1. What is UML? Explain the Importance of Modeling?
Answer:

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

In such cases, the programmer is still doing some modeling, albeit entirely mentally. He or she may even sketch out a few ideas on a white board or on a napkin. However, there are several problems with this.

First, communicating those conceptual models to others is error-prone unless everyone involved speaks the same language. Typically, projects and organizations develop their own language, and it is difficult to understand what's going on if you are an outsider or new to the group.

Second, there are some things about a software system you can't understand unless you build models that transcend the textual programming language. For example, the meaning of a class hierarchy can be inferred, but not directly grasped, by staring at the code for all the classes in the hierarchy. Similarly, the physical distribution and possible migration of the objects in a Web based system can be inferred, but not directly grasped, by studying the system's code. Third, if the developer who cut the code never wrote down the models that are in his or her head, that information would be lost forever or, at best, only partially recreatable from the implementation, once that developer moved on.

Third issue: An explicit model facilitates communication. Some things are best modeled textually; others are best modeled graphically. Indeed, in all interesting systems, there are structures that transcend what can be represented in a programming language. The UML is such a graphical language. This addresses the second problem described earlier.

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

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

The UML Is a Language for Documenting
A healthy software organization produces all sorts of artifacts in addition to raw executable code.
These artifacts include (but are not limited to)

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

- Releases

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

The Importance of Modeling

If you really want to build the software equivalent of a house or a high rise, the problem is more than just a matter of writing lots of software• in fact, the trick is in creating the right software and in figuring out how to write less software. This makes quality software development an issue of architecture and process and tools. Even so, many projects start out looking like dog houses but grow to the magnitude of a high rise simply because they are a victim of their own success. There comes a time when, if there was no consideration given to architecture, process, or tools, that the dog house, now grown into a high rise, collapses of its own weight. The collapse of a dog house may annoy your dog; the failure of a high rise will materially affect its tenants.

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.

2. **What is modeling? State and explain the Principles of Modeling?**

Answer:

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 software, the models you choose can greatly affect your world view. If you build a system through the eyes of a database developer, you will likely focus on entity-relationship models that push behavior into triggers and stored procedures. If you build a system through the eyes of a structured analyst, you will likely end up with models that are algorithmic-centric, with data flowing from process to process. If you build a system through the eyes of an object-oriented developer, you'll end up with a system whose architecture is centered around a sea of classes and the patterns of interaction that direct how those classes work together. Any of these approaches might be right for a given application and development culture, although experience suggests that the object-oriented view is superior in crafting resilient architectures, even for systems that might have a large database or computational element. That fact notwithstanding, the point is that each world view leads to a different kind of system, with different costs and benefits

Second,
Every model may be expressed at different levels of precision.

The same is true with software models. Sometimes, a quick and simple executable model of the user interface is exactly what you need; at other times, you have to get down and dirty with the bits, such as when you are specifying cross-system interfaces or wrestling with networking bottlenecks. In any case, the best kinds of models are those that let you choose your degree of detail, depending on who is doing the viewing and why they need to view it. An analyst or an end user will want to focus on issues of what; a developer will want to focus on issues of how. Both of these stakeholders will want to visualize a system at different levels of detail at different times.

Third,
The best models are connected to reality.

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.

The operative phrase here is "nearly independent." In this context, it means having models that can be built and studied separately but that are still interrelated. As in the case of a building, you can study electrical plans in isolation, but you can also see their mapping to the floor plan and perhaps even their interaction with the routing of pipes in the plumbing plan.

3. **What are the categories of building blocks in the UML?**

Answer:

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.

1. **Things in the UML**

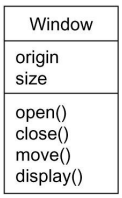
There are four kinds of things in the UML:

1. Structural things
2. Behavioral things
3. Grouping things
4. An notational things

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

1. **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,



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.

Interfaces

Third, **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, collaboration is rendered as an ellipse with dashed lines, usually including only its name.

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 collaboration. Graphically, a use case is rendered as an ellipse with solid lines, usually including only its name.

Use Cases

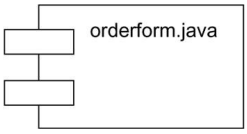
The remaining three things• active classes, components, and nodes• are all class-like, meaning they also describe a set of objects that share the same attributes, operations, relationships, and semantics. However, these three are different enough and are necessary for modeling certain aspects of an object-oriented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

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.

Active Classes

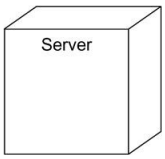
The remaining two elements• component, and nodes• are also different. They represent physical things, whereas the previous five things represent conceptual or logical things.

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.



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.



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).

2. **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 and links. Graphically, a message is rendered as a directed line, almost always including the name of its operation.

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.

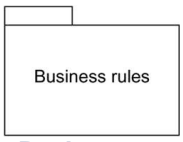
A state machine involves a number of other elements, including states, transitions (the flow from state to state), events (things that trigger a transition), and activities (the response to a transition). Graphically, a state is rendered as a rounded rectangle, usually including its name and its substates.

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.

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

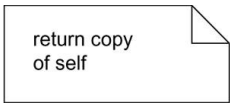
A *package* is a general-purpose mechanism for organizing elements into groups. Structural things, behavioral things, and even other grouping things may be placed in a package. Unlike components (which exist at run time), a package is purely conceptual (meaning that it exists only at development time).
Graphically, a package is rendered as a tabbed folder, usually including only its name and, sometimes, its contents.



Packages

4. **Annotational Things:** ***Annotational things*** are the explanatory parts of UML models. These are the comments you may apply to describe, illuminate, and remark about any element in a model. There is one primary kind of annotational thing, called a note.

A *note* is simply a symbol for rendering constraints and comments attached to an element or a collection of elements. Graphically, a note is rendered as a rectangle with a dog-eared corner, together with a textual or graphical comment.



Notes

2. Relationships in the UML

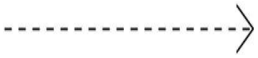
There are four kinds of relationships in the UML:

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).

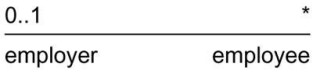
Graphically, a dependency is rendered as a dashed line, possibly directed, and occasionally including a label.



Dependency

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.

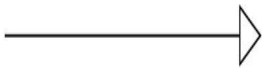
Graphically, an association is rendered as a solid line, possibly directed, occasionally including a label, and often containing other adornments, such as multiplicity and role names.



Association

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.



Generalization

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.

Realization

3. 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).

You draw diagrams to visualize a system from different perspectives, so a diagram is a projection into a system.

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.

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 the objects that send and receive messages. Sequence diagrams and collaboration diagrams are isomorphic, meaning that you can take one and transform it into the other.

Both **sequence diagrams and collaboration** diagrams are kinds of interaction diagrams. 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 ***state chart 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 state chart 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 architecture. They are related to component diagrams in that a node typically encloses one or more components.

4. Explain Common Mechanisms in the UML?

Answer:

The four common mechanisms that apply consistently throughout the language.

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

Specifications

The UML is more than just a graphical language. Rather, behind every part of its graphical notation there is a specification that provides a textual statement of the syntax and semantics of that building block.

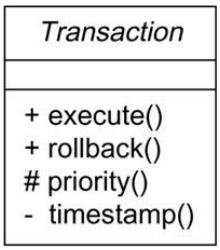
For example, behind a class icon is a specification that provides the full set of attributes, operations (including their full signatures), and behaviors that the class embodies; visually, that class icon might only show a small part of this specification.

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.

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



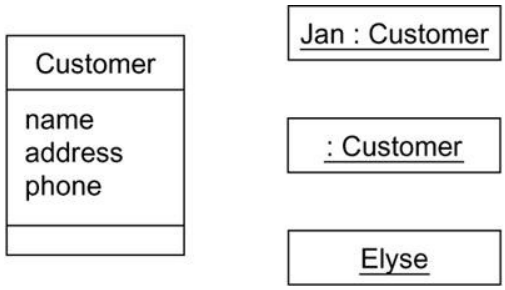
Adornments

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

Common Divisions

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

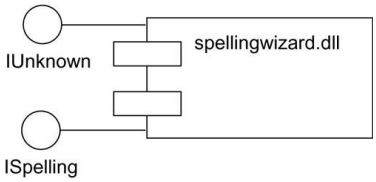
First, there is the division of class and object. A class is an abstraction; an object is one concrete manifestation of that abstraction. In the UML, you can model classes as well as objects, as shown below.



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,



Interfaces and Implementations

In this figure, there is one component named `spellingwizard.dll` that implements two interfaces, `I Unknown` and `I Spelling`.

Extensibility Mechanisms

The UML's extensibility mechanisms include

- Stereotypes
- Tagged values
- Constraints

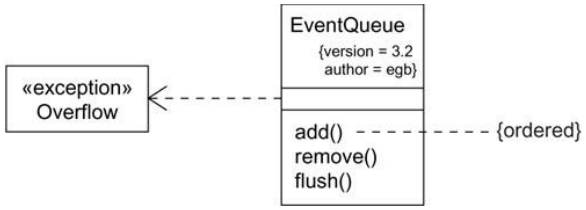
A ***stereotype*** **extends** the vocabulary of the UML, allowing you to create new kinds of building blocks that are derived from existing ones but that are specific to your problem.

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`.

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 below Figure for example, the class `Event Queue` is extended by marking its version and author explicitly.

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



Extensibility Mechanisms

5. **What is the need of the Architecture? Explain UML Architecture?**

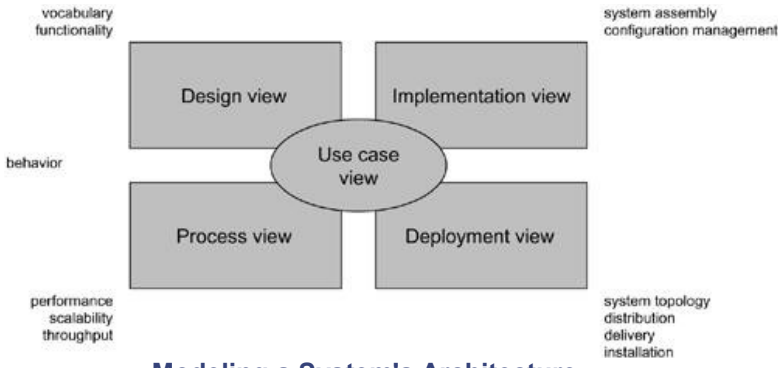
Answer:

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.

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.



Modeling a System's Architecture

The ***use case view*** of a system encompasses the use cases that describe the behavior of the system as seen by its end users, analysts, and testers. This view doesn't really specify the organization of a software system. Rather, it exists to specify the forces that shape the system's architecture. With the UML,

The static aspects of this view are captured in use case diagrams; the dynamic aspects of this view are captured in interaction diagrams, state chart diagrams, and activity diagrams.

The ***design view*** of a system encompasses the classes, interfaces, and collaborations that form the vocabulary of the problem and its solution. This view primarily supports the functional requirements of the system, meaning the services that the system should provide to its end users. With the UML,

The static aspects of this view are captured in class diagrams and object diagrams; the dynamic aspects of this view are captured in interaction diagrams, state chart 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, state chart diagrams, and activity diagrams.

The ***deployment view*** of a system encompasses the nodes that form the system's hardware topology on which the system executes. This view primarily addresses the distribution, delivery, and installation of the parts that make up the physical system. With the UML,

The static aspects of this view are captured in deployment diagrams; the dynamic aspects of this view are captured in interaction diagrams, state chart diagrams, and activity diagrams.

6. **Explain the various phases and workflows of an unified process for software development?**

Answer:

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

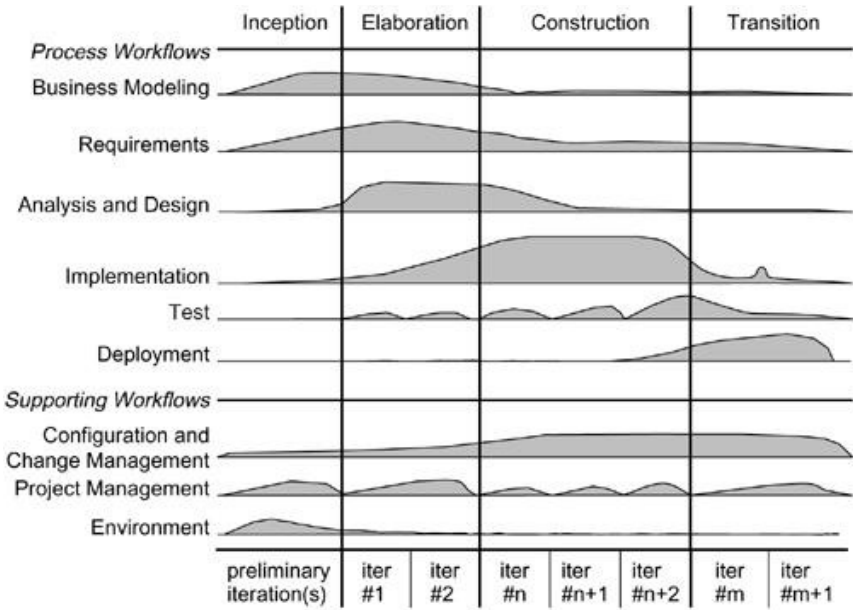
- Use case driven
- Architecture-centric
- Iterative and incremental

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

Architecture-centric means that a system's architecture is used as a primary artifact for conceptualizing, 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 below 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.



Software Development Life Cycle

Inception is the first phase of the process, when the seed idea for the development is brought up to 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 are defined. In this phase, the system's requirements are articulated, prioritized, and baseline. 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 executable architectural 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 the user 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 iteration. Iteration is a distinct set of activities, with a base lined 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.

1. **What are interactions? Discuss the terms and concepts of interactions in detail?**

Answer:

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 the interactions 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`), an object 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

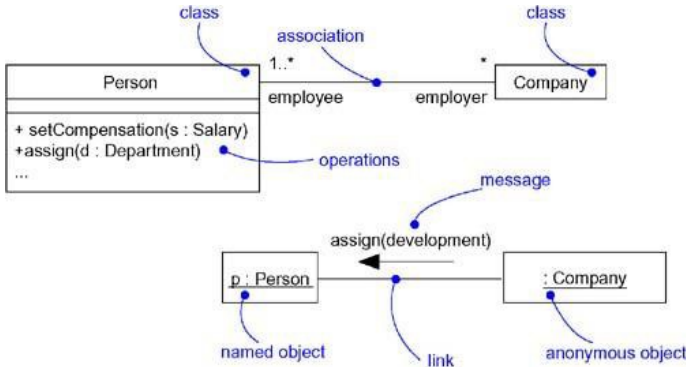
The objects that participate in an interaction are either concrete things or prototypical things. As a concrete thing, an object represents something in the real world. For example, `p`, an instance of the class `Person`, might denote a particular human. Alternately, as a prototypical thing, `p` might 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.



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

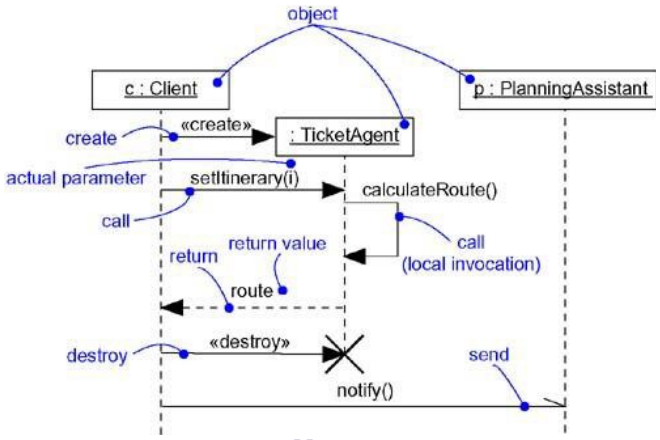
Messages

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.

- **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



Messages

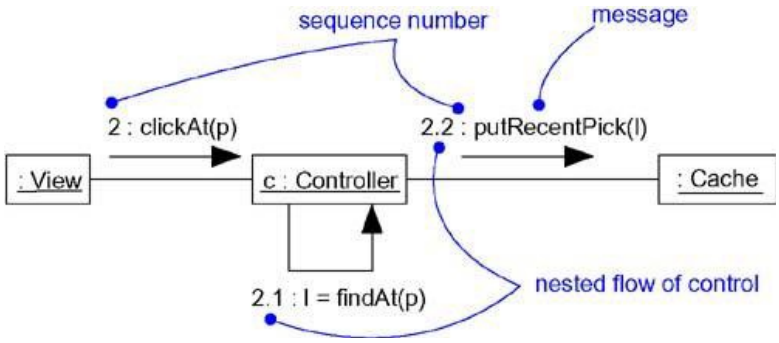
The most common kind of message you'll model is the call, in which one object invokes an operation of another (or the same) object. An object can't just call any random operation. If an object, such as `c` in the example above, calls the operation `setItinerary` on an instance of the class `TicketAgent`, 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), it must 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

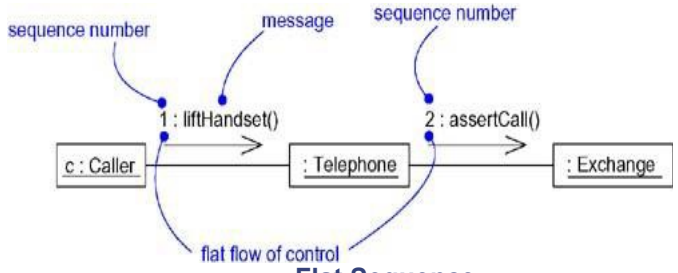
When an object passes a message to another object (in effect, delegating some action to the receiver), the receiving object might in turn send a message to another object, which might send a message to yet a different object, and so on. This stream of messages forms a sequence. Any sequence must have 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 is up.

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.



Procedural Sequence

Less common but also possible, as Figure 15-5 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.



Flat Sequence

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.

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.

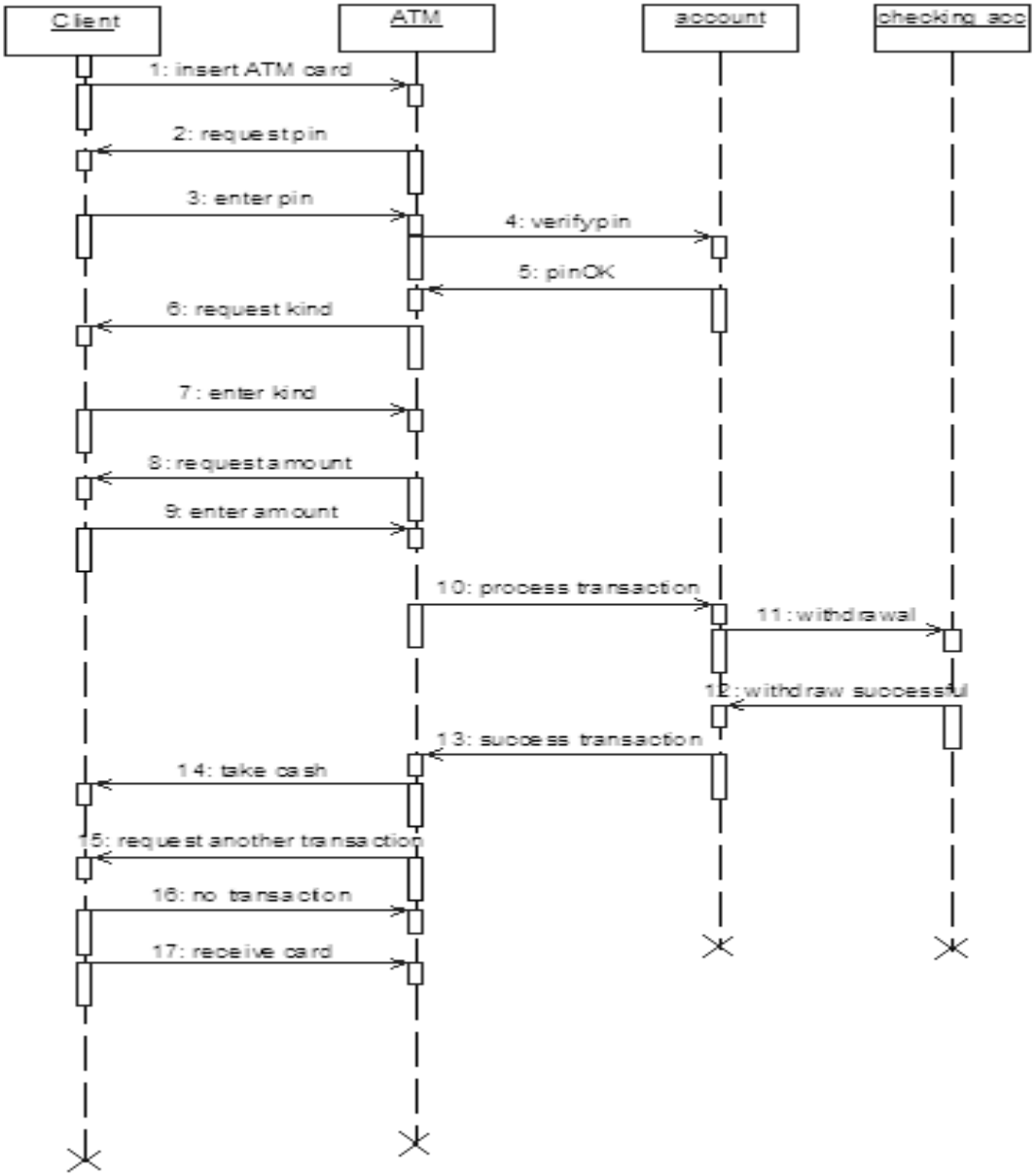
In the UML, the first kind of representation is called a sequence diagram; the second kind of representation is called a collaboration diagram. Both sequence diagrams and collaboration diagrams are kinds of interaction diagrams.

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.

2. **Draw a sequence diagram for customer interaction with bank ATM systems?**

Answer:



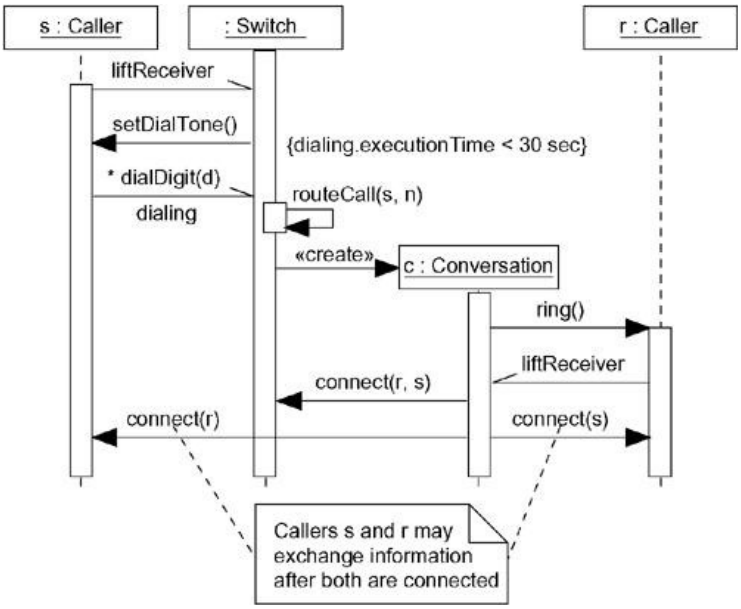
3. **What are the common modeling techniques for interaction diagrams Explain with examples?**

Answer:

Modeling Flows of Control by Time Ordering

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 post conditions to each message.



Modeling Flows of Control by Time Ordering

For example, Figure, shows a sequence diagram that specifies the flow of control involved in 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`) dispatching a 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 a timing 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.

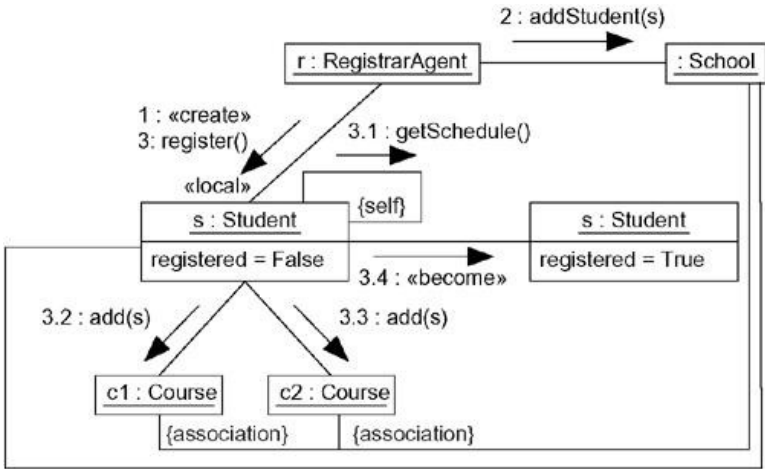
Modeling Flows of Control by Organization

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 one another.

- 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 post conditions to each message.



Modeling Flows of Control by Organization

For example, Figure 18-5 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` objects for 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.

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 and collaboration 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 (int i = 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. By walking 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 and collaboration 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.

4. **Discuss the concepts of activity diagram with neat diagram?**

Answer:

An ***activity diagram*** shows the flow from activity to activity. An is an ongoing non-atomic 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

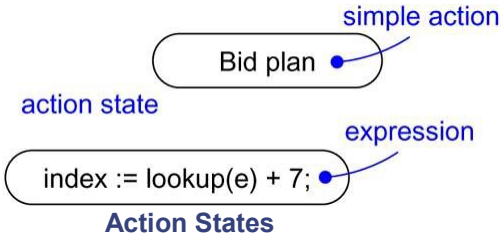
- Activity states and action states
- Transitions
- Objects

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.

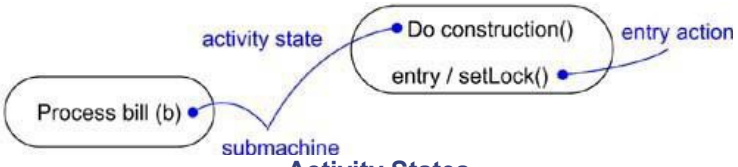
These executable, atomic computations are called action states because they are states of the system, each representing the execution of an action. As Figure shows, you represent an action state using a lozenge shape (a symbol with horizontal top and bottom and convex sides). Inside that shape, you may write any expression.

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.



Action States

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.

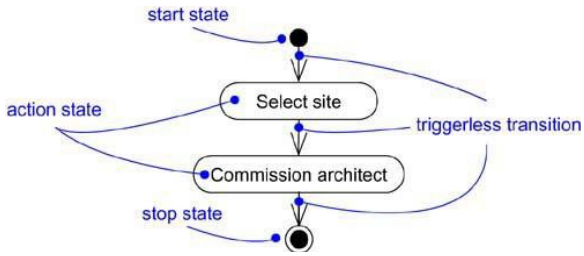


Activity States

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.



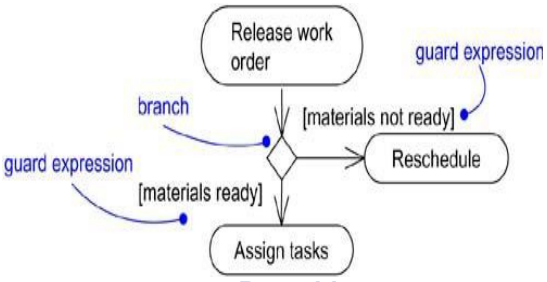
Trigger less Transitions

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

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).



Branching

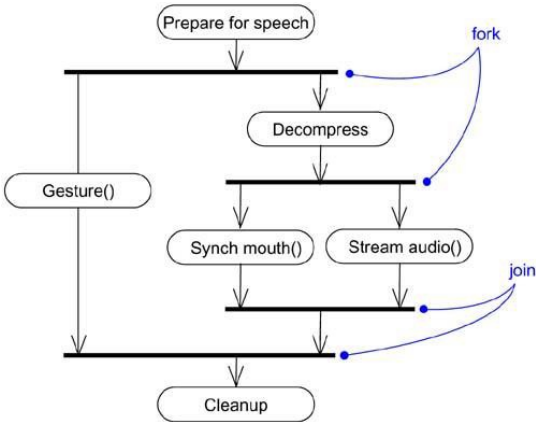
As a convenience, you can use the keyword `else` to mark one outgoing transition, representing the path taken if no other guard expression evaluates to true.

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-animatronics 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 continue 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

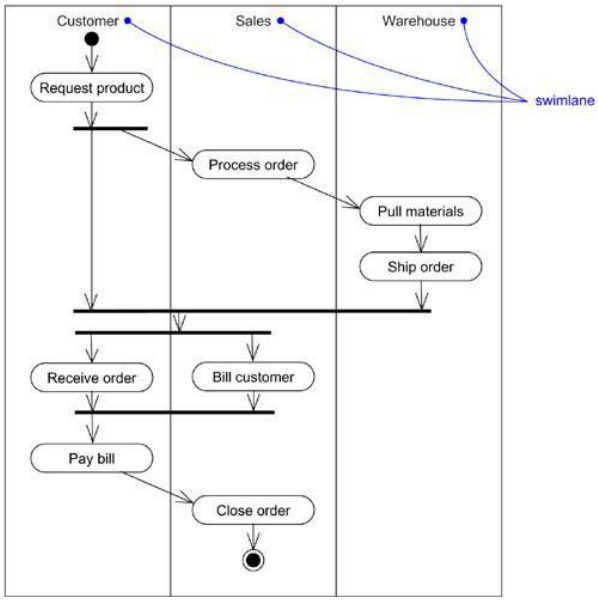
(in the case of a system deployed across multiple nodes) or sequential yet interleaved (in the case of a system deployed across one node), thus giving only the illusion of true concurrency.



Forking and Joining

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.



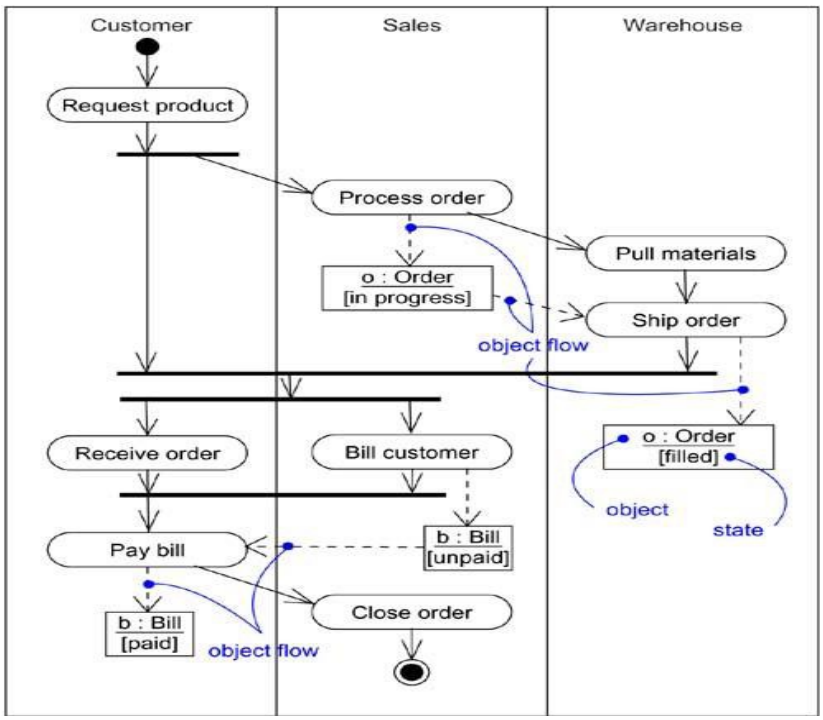
Swimlanes

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 may eventually 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.



Object Flow

5. **Explain the common modeling techniques for activity diagram?**

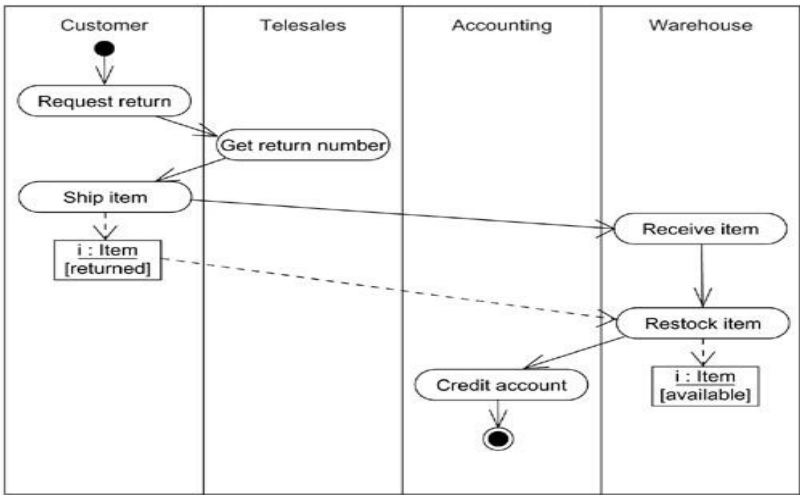
Answer:

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 post conditions 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 action states.
• 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.



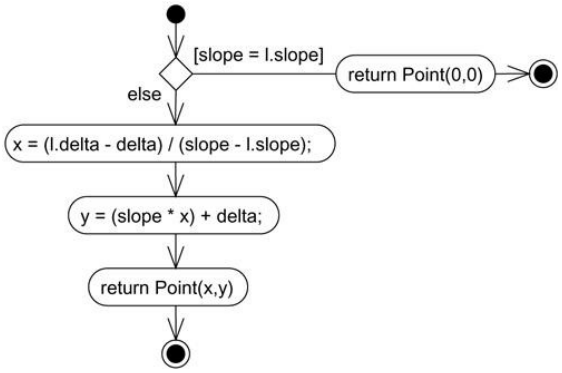
Modeling a Workflow

For example, Figure shows an activity diagram for a retail business, which specifies the workflow 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 diagram indicates, one significant object (`i`, an instance of `Item`) also flows the process, changing from the `returned` to the `available` state

Modeling an Operation

An activity diagram can be attached to any modeling element for the purpose of visualizing, specifying, constructing, and documenting that element's behavior. You can attach activity diagrams to classes, interfaces, components, nodes, use cases, and collaborations. The most common element to which you'll attach an activity diagram is 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 post conditions 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 action states.
• 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.



Modeling an Operation

For example, in the context of the class `Line`, Figure 19-10 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).

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.

6. **Enumerate the steps to modeling the behavior of an Element?**

Answer:

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

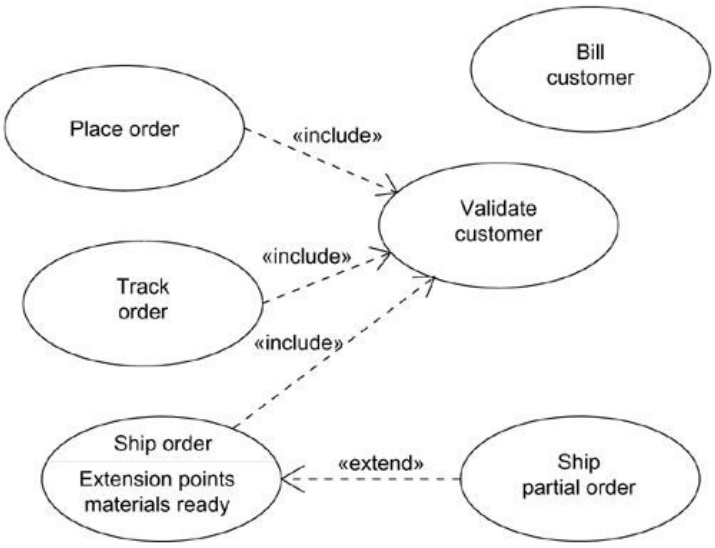
degree sufficient for developers to construct its inside view. Use cases provide a forum for your domain experts, end users, and developers to communicate to one another.

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.



Modeling the Behavior of an Element

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 16-6 shows, you can model the behavior of such a system by declaring these behaviors as use cases (`Place order`, `Track order`, `Ship order`, and `Bill customer`). Common behavior can be factored out (`Validate customer`) and variants (`Ship partial order`) can be distinguished, as well. For each of these use cases, you would include a specification of the behavior, either by text, state machine, or interactions.

1. **Define class, Explain how depicts attributes, operations and responsibilities of a class with suitable example?**

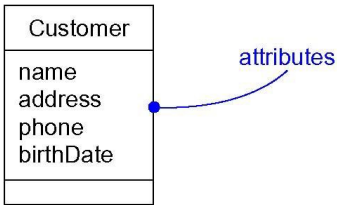
Answer:

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.

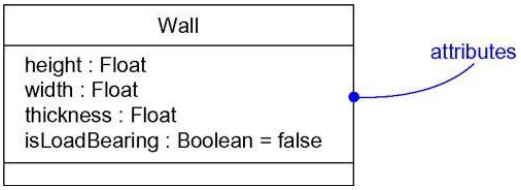
An ***attribute*** is a named property of a class that describes a range of values that instances of the property may hold. A class may have any number of attributes or no attributes at all. An attribute represents some property of the thing you are modeling that is shared by all objects of that class.

For example, every wall has a height, width, and thickness; you might model your customers in such a way that each has a name, address, phone number, and date of birth. An attribute is therefore an abstraction of the kind of data or state an object of the class might encompass. At a given moment, an object of a class will have specific values for every one of its class's attributes.

Graphically, attributes are listed in a compartment just below the class name. Attributes may be drawn showing only their names.



Attributes

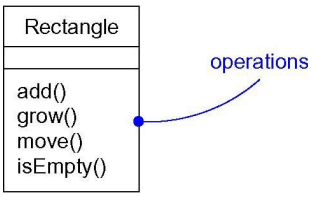


Attributes and Their Class

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.

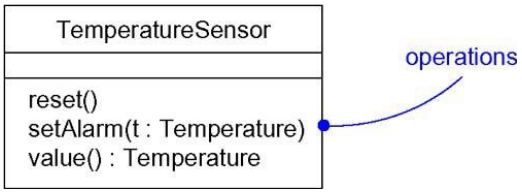
For example, in a windowing library such as the one found in Java's `awt` package, all objects of the class `Rectangle` can be moved, resized, or queried for their properties. Often (but not always), invoking an operation on an object changes the object's data or state.

Graphically, operations are listed in a compartment just below the class attributes. Operations may be drawn showing only their names.



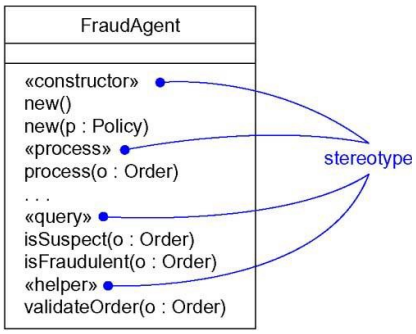
Operations

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.



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 ("`. . .`").



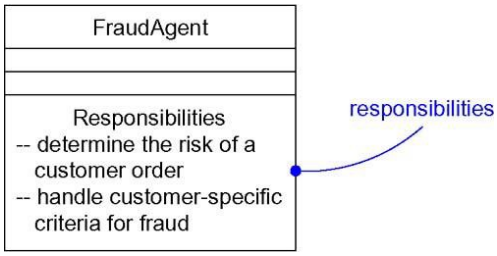
Stereotypes for Class Features

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.



Responsibilities

2. **Give a detail note on stereotypes and tagged values?**

Answer:

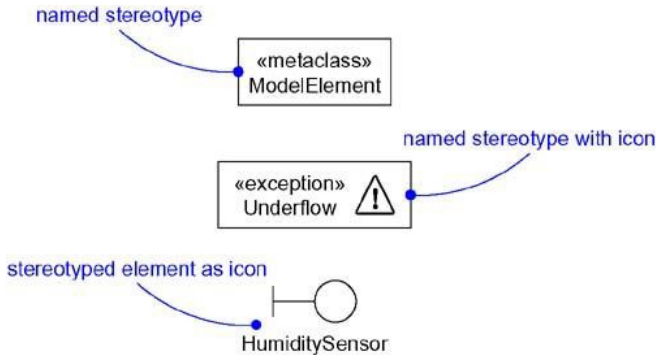
Stereotypes

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 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.

The UML by creating a new building block just like an existing one but with its own special properties (each stereotype may provide its own set of tagged values), semantics (each stereotype may provide its own constraints), and notation (each stereotype may provide its own icon).

In its simplest form, a stereotype is rendered as a name enclosed by guillemets (for example, `»name·d`) and placed above the name of another element. As a visual cue, you may define an icon 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.

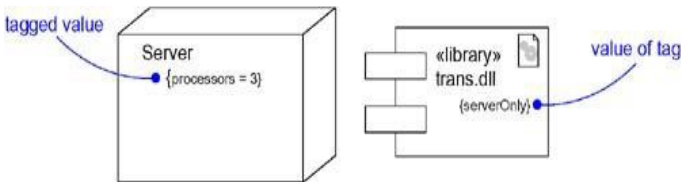


Stereotypes

Tagged Values

Everything 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.



Tagged Values

In its simplest form, a tagged value is rendered as a string enclosed by brackets and placed below the name of another element. That string includes a name (the tag), a separator (the symbol =), and a value (of the tag). You can specify just the value if its meaning is unambiguous, such as when the value is the name of enumeration.

1. **What is Object-Orie nted Analysis and Design?**

During object-oriented analysis there is an emphasis on finding and describing the objects or concepts in the problem domain. For example, in the case of the flight information system, some of the concepts include Plane, Flight, and Pilot. During object-oriented design (or simply, object design) there is an emphasis on defining software objects and how they collaborate to fulfill the requirements. The combination of these two concepts shortly known as object oriented analysis and design.

2. **What is the UML?**

The Unified Modeling Language is a visual language for specifying, constructing and documenting the artifacts of systems.

3. **What are the three ways and perspectives to Apply UML?**

Ways-UML as sketch, UML as blueprint, UML as programming language Perspectives-Conceptual perspective, Specification (software) perspective, Implementation (Software) perspective.

4. **What is Inception?**

Inception is the initial short step to establish a common vision and basic scope for the Project. It will include analysis of perhaps 10% of the use cases, analysis of the critical non- Functional requirement, creation of a business case, and preparation of the development Environment so that

programming can start in the elaboration phase. Inception in one Sentence: Envision the product scope, vision, and business case.

5. What are Actors?

An actor is something with behavior, such as a person (identified by role), computer system, or organization; for example, a cashier.

6. What is a scenario?

A scenario is a specific sequence of actions and interactions between actors and the system; it is also called a use case instance. It is one particular story of using a system, or one path through the use case; for example, the scenario of successfully purchasing items with cash, or the scenario of failing to purchase items because of a credit payment denial.

7. Define Use case.

A use case is a collection of related success and failure scenarios that describe an actor using a system to support a goal. Use cases are text documents, not diagrams, and use-case modeling is primarily an act of writing text, not drawing diagrams.

8. What are Use Case Diagrams?

A use case diagram is an excellent picture of the system context; it makes a good context diagram that is, showing the boundary of a system, what lies outside of it, and how it gets used. It serves as a communication tool that summarizes the behavior of a system and its actors.

9. What are Activity Diagrams?

A diagram which is useful to visualize workflows and business processes. These can be a useful alternative or adjunct to writing the use case text, especially for business use cases that describe complex workflows involving many parties and concurrent actions.

10. Explain about Unified process phases.

- Iterative Development

-Additional UP Best Practices and Concepts
-The UP Phases and Schedule
-The UP Disciplines (was Workflows)
-Process Customization and the Development *Case*
-The Agile UP
-The Sequential "Waterfall

11. Explain about Use-Case Model and its Writing Requirements in Context.

-Background
-Use Cases and Adding Value
-Use Cases and Functional Requirements
-Use Case Types and Formats
-Fully Dressed Example: Process Sale

12. What is Elaboration?

Elaboration is the initial series of iterations during which the team does serious investigation, implements (programs and tests) the core architecture, clarifies most requirements, and tackles the high-risk issues. In the UP, "risk" includes business value. Therefore, early work may include implementing scenarios that are deemed important, but are not especially technically risky.

13. What is Aggregation?

Aggregation is a vague kind of association in the UML that loosely suggests whole-part relationships (as do many ordinary associations). It has no meaningful distinct semantics in the UML versus a plain association, but the term is defined in the UML.

14. Write the characteristics of an object.

Identity, classification, polymorphism, and inheritance

15. What is an attribute? Give example.

An attribute is a data value held by the objects in a class .Example: name, age and weight are attributes of Person class.

16. What is multiple inheritance?

When one class inherits its state (attributes) and behavior from more than one super class, it is referred to as multiple inheritances.

17. Write the four quality measures for software development?

Correspondence, correctness, verification, and validation

18. What is polymorphism? Give an example.

Polymorphism means that the same operation may behave differently on different classes. Ex. Move operation. (Behave differently on the window class and chess Piece class).

19. What is a meta-class?

A meta-class is a class about a class. They are normally used to provide instance variables and operations.

20. What is the need of an Object diagram?

An object diagram is used to show the existence of objects and their relationships in the logical design of a system.

21. What is an association? Give one example.

An association is the relationship between the classes. Ex person and company are the classes, works-for is the association name. *Works_for.*

22. Define Prototype?

A prototype is a version of a software product developed in the early stages of the product's life cycle for specific, experimental purposes. A prototype enables you to fully understand how easy or difficult it will be to implement some of the features of the system.

23. Write any two advantages of modeling?

The main reason for modeling is the reduction of complexity. The cost of the modeling analysis is much lower than the cost of similar experimentation conducted with real time.

24. What are Conceptual Classes?

The domain model illustrates conceptual classes or vocabulary in the domain. Informally, a conceptual class is an idea, thing, or object. More formally, a conceptual class may be considered in terms of its symbol, intension, and extension

a) Symbol words or images representing a conceptual class.
b) Intension the definition of a conceptual class.
c) Extension the set of examples to which the conceptual class applies

25. What is composition?

Composition, also known as composite aggregation, is a strong kind of whole-part aggregation and is useful to show in some models. A composition relationship implies that 1) an instance of the part (such as a Square) belongs to only one composite instance (such as one Board) at a time, 2) the part must always belong to a composite (no free- floating Fingers), and 3) the composite is responsible for the creation and deletion of its parts either by itself creating/deleting the parts, or by collaborating with other objects.