

Chapter 2: Unified Modeling Language (UML)

OBJECTIVES

After the completion of this chapter students will be able to:

- ⇒ Understand basics of UML and its modeling diagrams.
- ⇒ Learn What the UML is
- ⇒ Model software using UML
- ⇒ Appreciate that UML is a language, *not* a methodology

INTRODUCTION

Before we start writing documentation for software that solves a problem, you should have a thorough understanding of Unified Modeling Language. You should also know how to organize the different diagrams that constitute the unified modeling language. In the first chapter we have talked about Unified Modeling Language in brief. But in this chapter we discuss the history of unified modeling language, the different diagrams in unified modeling language and the organization of those diagrams in detail. The concepts present here are crucial in describing the overall system using unified modeling language.

We introduce unified modeling language's things, diagrams and relationships, three of the building blocks that are vital in describing our system. We devote this chapter to better explain unified modeling language using Emergency Report case study.

2.1. An Overview Of UML

The Unified Modeling Language (UML) is a standard graphical language for modeling object-oriented software. It was developed in the mid-1990s as a collaborative effort by James Rumbaugh, Grady Booch and Ivar Jacobson, each of whom had developed their own notation in the early 1990s. The unified modeling language is a language for specifying, constructing, visualizing, and documenting the software system and its components. The UML is a graphical language with sets of rules and semantics. The rules and semantics of a model are expressed in English, in a form known as *object constraint language* (OCL). OCL is a specification language that uses simple logic for specifying the properties of a system. The UML is not intended to be a visual programming language in the sense of having all the necessary visual and semantic support to

replace programming languages. However, the UML does have a tight mapping to a family of object-oriented languages, so that you can get the best of both worlds.

The goals of the unification efforts were to keep it simple; to cast away elements of existing Booch, OMT, and OOSE methods that did not work in practice; to add elements from other methods that were more effective; and to invent new methods only when an existing solution was unavailable. Because the UML authors, in effect, were designing a language (albeit a graphical one), they had to strike a proper balance between minimalism (everything is text and boxes) and over engineering (having a symbol or fig for every conceivable modeling element).

The primary goals in the design of the UML were as follows:

- Provide users a ready-to-use, expressive visual modeling language so they can develop and exchange meaningful models.
- Provide extensibility and specialization mechanisms to extend the core concepts.
- Be independent of particular programming languages and development processes.
- Provide a formal basis for understanding the modeling language.
- Encourage the growth of the OO tools market.
- Support higher-level development concepts.
- Integrate best practices and methodologies.

2.2. Building Blocks Of UML

As UML describes the real time systems it is very important to make a conceptual model and then proceed gradually. Conceptual model of UML can be mastered by learning the following three major elements:

- UML building blocks
- Rules to connect the building blocks
- Common mechanisms of UML

This chapter describes all the UML building blocks. The building blocks of UML can be defined as:

- Things
- Relationships
- Diagrams

(1) Things:

Things are the most important building blocks of UML. Things can be:

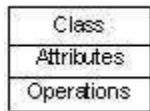
- Structural
- Behavioral
- Grouping
- An notational

Structural things:

The **Structural things** define the static part of the model. They represent physical and conceptual elements. Following are the brief descriptions of the structural things.

Class:

Class represents set of objects having similar responsibilities.



Interface:

Interface defines a set of operations which specify the responsibility of a class.



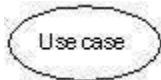
Collaboration:

Collaboration defines interaction between elements.



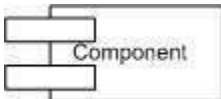
Use case:

Use case represents a set of actions performed by a system for a specific goal.



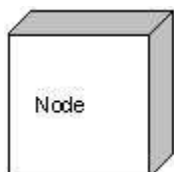
Component:

Component describes physical part of a system.



Node:

A node can be defined as a physical element that exists at run time.

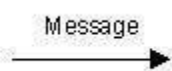


Behavioral things:

A **behavioral thing** consists of the dynamic parts of UML models. Following are the behavioral things:

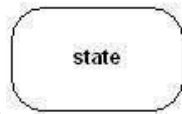
Interaction:

Interaction is defined as a behavior that consists of a group of messages exchanged among elements to accomplish a specific task.



State machine:

State machine is useful when the state of an object in its life cycle is important. It defines the sequence of states an object goes through in response to events. Events are external factors



responsible for state change.

Grouping things:

Grouping things can be defined as a mechanism to group elements of a UML model together. There is only one grouping thing available:

Package:

Package is the only one grouping thing available for gathering structural and behavioral things.

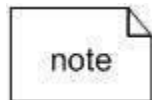


An notational things:

An notational things can be defined as a mechanism to capture remarks, descriptions, and comments of UML model elements. **Note** is the only one An notational thing available.

Note:

A note is used to render comments, constraints etc. of an UML element.



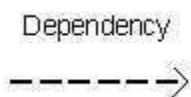
(2) Relationship :

Relationship is another most important building block of UML. It shows how elements are associated with each other and this association describes the functionality of an application.

There are four kinds of relationships available.

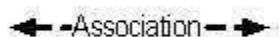
Dependency:

Dependency is a relationship between two things in which change in one element also affects the other one.



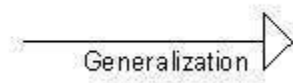
Association:

Association is basically a set of links that connects elements of an UML model. It also describes how many objects are taking part in that relationship.



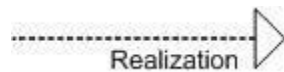
Generalization:

Generalization can be defined as a relationship which connects a specialized element with a generalized element. It basically describes inheritance relationship in the world of objects.



Realization:

Realization can be defined as a relationship in which two elements are connected. One element describes some responsibility which is not implemented and the other one implements them. This relationship exists in case of interfaces.



2.3. UML Diagrams:

UML diagrams are the ultimate output of the entire discussion. All the elements, relationships are used to make a complete UML diagram and the diagram represents a system.

The visual effect of the UML diagram is the most important part of the entire process. All the other elements are used to make it a complete one.

UML includes the following nine diagrams and the details are described in the following chapters.

- Class diagram
- Collaboration diagram
- Deployment diagram
- Object diagram
- Activity diagram
- Component diagram
- Use case diagram
- State chart diagram

2.3.1. Use Case Diagrams

Use cases and actors

Actors are external entities that interact with the system. Examples of actors include a user role (e.g., a system administrator, a bank customer, a bank teller) or another system (e.g., a central database, a fabrication line). Actors have unique names and descriptions.

Use cases describe the behavior of the system as seen from an actor's point of view. Behavior described by the use case model is also called **external behavior**. A use case describes a function provided by the system as a set of events that yields a visible result for the actors. Actors initiate a use case to access the system functionality. The use case can then initiate other use cases and gather more information from the actors. When actors and use cases exchange information, they are said to **communicate**. We will see later that we represent these exchanges with communication relationships.

For example, in an accident management system, field officers, such as, a police officer or a fireman have access to a wireless computer that enables them to interact with a dispatcher. The dispatcher in turn can visualize the current status of all its resources, such as, police cars or trucks, on a computer screen and dispatch a resource by issuing commands from a workstation. In this example, FieldOfficer and Dispatcher are actors.

Figure 2.1 depicts the actor FieldOfficer who invokes the use case ReportEmergency to notify the actor Dispatcher of a new emergency. As a response, the Dispatcher invokes the OpenIncident use case to create an incident report and initiate the incident handling. The Dispatcher enters preliminary information from the FieldOfficer in the incident database and orders additional units to the scene with the AllocateResources use case.

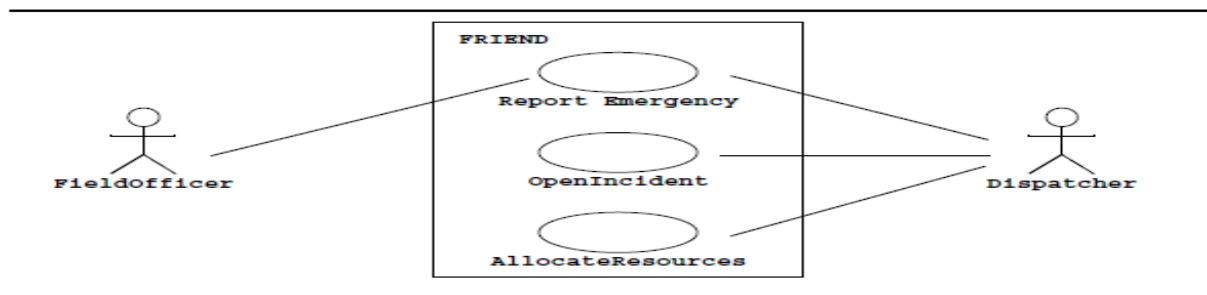


Figure 2.1 An example of a UML use case diagram

Incident initiation in an accident management system. Associations between actors and use cases represent information flows. In UML, these associations are bidirectional: They can represent the actor initiating a use case (e.g., FieldOfficer initiates ReportEmergency) or a use case providing information to an actor (e.g., ReportEmergency notifies Dispatcher).

To describe a use case, we use a template composed of six fields (see also Figure 2.2):

- The **name** of the use case is unique across the system so that developers (and project participants) can unambiguously refer to the use case.
- **Participating actors** are actors interacting with the use case.
- **Entry conditions** describe the conditions that need to be satisfied before the use case is initiated.
- The **flow of events** describes the sequence of actions of the use case, which are numbered for reference. The common case (i.e., cases that occur frequently) and the exceptional cases (i.e., cases that seldom occur, such as errors and unusual conditions) are described separately in different use cases for clarity.
- **Exit conditions** describe the conditions that are satisfied after the completion of the use case.
- **Special requirements** are requirements that are not related to the functionality of the system. These include constraints on the performance of the system, its implementation, the hardware platforms it runs on, and so on.

Use cases are written in natural language. This enables developers to use them for communicating with the client and the users, who generally do not have an extensive knowledge of software

engineering notations. The use of natural language also enables participants from other disciplines to understand the requirements of the system.

<i>Use case name</i>	ReportEmergency
<i>Participating actor</i>	Invoked by FieldOfficer Communicates with Dispatcher
<i>Entry condition</i>	1. The FieldOfficer activates the "Report Emergency" function of her terminal. FRIEND responds by presenting a form to the officer.
<i>Flow of events</i>	2. The FieldOfficer fills the form by selecting the emergency level, type, location, and brief description of the situation. The FieldOfficer also describes possible responses to the emergency situation. Once the form is completed, the FieldOfficer submits the form, at which point the Dispatcher is notified. 3. The Dispatcher reviews the submitted information and creates an incident in the database by invoking the OpenIncident use case. The Dispatcher selects a response and acknowledges the emergency report.
<i>Exit condition</i>	4. The FieldOfficer receives the acknowledgment and the selected response.
<i>Special requirements</i>	The FieldOfficer's report is acknowledged within 30 seconds. The selected response arrives no later than 30 seconds after it is sent by the Dispatcher.

Figure 2.2 An example of a use case: the ReportEmergency use case.

Scenarios

A use case is an abstraction that describes all possible scenarios involving the described functionality. A **scenario** is an instance of a use case describing a concrete set of actions. Scenarios are used as examples for illustrating common cases. Their focus is on understandability. Use cases are used to describe all possible cases. Their focus is on completeness. We describe a scenario using a template with three fields:

- The **name** of the scenario enables us to refer to it unambiguously. The name of a scenario is underlined to indicate that it is an instance.
- The **participating actor instances** field indicates which actor instances are involved in this scenario. Actor instances also have underlined names.
- The **flow of events** of a scenario describes the sequence of events step by step.

Communication relationships

Actors and use cases **communicate** when information is exchanged between them. Communication relationships are depicted by a solid line between the actor and use case symbol. In Figure 2.1, the actors FieldOfficer and Dispatcher communicate with the ReportEmergency use case. Only the actor Dispatcher communicates with the use cases OpenIncident and AllocateResources. Communication relationships between actors and use cases can be used to denote access to functionality. In the case of our example, a FieldOfficer and a Dispatcher are provided with different interfaces to the system and have access to different functionalities.

Include relationships

When describing a complex system, its use case model can become quite complex and can contain redundancy. We reduce the complexity of the model by identifying commonalities in different use cases. For example, assume that the Dispatcher can press at any time a key to access Help. This

can be modeled by a use case HelpDispatcher that is included by the use cases OpenIncident and AllocateResources (and any other use cases accessible by the Dispatcher). The resulting model only describes the HelpDispatcher functionality once, thus reducing complexity. Two use cases are related by an include relationship if one of them includes the second one in its flow of events. In UML, **include relationships** are depicted by a dashed arrow originating from the including use case. Include relationships are labeled with the string <<include>>.

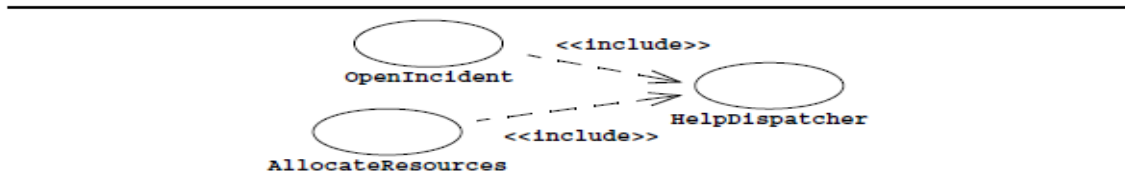


Figure 2.3 An example of an <<include>>relationship (UML use case diagram).

We represent include relationships in the use case itself with one of two ways. If the included use case can be included at any point in the flow of events (e.g., the HelpDispatcher use case), we indicate the inclusion in the *Special requirements* section of the use case. If the included use case is explicitly invoked during an event, we indicate the inclusion in the flow of events.

Extend relationships

Extend relationships are an alternate means for reducing complexity in the use case model. A use case can extend another use case by adding events. An extend relationship indicates that an instance of an extended use case may include (under certain conditions) the behavior specified by the extending use case. A typical application of extend relationships is the specification of exceptional behavior. For example (Figure 2-19), assume that the network connection between the Dispatcher and the FieldOfficer can be interrupted at any time. (e.g., if the FieldOfficer enters a tunnel). The use case ConnectionDown describes the set of events taken by the system and the actors while the connection is lost. ConnectionDown extends the use cases OpenIncident and AllocateResources. Separating exceptional behavior from common behavior enables us to write shorter and more focused use cases.

In the textual representation of a use case, we represent extend relationships as entry conditions of the extending use case. For example, the extend relationships depicted in Figure 2.4 are represented as an entry condition of the ConnectionDown use case

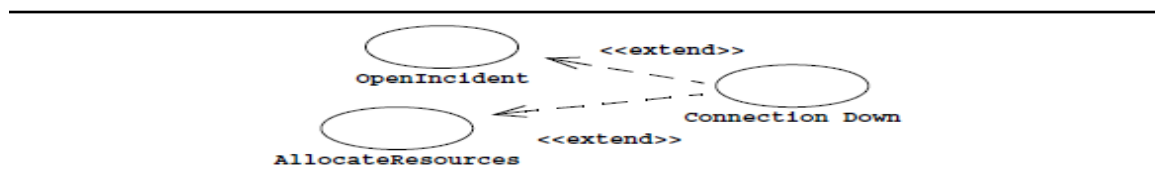


Figure 2.4 An example of an <<extend>>relationship (UML use case diagram).

Generalization relationships

Generalization/specialization relationships are a third mechanism for reducing the complexity of a model. A use case can specialize a more general one by adding more detail. For example, FieldOfficers are required to authenticate before they can use FRIEND. During early stages of requirements elicitation, authentication is modeled as a high-level Authenticate use case. Later, developers describe the Authenticate use case in more detail and allow for several different hardware platforms. This refinement activity results in two more use cases, AuthenticateWithPassword which enables FieldOfficers to login without any specific hardware, and AuthenticateWithCard, which enables FieldOfficers to login using a smart card. The two new use cases are represented as specializations of the Authenticate use case.

2.3.2. Class Diagrams

Classes and objects

Class diagrams describe the structure of the system in terms of classes and objects. **Classes** are abstractions that specify the attributes and behavior of a set of objects. **Objects** are entities that encapsulate state and behavior. Each object has an identity: It can be referred individually and is distinguishable from other objects.

In UML, classes and objects are depicted by boxes including three compartments. The top compartment displays the name of the class or object. The center compartment displays its attributes, and the bottom compartment displays its operations. The attribute and operation compartment can be omitted for clarity. Object names are underlined to indicate that they are instances. By convention, class names start with an uppercase letter. Objects in object diagrams may be given names (followed by their class) for ease of reference. In that case, their name starts with a lowercase letter.

The main symbols shown on class diagrams are:

- **Classes**, which represent the types of data themselves.
- **Associations**, which show how instances of classes reference instances of other classes.
- **Attributes**, which are simple data found in instances.
- **Operations**, which represent the functions performed by the instances.
- **Generalizations**, which are used to arrange classes into inheritance hierarchies.

In the FRIEND, Bob and Alice are field officers, represented in the system as FieldOfficer objects called bob: FieldOfficer and alice:FieldOfficer. FieldOfficer is a class describing all FieldOfficer objects, whereas Bob and Alice are represented by two individual FieldOfficer objects.

In Figure 2.5, the FieldOfficer class has two attributes: a name and a badgeNumber. This indicates that all FieldOfficer objects have these two attributes. the bob:FieldOfficer and alice:FieldOfficer objects have specific values for these attributes: “Bob. D.” and “Alice W.”, respectively. In Figure 2.5, the FieldOfficer.name attribute is of type String, which indicates that only instances of String can be assigned to the FieldOfficer.name attribute. The type of an attribute is used to specify the valid range of values the attribute can take. Note that when attribute types are not essential to the

definition of the system, attribute type decisions can be delayed until object design. This allows the developers to concentrate on the functionality of the system and to minimize the number of trivial changes when the functionality of the system is revised.

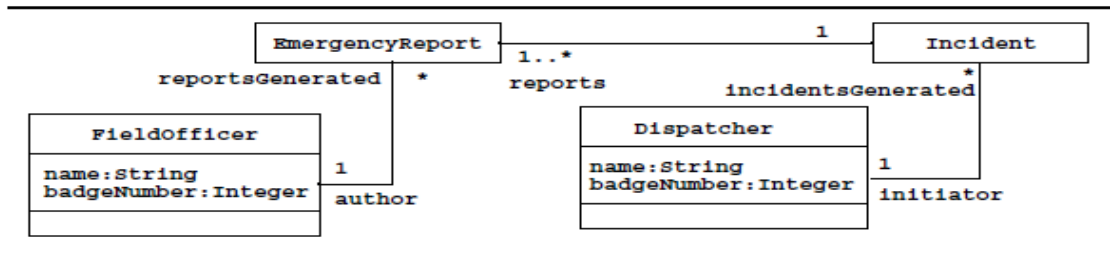


Figure 2.5 An example of a UML class diagram

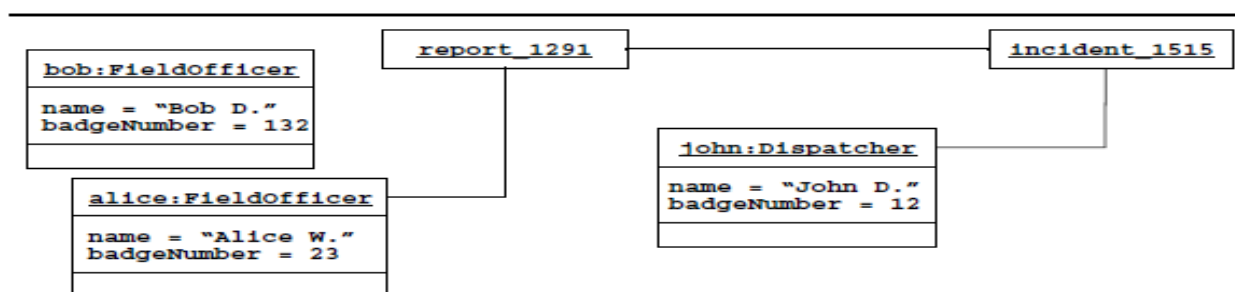
Classes that participate in the ReportEmergency use case. Detailed type information is usually omitted until object design.

Associations and links

A **link** represents a connection between two objects. **Associations** are relationships between classes and represent groups of links. Each FieldOfficer object also has a list of EmergencyReports that has been written by the FieldOfficer. In Figure 2.5, the line between the FieldOfficer class and the EmergencyReport class is an association. In Figure 2.6, the line between the alice:FieldOfficer object and the report_1291:EmergencyReport object is a link. This link represents state that is kept in the system to denote that alice:FieldOfficer generated report_1291:EmergencyReport.

Roles

Each end of an association can be labeled by a string called **role**. In Figure 2.5, the roles of the association between the EmergencyReport and FieldOfficer classes are author and reportsGenerated. Labeling the end of associations with roles allows us to distinguish multiple



associations originating from a class. Moreover, roles clarify the purpose of the association.

Figure 2.6 An example of a UML object diagram: objects that participate in the warehouse On Fire scenario.

Multiplicity

Each end of an association can be labeled by a set of integers indicating the number of links that can legitimately originate from an instance of the class connected to the association end. The association end author has a multiplicity of 1. This means that all EmergencyReports are written

by exactly one FieldOfficer. In other terms, each EmergencyReport object has exactly one link to an object of class FieldOfficer. The multiplicity of the association end reportsGenerated role is “many,” shown as a star. The “many” multiplicity is shorthand for 0..n. This means that any given FieldOfficer may be the author of zero or more EmergencyReports.

Association class

Associations are similar to classes, in that they can have attributes and operations attached to them. Such an association is called an **association class** and is depicted by a class symbol, containing the attributes and operations, connected to the association symbol with a dashed line.

For example, in Figure 2.7, the allocation of FieldOfficers to an Incident is modeled as an association class with attributes role and notificationTime.

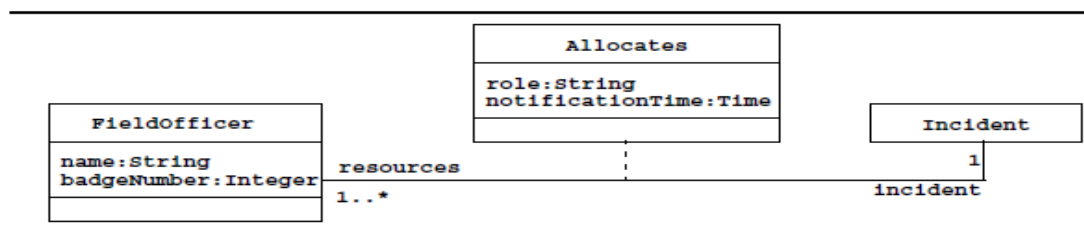


Figure 2.7 An example of an association class (UML class diagram).

Any association class can be transformed into a class and simple associations as shown in Figure 2.8. Although both representations are similar, the association class representation is clearer: An association cannot exist without the classes it links. Similarly, the Allocation object cannot exist without a FieldOfficer and an Incident object. Although Figure 2.8 carries the same information, this diagram requires careful examination of the association multiplicity.

Aggregation

Associations are used to represent a wide range of connections among a set of objects. In practice, a special case of association occurs frequently: composition. For example, a State contains many Counties, which in turn contains many Townships. A PoliceStation is composed of PoliceOfficers. Another example is a Directory that contains a number of Files. Such relationships could be modeled using a one-to-many association. Instead, UML provides the concept of an **aggregation** to denote composition. An aggregation is denoted by a simple line with a diamond at the container end of the association. Although one-to-many associations and aggregations can be used alternatively, aggregations are preferable because they emphasize the hierarchical aspects of the relationship.

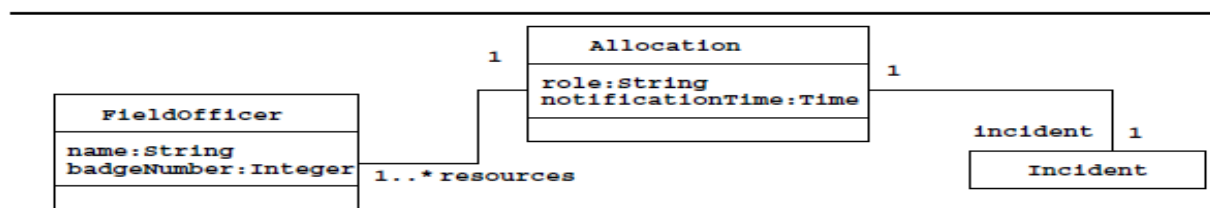
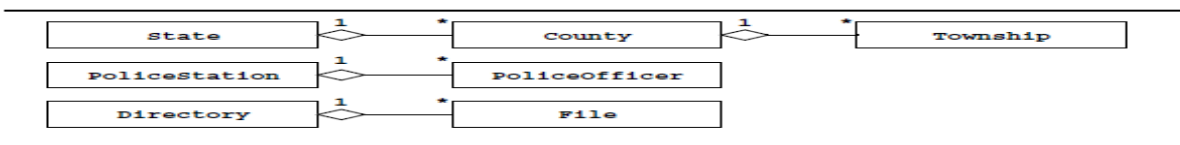


Figure 2.8 Alternative model for Allocation (UML class diagram).**Generalization**

Generalization is the relationship between a general class and one or more specialized classes. Generalization enables us to describe all the attributes and operations that are common to a set of classes. For example, FieldOfficer and Dispatcher both have name and badgeNumber attributes. However, FieldOfficer has an association with EmergencyReport, whereas Dispatcher has an association with Incident. The common attributes of FieldOfficer and Dispatcher can be modeled by introducing a *PoliceOfficer* class that is specialized by the FieldOfficer and the Dispatcher classes (see Figure 2.9). *PoliceOfficer*, the generalization, is called a **superclass**. FieldOfficer and Dispatcher, the specializations, are called the **subclasses**.

**Figure 2.9** An example of a generalization (UML class diagram).

PoliceOfficer is an abstract class which defines the common attributes and operations of the FieldOfficer and Dispatcher classes.

For example, consider a watch with two buttons (hereafter called 2Bwatch). Setting the time on 2Bwatch requires the actor 2BWatchOwner to first press both buttons simultaneously, after which 2Bwatch enters the set time mode. In the set time mode, 2Bwatch blinks the number being changed (e.g., the hours, the minutes, or the seconds, day, month, year). Initially, when the 2BWatchOwner enters the set time mode, the hours blink. If the actor presses the first button, the next number blinks (e.g., if the hours are blinking and the actor presses the first button, the hours stop blinking and the minutes start blinking. If the actor presses the second button, the blinking number is incremented by one unit. If the blinking number reaches the end of its range, it is reset to the beginning of its range (e.g., assume the minutes are blinking and its current value is 59, its new value is set to 0 if the actor presses the second button). The actor exits the set time mode by pressing both buttons simultaneously. Figure 2.10 depicts a sequence diagram for an actor setting his 2Bwatch one minute ahead.

Each column represents an object that participates in the interaction. The vertical axis represents time from top to bottom. Messages are shown by arrows. Labels on arrows represent message names and may contain arguments. Activations (i.e., executing methods) are depicted by vertical rectangles. Actors are shown as the leftmost column.

Sequence diagrams can be used to describe either an abstract sequence (i.e., all possible interactions) or concrete sequences (i.e., one possible interaction, as in Figure 2.10). When describing all possible interactions, sequence diagrams also provide notations for conditionals and iterators.

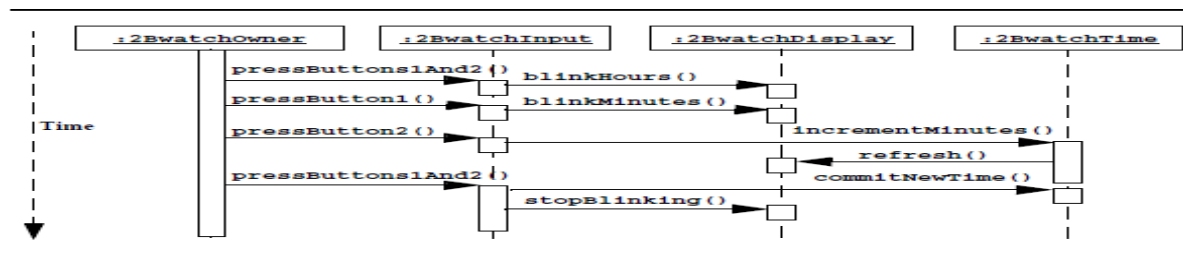


Figure 2.10 Example of a sequence diagram: setting the time on 2Bwatch.

2.3.3. State chart Diagram

A **UML statechart** is a notation for describing the sequence of states an object goes through in response to external events. Statecharts are extensions of the finite state machine model. On the one hand, statecharts provide notation for nesting states and state machines (i.e., a state can be described by a state machine). On the other hand, statecharts provide notation for binding transitions with message sends and conditions on objects.

A **state** is a condition that an object satisfies. A state can be thought of as an abstraction of the attribute values of a class. For example, an Incident object in FRIEND can be in four states: Active, Inactive, Closed, and Archived. An active Incident denotes a situation that requires a response (e.g., an ongoing fire, a traffic accident). An inactive Incident denotes a situation that was handled but for which reports still need to be written (e.g., the fire has been put out but damage estimates have not yet been performed). A closed Incident denotes a situation that has been handled and documented. An archived Incident is a closed Incident whose documentation has been moved to off-site storage.

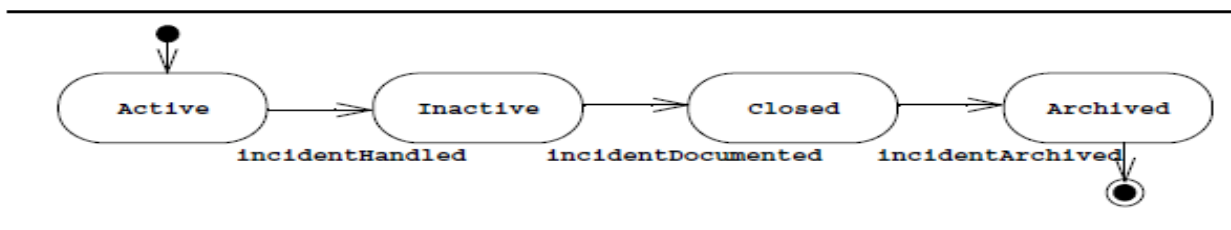


Figure 2.11 A UML statechart diagram for the Incident class.

A **transition** represents changes of state triggered by events, conditions, or time. there are three transitions: from the Active state into the Inactive state, from the Inactive state to the Closed state, and from the Closed state to the Archived state.

A state is depicted by a rounded rectangle. A transition is depicted by arrows connecting two states. States are labeled with their name. A small solid black circle indicates the initial state. A circle surrounding a small solid black circle indicates a final state.

displays another example, a state chart for the 2Bwatch (for which we constructed a sequence diagram). At the highest level of abstraction, 2Bwatch has two states, MeasureTime and SetTime. 2Bwatch changes states when the user presses and releases both buttons simultaneously. When 2Bwatch is first powered on, it is in the SetTime state. This is indicated by the small solid black

circle, which represents the initial state. When the battery of the watch runs out, the 2Bwatch is permanently out of order.

The statechart diagram does not represent the details of measuring or setting the time. These details have been abstracted away from the statechart diagram and can be modeled separately using either internal transitions or a nested statechart. Internal transitions are transitions that remain within a single state. They can also have actions associated with them. Entry and exit are displayed as an internal transition, given that their actions do not depend on the originating and destination states. Nested statecharts reduce complexity. They can be used instead of internal transitions. In Figure 2.20, the current number is modeled as nested state, whereas actions corresponding to modifying the current number are modeled using internal transitions. Note that each state could be modeled as a nested statechart (e.g., the BlinkHours statechart would have 24 substates that correspond to the hours in the day; transitions between these states would correspond to pressing the second button).

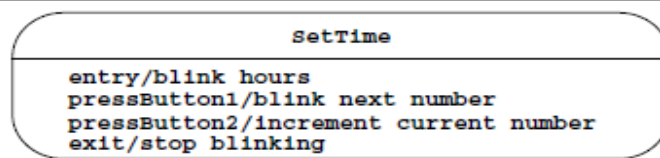


Figure 2.12 Internal transitions associated with the SetTime state (UML statechart diagram).

2.3.4. Activity diagrams

The outgoing transitions are triggered by the completion of an action associated with the state. This is called an **action state**. By convention, the name of a state denotes a condition, whereas the name of an action state denotes an action. **Activity diagrams** are statechart diagrams whose states are action states. is an activity diagram corresponding to the state diagram. An alternate and equivalent view of activity diagrams is to interpret action states as control flow between activities and transitions; that is, the arrows are interpreted as sequential constraints between activities.

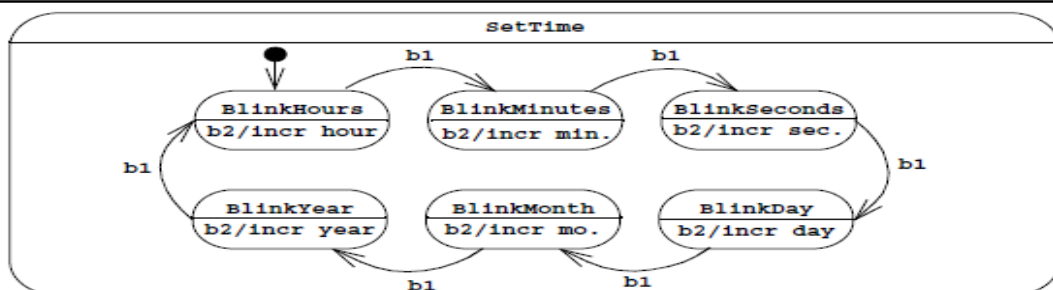


Figure 2.13 Refined statechart associated with the SetTime state (UML statechart diagram).

During the action state `HandleIncident`, the Dispatcher receives reports and allocates resources. Once the Incident is closed, the Incident moves to the `DocumentIncident` activity during which all participating FieldOfficers and Dispatchers document the Incident. Finally, the `ArchiveIncident` activity represents the archival of the Incident related information onto slow access medium.

Decisions are branches in the control flow. They denote alternative transitions based on a condition of the state of an object or a set of objects. Decisions are depicted by a diamond with one or more incoming arrows and two or more outgoing arrows. The outgoing arrows are labeled with the conditions that select a branch in the control flow. The set of all outgoing transitions from a decision represents the set of all possible outcomes.

2.3.5. Diagram organization

Models of complex systems quickly become complex as developers refine them. The complexity of models can be dealt with by grouping related elements into **packages**. A package is a grouping of model elements, such as use cases, classes, or activities, defining scopes of understanding.

For example, Figure 2.14 depicts use cases of the FRIEND system, grouped by actors. Packages are displayed as rectangles with a tab attached to their upper-left corner. Use cases dealing with incident management (e.g., creating, resource allocation, documentation) are grouped in the `IncidentManagement` package. Use cases dealing with incident archive (e.g., archiving an incident, generating reports from archived incidents) are grouped in the `IncidentArchive` package. Use cases dealing with system administration (e.g., adding users, registering end stations) are grouped in the `SysAdministration` package. This enables the client and the developers to organize use cases into related groups and to focus on only a limited set of use cases at a time.

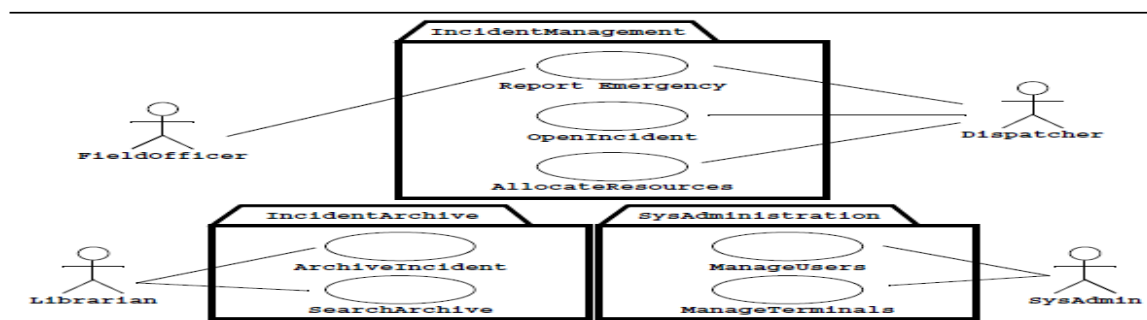
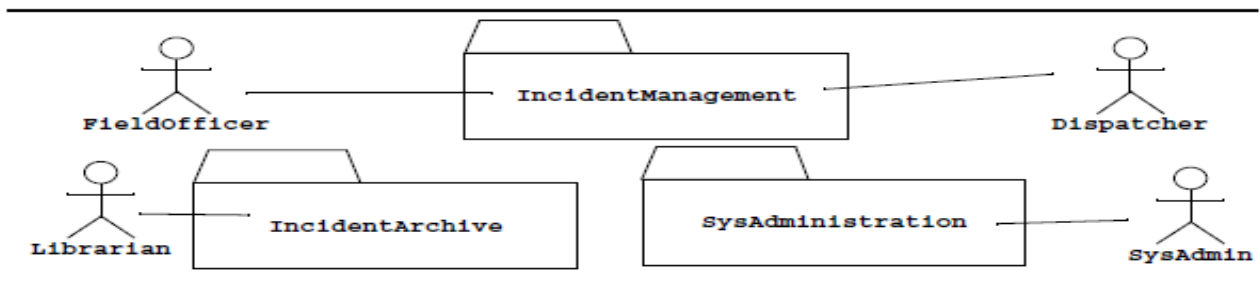


Figure 2.14 Example of packages: use cases of FRIEND organized by actors (UML use case diagram).

Figures 2.14 and 2.15 are examples of class diagrams using packages. Classes from the ReportEmergency use case are organized according to the site where objects are created. FieldOfficer and EmergencyReport are part of the FieldStation package, and Dispatcher and Incident are part of the DispatcherStation. Figure 2.14 displays the packages with the model elements they contain while Figure 2.15 displays the same information without the contents of each package. Figure 2.15 is a higher level picture of the system and can be used for discussing system-level issues, whereas Figure 2.14 is a more detailed view, which can be used to discuss the content of specific packages.

Figure 2.15 Example of packages:



This figure displays the same packages as Figure 2-39 except that the details of each packages are suppressed (UML use case diagram).

Packages (Figure 2.16) are used to deal with complexity the same way a user organizes files and subdirectories into directories. However, packages are not necessarily hierarchical: The same class may appear in more than one package. To reduce inconsistencies, classes (more generally model elements) are owned by exactly one package, whereas the other packages are said to refer to the modeling element. Note that packages are organizing constructs, not objects.

They have no behavior associated with them and cannot send and receive messages. A **note** is a comment attached to a diagram. Notes are used by developers for attaching information to models and model elements. This is an ideal mechanism for recording outstanding issues relevant to a model, clarifying a complex point, or recording to-dos or reminders. Although notes have no semantics per se, they are sometimes used to express constraints that cannot otherwise be expressed in UML.

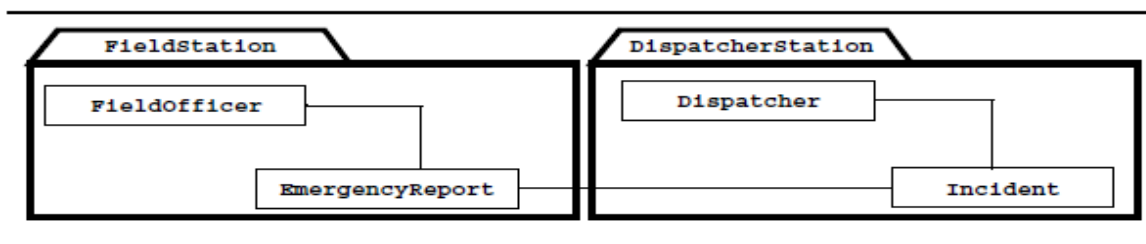


Figure 2.16 Example of packages.

2.3.6. Diagram Extensions

The goal of the UML designers is to provide a set of notations to model a broad class of software systems. They also recognized that a fixed set of notations could not achieve this goal, because it is impossible to anticipate the needs encountered in all application and solution domains. For this reason, UML provides a number of extension mechanisms enabling the modeler to extend the language. In this section, we describe two such mechanisms, **stereotypes** and **constraints**.

A **stereotype** is a string enclosed by angle brackets (e.g., <<subsystem>>), which is attached to a UML element, such as a class or an association. This enables modelers to create new kinds of building blocks that are needed in their domain. For example, during analysis, we classify objects into three types: entity, boundary, and control. The base UML language knows only objects. To introduce these three additional types, we use three stereotypes, <<entity>>, <<boundary>>, and <<control>> to represent the object type (Figure 2.17).

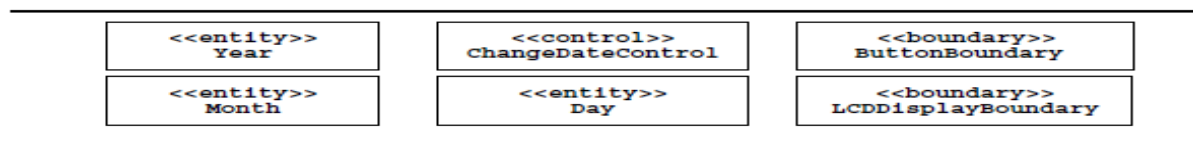


Figure 2.17 Examples of stereotypes (UML class diagram).

A **constraint** is a rule that is attached to a UML building block. This allows to represent phenomena that cannot otherwise be expressed with UML.

Summary

- The Unified Modeling Language (UML) is a standard graphical language for modeling object-oriented software. The Unified Modeling Language (UML) is a language for specifying, constructing, visualizing, and documenting the software system and its components. Object Constraint language (OCL) is a specification language that uses simple logic for specifying the properties of a system.
- Unified Modeling Language has things, diagrams, and relationships as its building blocks.
- **Structural things** define the static part of the model. **Behavioral things** consist the dynamic model of the UML models.
- **Annotational things** can be defined as a mechanism to capture remarks, descriptions, and comments of UML model elements.
- **Grouping things** can be defined as a mechanism to group elements of a UML model together.
- Generalization can be defined as a relationship which connects a specialized element with a generalized element.
- Static and Dynamic diagrams are the two types of diagrams in UML. Actors are external entities that interact with the system.

- Use Cases describe the behavior of the system as seen from an actor's point of view. Scenario is an instance of a use case describing concrete set of actions.
- **Sequence diagrams** describe patterns of communication among a set of interacting objects. A **UML statechart** is a notation for describing the sequence of states an object goes through in response to external events.
- State charts are extensions of the finite state machine model. A **state** is a condition that an object satisfies. A **transition** represents changes of state triggered by events, conditions, or time. **Activity diagrams** are statechart diagrams whose states are action states. A **stereotype** is a string enclosed by angle brackets (e.g., <<subsystem>>), which is attached to a UML element, such as a class or an association. A **constraint** is a rule that is attached to a UML building block.

Review Questions & Problems

1. Which of the following are the primary goals in the design of the UML?
 - A. Encourage the growth of the OO tools market
 - B. Support higher-level development concepts
 - C. Integrate best practice and methodologies
 - D. All
2. _____ Can be defined as a relationship which connects as a specialized element with a generalized element.
 - A. Generalization.
 - B. Inheritance
 - C. Polymorphism
 - D. A & B
3. Which of the following is not among the type of the structural things?
 - A. Class
 - B. Interface
 - C. Collaboration
 - D. Interaction
4. _____ is a relationship between two things in which changes in one element also effects the other one.
 - A. Association
 - B. Aggregation
 - C. Dependency
 - D. Realization
5. _____ Can be defined as a relationship in which two elements are connected. One element describes the function and other one implements them.
 - A. Realization
 - B. Interfaces
 - C. Generalization
 - D. A & B
6. What kind of association exists between a student and Identification card?
 - A. One to One
 - B. One to Many
 - C. Many to Many
 - D. None
7. Which type to diagram is among the UML Static Diagram?
 - A. Class Diagram
 - B. Sequence Diagram
 - C. Activity Diagram
 - D. State Chart Diagram
8. In State chart diagram, _____ represents changes of state triggered by events, conditions, or time.

References:

- The Unified Modeling Language User Guide SECOND EDITION Addison Wesley Professional USA