Unit-2 Part-1

Basic Structural Modelling

1. Classes
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.

You use classes to capture the vocabulary of the system you are developing. These classes may include abstractions that are part of the problem domain, as well as classes that make up an implementation.

You can use classes to represent software things, hardware things, and even things that are purely conceptual.

Graphically, a class is rendered as a rectangle.

![Figure 4-1 Classes]

Names
A class name must be unique within its enclosing package. 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 4-2 shows.

![Figure 4-2 Simple and Path Names]

Attributes
Attributes are related to the semantics of aggregation. 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, as shown in Figure 4-3.
You can further specify an attribute by stating its class and possibly a default initial value, as shown Figure 4-4.

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 4-5.

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 4-7.
Responsibilities
Responsibilities are an example of a defined stereotype. 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. Graphically, responsibilities can be drawn in a separate compartment at the bottom of the class icon, as shown in Figure 4-8.

Responsibilities are just free-form text. In practice, a single responsibility is written as a phrase, a sentence, or (at most) a short paragraph.

2. Relationships

A relationship is a connection among things. In object-oriented modeling, there are three kinds of relationships that are especially important: dependencies, which represent using relationships among classes (including refinement, trace, and bind relationships); generalizations, which link generalized classes to their specializations; and associations, which represent structural relationships among objects.

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 not necessarily 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 5-2 Dependencies](image)

A dependency can have a name, although names are rarely needed unless you have a model with many dependencies and you need to refer to or distinguish among dependencies. More commonly, you'll use stereotypes to distinguish different flavors of 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.

Graphically, generalization is rendered as a solid directed line with a large open arrowhead, pointing to the parent, as shown in Figure 5-3. Use generalizations when you want to show parent/child relationships.

![Figure 5-3 Generalization](image)

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.

**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 object of 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 5-4.

**Figure 5-4 Association Names**

![Figure 5-4 Association Names](image)

**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 5-5, a Person playing the role of employee is associated with a Company playing the role of employer.

**Figure 5-5 Roles**

![Figure 5-5 Roles](image)

**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 5-6. 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 5-6 Multiplicity**

![Figure 5-6 Multiplicity](image)

You can specify more complex multiplicities by using a list, such as 0..1, 3..4, 6..*, which would mean "any number of objects other than 2 or 5."
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 5-7.

![Figure 5-7 Aggregation](image)

The meaning of this simple form of aggregation is entirely conceptual. The open diamond distinguishes the "whole" from the "part," no more, no less. This means that 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.

3. Common Mechanisms
The UML is made simpler by the presence of four common mechanisms that apply consistently throughout the language: specifications, adornments, common divisions, and extensibility mechanisms.

Notes are the most important kind of adornment that stands alone. A note is a graphical symbol for rendering constraints or comments attached to an element or a collection of elements. You use notes to attach information to a model, such as requirements, observations, reviews, and explanations.

The UML’s extensibility mechanisms permit you to extend the language in controlled ways. These mechanisms include stereotypes, tagged values, and 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. A tagged value extends the properties of a UML building block, allowing you to create new information in that element’s specification. A constraint extends the semantics of a UML building block, allowing you to add new rules or modify existing ones. You use these mechanisms to tailor the UML to the specific needs of your domain and your development culture.
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.
A note may contain any combination of text or graphics. If your implementation allows it, you can put a live URL inside a note, or even link to or embed another document. In this way, you can use the UML to organize all the artifacts you might generate or use during development, as Figure 6-3 illustrates.

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.

Stereotypes
The UML provides a language for structural things, behavioral things, grouping things, and notational things. These four basic kinds of things address the overwhelming majority of the systems you'll need to model. However, sometimes you'll want to introduce new things that speak the vocabulary of your domain and look like primitive building blocks.

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. 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 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 6-5.

Figure 6-5 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.

For example, as Figure 6-6 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 6-6 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 6-7 shows, you might want to specify that, across a given association, communication is encrypted.

Figure 6-7 Constraint
Constraints 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).

5. Class Diagrams
Class diagrams are the most common diagram found in modeling object-oriented systems. A class diagram shows a set of classes, interfaces, and collaborations and their relationships. You use class diagrams to model the static design view of a system. For the most part, this involves modeling the vocabulary of the system, modeling collaborations, or modeling schemas.
Class diagrams are also the foundation for a couple of related diagrams: component diagrams and deployment 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.

Contents
Class diagrams commonly contain the following things:
- Classes
- Interfaces
- Collaborations
- Dependency, generalization, and association relationships

With the UML, you use class diagrams to visualize the static aspects of these building blocks and their relationships and to specify their details for construction, as you can see in Figure 8-1.

![Figure 8-1 A Class Diagram](image)
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

1. 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.

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.

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 their parent, Motor.
2. 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.
3. 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.

*Forward engineering* is the process of transforming a model into code through a mapping to an implementation 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.
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.

```java
public abstract class EventHandler {
    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 a specific 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.
- Using your tool, create a class diagram by querying the model. For example, you might start with one or more classes, then expand the diagram by following specific relationships or other neighboring classes. Expose or hide details of the contents of this class diagram as necessary to communicate your intent.

### 6. Object Diagrams

Object diagrams model the instances of things contained in class diagrams. An object diagram shows a set of objects and their relationships at a point in time.

You use object diagrams to model the static design view or static process view of a system. This involves modeling a snapshot of the system at a moment in time and rendering a set of objects, their state, and their relationships.
An object diagram, therefore, expresses the static part of an interaction, consisting of the objects that collaborate, but without any of the messages passed among them. In both cases, an object diagram freezes a moment in time, as in Figure 14-1.

**Figure 14-1 An Object Diagram**

![Object Diagram](image)

**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.

**Contents**
Object diagrams commonly contain
- Objects
- Links

**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

**Common Modeling Techniques**

1. **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.

   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 elements that 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 associations among them.

For example, Figure 14-2 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.

**Figure 14-2 Modeling Object Structures**

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.

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 necessary to 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.
7. Interaction Diagrams
Sequence diagrams and collaboration diagrams—both of which are called interaction diagrams—are two of the five diagrams used in the UML for modeling the dynamic aspects of systems. 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; a collaboration diagram is an interaction diagram that emphasizes the structural organization of the objects that send and receive messages.

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.

Contents
Interaction diagrams commonly contain
· Objects
· Links
· Messages

A. Sequence Diagrams
A sequence diagram emphasizes the time ordering of messages. As Figure 18-2 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.

Sequence diagrams have two features that distinguish them from collaboration diagrams.
You can specify the vitality of an object or a link by marking it with a new destroyed, or transient constraint.

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.

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.

B. Collaboration Diagrams

A collaboration diagram emphasizes the organization of the objects that participate in an interaction.

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 local to the sender).

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.

As Figure 18-3 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 18-3 Collaboration Diagram**

![Collaboration Diagram](image)

**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, the sequence 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 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 behaviour 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
1. 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.

2. 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 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 postconditions to each message.

3. 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.

```java
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.

Forward engineering is straightforward; reverse engineering is hard. It's easy to get too much information from simple reverse engineering, and so the hard part is being clever about what details to keep.

Unit-2 Part-2

1. Use Case Diagrams
Use case diagrams are one of the five diagrams in the UML for modeling the dynamic aspects of systems (activity diagrams, statechart diagrams, sequence diagrams, and collaboration diagrams are four other kinds of diagrams in the UML for modeling the dynamic aspects of systems). Use case diagrams are central to modeling the behavior of a system, a subsystem, or a class. Each one shows a set of use cases and actors and their relationships.
You apply use case diagrams to model the use case view of a system. For the most part, this involves modeling the context of a system, subsystem, or class, or modeling the requirements of the behavior of these elements.
As Figure 17-1 shows, you can provide a use case diagram to model the behavior of that box• which most people would call a cellular phone.

Figure 17-1 A Use Case Diagram
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

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
1. 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.

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 17-2 shows the context of a credit card validation system, with an emphasis on the actors that surround the system.

Figure 17-2 Modeling the Context of a System

2. 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 implement the system.

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.
3. 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.

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.

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 the system, 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.

2. Activity Diagrams

Activity diagrams are one of the five diagrams in the UML for modeling the dynamic aspects of systems. An activity diagram is essentially a flowchart, showing flow of control from activity to activity. You use activity diagrams to model the dynamic aspects of a system. For the most part, this involves modeling the sequential (and possibly concurrent) steps in a computational process. Consider the workflow associated with building a house.

An activity is an ongoing nonatomic execution within a state machine. Activities ultimately result in some action, which is made up of executable atomic computations that results in a change in state of the system or the return of a value.

Activity diagrams are not only important for modeling the dynamic aspects of a system, but also for constructing executable systems through forward and reverse engineering.
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. 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—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 19-2 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 19-2 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.

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 19-3 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.

Action states and activity states are just special kinds of states in a state machine. When you enter an action or activity state, you simply perform the action or the activity; when you finish, control passes to the next action or activity. Activity states are somewhat of a shorthand, therefore. An activity state is semantically equivalent to expanding its activity graph (and transitively so) in place until you only see actions.

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 19-4 shows.

Semantically, these are called triggerless, or completion, transitions because control passes immediately once the work of the source state is done.

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 19-5 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 19-5 Branching**

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.

**Figure 19-6 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 19-7. A swimlane specifies a locus of activities.

There's a loose connection between swimlanes and concurrent flows of control. Conceptually, the activities of each swimlane are generally but not always considered separate from the activities of neighboring swimlanes. That makes sense because, in the real world, the business organizations that generally map to these swimlanes are independent and concurrent.
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).

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 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**

1. **Modeling a Workflow**

No software-intensive system exists in isolation; there's always some context in which a system lives, and that context always encompasses actors that interact with the system. You can model the business processes for the way these various automated and human systems collaborate by using activity diagrams.

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 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.

For example, Figure 19-9 shows an activity diagram for a retail business, which specifies the workflow involved when a customer returns an item from a mail order.
2. Modeling an Operation
The most common element to which you’ll attach an activity diagram is an operation. Used in this manner, an activity diagram is simply a flowchart of an operation's actions. An activity diagram's primary advantage is that all the elements in the diagram are semantically tied to a rich underlying model. For example, any other operation or signal that an action state references can be type checked against the class of the target object.

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 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.

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.

```cpp
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.

### 3. Events and Signals

In the real world, things happen. Not only do things happen, but lots of things may happen at the same time, and at the most unexpected times. "Things that happen" are called events, and each one represents the specification of a significant occurrence that has a location in time and space. In the context of state machines, you use events to model the occurrence of a stimulus that can trigger a state transition. Events may include signals, calls, the passing of time, or a change in state.

Events may be synchronous or asynchronous, so modeling events is wrapped up in the modelling of processes and threads.

In the UML, each thing that happens is modeled as an event. The UML provides a graphical representation of an event, as Figure 20-1 shows. This notation permits you to visualize the declaration of events (such as the signal OffHook), as well as the use of events to trigger a state transition (such as the signal OffHook, which causes a transition from the Active to the Idle state of a telephone).

![Figure 20-1 Events](image)

**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.
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.
In the UML, as Figure 20-2 shows, you model signals (and exceptions) as stereotyped classes. You can use a dependency, stereotyped as send, to indicate that an operation sends a particular signal.

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.
As Figure 20-3 shows, modeling a call event is indistinguishable from modeling a signal event.

Time and Change Events
A time event is an event that represents the passage of time. As Figure 20-4 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.
4. State Machines

Using a state machine, you can model the behavior of an individual object. 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.

You use state machines to model the dynamic aspects of a system. For the most part, this involves specifying the lifetime of the instances of a class, a use case, or an entire system. An activity 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 model or a return of a value. The state of an object 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.

You can visualize a state machine in two ways: by emphasizing the flow of control from activity to activity (using activity diagrams), or by emphasizing the potential states of the objects and the transitions among those states (using statechart diagrams).

You use state machines to model the behavior of any modeling element, although, most commonly, that will be a class, a use case, or an entire system. State machines may be visualized in two ways. First, using activity diagrams, you can focus on the activities that take place within the object. Second, using statechart diagrams, you can focus on the event ordered behavior of an object, which is especially useful in modeling reactive systems.

The UML provides a graphical representation of states, transitions, events, and actions, as Figure 21-1 shows. This notation permits you to visualize the behavior of an object in a way that lets you emphasize the important elements in the life of that object.
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.

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).

As Figure 21-2 shows, you represent a state as a rectangle with rounded corners.

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.
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 21-3 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.

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. An event trigger may be polymorphic. For example, if you’ve specified a family of signals, then a transition whose trigger event is S can be triggered by S, as well as by any children of S.

Guard
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.
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. 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.

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.

Entry and exit actions may not have arguments or guard conditions. However, the entry action at the top level of a state machine for a class may have parameters that represent the arguments that the machine receives when the object is created.

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. Internal transitions may have events with parameters and guard conditions. As such, internal transitions are essentially interrupts.

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.

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 it appears as if they just occurred.

The implementation of deferred events requires the presence of an internal queue of events. If an event happens but is listed as deferred, it is queued. Events are taken off this queue as soon as the object enters a state that does not defer these events.
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.

5. Processes and Threads
In the UML, you model each independent flow of control as an active object that represents a process or thread that can initiate control activity. A process is a heavyweight flow that can execute concurrently with other processes; a thread is a lightweight flow that can execute concurrently with other threads within the same process. Building abstractions so that they work safely in the presence of multiple flows of control is hard. In particular, you have to consider approaches to communication and synchronization that are more complex than for sequential systems. You also have to be careful to neither over-engineer your process view (too many concurrent flows and your system ends up thrashing) nor under engineer it (insufficient concurrency does not optimize the system's throughput.

In the UML, each independent flow of control is modeled as an active object. An active object is a process or thread that can initiate control activity. As for every kind of object, an active object is an instance of a class. In this case, an active object is an instance of an active class. Also as for every kind of object, active objects can communicate with one another by passing messages, although here, message passing must be extended with certain concurrency semantics, to help you to synchronize the interactions among independent flows.

The UML provides a graphical representation of an active class, as Figure 22-1 shows. Active classes are kinds of classes, so have all the usual compartments for class name, attributes, and operations. Active classes often receive signals, which you typically enumerate in an extra compartment.

![Figure 22-1 Active Class](image)

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 that can 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.

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. You can achieve true concurrency in one of three ways: first, by distributing active objects across multiple nodes; second, by placing active objects on nodes with multiple processors; and third, by a combination of both methods.

**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.

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. 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.

**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 22-2.

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. It is possible to model variations of synchronous and asynchronous message passing by using constraints. For example, to model a balking rendezvous as found in Ada, you’d use a synchronous message with a constraint such as \{wait = 0\}, saying that the caller will not wait for the receiver. Similarly, you can model a time out by using a constraint such as \{wait = 1 ms\}, saying that the caller will wait no more than one millisecond for the receiver to accept the message.

**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 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 every language that supports concurrency, you can build support for all these properties by constructing them out of semaphores.

As Figure 22-3 shows, you can attach these properties to an operation.

**Figure 22-3 Synchronization**

**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.

**6. Time and Space**
Modeling time and space is an essential element of any real time and/or distributed system. You use a number of the UML’s features, including timing marks, time expressions, constraints, and tagged values, to visualize, specify, construct, and document these systems.
When you start to model most software systems, you can usually assume a frictionless environment—messages are sent in zero time, networks never go down, workstations never fail, the load across your network is always evenly balanced. Unfortunately, the real world does not work that way—messages do take time to deliver (and, sometimes, never get delivered), networks do go down, workstations do fail, and a network’s load is often unbalanced. Therefore, when you encounter systems that must operate in the real world, you have to take into account the issues of time and space.

To represent the modeling needs of real time and distributed systems, the UML provides a graphic representation for timing marks, time expressions, timing constraints, and location, as Figure 23-1 shows.

### Figure 23-1 Timing Constraints and Location

```
<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td>m</td>
<td>c</td>
</tr>
<tr>
<td>: Client</td>
<td>: MapServer</td>
<td>: MapCache</td>
</tr>
<tr>
<td>(location = Console)</td>
<td>(location = Server)</td>
<td>(location = MissionServer)</td>
</tr>
</tbody>
</table>

a : getMap(region)  
b : getMap(region)
```

### 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

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 arbitrarily complex 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 23-2, you can place these time expressions in a timing constraint to specify the timing behavior 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.
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 23-3 shows, the executable component vision.exe may reside on the node named KioskServer.

7. Statechart Diagrams

Statechart diagrams are one of the five diagrams in the UML for modeling the dynamic aspects of systems. A statechart diagram shows a state machine. An activity diagram is a special case of a statechart diagram in which all or most of the states are activity states and all or most of the transitions are triggered by completion of activities in the source state. Thus, both activity and statechart diagrams are useful in modeling the lifetime of an object. However, whereas an activity diagram shows flow of control from activity to activity, a statechart diagram shows flow of control from state to state.

In the UML, you model the event-ordered behavior of an object by using statechart diagrams. As Figure 24-1 shows, a statechart diagram is simply a presentation of a state machine, emphasizing the flow of control from state to state.
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

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 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
Common Modeling Technique

Modeling Reactive Objects
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 24-2 shows the statechart diagram for parsing a simple context-free language, 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 24-2 Modeling Reactive Objects

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.
class MessageParser {
  public
boolean put(char c) {
    switch (state) {
    case Waiting:
        if (c == '<') {
            state = GettingToken;
            token = new StringBuffer();
            body = new StringBuffer();
        }
        break;
    case GettingToken:
        if (c == '>')
            state = GettingBody;
        else
            token.append(c);
        break;
    case GettingBody:
        if (c == ';')
            state = Waiting;
        else
            body.append(c);
        return true;
    }
    return false;
}

StringBuffer getToken() {
    return token;
}

StringBuffer getBody() {
    return body;
}

private
final static int Waiting = 0;
final static int GettingToken = 1;
final static int GettingBody = 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 practically not 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.
1. Components
Components live in the material world of bits and therefore are an important building block in modeling the physical aspects of a system. A component is a physical and replaceable part of a system that conforms to and provides the realization of a set of interfaces. You use components to model the physical things that may reside on a node, such as executables, libraries, tables, files, and documents.
A component typically represents the physical packaging of otherwise logical elements, such as classes, interfaces, and collaborations. Good components define crisp abstractions with well-defined interfaces, making it possible to easily replace older components with newer, compatible ones.

The end product of a construction company is a physical building that exists in the real world. You build logical models to visualize, specify, and document your decisions about the building envelope; the placement of walls, doors, and windows; the routing of electrical and plumbing systems; and the overall architectural style. When you actually construct the building, these walls, doors, windows, and other conceptual things get turned into real, physical things.

The UML provides a graphical representation of a component, as Figure 25-1 shows. This canonical notation permits you to visualize a component apart from any operating system or programming language. Using stereotypes, one of the UML's extensibility mechanisms, you can tailor this notation to represent specific kinds of components.

**Figure 25-1 Components**

![Component Diagram]

**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**
A component name must be unique within its enclosing package.
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 25-2. 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.
2. Deployment

Nodes, just like components, live in the material world and are an important building block in modeling the physical aspects of a system. 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.

You use nodes to model the topology of the hardware on which your system executes. A node typically represents a processor or a device on which components may be deployed. Good nodes crisply represent the vocabulary of the hardware in your solution domain.

The UML provides a graphical representation of node, as Figure 26-1 shows. This canonical notation permits you to visualize a node apart from any specific hardware. Using stereotypes— one of the UML’s extensibility mechanisms— you can (and often will) tailor this notation to represent specific kinds of processors and devices.

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

A node name must be unique within its enclosing package. 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 26-2. Just as with classes, you may draw nodes adorned with tagged values or with additional compartments to expose their details.
3. Component Diagrams

Component diagrams are one of the two kinds of diagrams found in modeling the physical aspects of object-oriented systems. A component diagram shows the organization and dependencies among a set of components. You use component diagrams to model the static implementation view of a system. This involves modeling the physical things that reside on a node, such as executables, libraries, tables, files, and documents. Component diagrams are essentially class diagrams that focus on a system’s components.

You create use case diagrams to reason about the desired behaviour of your system. You specify the vocabulary of your domain with class diagrams. You create sequence diagrams, collaboration diagrams, statechart diagrams, and activity diagrams to specify the way the things in your vocabulary work together to carry out this behavior. Eventually, you will turn these logical blueprints into things that live in the world of bits, such as executables, libraries, tables, files, and documents. You'll find that you must build some of these components from scratch, but you'll also end up reusing older components in new ways.

With the UML, you use component diagrams to visualize the static aspect of these physical components and their relationships and to specify their details for construction, as in Figure 29-1.

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

**Common Uses**
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**
1. **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). 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.
   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 29-2 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 29-2 Modeling Source Code

2. 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.

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 29-3 models part of the executable release for an autonomous robot.

Figure 29-3 Modeling an Executable Release

3. 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.

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 crosstable 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 29-4 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.

4. Modeling Adaptable Systems

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 29-5 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.
5. Forward and Reverse Engineering

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.

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 a new 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 29-6 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.

4. Deployment Diagrams

Deployment diagrams are one of the two kinds of diagrams used in modeling the physical aspects of an object-oriented system. A deployment diagram shows the configuration of run time processing nodes and the components that live on them.

You use deployment diagrams to model the static deployment view of a system. For the most part, this involves modeling the topology of the hardware on which your system executes. Deployment diagrams are essentially class diagrams that focus on a system’s nodes. Deployment diagrams are not only important for visualizing, specifying, and documenting embedded, client/server, and distributed systems, but also for managing executable systems through forward and reverse engineering.

With the UML, you use deployment diagrams to visualize the static aspect of these physical nodes and their relationships and to specify their details for construction, as in Figure 30-1.
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

**Common Uses**

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**

**1. 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.

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 30-2 shows the hardware for a simple autonomous robot. You'll find one node (Pentium motherboard) stereotyped as a processor.

![Figure 30-2: Modeling an Embedded System](image)

**2. Modeling a Client/Server System**

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 30-3 shows the topology of a human resources system, which follows a classical client/server architecture.

**Figure 30-3 Modeling a Client/Server System**

![Figure 30-3](image)

3. **Modeling a Fully Distributed System**
   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 by using 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 30-4 shows the topology of a fully distributed system.

**Figure 30-4 Modeling a Fully Distributed System**

![Figure 30-4](image)

4. **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.
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.