of real time systems for which you'll use timing constraints.

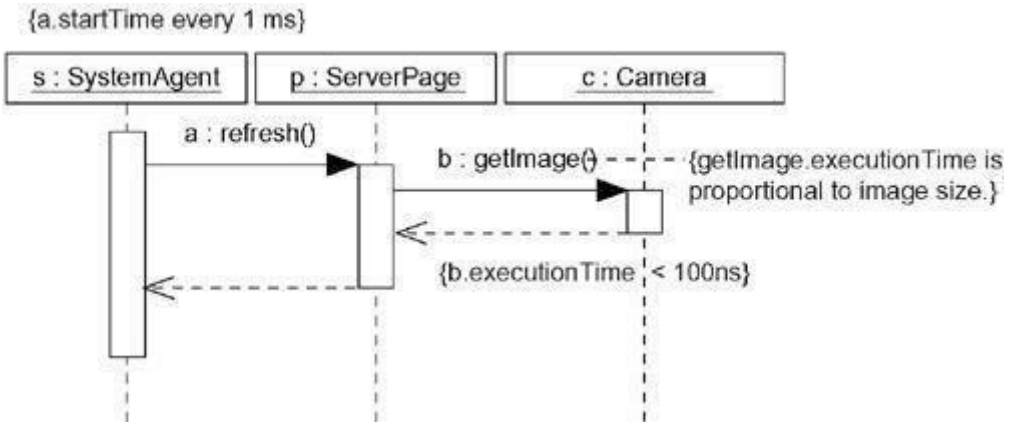
To model timing constraints,

- For each event in an interaction, consider whether it must start at some absolute time. Model that real time property as a timing constraint on the message.
- For each interesting sequence of messages in an interaction, consider whether there is an associated maximum relative time for that sequence. Model that real time property as a timing constraint on the sequence.

- For each time critical operation in each class, consider its time complexity. Model those semantics as timing constraints on the operation.

For example, as shown in Figure the left-most constraint specifies the repeating start time the call event **refresh.** Similarly, the center timing constraint specifies the maximum duration for calls to **getImage.** Finally, the right-most constraint specifies the time complexity of the call event **getImage.**

Figure Modeling Timing Constraint



Note

Observe that **executionTime**may be applied to actions such as**getImage**, as wellas to timing marks such as **a** and**b**. Also, timing constraints such as these may be written as free- form text. If you want to specify your semantics more precisely, you can use the UML's Object Constraint Language (OCL), described further in *The UnifiedModeling Language Reference Manual.*

Often, you'll choose short names for messages, so that you don't confuse them with operation names.

Modeling the Distribution of Objects

When you model the topology of a distributed system, you'll want to consider the physical placement of both components and class instances. If your focus is the configuration management of the deployed system, modeling the distribution of components is especially important in order to visualize, specify, construct, and document the placement of physical things such as executables, libraries, and tables. If your focus is the functionality, scalability, and throughput of the system, modeling the distribution of objects is what's important.

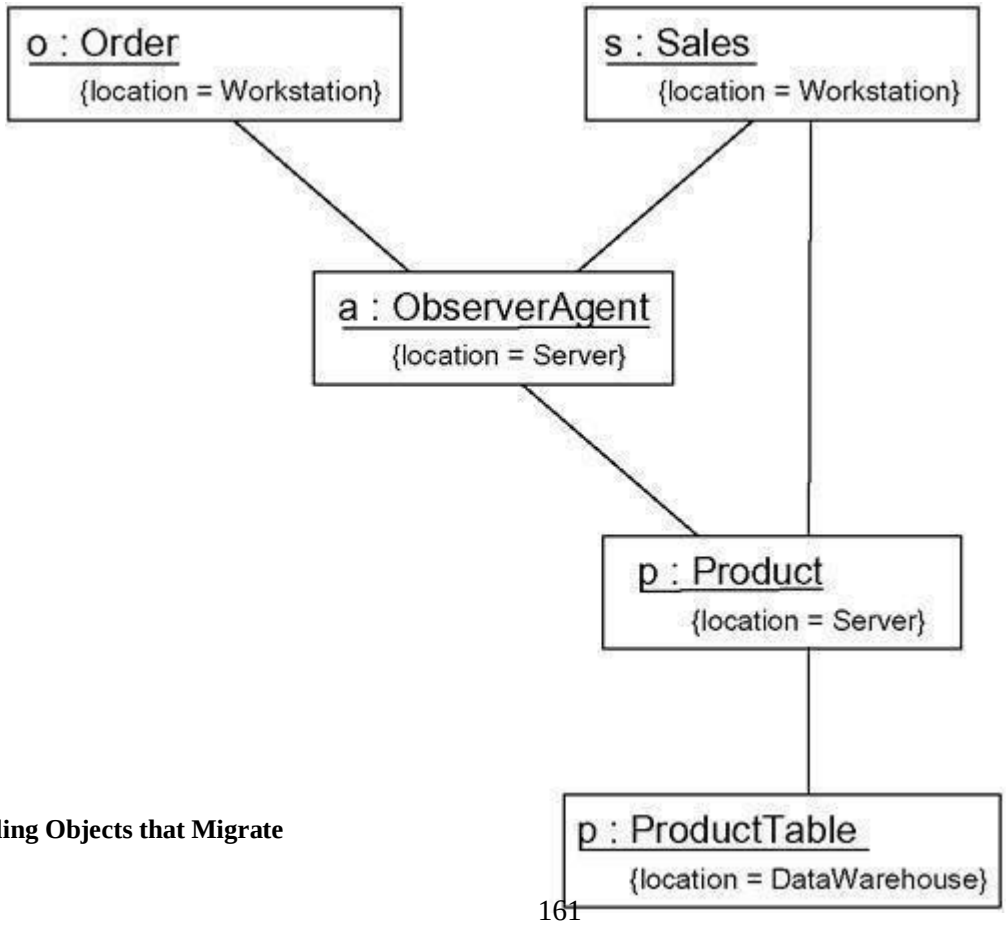
Deciding how to distribute the objects in a system is a wicked problem, and not just because the problems of distribution interact with the problems of concurrency. Naive solutions tend to yield profoundly poor performance, and over-engineering solutions aren't much better. In fact, they are probably worse because they usually end up being brittle.

To model the distribution of objects,

- For each interesting class of objects in your system, consider its locality of reference. In other words, consider all its neighbors and their locations. A tightly coupled locality will have neighboring objects close by; a loosely coupled one will have distant objects (and thus, there will be latency in communicating with them). Tentatively allocate objects closest to the actors that manipulate them.
- Next consider patterns of interaction among related sets of objects. Co-locate sets of objects that have high degrees of interaction, to reduce the cost of communication. Partition sets of objects that have low degrees of interaction.
- Next consider the distribution of responsibilities across the system. Redistribute your objects to balance the load of each node.
- Consider also issues of security, volatility, and quality of service, and redistribute your objects as appropriate.
- Render this allocation in one of two ways:
    1. By nesting objects in the nodes of a deployment diagram
    2. By explicitly indicating the location of the object as a tagged value

Figure provides an object diagram that models the distribution of certain objects in a retailsystem. The value of this diagram is that it lets you visualize the physical distribution of certain key objects. As the diagram shows, two objects reside on a **Workstation** (the **Order** and **Sales** objects), two objects reside on a **Server** (the**ObserverAgent**and the **Product** objects), and one object resides on a **DataWarehouse** (the **ProductTable** object).

Figure Modeling the Distribution of Objects



Modeling Objects that Migrate

For many distributed systems, components and objects, once loaded on the system, stay put. For their lifetime, from creation to destruction, they never leave the node on which they were born. There are certain classes of distributed systems, however, for which things move about, usually for one of two reasons.

First, you'll find objects migrating in order to move closer to actors and other objects they need to work with to do their job better. For example, in a global shipping system, you'd see objects that represent ships, containers, and manifests moving from node to node to track their physical counterpart. If you have a ship in Hong Kong, it makes for better locality of reference to put the object representing the ship, its containers, and its manifest on a node in Hong Kong. When that ship sails to San Diego, you'd want to move the associated objects, as well.

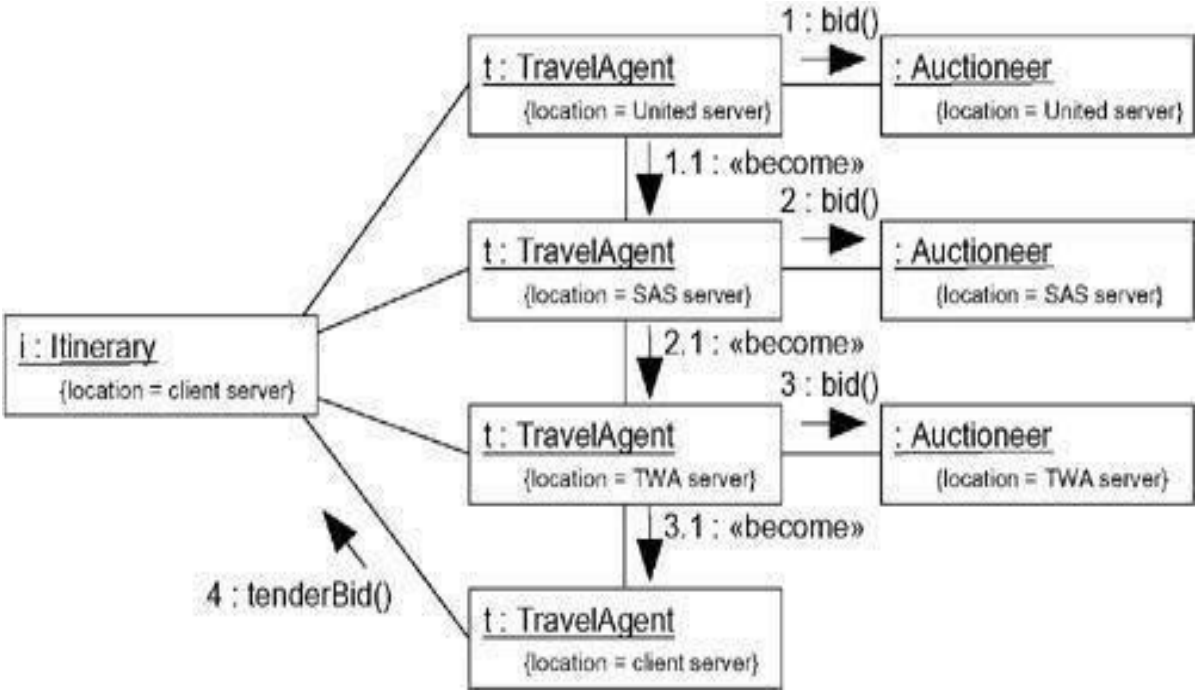
Second, you'll find objects migrating in response to the failure of a node or connection or to balance the load across multiple nodes. For example, in an air traffic control system, the failure of one node cannot be allowed to stall a nation's entire operations. Rather, a failure-tolerant system such as this will migrate elements to other nodes. Performance and throughput may be reduced,
but safe functionality will be preserved. Similarly, and especially in Web-based systems that must deal with unpredictable peaks in demand, you'll often want to build mechanisms to automatically balance the processing load, perhaps by migrating components and objects to underused nodes.

Deciding how to migrate the objects in a system is an even more wicked problem than simple static distribution because migration raises difficult problems of synchronization and preservation of identity.

To model the migration of objects,

Figure provides a collaboration diagram that models the migration of a Web agent thatmoves from node to node, collecting information and bidding on resources in order to automatically deliver a lowest-cost travel ticket. Specifically, this diagram shows an instance (named **t**) of the class **TravelAgent** migrating from one server to another. Along the way, the object interacts with anonymous **Auctioneer** instances at each node, eventually delivering a bid for the **Itinerary** object, located on the **client server.**

Figure Modeling Objects that Migrate



Statechart Diagrams

Terms and Concepts

A *statechart diagram* shows a state machine, emphasizing the flow of control from state to state. A *state machine* is a behavior that specifies the sequences of states an object goes through during its lifetime in response to events, together with its responses to those events. A *state* is a condition or situation in the life of an object during which it satisfies some condition, performs some activity, or waits for some event. An *event* is the specification of a significant occurrence that has a location in time and space. In the context of state machines, an event is an occurrence of a stimulus that can trigger a state transition. A *transition* is a relationship between two states indicating that an object in the first state will perform certain actions and enter the second state when a specified event occurs and specified conditions are satisfied. An *activity* is ongoing nonatomic execution within a state machine. An *action* is an executable atomic computation that results in a change in state of the model or the return of a value. Graphically, a statechart diagram is a collection of vertices and arcs.

Common Properties

A statechart diagram is just a special kind of diagram and shares the same common properties as do all other diagrams• that is, a name and graphical contents that are a projection into a model. What distinguishes a statechart diagram from all other kinds of diagrams is its content.

Contents

Statechart diagrams commonly contain

- Simple states and composite states
- Transitions, including events and actions distinguishes an activity diagram from a statechart diagram is that an activity diagram is basically a projection of the elements found in an activity graph, a special case of a state machine in which all or most states are activity states and in which all or most transitions are triggered by completion of activities in the source state.

Common Uses

You use statechart diagrams to model the dynamic aspects of a system. These dynamic aspects may involve the event- ordered behavior of any kind of object in any view of a system's architecture, including classes (which includes active classes), interfaces, components, and nodes.

When you use a statechart diagram to model some dynamic aspect of a system, you do so in the context of virtually any modeling element. Typically, however, you'll use statechart diagrams in the context of the system as a whole, a subsystem, or a class. You can also attach statechart diagrams to use cases (to model a scenario).

When you model the dynamic aspects of a system, a class, or a use case, you'll typically use statechart diagrams in one way.

- To model reactive objects

A reactive — or event-driven — object is one whose behavior is best characterized by its response to events dispatched from outside its context. A reactive object is typically idle until it receives an event. When it receives an event, its response usually depends on previous events. After the object responds to an event, it becomes idle again, waiting for the next event. For these kinds of objects, you'll focus on the stable states of that object, the events that trigger a transition from state to state, and the actions that occur on each state change.

Common Modeling Technique

Modeling Reactive Objects

The most common purpose for which you'll use statechart diagrams is to model the behavior of reactive objects, especially instances of classes, use cases, and the system as a whole. Whereasinteractions model the behavior of a society of objects working together, a statechart diagram models the behavior of a single object over its lifetime. Whereas an activity diagram models the flow of control from activity to activity, a statechart diagram models the flow of control from event to event.

When you model the behavior of a reactive object, you essentially specify three things: the stable states in which that object may live, the events that trigger a transition from state to state, and the actions that occur on each state change. Modeling the behavior of a reactive object also involves modeling the lifetime of an object, starting at the time of the object's creation and continuing until its destruction, highlighting the stable states in which the object may be found.

A stable state represents a condition in which an object may exist for some identifiable period of time. When an event occurs, the object may transition from state to state. These events may also trigger self- and internal transitions, in which the source and the target of the transition are the same state. In reaction to an event or a state change, the object may respond by dispatching an action.

To model a reactive object,

- Choose the context for the state machine, whether it is a class, a use case, or the system as a whole.
- Choose the initial and final states for the object. To guide the rest of your model, possibly state the pre- and postconditions of the initial and final states, respectively.
- Decide on the stable states of the object by considering the conditions in which the object may exist for some identifiable period of time. Start with the high-level states of the object and only then consider its possible substates.
- Decide on the meaningful partial ordering of stable states over the lifetime of the object.
- Decide on the events that may trigger a transition from state to state. Model these events as triggers to transitions that move from one legal ordering of states to another.
- Attach actions to these transitions (as in a Mealy machine) and/or to these states (as in a Moore machine).
- Consider ways to simplify your machine by using substates, branches, forks, joins, and history states.
- Check that all states are reachable under some combination of events.
- Check that no state is a dead end from which no combination of events will transition the object out of that state.
- Trace through the state machine, either manually or by using tools, to check it against expected sequences of events and their responses.

For example, Figure shows the statechart diagram for parsing a simple context- freelanguage, such as you might find in systems that stream in or stream out messages to XML. In this case, the machine is designed to parse a stream of characters that match the syntax

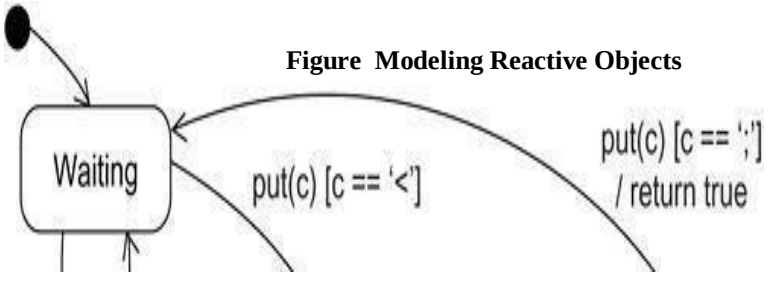


Figure Modeling Reactive Objects

The first string represents a tag; the second string represents the body of the message. Given a stream ofcharacters, only well-formed messages that follow this syntax may be accepted.

As the figure shows, there are only three stable states for this state machine: **Waiting**, **GettingToken**, and**GettingBody.** This statechart is designed as a Mealy machine, withactions tied to transitions. In fact, there is only one event of interest in this state machine, the invocation of **put** with the actual parameter **c** (a character). While **Waiting**, this machine throws away any character that does not designate the start of a token (as specified by the guard condition). When the start of a token is received, the state of the object changes to **GettingToken.** While in that state, the machine saves any character that does not designatethe end of a token (as specified by the guard condition). When the end of a token is received, the state of the object changes to **GettingBody.** While in that state, the machine saves any character that does not designate the end of a message body (as specified by the guard condition). When the end of a message is received, the state of the object changes to **Waiting**,and a value is returned indicating that the message has been parsed (and the machine is ready to receive another message).Note that this statechart specifies a machine that runs continuously; there is no final state.

Forward and Reverse Engineering

Forward engineering(the creation of code from a model) is possible for statechart diagrams,especially if the context of the diagram is a class. For example, using the previous statechart diagram, a forward engineering tool could generate the following Java code for the class
MessageParser.

```
classMessageParser { public
  boolean put(char c) { switch (state) {
     case Waiting:
      if (c == '<') {
        state = GettingToken;
        token = new StringBuffer(); body = new
        StringBuffer();
      }
      break;
```

```
            caseGettingToken : if (c == '>')
                state = GettingBody; else
                token.append(c);
              break;
            caseGettingBody : if (c == ';')
               state = Waiting; else
                    body.append(c); return true;
       }
       return false;
     }
     StringBuffergetToken() { return token;
     }
     StringBuffergetBody() { return body;
     }
  private
     final static int Waiting = 0; final static intGettingToken =
     1; final static intGettingBody = 2; int state = Waiting;
     StringBuffer token, body;
  }
```

This requires a little cleverness. The forward engineering tool must generate the necessary private attributes and final static constants.

Reverse engineering (the creation of a model from code) is theoretically possible, but practicallynot very useful. The choice of what constitutes a meaningful state is in the eye of the designer.Reverse engineering tools have no capacity for abstraction and therefore cannot automatically produce meaningful statechart diagrams. More interesting than the reverse engineering of a model from code is the animation of a model against the execution of a deployed system. For example, given the previous diagram, a tool could animate the states in the diagram as they were reached in the running system. Similarly, the firing of transitions could be animated, showing the receipt of events and the resulting dispatch of actions. Under the control of a debugger, you could control the speed of execution, setting breakpoints to stop the action at interesting states to examine the attribute values of individual objects.

Components

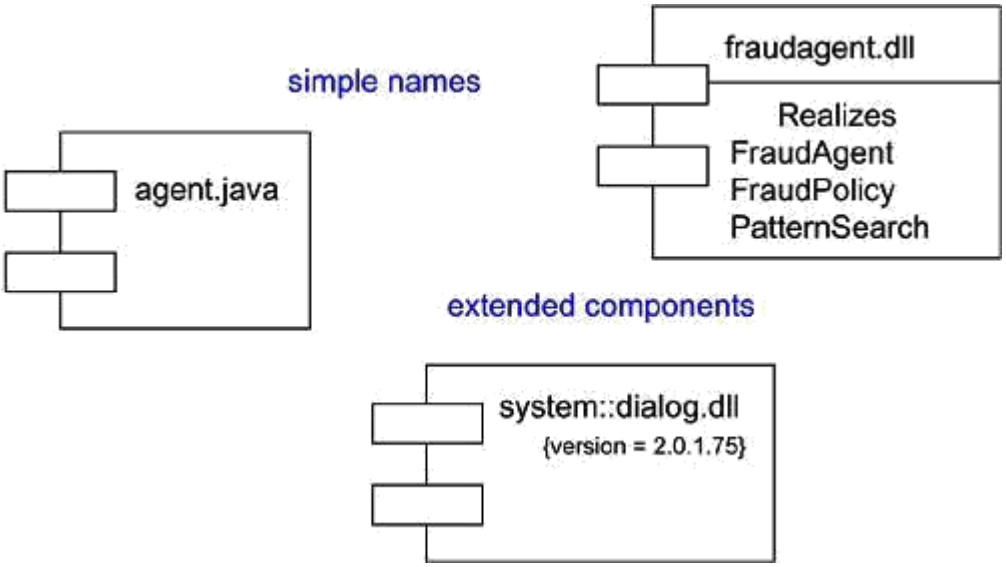
Terms and Concepts

A *component* is a physical and replaceable part of a system that conforms to and provides the realization of a set of interfaces. Graphically, a component is rendered as a rectangle with tabs.

Names

Every component must have a name that distinguishes it from other components. A *name* is a textual string. That name alone is known as a *simple name;* a *path name* is the component name prefixed by the name of the package in which that component lives. A component is typically drawn showing only its name, as in Figure . Just as with classes, you may draw components adorned with tagged values or with additional compartments to expose their details, as you see in the figure.

Figure Simple and Extended Component



Components and Classes

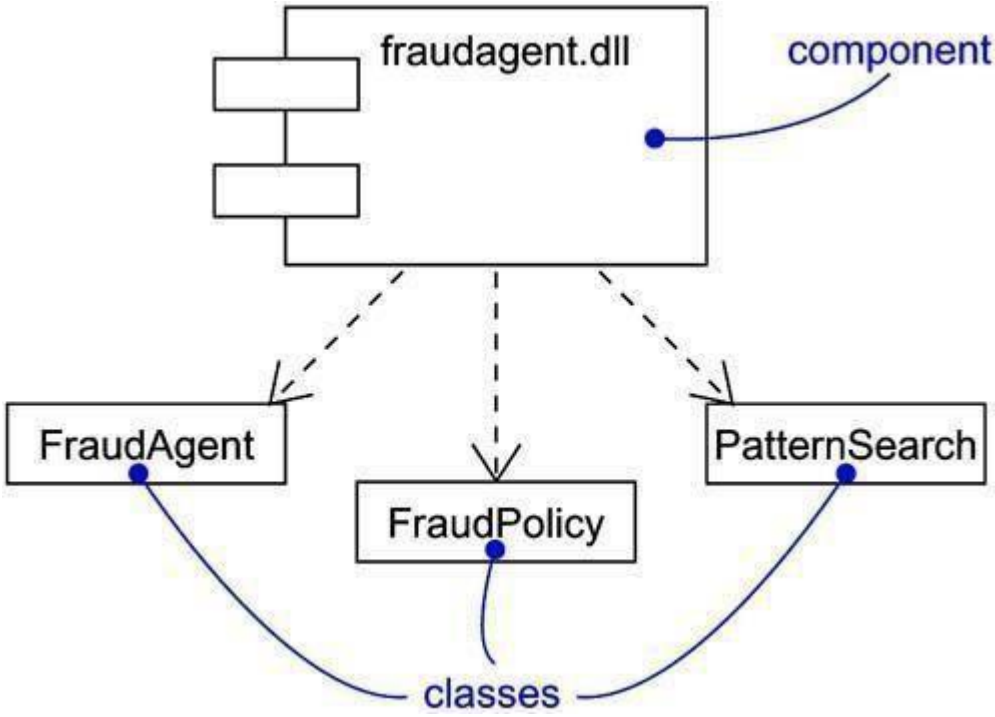
In many ways, components are like classes: Both have names; both may realize a set of interfaces; both may participate in dependency, generalization, and association relationships; both may be nested; both may have instances; both may be participants in interactions. However, there are some significant differences between components and classes.

- Classes represent logical abstractions; components represent physical things that live in the world of bits. In short, components may live on nodes, classes may not.
- Components represent the physical packaging of otherwise logical components and are at a different level of abstraction.
- Classes may have attributes and operations directly. In general, components only have operations that are reachable only through their interfaces.

The first difference is the most important. When modeling a system, deciding if you should use a class or a component involves a simple decision• if the thing you are modeling lives directly on a node, use a component; otherwise, use a class.

The second difference suggests a relationship between classes and components. In particular, a component is the physical implementation of a set of other logical elements, such as classes and collaborations. As Figure shows, the relationship between a component and the classes it implements can be shown explicitly by using a dependency relationship. Most of the time, you'll never need to visualize these relationships graphically. Rather, you will keep them as a part of the component's specification.

Figure Components and Classes



The third difference points out how interfaces bridge components and classes. As described in more detail in the next section, components and classes may both realize an interface, but a component's services are usually available only through its interfaces.

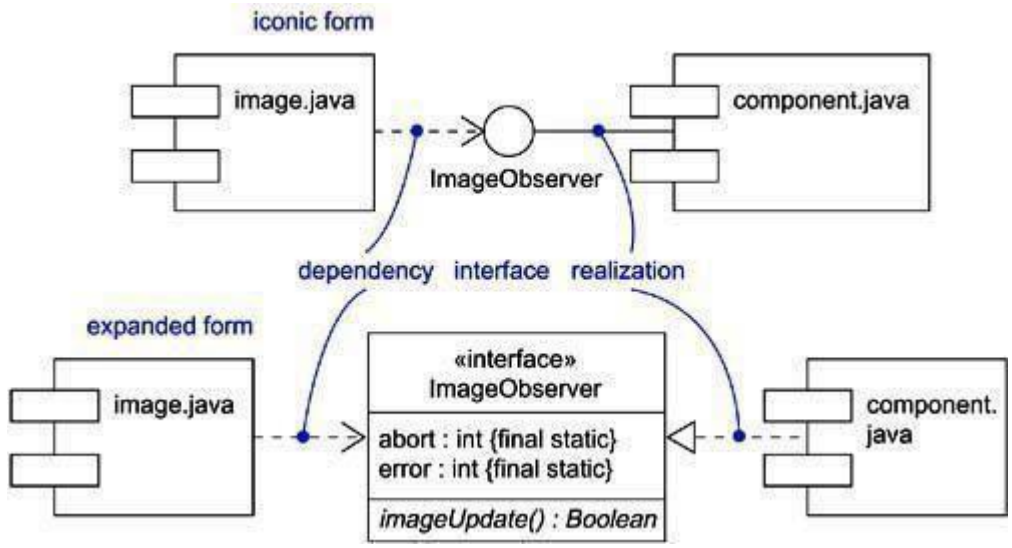
Components and Interfaces

An interface is a collection of operations that are used to specify a service of a class or a component. The relationship between component and interface is important. All the most common component-based operating system facilities (such as COM+, CORBA, and Enterprise Java Beans) use interfaces as the glue that binds components together.

Using one of these facilities, you decompose your physical implementation by specifying interfaces that represent the major seams in the system. You then provide components that realize the interfaces, along with other components that access the services through their interfaces. This mechanism permits you to deploy a system whose services are somewhat location-independent and, as discussed in the next section, replaceable.

As Figure indicates, you can show the relationship between a component and its interfaces in one of two ways. The first (and most common) style renders the interface in its elided, iconic form. The component that realizes the interface is connected to the interface using an elided realization relationship. The second style renders the interface in its expanded form, perhaps revealing its operations. The component that realizes the interface is connected to the interface using a full realization relationship. In both cases, the component that accesses the services of the other component through the interface is connected to the interface using a dependencyrelationship.

Figure Components and Interfaces



An interface that a component realizes is called an *export interface*, meaning an interface that the component provides as a service to other components. A component may provide many export interfaces. The interface that a component uses is called an *import interface*, meaning an interface that the component conforms to and so builds on. A component may conform to many import interfaces. Also, a component may both import and export interfaces.

A given interface may be exported by one component and imported by another. The fact that this interface lies between the two components breaks the direct dependency between the components. A component that uses a given interface will function properly no matter what component realizes that interface. Of course, a component can be used in a context if and only if all its import interfaces are provided by the export interfaces of other components.

Binary Replaceability

The basic intent of every component-based operating system facility is to permit the assembly of systems from binary replaceable parts. This means that you can create a system out of components and then evolve that system by adding new components and replacing old ones, without rebuilding the system. Interfaces are the key to making this happen. When you specify an interface, you can drop into the executable system any component that conforms to or provides that interface. You can extend the system by making the components provide new services through other interfaces, which, in turn, other components can discover and use. These semantics explain the intent behind the definition of components in the UML. A component is a physical and replaceable part of a system that conforms to and provides the realization of a set of interfaces.

First, a component is *physical.* It lives in the world of bits, not concepts.

Second, a component is *replaceable.* A component is substitutable• it is possible to replace a component with another that conforms to the same interfaces. Typically, the mechanism of inserting or replacing a component to form a run time system is transparent to the component user and is enabled by object models (such as COM+ and Enterprise Java Beans) that require little or no intervening transformation or by tools that automate the mechanism.

Third, a component is *part of a system.* A component rarely stands alone. Rather, a given component collaborates with other components and in so doing exists in the architectural or technology context in which it is intended to be used. A component is logically and physically cohesive and thus denotes a meaningful structural and/or behavioral

chunk of a larger system. A component may be reused across many systems. Therefore, a component represents a fundamental building block on which systems can be designed and composed. This definition is recursive• a system at one level of abstraction may simply be a component at a higher level of abstraction.

Fourth, as discussed in the previous section, a component *conforms to and provides the realization of a set of interfaces.*

Kinds of Components

Three kinds of components may be distinguished.

First, there are *deployment components.* These are the components necessary and sufficient to form an executable system, such as dynamic libraries (DLLs) and executables (EXEs). The UML's definition of component is broad enough to address classic object models, such as COM+, CORBA, and Enterprise Java Beans, as well as alternative object models, perhaps involving dynamic Web pages, database tables, and executables using proprietary communication mechanisms.

Second, there are *work product components.* These components are essentially the residue of the development process, consisting of things such as source code files and data files from which deployment components are created. These components do not directly participate in an executable system but are the work products of development that are used to create the executable system.

Third are *execution components.* These components are created as a consequence of an executing system, such as a COM+ object, which is instantiated from a DLL.

Organizing Components

You can organize components by grouping them in packages in the same manner in which you organize classes.

You can also organize components by specifying dependency, generalization, association (including aggregation), and realization relationships among them.

Standard Elements

All the UML's extensibility mechanisms apply to components. Most often, you'll use tagged values to extend component properties (such as specifying the version of a development component) and stereotypes to specify new kinds of components (such as operating system-specific components).

The UML defines five standard stereotypes that apply to components:

1. **executable**	Specifies a component that may be executed on a node
2. **library**	Specifies a static or dynamic object library
3. **table**	Specifies a component that represents a database table
4. **file**	Specifies a component that represents a document containing source code or Data
5. **document**	Specifies a component that represents a document

Common Modeling Techniques

Modeling Executables and Libraries

The most common purpose for which you'll use components is to model the deployment components that make up your implementation. If you are deploying a trivial system whose implementation consists of exactly one executable file, you will not need to do any component modeling. If, on the other hand, the system you are deploying is made up of several executables and associated object libraries, doing component modeling will help you to visualize, specify, construct, and document the decisions you've made about the physical system. Component modeling is even more important if you want to control the versioning and configuration management of these parts as your system evolves.

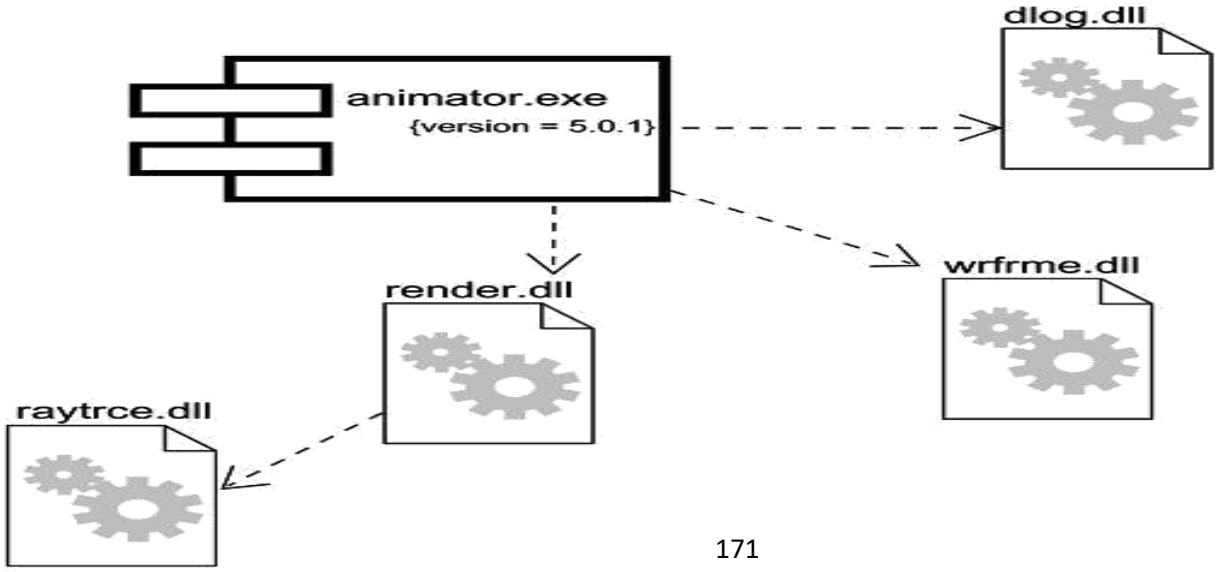
For most systems, these deployment components are drawn from the decisions you make about how to segment the physical implementation of your system. These decisions will be affected by a number of technical issues (such as your choice of component-based operating system facilities), configuration management issues (such as your decisions about which parts will likely change over time), and reuse issues (that is, deciding which components you can reuse in or from other systems).

To model executables and libraries,

- Identify the partitioning of your physical system. Consider the impact of technical, configuration management, and reuse issues.
- Model any executables and libraries as components, using the appropriate standard elements. If your implementation introduces new kinds of components, introduce a new appropriate stereotype.
- If it's important for you to manage the seams in your system, model the significant interfaces that some components use and others realize.
- As necessary to communicate your intent, model the relationships among these executables, libraries, and interfaces. Most often, you'll want to model the dependencies among these parts in order to visualize the impact of change.

For example, Figure shows a set of components drawn from a personal productivity tool that runs on a single personal computer. This figure includes one executable (**animator.exe**, with a tagged value noting its version number) and four libraries (**dlog.dll**, **wrfrme.dll**, **render.dll**, and **raytrce.dll**), all of which use the UML's standard elements for executables and libraries, respectively. This diagram also presents the dependencies among these components.

Figure Modeling Executables and Libraries



As your models get bigger, you will find that many components tend to cluster together in groups that are conceptually and semantically related. In the UML, you can use packages to model these clusters of components.For larger systems that are deployed across several computers, you'll want to model the way your components are distributed by asserting the nodes on which they are located.

Modeling Tables, Files, and Documents

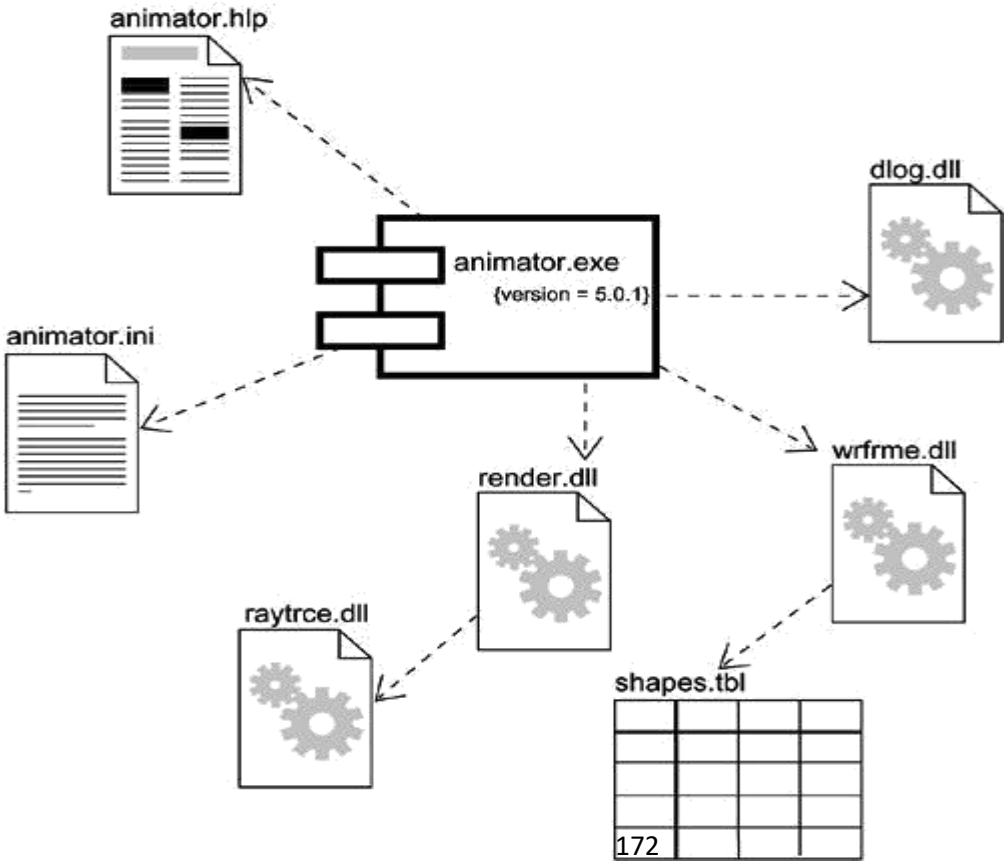
Modeling the executables and libraries that make up the physical implementation of your system is useful, but often you'll find there are a host of ancillary deployment components that are neither executables nor libraries and yet are critical to the physical deployment of your system. For example, your implementation might include data files, help documents, scripts, log files, initialization files, and installation/removal files. Modeling these components is an important part of controlling the configuration of your system. Fortunately, you can use UML components to model all of these artifacts.

To model tables, files, and documents,

- Identify the ancillary components that are part of the physical implementation of your system.
- Model these things as components. If your implementation introduces new kinds of artifacts, introduce a new appropriate stereotype.
- As necessary to communicate your intent, model the relationships among these ancillary components and the other executables, libraries, and interfaces in your system. Most often, you'll want to model the dependencies among these parts in order to visualize the impact of change.

For example, Figure builds on the previous figure and shows the tables, files, and documents that are part of the deployed system surrounding the executable **animator.exe.** This figure includes one document (**animator.hlp**), one simple file (**animator.ini**), and one database table (**shapes.tbl**), all of which use the UML's standard elements for cuments, files, and tables, respectively.

Figure Modeling Tables, Files, and Documents



Modeling databases can get complicated when you start dealing with multiple tables, triggers, and stored procedures. To visualize, specify, construct, and document these features, you'll need to model the logical schema, as well as the physical databases.

Modeling an API

If you are a developer who's assembling a system from component parts, you'll often want to see the application programming interfaces (APIs) that you can use to glue these parts together. APIs represent the programmatic seams in your system, which you can model using interfaces and components.

An API is essentially an interface that is realized by one or more components. As a developer, you'll really care only about the interface itself; which component realizes an interface's operations is not relevant as long as *some* component realizes it. From a system configuration management perspective, though, these realizations are important because you need to ensure that, when you publish an API, there's some realization available that carries out the API's obligations. Fortunately, with the UML, you can model both perspectives.

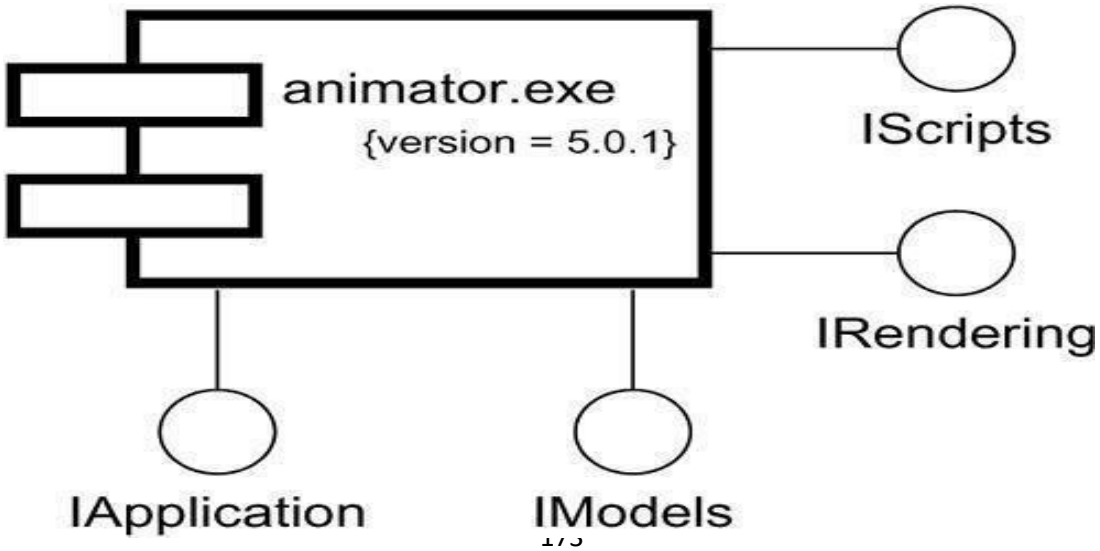
The operations associated with any semantically rich API will be fairly extensive, so most of the time you won't need to visualize all these operations at once. Instead, you'll tend to keep theoperations in the backplane of your models and use interfaces as handles with which you can find these sets of operations. If you want to construct executable systems against these APIs, you will need to add enough detail so that your development tools can compile against the properties of your interfaces. Along with the signatures of each operation, you'll probably also want to include uses cases that explain how to use each interface.

To model an API,

- Identify the programmatic seams in your system and model each seam as an interface, collecting the attributes and operations that form this edge.
- Expose only those properties of the interface that are important to visualize in the given context; otherwise, hide these properties, keeping them in the interface's specification for reference, asnecessary.
- Model the realization of each API only insofar as it is important to show the configuration of a specific implementation.

Figure exposes the APIs of the executable in the previous two figures. You'll see four interfaces that form the API of the executable: **IApplication**, **IModels**, **IRendering**, and **IScripts.**

Figure Modeling an API



Modeling Source Code

The most common purpose for which you'll use components is to model the physical parts that make up your implementation. This also includes the modeling of all the ancillary parts of your deployed system, including tables, files, documents, and APIs. The second most common purpose for which you'll use components is to model the configuration of all the source code files that your development tools use to create these components. These represent the work product components of your development process.

Modeling source code graphically is particularly useful for visualizing the compilation dependencies among your source code files and for managing the splitting and merging ofgroups of these files when you fork and join development paths. In this manner, UML components can be the graphical interface to your configuration management and version control tools.

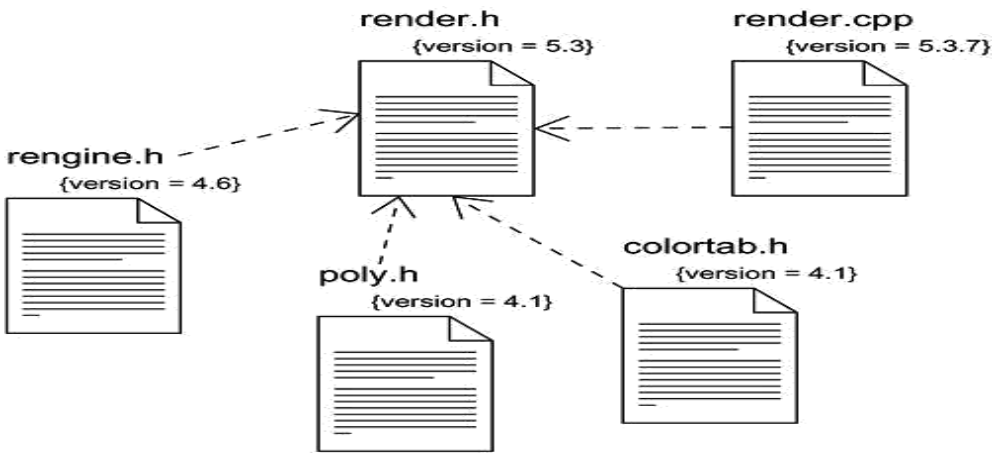
For most systems, source code files are drawn from the decisions you make about how to segment the files your development environment needs. These files are used to store the details of your classes, interfaces, collaborations, and other logical elements as an intermediate step to creating the physical, binary components that are derived from these elements by your tools. Most of the time, these tools will impose a style of organization (one or two files per class is common), but you'll still want to visualize the relationships among these files. How you organize groups of these files using packages and how you manage versions of these files is driven by your decisions about how to manage change.

To model source code,

- Depending on the constraints imposed by your development tools, model the files used to store the details of all your logical elements, along with their compilation dependencies.
- If it's important for you to bolt these models to your configuration management and version control tools, you'll want to include tagged values, such as version, author, and check in/check out information, for each file that's under configuration management.
- As far as possible, let your development tools manage the relationships among these files, and use the UML only to visualize and document these relationships.

For example, Figure shows some source code files that are used to build the library r**ender.dll** from the previous examples. This figure includes four header files (**render.h**, **rengine.h**, **poly.h**, and **colortab.h**) that represent the source code for the specification of certain classes. There is also one implementation file (**render.cpp**) that represents the implementation of one of these headers.

Figure Modeling Source Code



As your models get bigger, you will find that many source code files tend to cluster together in groups that are conceptually and semantically related. Most of the time, your development tools will place these groups in separate directories. In the UML, you can use packages to model these clusters of source code files.

In the UML, it is possible to visualize the relationship of a class to its source code file and, in turn, the relationship of a source code file to its executable or library by using trace relationships. However, you'll rarely need to go to this detail of modeling.

Deployment

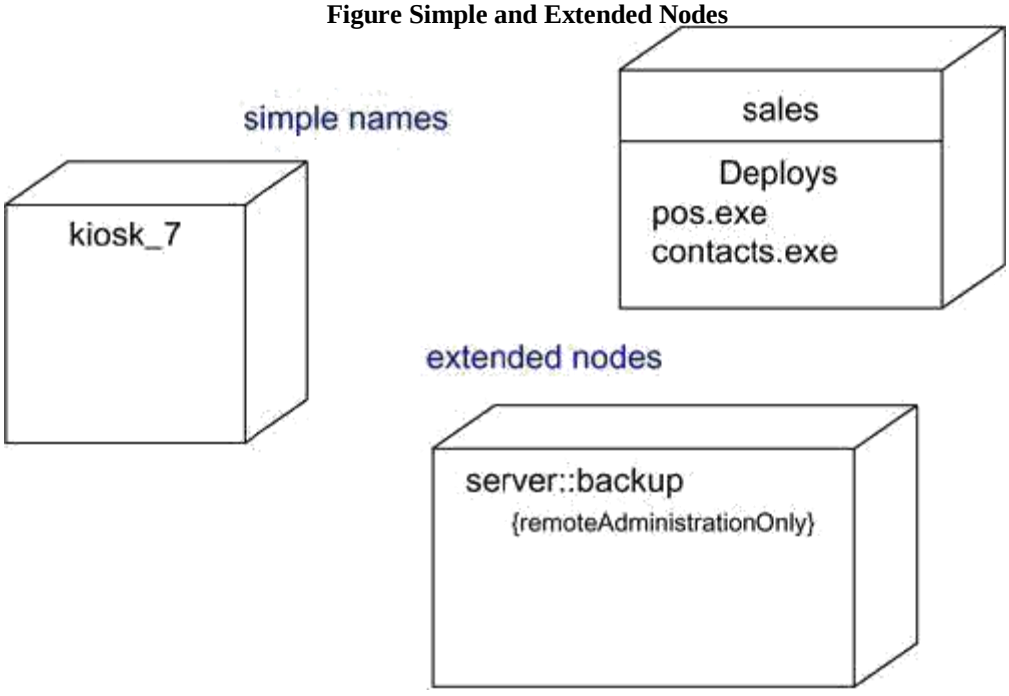
Terms and Concepts

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. Graphically, a node is rendered as a cube.

Names

Every node must have a name that distinguishes it from other nodes. A *name* is a textual string. That name alone is known as a *simple name;* a *path name* is the node name prefixed by the name of the package in which that node lives. A node is typically drawn showing only its name, as in Figure . Just as with classes, you may draw nodes adorned with tagged values or with additional compartments to expose their details.

Figure Simple and Extended Nodes



Note

A node name may be text consisting of any number of letters, numbers, and certain punctuation marks (except for marks such as the colon, which is used to separate a node name and the name of its enclosing package) and may continue over several lines. In practice, node names are short nouns or noun phrases drawn from the vocabulary of the implementation.

Nodes and Components

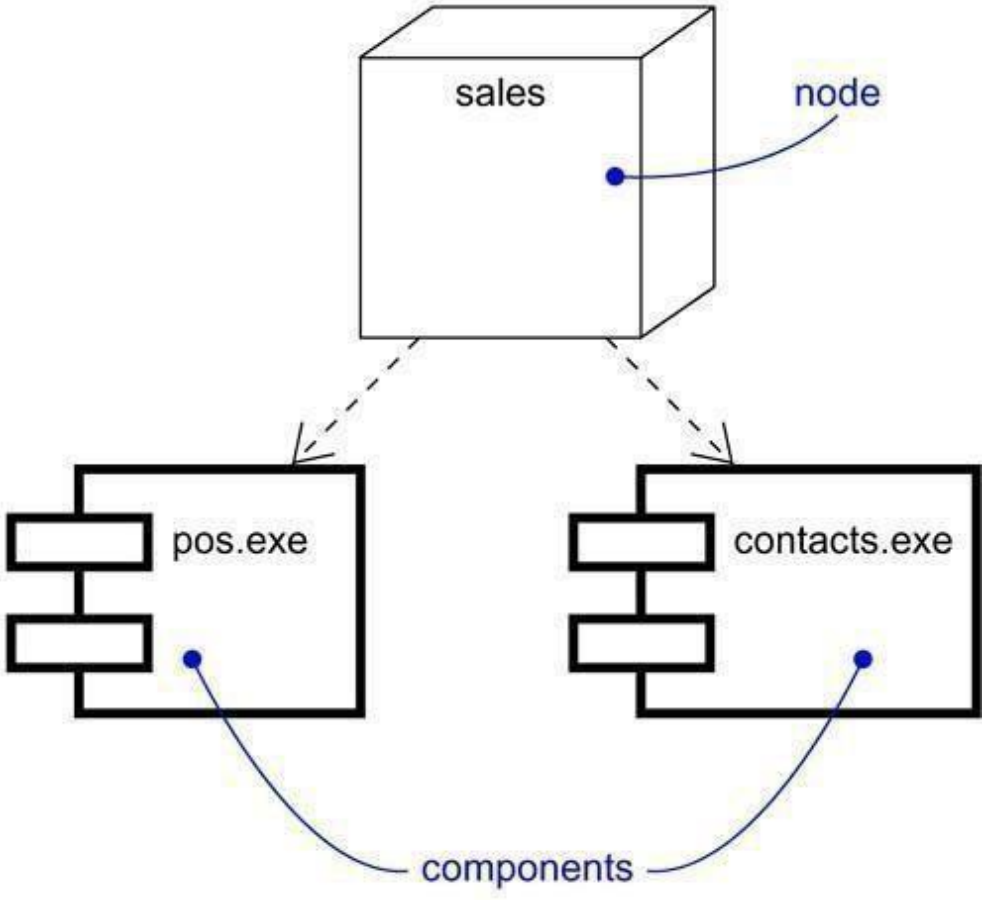
In many ways, nodes are a lot like components: Both have names; both may participate in dependency, generalization, and association relationships; both may be nested; both may have instances; both may be participants in interactions. However, there are some significant differences between nodes and components.

- Components are things that participate in the execution of a system; nodes are things that execute components.
- Components represent the physical packaging of otherwise logical elements; nodes represent the physical deployment of components.

This first difference is the most important. Simply put, nodes execute components; components are things that are executed by nodes.

The second difference suggests a relationship among classes, components, and nodes. In particular, a component is the materialization of a set of other logical elements, such as classes and collaborations, and a node is the location upon which components are deployed. A class may be implemented by one or more components, and, in turn, a component may be deployed on one or more nodes. As Figure shows, the relationship between a node and the components it deploys can be shown explicitly by using a dependency relationship. Most of the time, you won't need to visualize these relationships graphically but will keep them as a part of the node's specification.

Figure Nodes and Components



A set of objects or components that are allocated to a node as a group is called a *distribution unit.*

Note

Nodes are also class-like in that you can specify attributes and operations for them. For example, you might specify that a node provides the attributes **processorSpeed** and **memory,** as well as the operations **turnOn, turnOff,** and **suspend.**

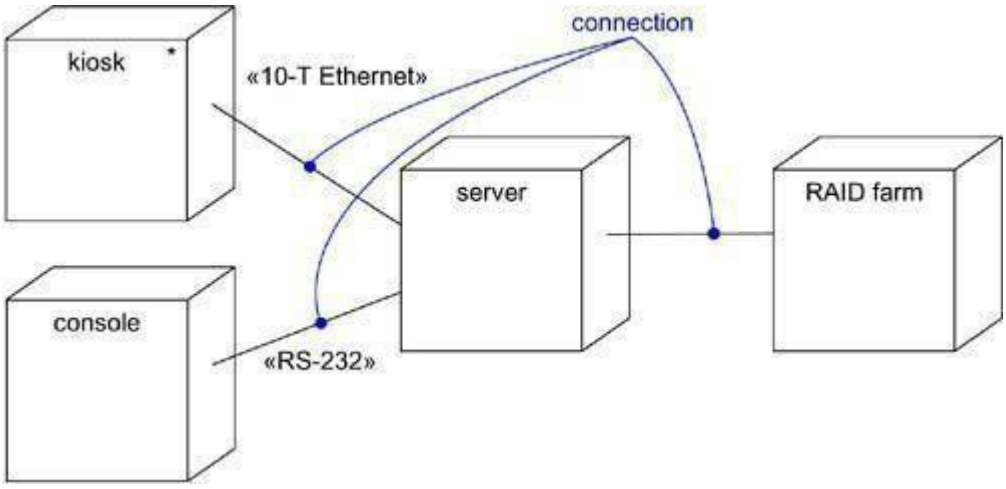
Organizing Nodes

You can organize nodes by grouping them in packages in the same manner in which you can organize classes and components.
You can also organize nodes by specifying dependency, generalization, and association (including aggregation) relationships among them.

Connections

The most common kind of relationship you'll use among nodes is an association. In this context, an association represents a physical connection among nodes, such as an Ethernet connection, a serial line, or a shared bus, as Figure shows. You can even use associations to model indirect connections, such as a satellite link between distant processors.

Figure Connections



Because nodes are class-like, you have the full power of associations at your disposal. This means that you can include roles, multiplicity, and constraints. As in the previous figure, you should stereotype these associations if you want to model new kinds of connections• for example, to distinguish between a 10-T Ethernet connection and an RS-232 serial connection.

Common Modeling Techniques

Modeling Processors and Devices

Modeling the processors and devices that form the topology of a stand-alone, embedded, client/server, or distributed system is the most common use of nodes.Because all of the UML's extensibility mechanisms apply to nodes, you will often use stereotypes to specify new kinds of nodes that you can use to represent specific kinds of processors and devices. A *processor* is a

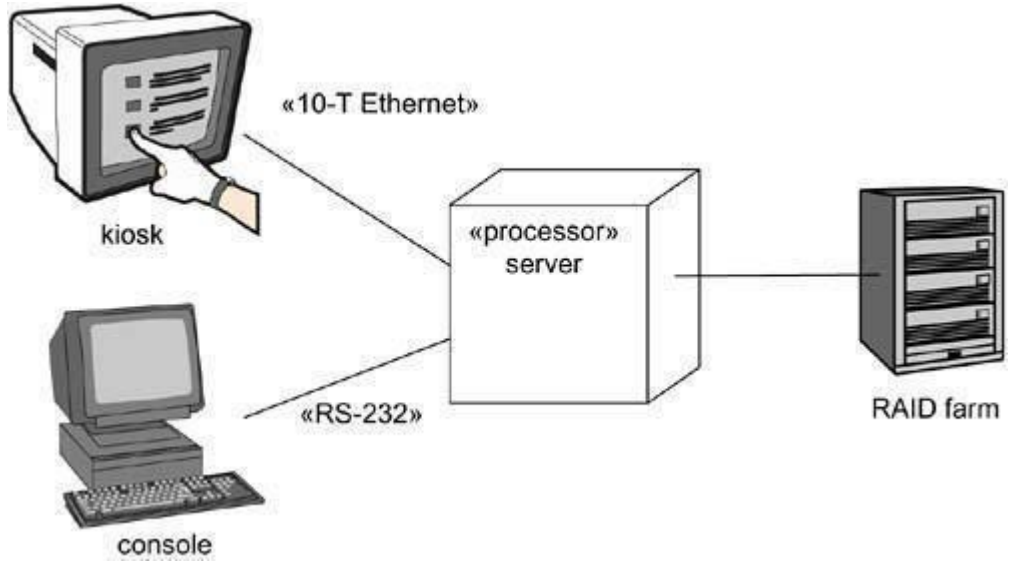
node that has processing capability, meaning that it can execute a component. A *device* is a node that has no processing capability (at least, none that are modeled at this level of abstraction) and, in general, represents something that interfaces to the real world.

To model processors and devices,

- Identify the computational elements of your system's deployment view and model each as a node.If these elements represent generic processors and devices, then stereotype them as such. If they are kinds of processors and devices that are part of the vocabulary of your domain, then specify an appropriate stereotype with an icon for each.
- As with class modeling, consider the attributes and operations that might apply to each node.

For example, Figure takes the previous diagram and stereotypes each node. The **server** is a node stereotyped as a generic processor; the **kiosk** and the **console** are nodes stereotyped as special kinds of processors; and the **RAID farm** is a node stereotyped as a special kind of device.

Figure Processors and Devices



Modeling the Distribution of Components

When you model the topology of a system, it's often useful to visualize or specify the physical distribution of its components across the processors and devices that make up the system.

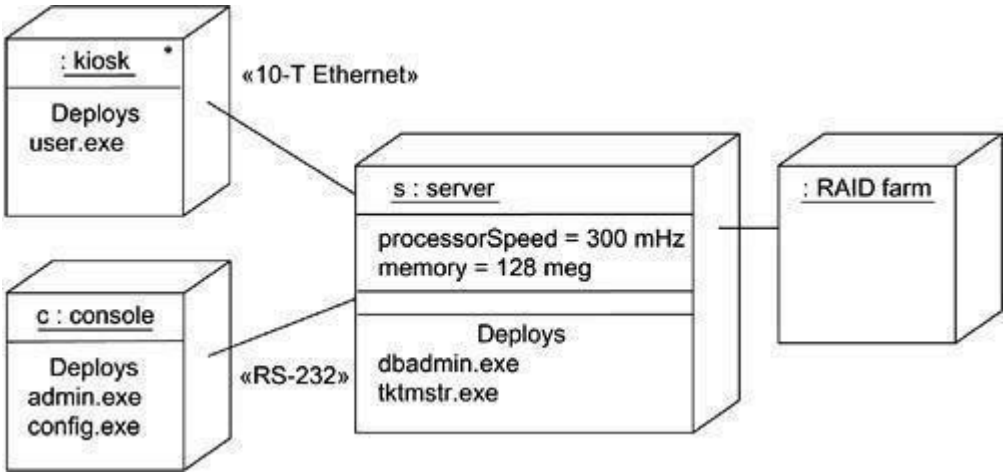
To model the distribution of components,

- For each significant component in your system, allocate it to a given node.

- Consider duplicate locations for components. It's not uncommon for the same kind of component (such as specific executables and libraries) to reside on multiple nodes simultaneously.
- Render this allocation in one of three ways.
    1. Don't make the allocation visible, but leave it as part of the backplane of your model• that is, in each node's specification.
    2. Using dependency relationships, connect each node with the components it deploys.
    3. List the components deployed on a node in an additional compartment.

Using the third approach, Figure takes the earlier diagrams and specifies the executable components that reside on each node. This diagram is a bit different from the previous ones in that it is an object diagram, visualizing specific instances of each node. In this case, the **RAID farm** and **kiosk** instances are both anonymous and the other two instances are named (c for the **console** and **s** for the **server**). Each processor in this figure is rendered with an additional compartment showing the component it deploys. The **server** object is also rendered with its attributes (**processorSpeed** and **memory**) and their values visible.

Figure Modeling the Distribution of Components.



Components need not be statically distributed across the nodes in a system. In the UML, it is possible to model the dynamic migration of components from node to node, as in an agent-based system or a high-reliability system that involves clustered servers and replicated databases.

Collaborations

Component Diagrams

Terms and Concepts

A *component diagram* shows a set of components and their relationships. Graphically, a component diagram is a collection of vertices and arcs.

Common Properties

A component diagram is just a special kind of diagram and shares the same common properties as do all other diagrams• a name and graphical contents that are a projection into a model. What distinguishes a component diagram from all other kinds of diagrams is its particular content.

Contents

Component diagrams commonly contain

- Components
- Interfaces
- Dependency, generalization, association, and realization relationships Like all

other diagrams, component diagrams may contain notes and constraints.

Component diagrams may also contain packages or subsystems, both of which are used to group elements of your model into larger chunks. Sometimes, you'll want to place instances in your component diagrams, as well, especially when you want to visualize one instance of a family of component-based systems.

Common Uses

You use component diagrams to model the static implementation view of a system. This view primarily supports the configuration management of a system's parts, made up of components that can be assembled in various ways to produce a running system.

When you model the static implementation view of a system, you'll typically use component diagrams in one of four ways.

1. To model source code

With most contemporary object- oriented programming languages, you'll cut code using integrated development environments that store your source code in files. You can use component diagrams to model the configuration management of these files, which represent work-product components.

2. To model executable releases

A release is a relatively complete and consistent set of artifacts delivered to an internal or external user. In the context of components, a release focuses on the parts necessary to deliver a running system. When you model a release using component diagrams, you are visualizing, specifying, and documenting the decisions about the physical parts that constitute your software• that is, its deployment components.

3. To model physical databases

Think of a physical database as the concrete realization of a schema, living in the world of bits. Schemas, in effect, offer an API to persistent information; the model of a physical database represents the storage of that information in the tables of a relational database or the pages of an object-oriented database. You use component diagrams to represent these and other kinds of physical databases.

4. To model adaptable systems

Some systems are quite static; their components enter the scene, participate in an execution, and then depart. Other systems are more dynamic, involving mobile agents or components that migrate for purposes of load balancing and failure recovery. You use component diagrams in conjunction with some of the UML's diagrams for modeling behavior to represent these kinds of systems.

Common Modeling Techniques

Modeling Source Code

If you develop software in Java, you'll usually save your source code in **.java** files. If you develop software using C++, you'll typically store your source code in header files (**.h** files) and bodies (**.cpp** files). If you use IDL to develop COM+ or CORBA applications, one interface from your design view will often expand into four source code files: the interface itself, the client proxy, the server stub, and a bridge class. As your application grows, no matter which language you use, you'll find yourself organizing these files into larger groups. Furthermore, during the construction phase of development, you'll probably end up creating new versions of some of these files for each new incremental release you produce, and you'll want to place these versions under the control of a configuration management system.

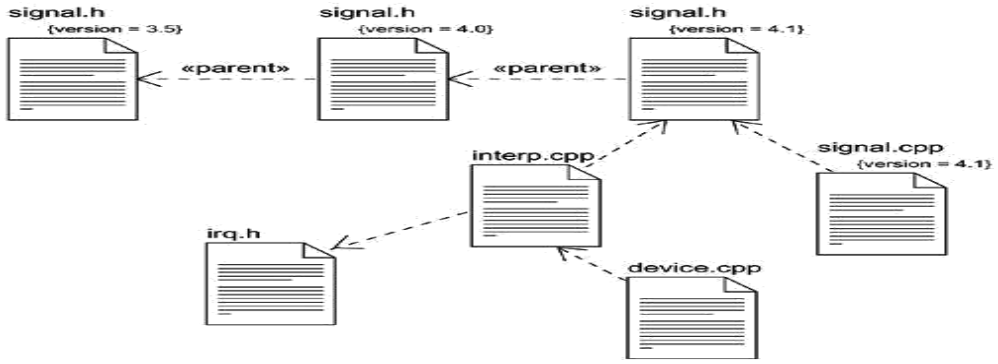
Much of the time, you will not need to model this aspect of a system directly. Instead, you'll let your development environment keep track of these files and their relationships. Sometimes, however, it's helpful to visualize these source code files and their relationships using component diagrams. Component diagrams used in this way typically contain only work-product components stereotyped as files, together with dependency relationships. For example, you might reverse engineer a set of source code files to visualize their web of compilation dependencies. You can go in the other direction by specifying the relationships among your source code files and then using those models as input to compilation tools, such as **make** on Unix. Similarly, you might want to use component diagrams to visualize the history of a set of source code files that are under configuration management. By extracting information from your configuration management system, such as the number of times a source code file has been checked out over a period of time, you can use that information to color component diagrams, showing "hot spots" of change among your source code files and areas of architectural churn.

To model a system's source code,

- Either by forward or reverse engineering, identify the set of source code files of interest and model them as components stereotyped as files.
- For larger systems, use packages to show groups of source code files.
- Consider exposing a tagged value indicating such information as the version number of the source code file, its author, and the date it was last changed. Use tools to manage the value of this tag.
- Model the compilation dependencies among these files using dependencies. Again, use tools to help generate and manage these dependencies.

For example, Figure shows five source code files. **signal.h** is a header file. Three of its versions are shown, tracing from new versions back to their older ancestors. Each variant of this source code file is rendered with a tagged value exposing its version number.

Figure Modeling Source Code



This header file (**signal.h**) is used by two other files (**interp.cpp** and **.signal.cpp**), both of which are bodies. One of these files (**interp.cpp**) has a compilation dependency to another header (**irq.h**); in turn, **device.cpp** has a compilation dependency to **interp.cpp.** Given this component diagram, it's easy to trace the impact of changes. For example, changing the source code file **signal.h** will require the recompilation of three other files: **signal.cpp**, **interp.cpp**, and transitively, **device.cpp.** As this diagram also shows, the file **irq.h** is not affected.

Diagrams such as this can easily be generated by reverse engineering from the information held by your development environment's configuration management tools.

Modeling an Executable Release

Releasing a simple application is easy: You throw the bits of a single executable file on a disk, and your users just run that executable. For these kinds of applications, you don't need component diagrams because there's nothing difficult to visualize, specify, construct, or document.

Releasing anything other than a simple application is not so easy. You need the main executable (usually, a **.exe** file), but you also need all its ancillary parts, such as libraries (commonly **.dll** files if you are working in the context of COM+, or **.class** and **.jar** files if you are working in the context of Java), databases, help files, and resource files. For distributed systems, you'll likely have multiple executables and other parts scattered across various nodes. If you are working with a system of applications, you'll find that some of these components are unique to each application but that many are shared among applications. As you evolve your system, controlling the configuration of these many components becomes an important activity• and a more difficult one because changes in the components associated with one application may affect the operation of other applications.

For this reason, you use component diagrams to visualize, specify, construct, and document the configuration of your executable releases, encompassing the deployment components that form each release and the relationships among those components. You can use component diagrams to forward engineer a new system and to reverse engineer an existing one.

When you create component diagrams such as these, you actually just model a part of the things and relationships that make up your system's implementation view. For this reason, each component diagram should focus on one set of components at a time.

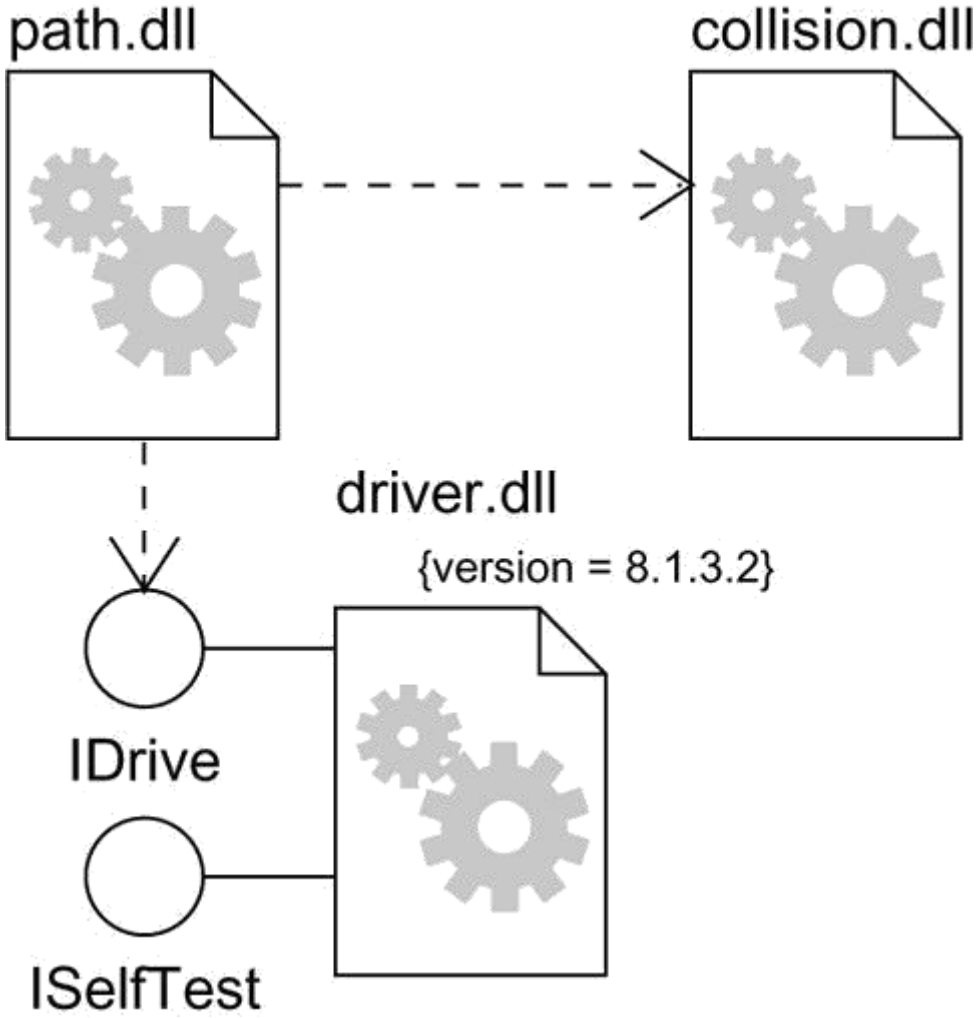
To model an executable release,

- Identify the set of components you'd like to model. Typically, this will involve some or all the components that live on one node, or the distribution of these sets of components across all the nodes in the system.
- Consider the stereotype of each component in this set. For most systems, you'll find a small number of different kinds of components (such as executables, libraries, tables, files, and documents). You can use the UML's extensibility mechanisms to provide visual cues for these stereotypes.
- For each component in this set, consider its relationship to its neighbors. Most often, this will involve interfaces that are exported (realized) by certain components and then imported (used) by others. If you want to expose the seams in your system, model these interfaces explicitly. If you want your model at a higher level of abstraction, elide these relationships by showing only dependencies among the components.

For example, Figure models part of the executable release for an autonomous robot. This figure focuses on the deployment components associated with the robot's driving and calculation functions. You'll find one component (**driver.dll**) that exports an interface (**IDrive**) that is, in turn, imported by another component (**path.dll**). **driver.dll** exports one other interface (**ISelfTest**) that is probably used by other

components in the system, although they are not shown here. There's one other component shown in this diagram (**collision.dll**), and it, too, exports a set of interfaces, although these details are elided: **path.dll** is shown with a dependency directly to **collision.dll.**

Figure Modeling an Executable Release



There are many more components involved in this system. However, this diagram only focuses on those deployment components that are directly involved in moving the robot. Note that in this component-based architecture, you could replace a specific version of **driver.dll** with another that realized the same (and perhaps additional) interfaces, and **path.dll** would still function properly. If you want to be explicit about the operations that **driver.dll** realizes, you could always render its interface using class notation, stereotyped as **»interface .**

Modeling a Physical Database

A logical database schema captures the vocabulary of a system's persistent data, along with the semantics of their relationships. Physically, these things are stored in a database for later retrieval, either a relational database, an object-oriented one, or a hybrid object/relational database. The UML is well suited to modeling physical databases, as well as logical database schemas.

*Physical database design is beyond the scope of this book; the focus here is simply to show you how you can model databases and tables using the UML.*Mapping a logical database schema to an object-oriented

database is straightforward because even complex inheritance lattices can be made persistent directly. Mapping a logical database schema to a relational database is not so simple, however. In the presence of inheritance, you have to make decisions about how to map classes to tables. Typically, you can apply one or a combination of three strategies.

1. Define a separate table for each class. This is a simple but naive approach because it introduces maintenance headaches when you add new child classes or modify your parent classes.
2. Collapse your inheritance lattices so that all instances of any class in a hierarchy has the same state. The downside with this approach is that you end up storing superfluous information for many instances.
3. Separate parent and child states into different tables. This approach best mirrors your inheritance lattice, but the downside is that traversing your data will require many cross-table joins.

When designing a physical database, you also have to make decisions about how to map operations defined in your logical database schema. Object- oriented databases make the mapping fairly transparent. But, with relational databases, you have to make some decisions about how these logical operations are implemented. Again, you have some choices.

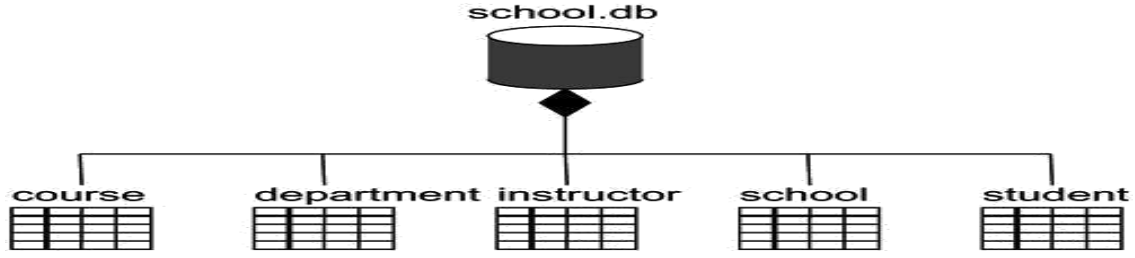
1. For simple CRUD (create, read, update, delete) operations, implement them with standard SQL or ODBC calls.
2. For more-complex behavior (such as business rules), map them to triggers or stored procedures.

Given these general guidelines, to model a physical database,

- Identify the classes in your model that represent your logical database schema.
- Select a strategy for mapping these classes to tables. You will also want to consider the physical distribution of your databases. Your mapping strategy will be affected by the location in which you want your data to live on your deployed system.
- To visualize, specify, construct, and document your mapping, create a component diagram that contains components stereotyped as tables.
- Where possible, use tools to help you transform your logical design into a physical design.

Figure shows a set of database tables drawn from an information system for a school. You will find one database (**school.db**, rendered as a component stereotyped as **database**) that's composed of five tables: **student**, **class**, **instructor**, **department**, and **course** (rendered as a component stereotyped as **table,** one of the UML's standard elements). In the corresponding logical database schema, there was no inheritance, so mapping to this physical database design is straightforward.

Figure Modeling a Physical Database



Although not shown in this example, you can specify the contents of each table. Components can have attributes, so a common idiom when modeling physical databases is to use these attributes to specify the columns of each table. Similarly, components can have operations, and these can be used to denote stored procedures.

Modeling Adaptable Systems

All the component diagrams shown thus far have been used to model static views. Their components spend their entire lives on one node. This is the most common situation you'll encounter, but especially in the domain of complex, distributed systems, you'll need to model dynamic views. For example, you might have a system that replicates its databases across several nodes, switching the one that is the primary database when a server goes down. Similarly, if you are modeling a globally distributed 24x7 operation (that is, a system that's up 24 hours a day, 7 days a week), you will likely encounter mobile agents, components that migrate from node to node to carry out some transaction. To model these dynamic views, you'll need to use a combination of component diagrams, object diagrams, and interaction diagrams.

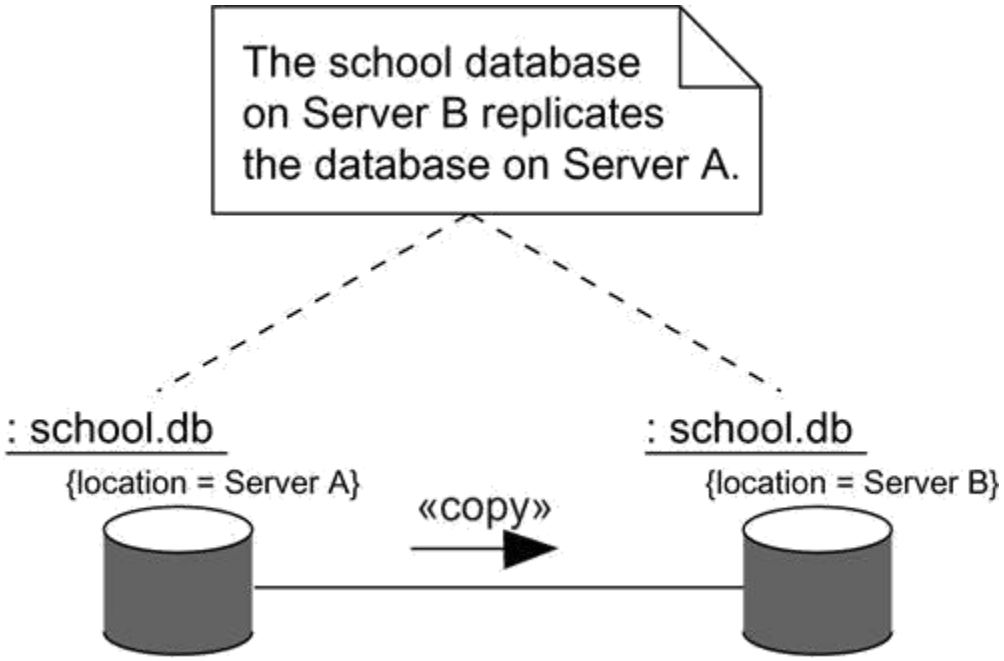
To model an adaptable system,

Consider the physical distribution of the components that may migrate from node to node. You can specify the location of a component instance by marking it with a location tagged value, which you can then render in a component diagram (although, technically speaking, a diagram that contains only instances is an object diagram).

- If you want to model the actions that cause a component to migrate, create a corresponding interaction diagram that contains component instances. You can illustrate a change of location by drawing the same instance more than once, but with different values for its location tagged value.

For example, Figure models the replication of the database from the previous figure. We show two instances of the component **school.db.** Both instances are anonymous, and both have a different value for their location tagged value. There's also a note, which explicitly specifies which instance replicates the other.

Figure Modeling Adaptable Systems



If you want to show the details of each database, you can render them in their canonical form• a component stereotyped as a **database.**Although not shown here, you could use an interaction diagram to model the dynamics of switching from one primary database to another.

Forward and Reverse Engineering

Forward engineering and reverse engineering components are pretty direct, because components are themselves physical things (executables, libraries, tables, files, and documents) that are therefore close to the running system. When you forward engineer a class or a collaboration, you really forward engineer to a component that represents the source code, binary library, or executable for that class or collaboration. Similarly, when you reverse engineer source code, binary libraries, or executables, you really reverse engineer to a component or set of components that, in turn, trace to classes or collaborations.

Choosing to forward engineer (the creation of code from a model) a class or collaboration to source code, a binary library, or an executable is a mapping decision you have to make. You'll want to take your logical models to source code if you are interested in controlling the configuration management of files that are then manipulated by a development environment. You'll want to take your logical models directly to binary libraries or executables if you are interested in managing the components that you'll actually deploy on a running system. In some cases, you'll want to do both. A class or collaboration may be denoted by source code, as well as by a binary library or executable.

To forward engineer a component diagram,

- For each component, identify the classes or collaborations that the component implements.
- Choose the target for each component. Your choice is basically between source code (a form that can be manipulated by development tools) or a binary library or executable (a form that can be dropped into a running system).
- Use tools to forward engineer your models.

Reverse engineering (the creation of a model from code) a component diagram is not a perfect process because there is always a loss of information. From source code, you can reverse engineer back to classes; this is the most common thing you'll do. Reverse engineering source code to components will uncover compilation dependencies among those files. For binary libraries, the best you can hope for is to denote the library as a component and then discover its interfaces by reverse engineering. This is the second most common thing you'll do with component diagrams. In fact, this is a useful way to approach a set of new libraries that may be otherwise poorly documented. For executables, the best you can hope for is to denote the executable as a component and then disassemble its code• something you'll rarely need to do unless you work in assembly language.

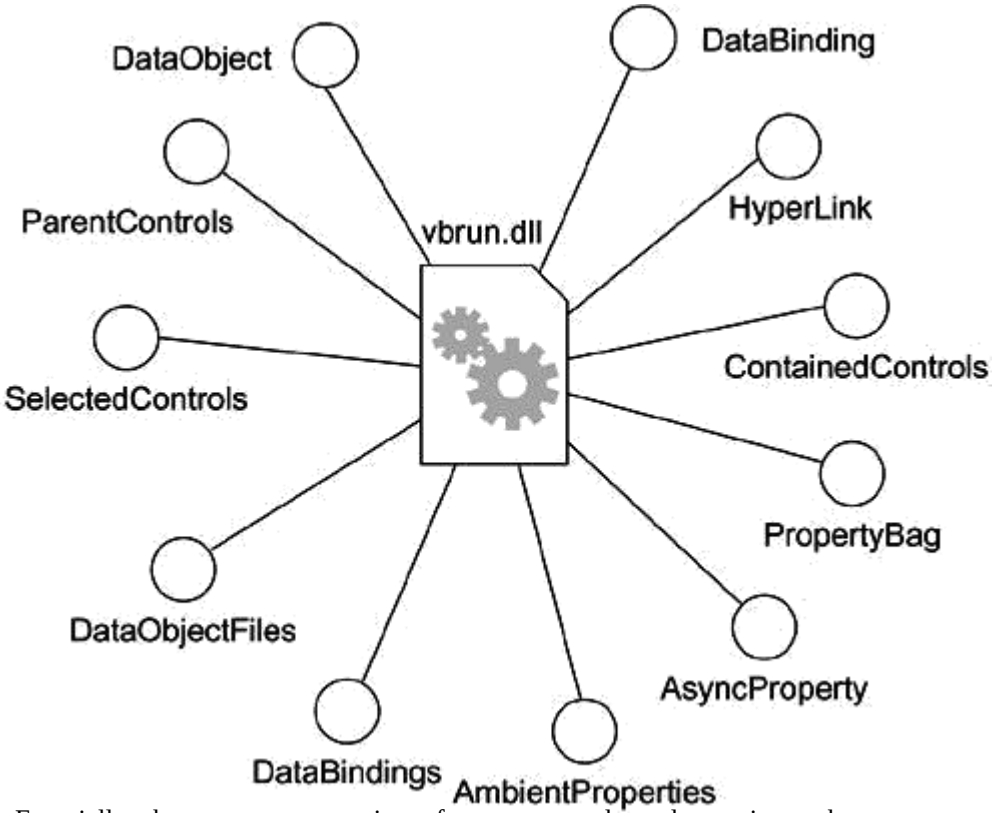
To reverse engineer a component diagram,

- Choose the target you want to reverse engineer. Source code can be reverse engineered to components and then classes. Binary libraries can be reverse engineered to uncover their interfaces. Executables can be reverse engineered the least.
- Using a tool, point to the code you'd like to reverse engineer. Use your tool to generate anew model or to modify an existing one that was previously forward engineered.
- Using your tool, create a component diagram by querying the model. For example, you might start with one or more components, then expand the diagram by following relationships or neighboring components. Expose or hide the details of the contents of this component diagram as necessary to communicate your intent.

For example, Figure provides a component diagram that represents the reverse engineering of the ActiveX component **vbrun.dll.** As the figure shows, the component realizes 11 interfaces. Given this diagram, you

can begin to understand the semantics of the component by next exploring the details of its interfaces.

Figure Reverse Engineering



Especially when you reverse engineer from source code, and sometimes when you reverse engineer from binary libraries and executables, you'll do so in the context of a configuration management system. This means that you'll often be working with specific versions of files or libraries, with all versions of a configuration compatible with one another. In these cases, you'll want to include a tagged value that represents the component version, which you can derive from your configuration management system. In this manner, you can use the UML to visualize the history of a component across various releases.

Deployment Diagrams

Terms and Concepts

A *deployment diagram* is a diagram that shows the configuration of run time processing nodes and the components that live on them. Graphically, a deployment diagram is a collection of vertices and arcs.

Common Properties

A deployment diagram is just a special kind of diagram and shares the same common properties as all other diagrams• a name and graphical contents that are a projection into a model. What distinguishes a deployment diagram from all other kinds of diagrams is its particular content.

Contents

Deployment diagrams commonly contain

- Nodes
- Dependency and association relationships

Like all other diagrams, deployment diagrams may contain notes and constraints.

Deployment diagrams may also contain components, each of which must live on some node. Deployment diagrams may also contain packages or subsystems, both of which are used to group elements of your model into larger chunks. Sometimes, you'll want to place instances in your deployment diagrams, as well, especially when you want to visualize one instance of a family of hardware topologies.

Note

In many ways, a deployment diagram is just a special kind of class diagram, which focuses on a system's nodes.

Common Uses

You use deployment diagrams to model the static deployment view of a system. This view primarily addresses the distribution, delivery, and installation of the parts that make up the physical system.

There are some kinds of systems for which deployment diagrams are unnecessary. If you are developing a piece of software that lives on one machine and interfaces only with standard devices on that machine that are already managed by the host operating system (for example, a personal computer's keyboard, display, and modem), you can ignore deployment diagrams. On the other hand, if you are developing a piece of software that interacts with devices that the host operating system does not typically manage or that is physically distributed across multiple processors, then using deployment diagrams will help you reason about your system's software-to-hardware mapping.

When you model the static deployment view of a system, you'll typically use deployment diagrams in one of three ways.

1. To model embedded systems

An embedded system is a software-intensive collection of hardware that interfaces with the physical world. Embedded systems involve software that controls devices such as motors, actuators, and displays and that, in turn, is controlled by external stimuli such as sensor input, movement, and temperature changes. You can use deployment diagrams to model the devices and processors that comprise an embedded system.

2. To model client/server systems

A client/server system is a common architecture focused on making a clear separation of concerns between the system's user interface (which lives on the client) and the system's persistent data (which lives on the server). Client/ server systems are one end of the continuum of distributed systems and require you to make decisions about the network connectivity of clients
to servers and about the physical distribution of your system's software components across the nodes.
You can model the topology of such systems by using deployment diagrams.

3. To model fully distributed systems

At the other end of the continuum of distributed systems are those that are widely, if not globally, distributed, typically encompassing multiple levels of servers. Such systems are often hosts to multiple versions of software components, some of which may even migrate from node to node. Crafting such systems requires you to make decisions that enable the continuous change in the system's topology. You can use deployment diagrams to visualize the system's current topology and distribution of components to reason about the impact of changes on that topology.

Common Modeling Techniques

Modeling an Embedded System

Developing an embedded system is far more than a software problem. You have to manage the physical world in which there are moving parts that break and in which signals are noisy and behavior is nonlinear. When you model such a system, you have to take into account its interface with the real world, and that means reasoning about unusual devices, as well as nodes.

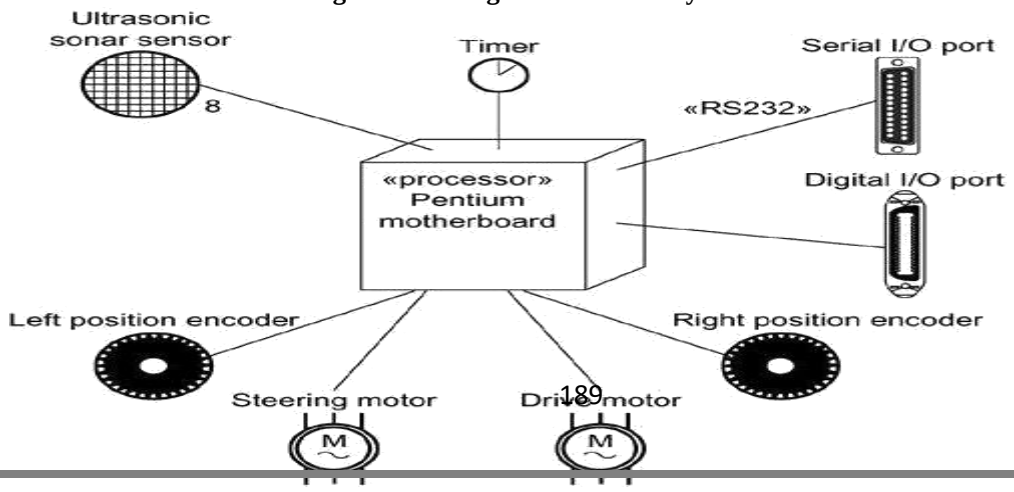
Deployment diagrams are useful in facilitating the communication between your project's hardware engineers and software developers. By using nodes that are stereotyped to look like familiar devices, you can create diagrams that are understandable by both groups. Deployment diagrams are also helpful in reasoning about hardware/software trade-offs. You'll use deployment diagrams to visualize, specify, construct, and document your system engineering decisions.

To model an embedded system,

- Identify the devices and nodes that are unique to your system.
- Provide visual cues, especially for unusual devices, by using the UML's extensibility mechanisms to define system-specific stereotypes with appropriate icons. At the very least, you'll want to distinguish processors (which contain software components) and devices (which, at that level of abstraction, don't directly contain software).
- Model the relationships among these processors and devices in a deployment diagram. Similarly, specify the relationship between the components in your system's implementation view and the nodes in your system's deployment view.
- As necessary, expand on any intelligent devices by modeling their structure with a more detailed deployment diagram.

For example, Figure shows the hardware for a simple autonomous robot. You'll find one node (**Pentium motherboard**) stereotyped as a processor.

Figure Modeling an Embedded System



Surrounding this node are eight devices, each stereotyped as a device and rendered with an icon that offers a clear visual cue to its real-world equivalent.

Modeling a Client/Server System

The moment you start developing a system whose software no longer resides on a single processor, you are faced with a host of decisions: How do you best distribute your software components across these nodes? How do they communicate? How do you deal with failure and noise? At one end of the spectrum of distributed systems, you'll encounter client/server systems, in which there's a clear separation of concerns between the system's user interface (typically managed by the client) and its data (typically managed by the server).

There are many variations on this theme. For example, you might choose to have a thin client, meaning that it has a limited amount of computational capacity and does little more than manage the user interface and visualization of information. Thin clients may not even host a lot of components but, rather, may be designed to load components from the server, as needed, as with Enterprise Java Beans. On the other hand, you might chose to have a thick client, meaning that it has a goodly amount of computational capacity and does more than just visualization. A thick client typically carries out some of the system's logic and business rules. The choice between thin and thick clients is an architectural decision that's influenced by a number of technical, economic, and political factors.

Either way, partitioning a system into its client and server parts involves making some hard decisions about where to physically place its software components and how to impose a balanced distribution of responsibilities among those components. For example, most management information systems are essentially three-tier architectures, which means that the system's GUI, business logic, and database are physically distributed. Deciding where to place the system's GUI and database are usually fairly obvious, so the hard part lies in deciding where the business logic lives.

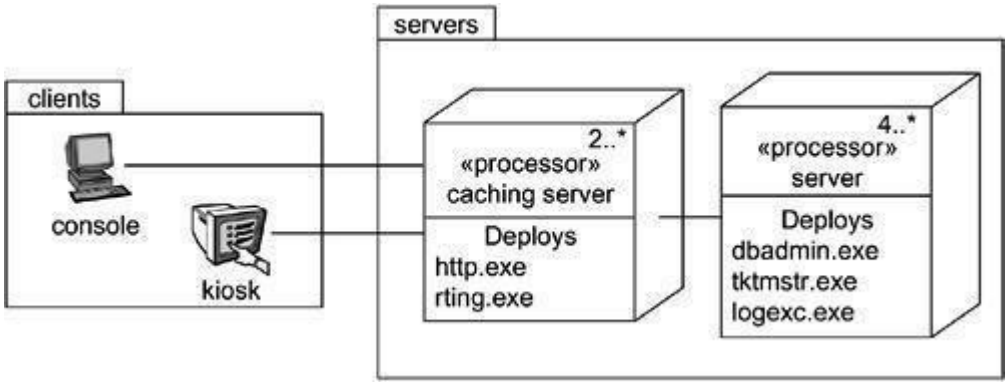
You can use the UML's deployment diagrams to visualize, specify, and document your decisions about the topology of your client/server system and how its software components are distributed across the client and server. Typically, you'll want to create one deployment diagram for the system as a whole, along with other, more detailed, diagrams that drill down to individual segments of the system.

To model a client/server system,

- Identify the nodes that represent your system's client and server processors.
- Highlight those devices that are germane to the behavior of your system. For example, you'll want to model special devices, such as credit card readers, badge readers, and display devices other than monitors, because their placement in the system's hardware topology are likely to be architecturally significant.
- Provide visual cues for these processors and devices via stereotyping.
- Model the topology of these nodes in a deployment diagram. Similarly, specify the relationship between the components in your system's implementation view and the nodes in your system's deployment view.

For example, Figure shows the topology of a human resources system, which follows a classical client/server architecture. This figure illustrates the client/server split explicitly by using the packages named **client** and **server.** The client package contains two nodes (**console** and **kiosk**), both of which are stereotyped and are visually distinguishable. The server package contains two kinds of nodes (**caching server** and **server**), and both of these have been adorned with some of the components that reside on each. Note also that **caching server** and **server** are marked with explicit multiplicities, specifying how many instances of each are expected in a particular deployed configuration. For example, this diagram indicates that there may be two or more **caching servers** in any deployed instance of the system.

Figure Modeling a Client/Server System



Modeling a Fully Distributed System

Distributed systems come in many forms, from simple two-processor systems to those that span many geographically dispersed nodes. The latter are typically never static. Nodes are added and removed as network traffic changes and processors fail; new and faster communication paths may be established in parallel with older, slower channels that are eventually decommissioned.

Not only may the topology of these systems change, but the distribution of their software components may change, as well. For example, database tables may be replicated across servers, only to be moved, as traffic dictates. For some global systems, components may follow the sun, migrating from server to server as the business day begins in one part of the world and ends in another.

Visualizing, specifying, and documenting the topology of fully distributed systems such as these are valuable activities for the systems administrator, who must keep tabs on an enterprise's computing assets. You can use the UML's deployment diagrams to reason about the topology of such systems. When you document fully distributed systems using deployment diagrams, you'll want to expand on the details of the system's networking devices, each of which you can represent as a stereotyped node.

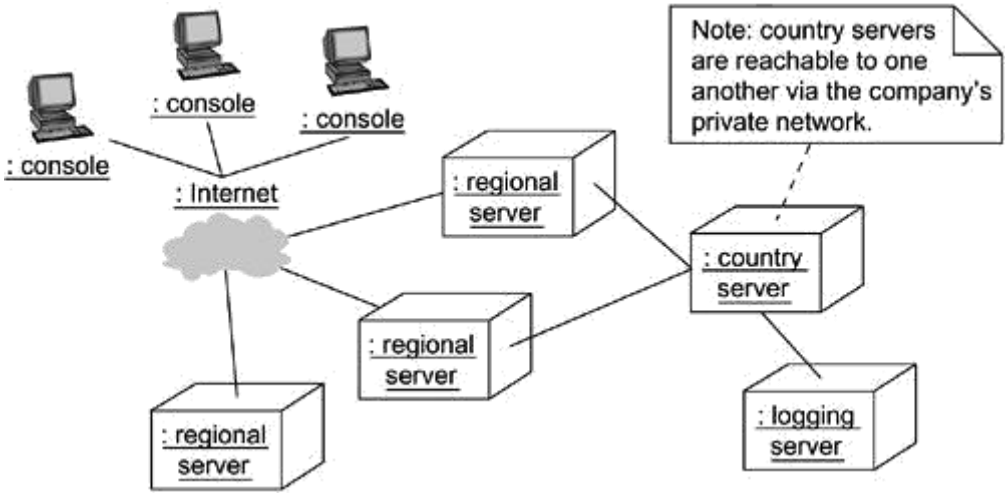
To model a fully distributed system,

- Identify the system's devices and processors as for simpler client/server systems.
- If you need to reason about the performance of the system's network or the impact of changes to the network, be sure to model these communication devices to the level of detail sufficient to make these assessments.
- Pay close attention to logical groupings of nodes, which you can specify byusing packages.
- Model these devices and processors using deployment diagrams. Where possible, use tools that discover the topology of your system by walking your system's network.
- If you need to focus on the dynamics of your system, introduce use case diagrams to specify the kinds of behavior you are interested in, and expand on these use cases with interaction diagrams.

Figure shows the topology of a fully distributed system. This particular deployment diagram is also an object diagram, for it contains only instances. You can see three consoles (anonymous instances of the stereotyped node **console**), which are linked to the **Internet** (clearly a singleton node). In turn, there are three instances of **regional servers,** which serve as front ends of **country servers,** only one of which is shown. As the note indicates, country servers are connected to one another, but their relationships are not

shown in this diagram.

Figure Modeling a Fully Distributed System



In this diagram, the Internet has been reified as a stereotyped node.

Forward and Reverse Engineering

There's only a modest amount of forward engineering (the creation of code from models) that you can do with deployment diagrams. For example, after specifying the physical distribution of components across the nodes in a deployment diagram, it is possible to use tools that then push these components out to the real world. For system administrators, using the UML in this way helps you visualize what can be a very complicated task.

Reverse engineering (the creation of models from code) from the real world back to deployment diagrams is of tremendous value, especially for fully distributed systems that are under constant change. You'll want to supply a set of stereotyped nodes that speak the language of your system's network administrators, in order to tailor the UML to their domain. The advantage of using the UML is that it offers a standard language that addresses not only their needs, but the needs of your project's software developers, as well.

To reverse engineer a deployment diagram,

- Choose the target that you want to reverse engineer. In some cases, you'll want to sweep across your entire network; in others, you can limit your search.
- Choose also the fidelity of your reverse engineering. In some cases, it's sufficient to reverse engineer just to the level of all the system's processors; in others, you'll want to reverse engineer the system's networking peripherals, as well.
- Use a tool that walks across your system, discovering its hardware topology. Record that topology in a deployment model.
- Along the way, you can use similar tools to discover the components that live on each node, which you can also record in a deployment model. You'll want to use an intelligent search, for even a basic personal computer can contain gigabytes of components, many of which may not be relevant to your system.
- Using your modeling tools, create a deployment diagram by querying the model. For example,

you might start with visualizing the basic client/server topology, then expand on the diagram by populating certain nodes with components of interest that live on them. Expose or hide the details of the contents of this deployment diagram as necessary to communicate your intent.

UNIT-5

The unified library Application

INTRODUCTION:

USECASE DIAGRAM:
A behavioral diagram that shows a set of use cases and actors and their relation ships.

USECASE:
It is a description of set of sequence of actions that a system performs that yields an observable result of value to a particular actor.

An use case is used to structure the behavioral things in a model and it is rendered as an ellipse with solid line along with its name.

Actor:

It specifies a coherent set of roles that users of use cases play when interacting with these use cases.

CLASS DIAGRAM:

A structural diagram that shows a set of classes, interfaces, collaborations, and their relationships.

OBJECT DIAGRAM:

A structural diagram that shows a set of objects and their relationships.

SEQUENCE DIAGRAM:

A behavioral diagram that shows an interaction, emphasizing the time ordering of messages.

COLLABORATION DIAGRAM:

A behavioural diagram that shows an interaction, emphasizing the structural organization of the objects that send and receive meassages.

STATECHART DIAGRAM:

A behavioural diagram that shows a state machine, emphasizing the event-ordered behavior of an object.

ACTIVITY DIAGRAM :

An activity diagram represents the execution state of a mechanism as a sequence of steps grouped sequentially as parallel conrol flow branches.It is a variant of state chart diagrams organized according to actions and internal behavior of a method or a usecase.

Activity diagram s are used to model the dynamic aspects of system.

- Swimlanes

- Forking
- Joining

COMPONENT DIAGRAM:

Component diagrams are basically used to model static view of the system. This can be achieved by modeling various physical components like libraries, tables, files etc. which are residing within a node.

Component diagrams are very essential for constructing executable systems. This can be done using concepts of forward and reverse engineering. The graphical representation of a component diagrams basically include collection of vertices and arcs.

DEPLOYMENT DIAGRAM:

Deployment is the stage of development that describes the configuration of the running system in a real time environment.For deployment,decisions should be made about configuration parameters,performece,resource allocation ,distribution and concurency.The component develop or reused should be deployed on some set o f hardware for execution.Nodes are used to model the topology of the hardware on which the system executes.A node usually represents a processor or a device on which components can be deployed .

Case Study 1: Library Application

1. Problem Statement:

A lightweight set of features for the first version of the library application might look like this:

> It is a support system for a library. The library lends books and magazines to borrowers, who are registered in the system, as are the books and magazines. The library handles the purchase of new titles for the library. Popular titles are bought in multiple copies. Old books and magazines are removed when they are out of date or in poor condition. The librarian is an employee of the library who interacts with the customers (borrowers) and whose work is supported by the system. A borrower can reserve a book or magazine that is not currently available in the library, so that when it's returned or purchased by the library, that borrower is notified. The reservation is cancelled when the borrower checks out the book or magazine or through an explicit canceling procedure. The librarian can easily create, update, and delete information about the titles, borrowers, loans, and reservations in the system.The system can run on all popular Web browser platforms (Internet Explorer 5.1+, Netscape 4.0+, and so on).The system is easy to extend with new functionality.

2. Identification of actors and use cases:

The use cases in the library system are as follows:

- Login
- Search
- Browse
- Make Reservation

- Remove Reservation
- Checkout Item
- Return Item
- Manage Titles
- Manage Items
- Manage Borrowers
- Manage Librarians
- Assume Identity of Borrower

The outline of the basic flow for the use case Checkout Item (which means that a Borrower can check out an Item) is described as follows:

1. The borrower chooses to perform a “Search” for desired titles/
2. The system prompts the borrower to enter Search criteria.
3. The borrower specifies the search criteria and submits the search.
4. The system locates matching titles and displays them to the borrower.
5. The borrower selects a title to check out.
6. The system displays the details of the title, as well as whether or not there is an available item to be checked out.
7. The borrower confirms that he or she wishes to checkout the item.
8. The system checks out the item.
9. Steps 1 to 8 can be repeated as often as desired by the borrower.
10. The borrower completes checkout.
11. The system notifies a librarian that the borrower has concluded the checkout item session and displays instructions for the borrower to collect the contents.

3: Identification of actors and use cases:

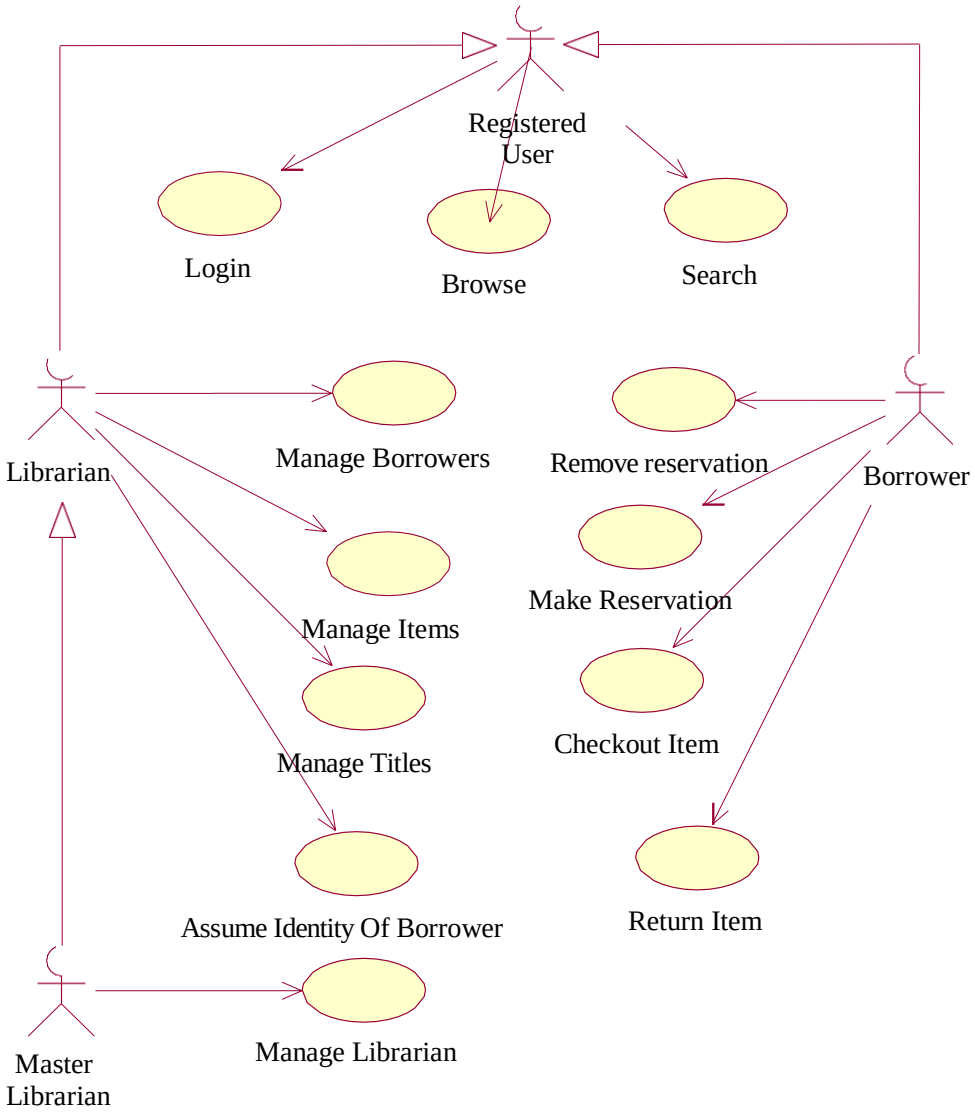
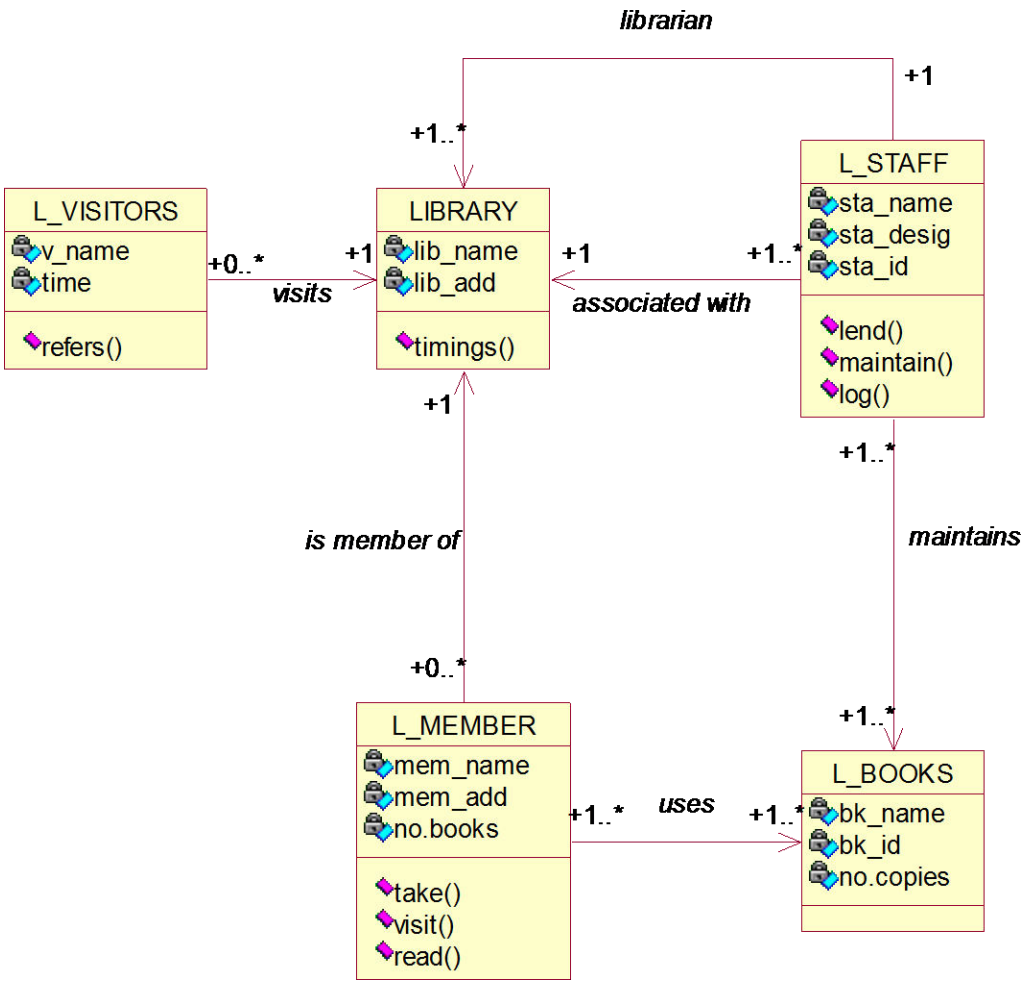
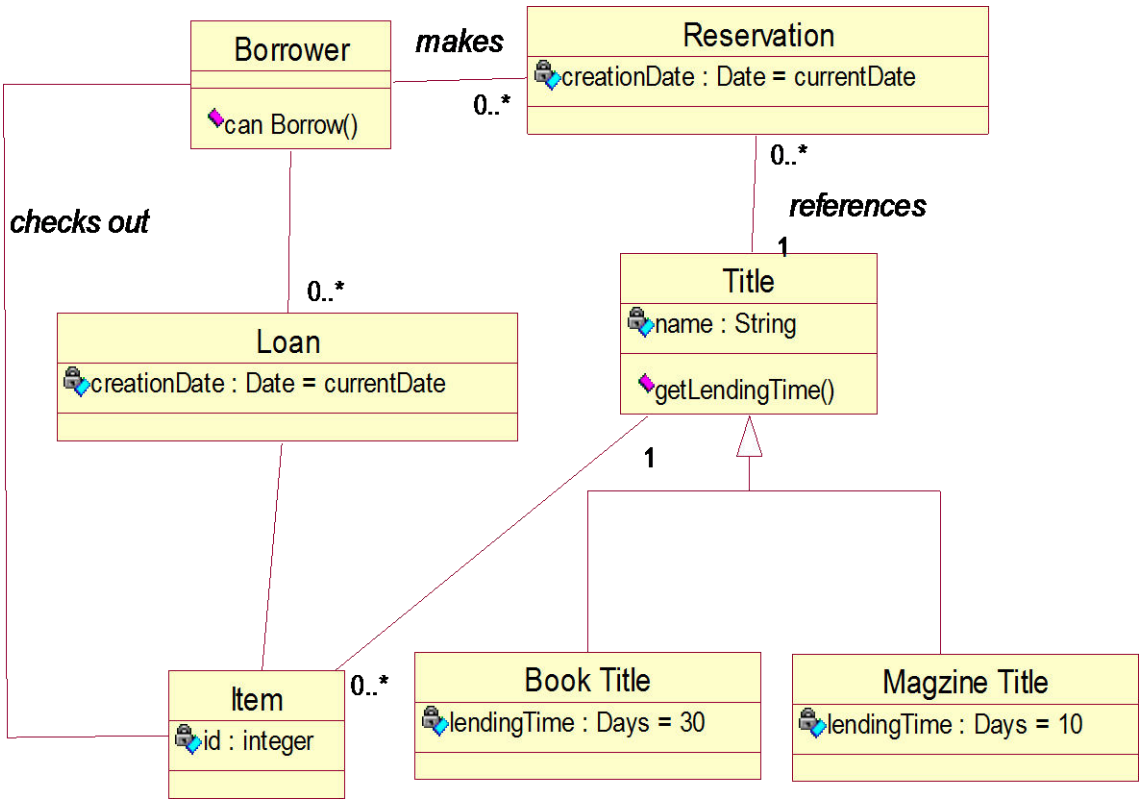


Fig: A use-case diagram for library system

Class diagram for library system:

Fig: classes for the library system





classes for the library system.

Sequence diagram for use case Return Item:

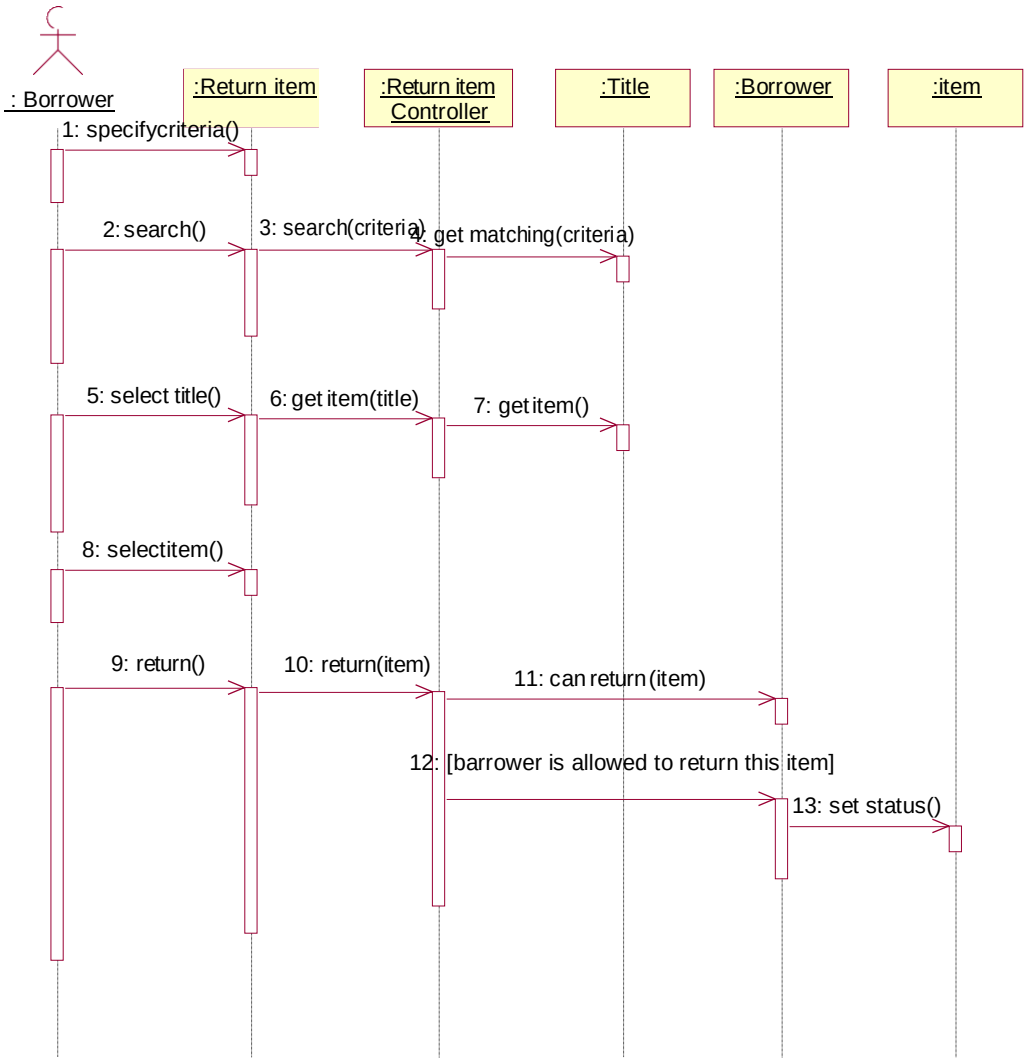
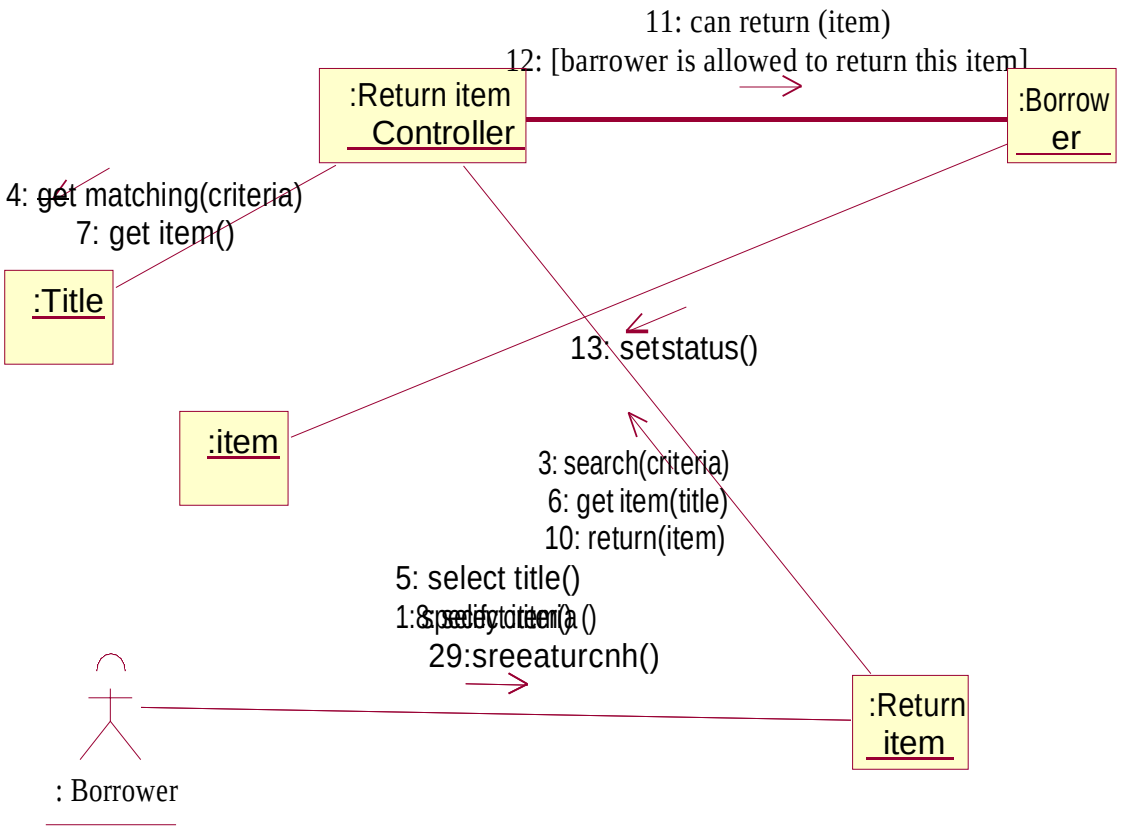
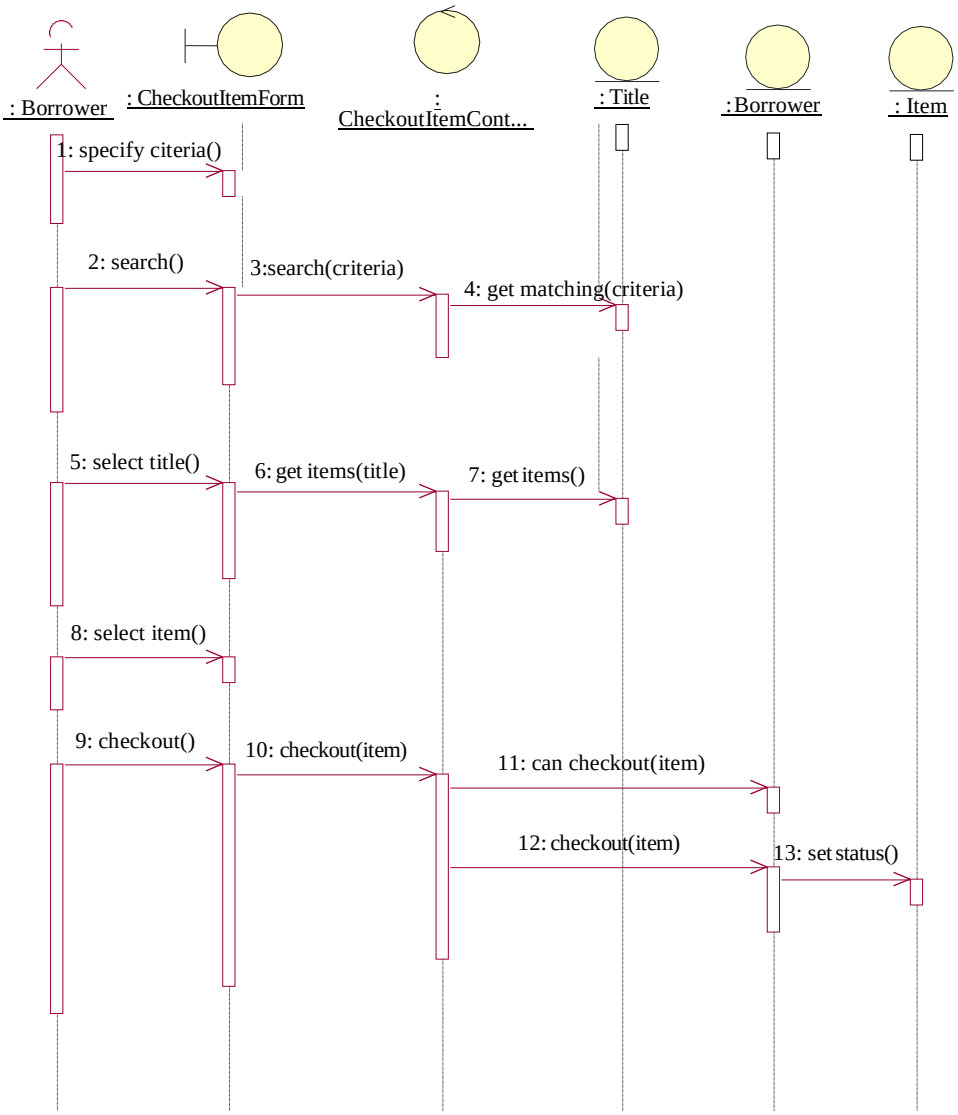


Fig: collaboration diagram for use case Return Item



Sequence diagram for use cases:

Checkout Item:


: Borrower
: CheckoutItemForm
: CheckoutItemCont...
: Title
: Borrower
: Item
1: specify citeria()
2: search()
3:search(criteria)
4: get matching(criteria)
5: select title()
6: get items(title)
7: get items()
8: select item()
9: checkout()
10: checkout(item)
11: can checkout(item)
12: checkout(item)
13: set status()

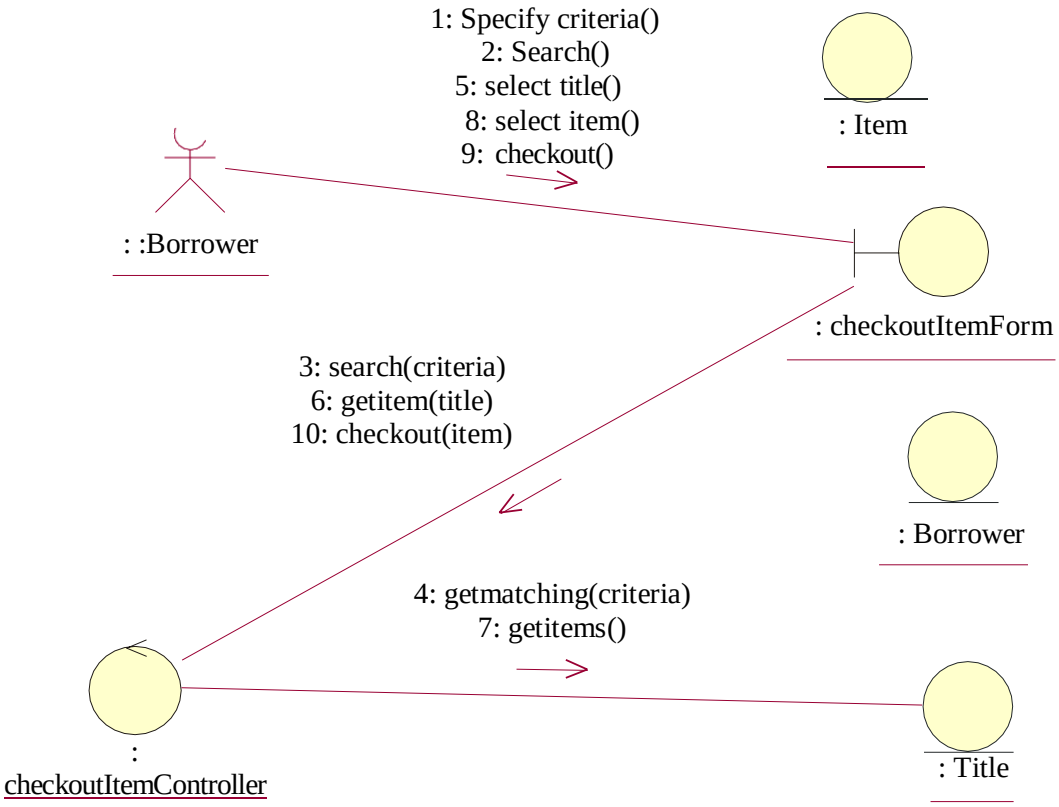
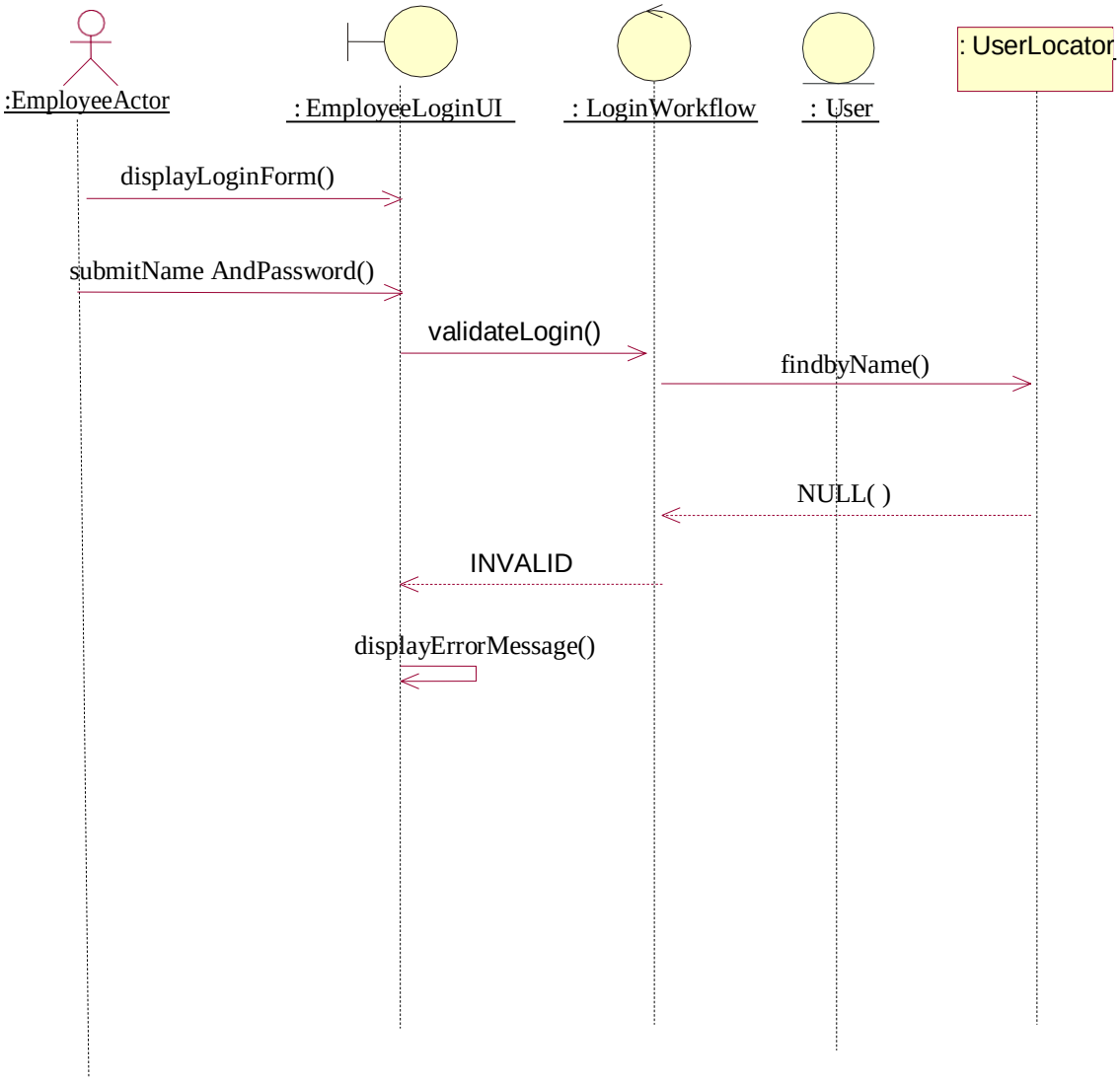


Fig: collaboration diagram for Checkout Item

Sequence diagram for use case login:



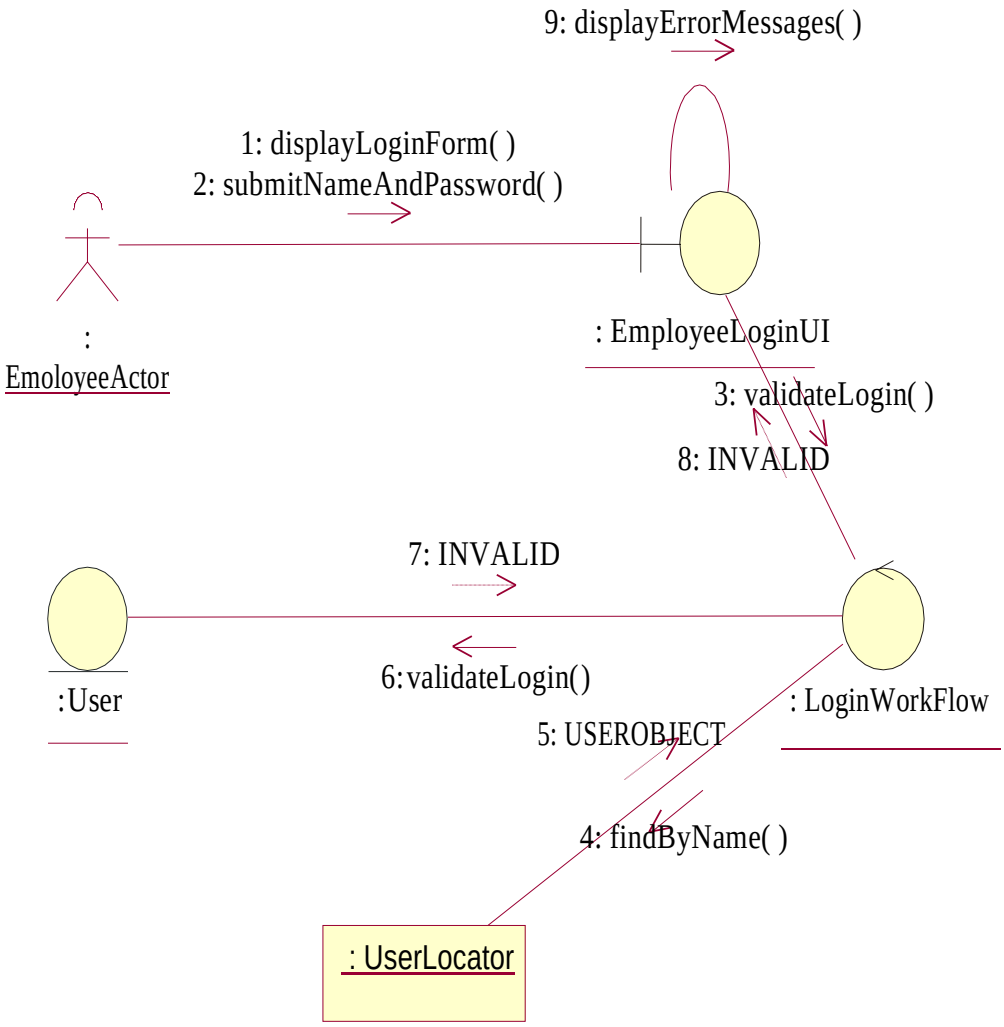
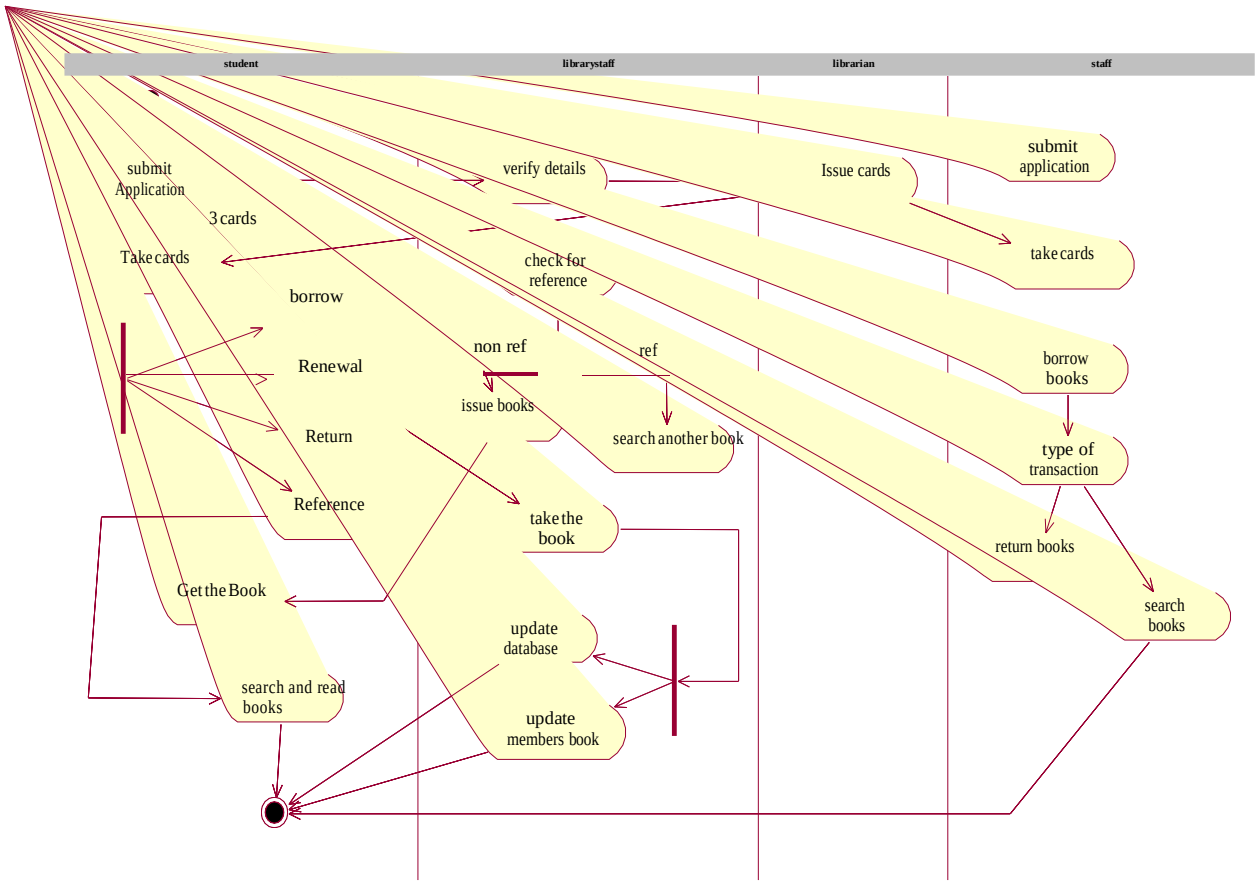
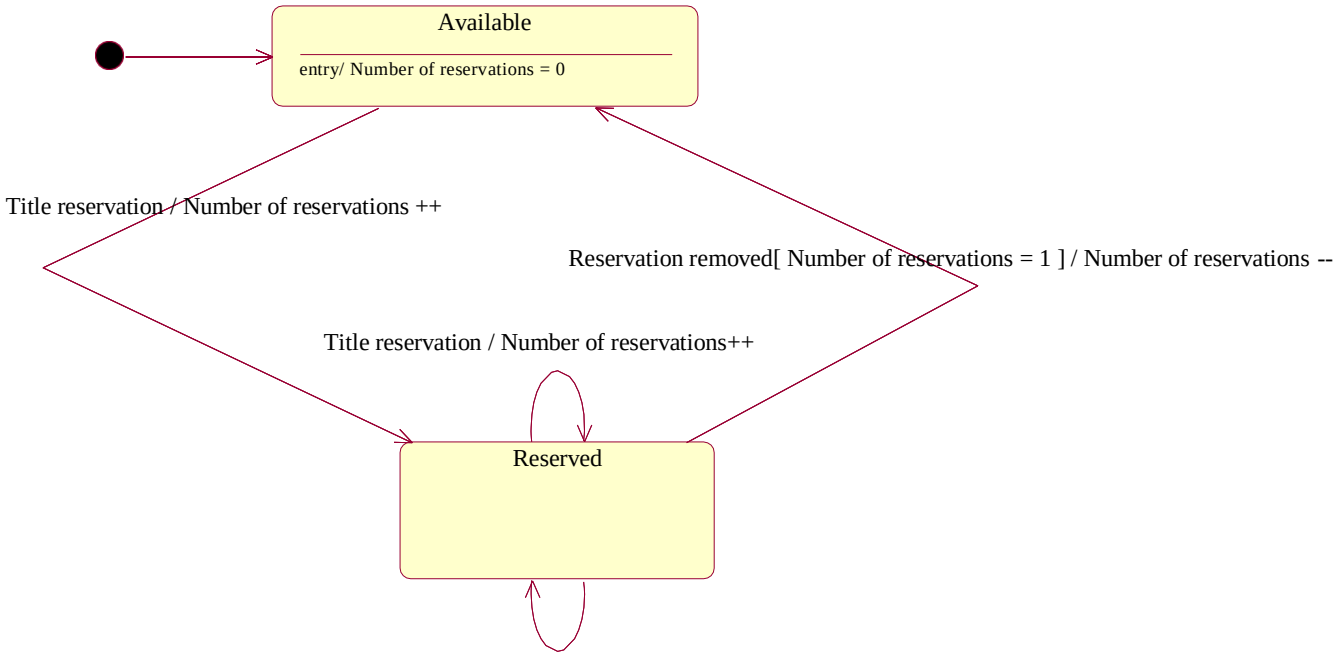


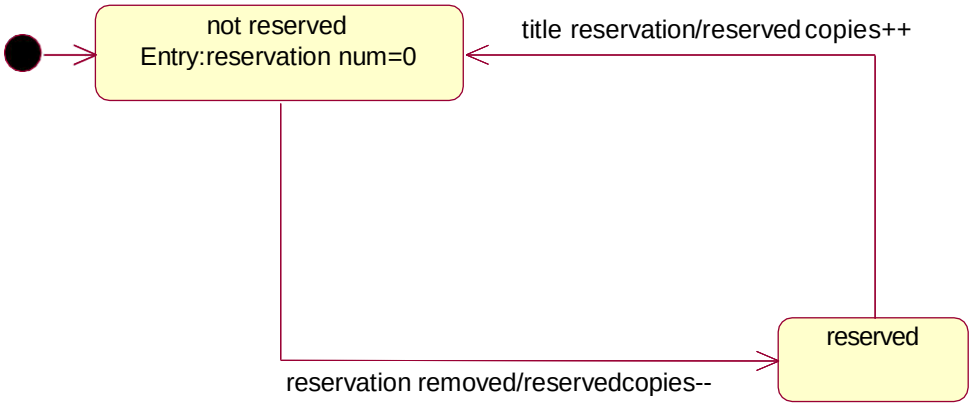
Fig: collaboration diagram for login

Activity diagram for library application:



STATE MACHINE DIAGRAM FOR THE TITLE CLASS:





STATE DIAGRAM FOR LIBRARY SYSTEM

Component diagram :

Component diagram for library application:

Figure1:

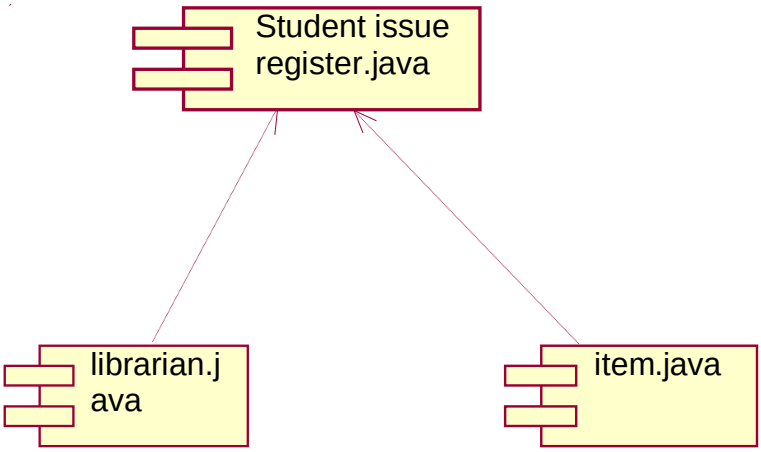
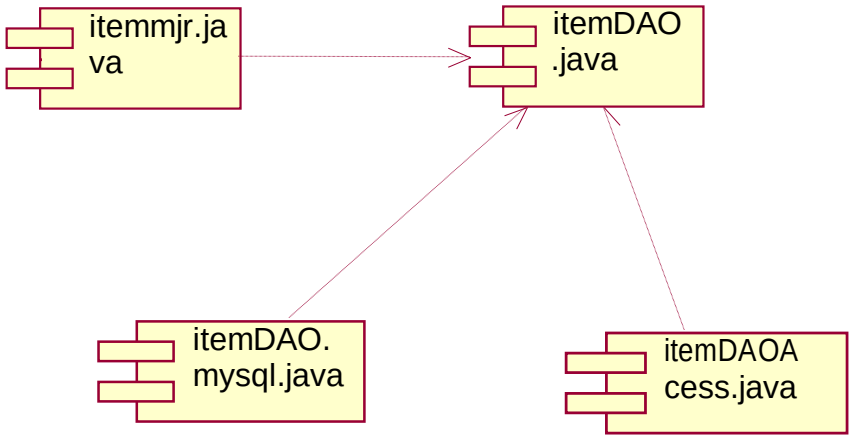


Figure2:



Deployment diagram for library applications

Deployment usually refers to transfering the project to the required end users along with the project documentation.Deployment diagrams are also essential in each application since it narrates packaged scenario of interaction following is a deployment diagram referring to unified library application scenario

Figure 1:

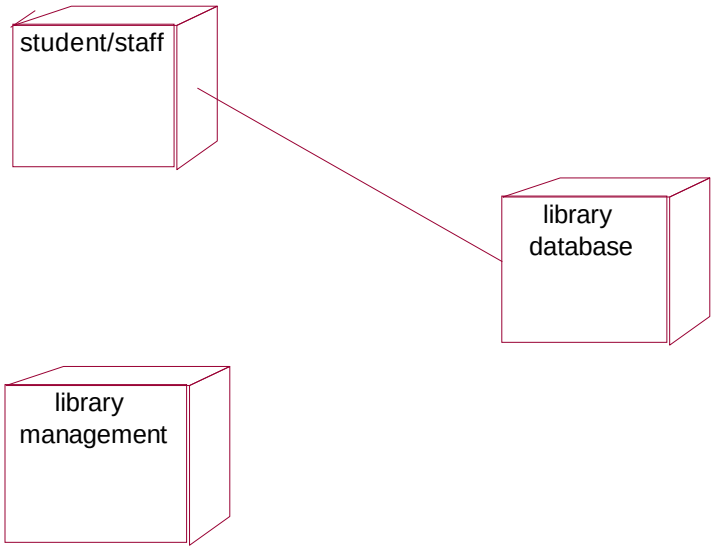
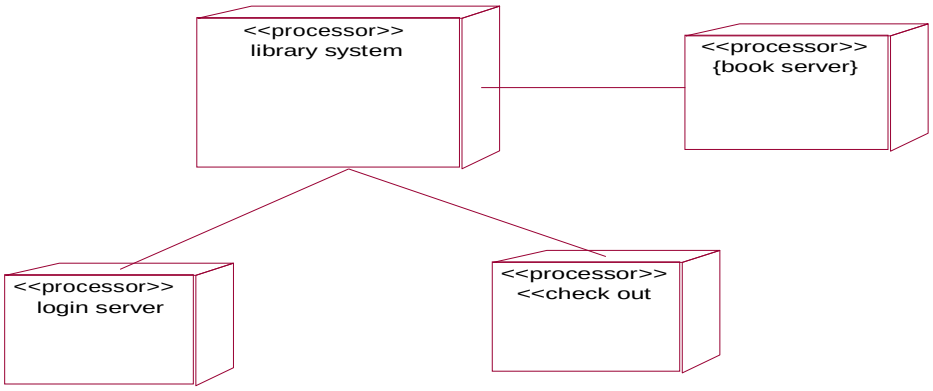

student/staff
library
database
library
management

Figure 2:


<<processor>>
library system
<<processor>>
{book server}
<<processor>>
login server
<<processor>>
<<check out