ICT - 4231
OBJECT ORIENTED SOFTWARE ENGINEERING

LECTURE - 2

CHAPTER – 2 : MODELING WITH UML

Moinul Islam, IIT, JU

CONTENTS

INTRODUCTION

“ *Every mechanic is familiar with the problem of the part you can’t buy because you can’t find it because the manufacturer considers it a part of something else.* ”

-----Robert Pirsig, in Zen and the Art of Motorcycle Maintenance

INTRODUCTION (CONT.)

- UML is a notation that resulted from the unification of OMT (Object Modeling Technique), Booch, and OOSE (Object-Oriented Software Engineering).
- UML has also been influenced by other object-oriented notations, such as those introduced by Mellor and Shlaer, Coad and Yourdon, Wirfs-Brock, and Martin and Odell, 1992].
- The goal of UML is to provide a standard notation that can be used by all object-oriented methods and to select and integrate the best elements of precursor notations.
- For example, UML includes the use case diagrams introduced by OOSE and uses many features of the OMT class diagrams. UML also includes new concepts that were not present in other major methods at the time, such as extension mechanisms and a constraint language.

INTRODUCTION (CONT.)

- UML has been designed for a broad range of applications. Hence, it provides constructs for a broad range of systems and activities (e.g., distributed systems, analysis, system design, deployment). System development focuses on three different models of the system:
  1. **Functional Model**: Depicted with use case diagrams, it outlines system functionality from the user's perspective.
  2. **Object Model**: Represented by class diagrams, it describes the system structure, evolving from analysis to system design and further to object design models.
  3. **Dynamic Model**: Portrayed through interaction diagrams, state machine diagrams, and activity diagrams, it elucidates the internal system behavior. Interaction diagrams show object message sequences, state machine diagrams depict individual object states and transitions, and activity diagrams illustrate control and data flows.

AN OVERVIEW OF UML

- we briefly introduce five UML notations:

  1. Use Case Diagrams
  2. Class Diagrams
  3. Interaction Diagrams
  4. State Machine Diagrams
  5. Activity Diagrams

USE CASE DIAGRAMS

- Use cases are used during requirements elicitation and analysis to represent the functionality of the system and focus on the behavior of the system from an external point of view.
- They describe system functions that produce visible outcomes for actors, entities interacting with the system (e.g., users, other systems).
- Identifying actors and use cases helps define the system's boundary, distinguishing tasks handled by the system from those by its environment.
- Actors exist outside this boundary, while use cases reside within, emphasizing the roles and tasks unique to the system.

USE CASE DIAGRAMS(CONT.)

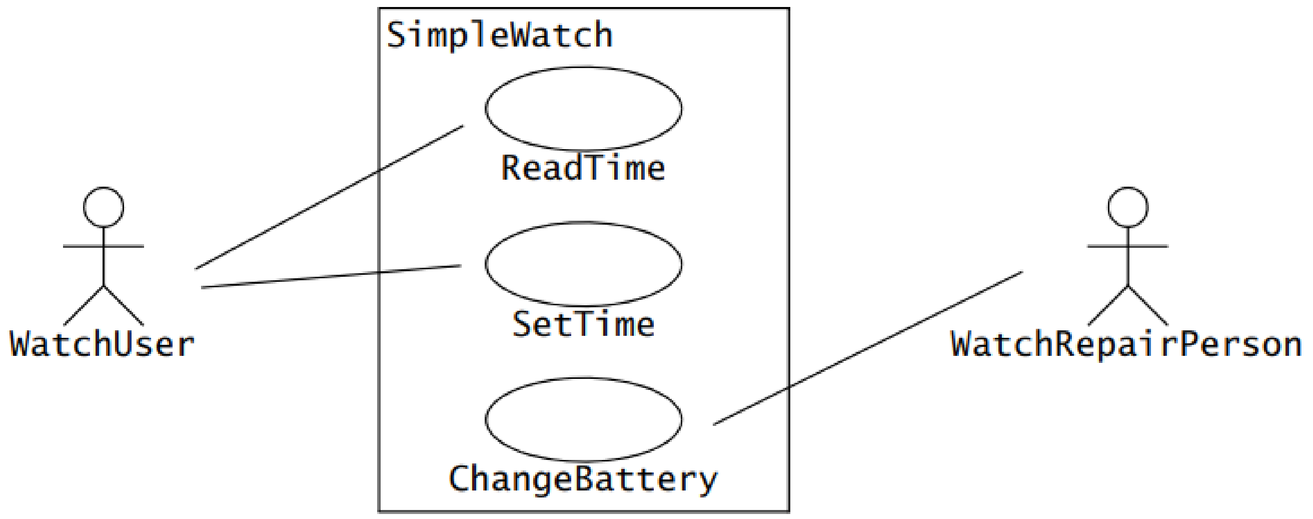


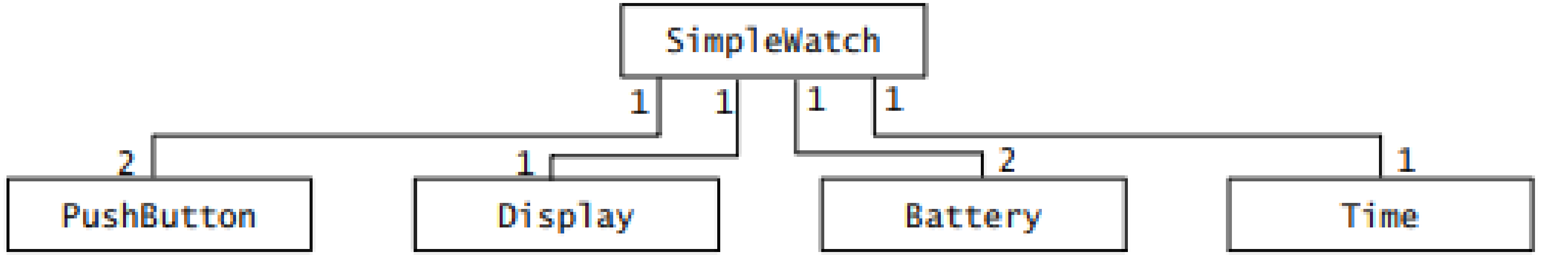
Figure 2-1 A UML use case diagram describing the functionality of a simple watch. The `WatchUser` actor may either consult the time on her watch (with the `ReadTime` use case) or set the time (with the `SetTime` use case). However, only the `WatchRepairPerson` actor can change the battery of the watch (with the `ChangeBattery` use case). Actors are represented with stick figures, use cases with ovals, and the boundary of the system with a box enclosing the use cases.

CLASS DIAGRAMS

- Class diagrams are used to describe the structure of the system.
- Classes are abstractions that specify the common structure and behavior of a set of objects.
- Objects are instances of classes that are created, modified, and destroyed during the execution of the system.
- An object has state that includes the values of its attributes and its links with other objects.
- Class diagrams describe the system in terms of objects, classes, attributes, operations, and their associations.

CLASS DIAGRAMS(CONT.)

- For example, Figure 2-2 is a class diagram describing the elements of all the watches of the SimpleWatch class. These watch objects all have an association to an object of the PushButton class, an object of the Display class, an object of the Time class, and an object of the Battery class. The numbers on the ends of associations denote the number of links each SimpleWatch object can have with an object of a given class. For example, a SimpleWatch has exactly two PushButtons, one Display, two Batteries, and one Time. Similarly, all PushButton, Display, Time, and Battery objects are associated with exactly one SimpleWatch object.

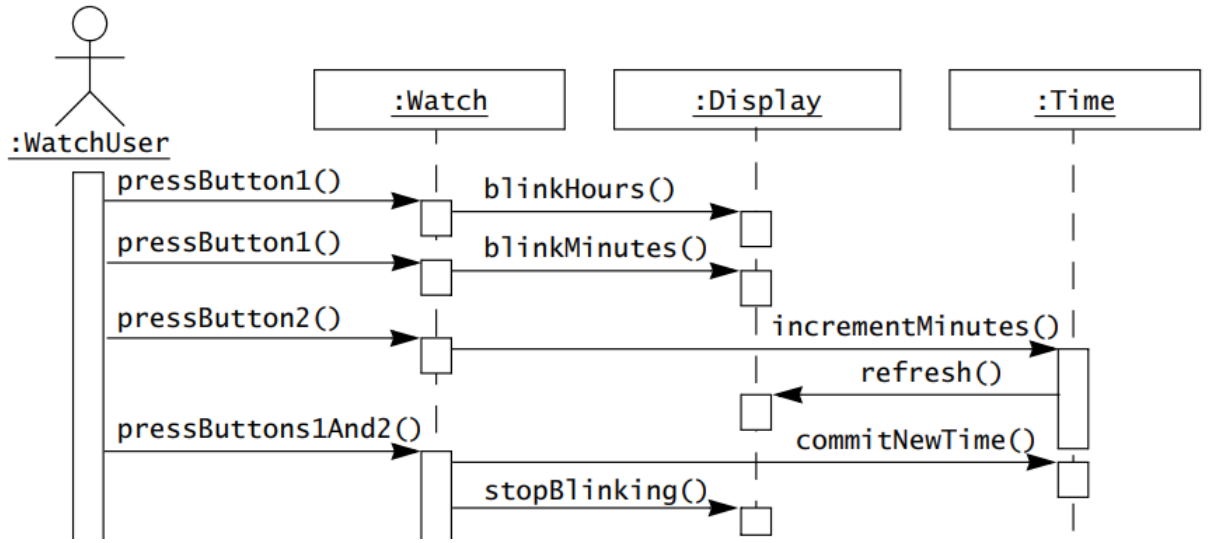


INTERACTION DIAGRAMS

- Interaction diagrams are used to formalize the dynamic behavior of the system and to visualize the communication among objects. They are useful for identifying additional objects that participate in the use cases. We call objects involved in a use case participating objects.
- An interaction diagram represents the interactions that take place among these objects. For example, Figure 2-3 is a special form of interaction diagram, called a sequence diagram.

INTERACTION DIAGRAMS(CONT.)

- **Sequence diagram** for the SetTime use case of our simple watch. The left-most column represents the WatchUser actor who initiates the use case. Labeled arrows represent stimuli that an actor or an object sends to other objects. In this case, the WatchUser presses button 1 twice and button 2 once to set her watch a minute ahead. The SetTime use case terminates when the WatchUser presses both buttons simultaneously.

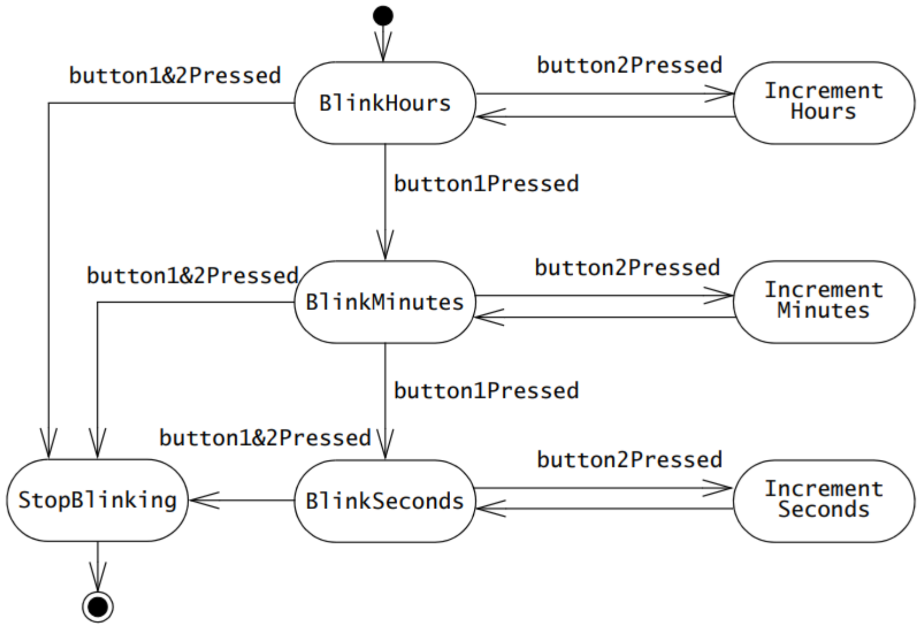


STATE MACHINE DIAGRAMS

- State machine diagrams describe the dynamic behavior of an individual object as a number of states and transitions between these states.
- A state represents a particular set of values for an object.
- Given a state, a transition represents a future state the object can move to and the conditions associated with the change of state.
- The sequence diagram focuses on the messages exchanged between objects as a result of external events created by actors. The state machine diagram focuses on the transitions between states as a result of external events for an individual object.

STATE MACHINE DIAGRAMS(CONT.)

- For example, Figure 2-4 is a state machine diagram for the Watch. A small black circle initiates that BlinkHours is the initial state. A circle surrounding a small black circle indicates that StopBlinking is a final state.

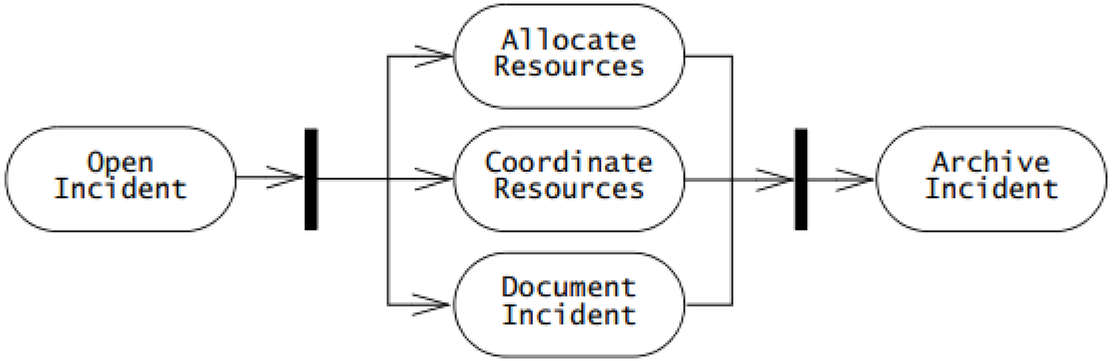


ACTIVITY DIAGRAMS

- An activity diagram describes the behavior of a system in terms of activities.
- Activities are modeling elements that represent the execution of a set of operations.
- The execution of an activity can be triggered by the completion of other activities, by the availability of objects, or by external events.
- Activity diagrams are similar to flowchart diagrams in that they can be used to represent control flow (i.e., the order in which operations occur) and data flow (i.e., the objects that are exchanged among operations).

ACTIVITY DIAGRAMS(CONT.)

- For example, Figure 2-5 is an activity diagram representing activities related to managing an Incident. Rounded rectangles represent activities; arrows between activities represent control flow; thick bars represent the synchronization of the control flow.
- The activity diagram of Figure 2-5 depicts that the AllocateResources, CoordinateResources, and DocumentIncident can be initiated only after the OpenIncident activity has been completed. Similarly, the ArchiveIncident activity can be initiated only after the completion of AllocateResources, Coordinate–Resources, and DocumentIncident. These latter three activities, however, can occur concurrently.



MODELING CONCEPTS

In this section, we describe the basic concepts of modeling.

- We define the terms system, model, and view, and discuss the purpose of modeling.
- Describe relationship to programming languages and terms such as data types, classes, instances, and objects.
- Describe how object-oriented modeling focuses on building an abstraction of the system environment as a basis for the system model.

SYSTEMS, MODELS, AND VIEWS

- **System:** A system is an organized set of communicating parts, particularly engineered systems designed for specific purposes.
  - Examples: Cars, watches, and payroll systems are cited as instances of engineered systems, each composed of interconnected subsystems.
- **Model:** Modeling is employed to address the complexity of systems, providing a means to construct abstractions that focus on relevant aspects while omitting unnecessary details.
  - Significance: Models, simpler representations of systems, help in a divide-and-conquer approach. The modeling process starts with a basic model, gradually refining it during development, and is crucial in fields like software engineering.
- **View:** Views are subsets of models designed to enhance understanding. They focus on specific aspects or levels of accuracy within a model.
  - Example: In the context of constructing an airplane, blueprints for the entire aircraft constitute a model, while a view might specifically address the fuel system, allowing for a more focused and understandable representation. Views can overlap, such as an electrical wiring view that includes wiring for both the fuel system and other components.

SYSTEMS, MODELS, AND VIEWS (CONT.)

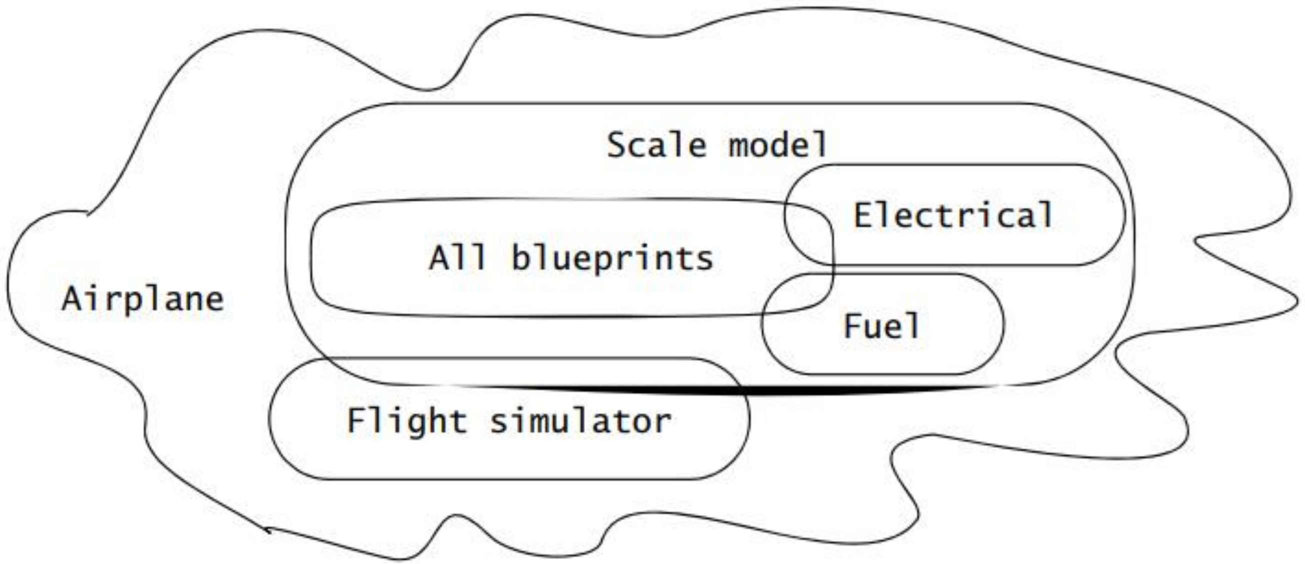


Figure 2-6 A model is an abstraction describing a subset of a system. A view depicts selected aspects of a model. Views and models of a single system may overlap each other.

NOTATIONS

- **Notations:** Notations are graphical or textual rules for representing views. A UML class diagram is a graphical view of the object model.
  In wiring diagrams, each connected line represents a different wire or bundle of wires. In UML class diagrams, a rectangle with a title represents a class. A line between two rectangles represents a relationship between the two corresponding classes. Note that different notations can be used to represent the same view (Figure 2-7).

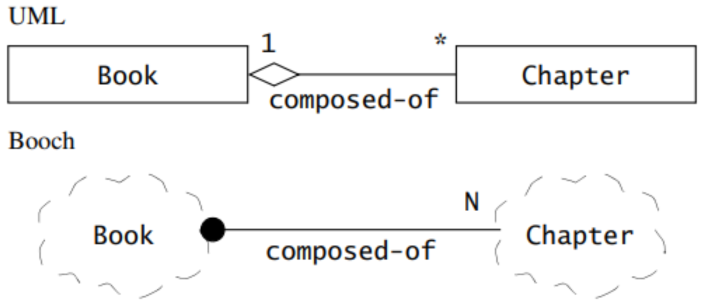


Figure 2-7 Example of describing a model with two different notations. The model includes two classes, Book and Chapter, with the relationship, Book is composed of Chapters. In UML, classes are depicted by rectangles and aggregation associations by a line terminated with a diamond. In the Booch notation, classes are depicted by clouds, and aggregation associations are depicted with a line terminated with a solid circle.

DATA TYPES, ABSTRACT DATA TYPES, AND INSTANCES

- **Data Types:**

  - Definition: In the context of programming languages, a data type is an abstraction with a unique name, representing a set of values (instances) along with defined operations.

  - Example: In Java, the data type "int" includes all signed integers between –232 and 232 – 1, with valid operations being arithmetic operations and functions/methods that use "int" as a parameter.

- **Abstract Data Types (ADTs):**

  - Definition: An abstract data type is a data type defined by an implementation-independent specification. Abstract data types enable developers to reason about a set of instances without looking at a specific implementation of the abstract data type.

  - Significance: ADTs, such as sets and sequences, can have various implementations optimizing different criteria. Developers interact with ADTs based on semantics, without needing to understand the internal representations.

DATA TYPES, ABSTRACT DATA TYPES, AND INSTANCES (CONT.)

- **Instances:**
  - Definition: Instances refer to individual values belonging to a specific data type or abstract data type, conforming to the defined characteristics and operations.
  - Example: For an abstract data type "Person," instances involve specific individuals, and operations like `getName()` and `getSocialSecurityNumber()` can be performed. The internal storage details, such as representing the social security number as a number or a string, are implementation decisions hidden from the rest of the system.

CLASSES, ABSTRACT CLASSES, AND OBJECTS

- **Classes:** Classes are abstractions in object-oriented modeling and programming languages that encapsulate both structure and behavior.
  - ➔ Unlike abstract data types, classes can be defined in terms of other classes using inheritance, where a subclass refines a superclass by adding new attributes and operations.
  - ➔ A class defines operations that can be applied to its instances and attributes that apply to all instances, each with a unique name and type.

- **Abstract Classes:** Abstract classes result from an inheritance relationship aimed at modeling shared attributes and operations. They are not meant to be instantiated and often represent generalized concepts in the application domain.
  - ➔ In chemistry, "OrganicCompound" is an abstract class, and in Java, "Collection" is an abstract class serving as a generalization for collection classes like LinkedList or ArrayList.

CLASSES, ABSTRACT CLASSES, AND OBJECTS (CONT.)

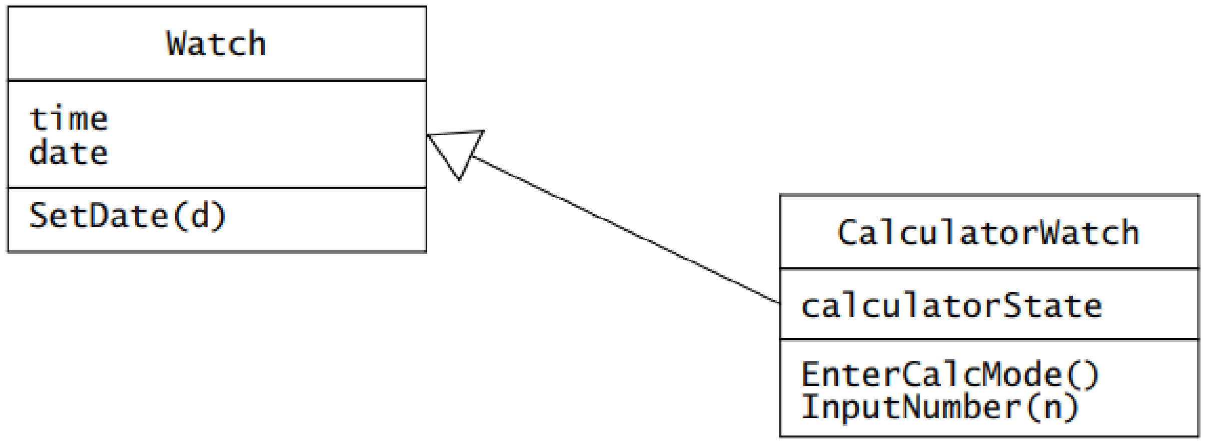


Figure 2-8 A UML class diagram depicting two classes, `Watch` and `CalculatorWatch`. `CalculatorWatch` is a refinement of `Watch`, providing calculator functionality not found in normal watches. In a UML class diagram, classes and objects are represented as boxes with three compartments: the first compartment depicts the name of the class, the second depicts its attributes, the third its operations. The second and third compartments can be omitted for brevity. An inheritance relationship is displayed by a line with a triangle. The triangle points to the superclass, and the other end is attached to the subclass.

CLASSES, ABSTRACT CLASSES, AND OBJECTS (CONT.)

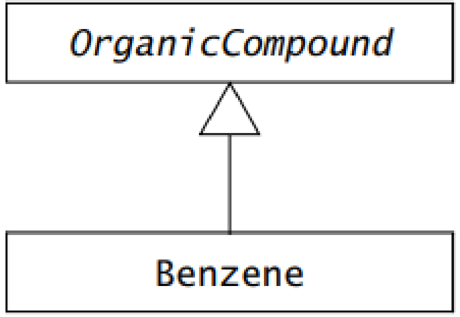


Figure 2-9 An example of abstract class (UML class diagram). *OrganicCompound* is never instantiated and only serves as a generalization class. The names of abstract classes are italicized.

CLASSES, ABSTRACT CLASSES, AND OBJECTS (CONT.)

- **Objects:** Objects are instances of classes, possessing identity and storing attribute values. Each object belongs to a specific class.
  - ➔ In UML, instances are depicted by rectangles with underlined names to distinguish them from classes. Objects have visibility of their attributes, and their attributes may be visible to other parts of the system in some programming languages like Java.

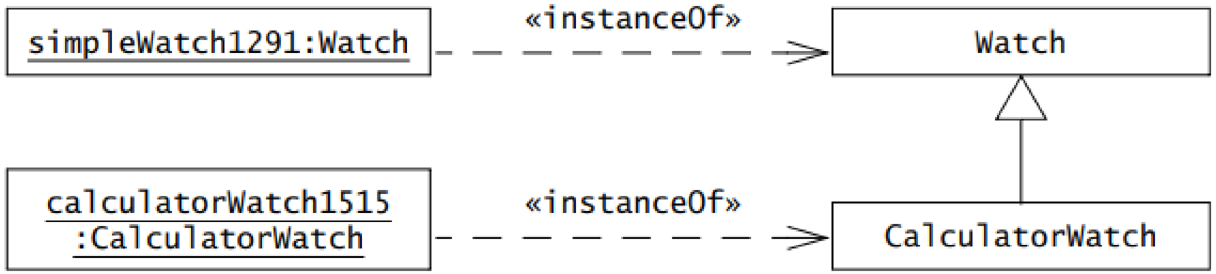


Figure 2-10 A UML class diagram depicting instances of two classes. simpleWatch1291 is an instance of Watch. calculatorWatch1515 is an instance of CalculatorWatch. Although the operations of Watch are also applicable to calculatorWatch1515, the latter is not an instance of the former.

EVENT CLASSES, EVENTS, AND MESSAGES

- **Event Classes:** Event classes serve as abstractions representing a specific kind of event, indicating a shared, common response from the system.
  - Example: An event class might be "ButtonPressEvent," encapsulating instances where a button is pressed, and the system responds uniformly to such occurrences.

- **Events:** Events are instances of event classes, representing actual occurrences in the system that trigger a response.
  - Example: An event instance could be "WatchUser presses the left button," signaling a tangible and relevant happening within the system.

- **Messages:** Messages are a means of communication between objects in a system, initiated by sending objects to request the execution of an operation in receiving objects. They consist of a name and a set of arguments.
  - Example: In a system, a message could be the "getTime()" message sent by the :Watch object to the :Time object, resulting in the retrieval of the current time. The message includes the name ("getTime") and any required arguments.

EVENT CLASSES, EVENTS, AND MESSAGES (CONT.)

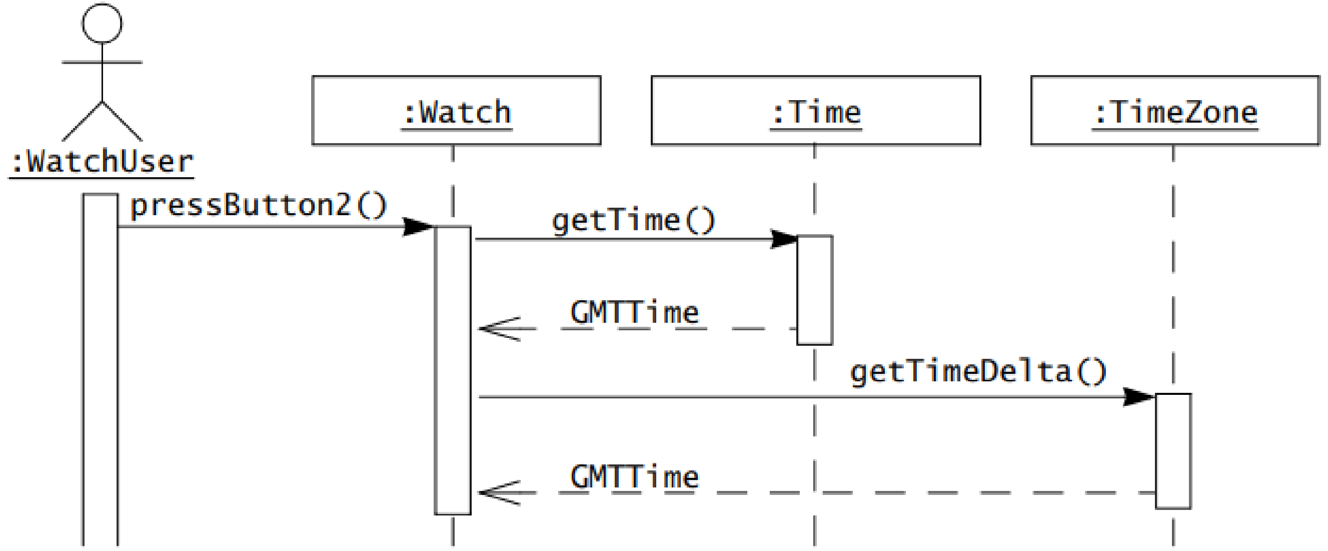
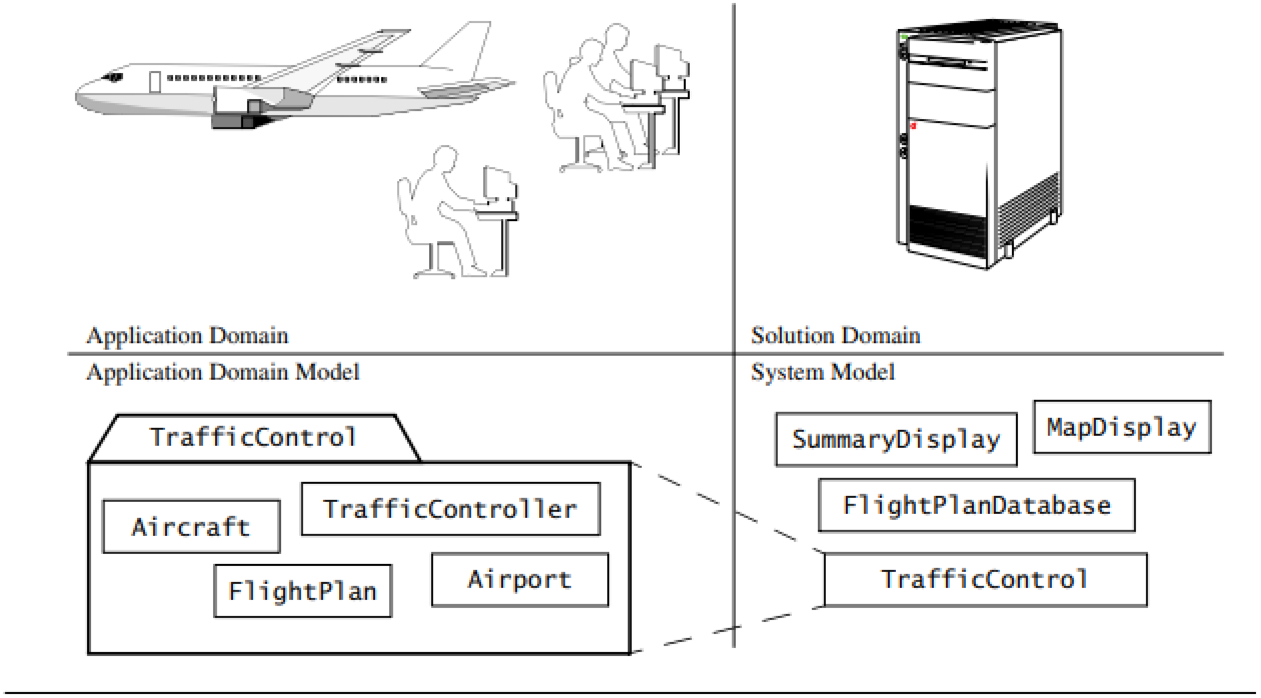


Figure 2-11 Examples of message sends (UML sequence diagram). The `Watch` object sends the `getTime()` message to a `Time` object to query the current Greenwich time. It then sends the `getTimeDelta()` message to a `TimeZone` object to query the difference to add to the Greenwich time. The dashed arrows represent the replies (i.e., message results that are sent back to the sender).

OBJECT-ORIENTED MODELING

- The **application domain** represents all aspects of the user's problem. This includes the physical environment, the users and other people, their work processes, and so on.
- The **solution domain** is the modeling space of all possible systems. Modeling in the solution domain represents the system design and object design activities of the development process. The solution domain model is much richer and more volatile than the application domain model.
- **Object-oriented analysis** is concerned with modeling the application domain. **Object-oriented design** is concerned with modeling the solution domain. Both modeling activities use the same representations (i.e., classes and objects).
- In object-oriented analysis and design, the application domain model is also part of the system model.
- For example, an air traffic control system has a TrafficController class to represent individual users, their preferences, and log information. The system also has an Aircraft class to represent information associated with the tracked aircraft. TrafficController and Aircraft are application domain concepts that are encoded into the system

OBJECT-ORIENTED MODELING (CONT.)



FALSIFICATION AND PROTOTYPING

- **Falsification:** Falsification is the process of demonstrating that a model does not accurately represent relevant details or has incorrectly represented them, indicating a mismatch with reality.
  - Example: In scientific research, the orbit of the planet Mercury falsified Newton's theory of gravity in favor of Einstein's general theory of relativity, which better matched observational data.

- **Prototyping:** Prototyping is a software development technique where developers construct a prototype, often focusing on aspects like the user interface, which is then presented to users for evaluation and modification.
  - Example: In software development, a user interface prototype is created, presented to potential users, and modified based on their feedback. This iterative process allows for the falsification of the initial prototype and refinement of the model representing the future system.

A DEEPER VIEW INTO UML

We now describe in detail the five main UML diagrams :

- **Use case diagrams** represent the functionality of the system from a user's point of view. They define the boundaries of the system.
- **Class diagrams** represent the static structure of a system in terms of objects, their attributes, operations, and relationships.
- **Interaction diagrams** represent the system's behavior in terms of interactions among a set of objects. They are used to identify objects in the application and implementation domains .
- **State machine diagrams** represent the behavior of nontrivial objects.
- **Activity diagrams** are flow diagrams used to represent the data flow or the control flow through a system.

UML: USE CASE DIAGRAMS

Actors: Actors are external entities that interact with the system. Actors have unique names and descriptions.

- ➔ 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).

Use cases: Use cases describe the behavior of the system as seen from an actor's point of view. Behavior described by use cases 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 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**.

UML: USE CASE DIAGRAMS

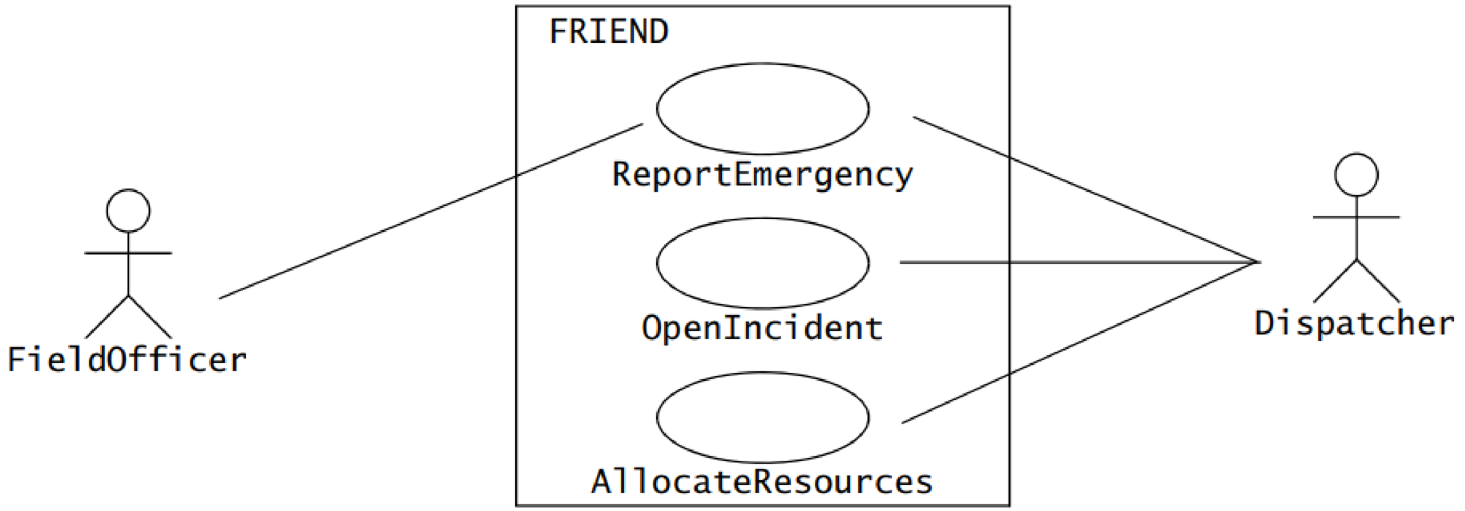


Figure 2-13 An example of a UML use case diagram for First Responder Interactive Emergency Navigational Database (FRIEND), an accident management system. Associations between actors and use cases denote information flows. These associations are bidirectional: they can represent the actor initiating a use case (FieldOfficer initiates ReportEmergency) or a use case providing information to an actor (ReportEmergency notifies Dispatcher). The box around the use cases represents the system boundary.

Use Case Diagrams (cont.)

Relationships

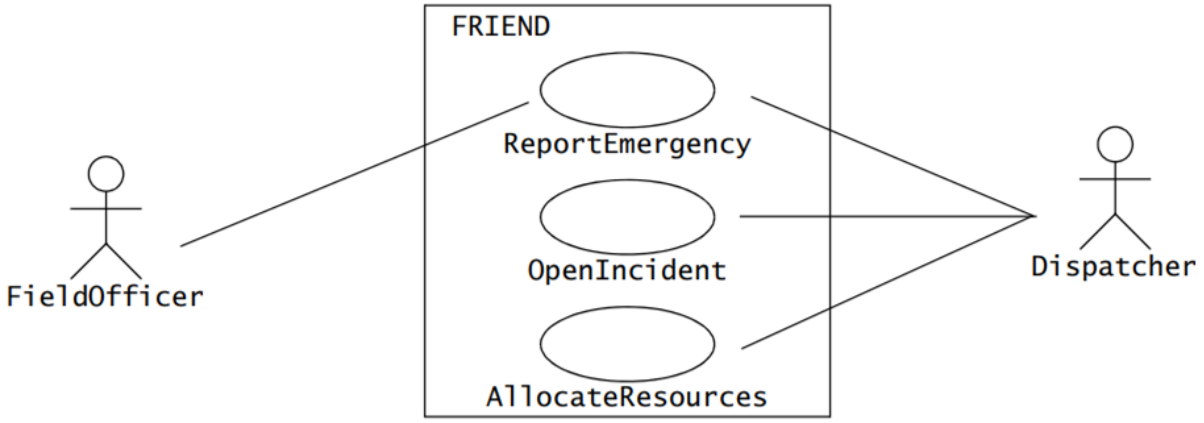
Use case diagrams can include four types of relationships:

1. communication,
2. inclusion,
3. extension, and
4. inheritance.

Use Case Diagrams (cont.)

Communication Relationship

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 the Figure, 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 functionality.



Use Case Diagrams (cont.)

Include Relationship

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 a street map. This can be modeled by a use case ViewMap 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 ViewMap functionality once, thus reducing complexity of the overall use case model.

➔ Two use cases are related by an include relationship if one of them includes the second one in its flow of events. In use case diagrams, include relationships are depicted by a dashed open arrow originating from the including use case (see Figure 2-15). Include relationships are labeled with the string «include»

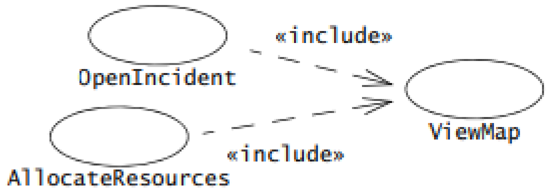


Figure 2-15 An example of an «include» relationship (UML use case diagram).

Use Case Diagrams (cont.)

Extend Relationship

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 2-17), 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.

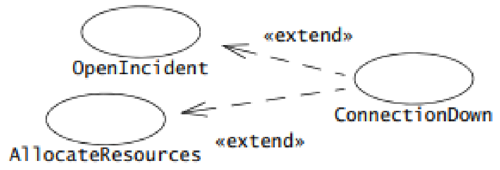


Figure 2-17 An example of an «extend» relationship (UML use case diagram).

Use Case Diagrams (Cont.)

Inheritance Relationship

- An inheritance relationship is a third mechanism for reducing the complexity of a model. One use case can specialize another more general one by adding more detail.

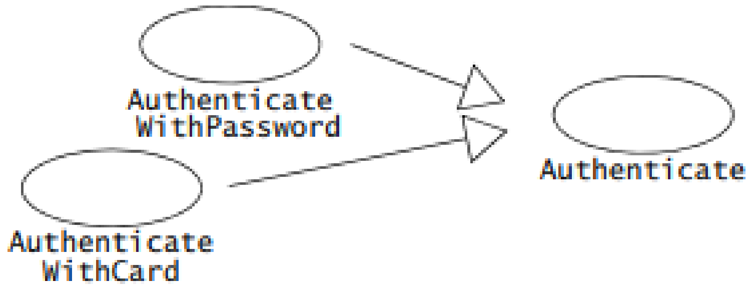


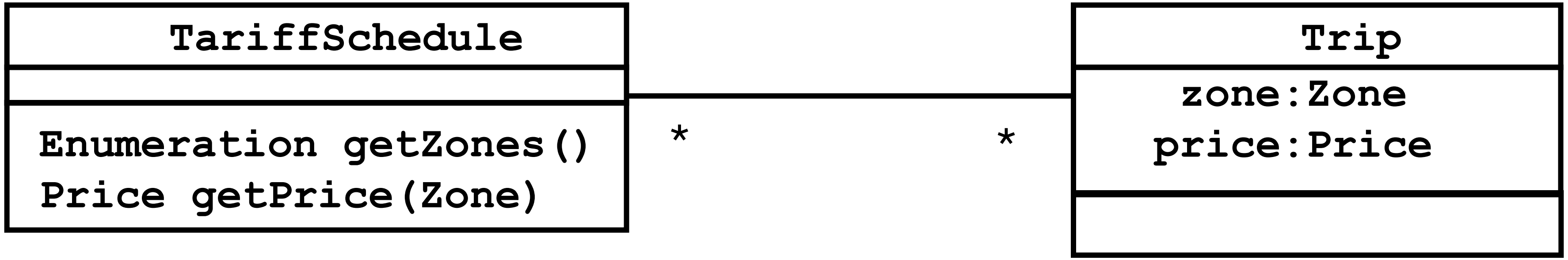
Figure 2-19 An example of an inheritance relationship (UML use case diagram). The Authenticate use case is a high-level use case describing, in general terms, the process of authentication. AuthenticateWithPassword and AuthenticateWithCard are two specializations of Authenticate.

Use Case Diagrams

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. We describe a scenario using a template with three fields:
  1. 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.
  2. The **participating actor instances** field indicates which actor instances are involved in this scenario. Actor instances also have underlined names.
  3. The **flow of events** of a scenario describes the sequence of events step by step.

CLASS DIAGRAMS



- Class diagrams represent the structure of the system.
- Class diagrams are used
  - **during requirements analysis to model problem domain concepts**
  - **during system design to model subsystems and interfaces**
  - **during object design to model classes.**

CLASS DIAGRAMS (CONT.)

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. A class is a collection of objects that share a set of attributes that distinguish the objects as members of the collection.
- **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 composed of 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 compartments can be omitted for clarity. Object names are underlined to indicate that they are instances.

CLASS DIAGRAMS (CONT.)

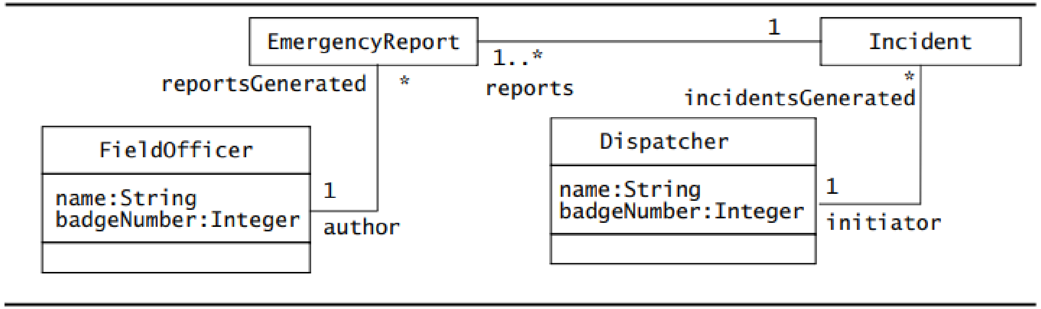


Figure 2-22 An example of a UML class diagram: classes that participate in the ReportEmergency use case. Detailed type information is usually omitted until object design (see Chapter 9, *Object Design: Specifying Interfaces*).

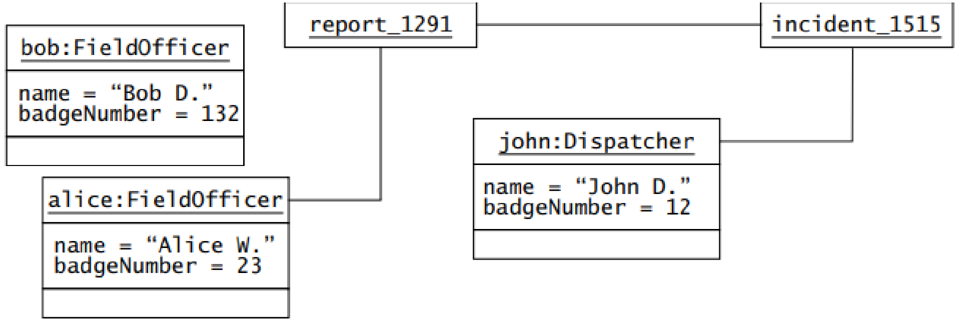
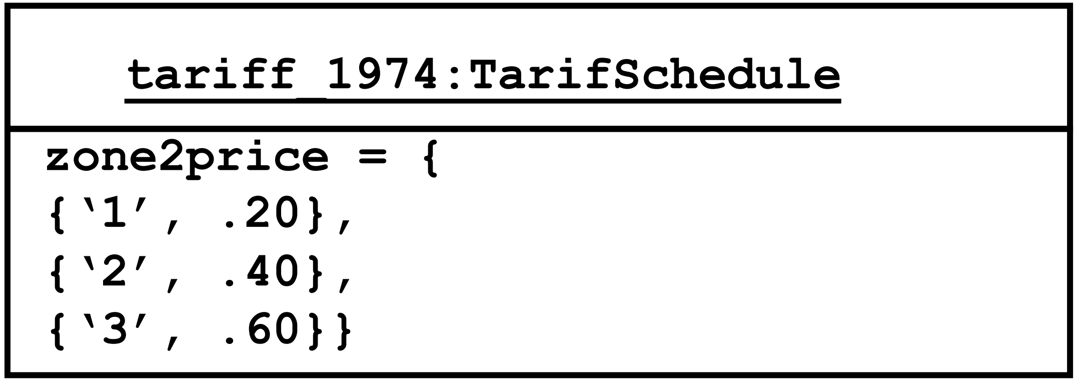


Figure 2-23 An example of a UML object diagram: objects that participate in warehouseOnFire.

CLASS DIAGRAMS (CONT.)

INSTANCES



- An ***instance*** represents a phenomenon.
- The name of an instance is underlined and can contain the class of the instance.
- The attributes are represented with their ***values***.

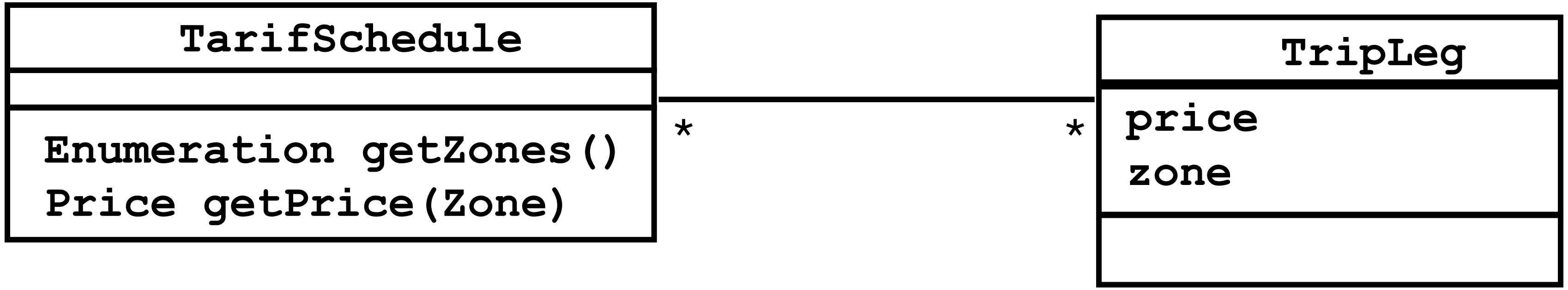
CLASS DIAGRAMS (CONT.)

ACTOR VS. INSTANCES

- What is the difference between an actor and a class and an instance?
- Actor:
  - **An entity outside the system to be modeled, interacting with the system ("Pilot")**
- Class:
  - **An abstraction modeling an entity in the problem domain, inside the system to be modeled ("Cockpit")**
- Object:
  - **A specific instance of a class ("Joe, the inspector").**

CLASS DIAGRAMS (CONT.)

ASSOCIATIONS AND LINKS



- Associations denote relationships between classes.
- The multiplicity of an association end denotes how many objects the source object can legitimately reference.
- A link represents a connection between two objects.

CLASS DIAGRAMS (CONT.)

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. This set of integers is called the multiplicity of the association end.
- Most of the associations we encounter belong to one of the following three types:
  1) A **one-to-one association** has a multiplicity 1 on each end. A one-to-one association between two classes.
  2) A **one-to-many association** has a multiplicity 1 on one end and 0..n (also represented by a star) or 1..n on the other.
  3) A **many-to-many association** has a multiplicity 0..n or 1..n on both ends. A many-to-many association between two classes. This is the most complex type of association.

CLASS DIAGRAMS (CONT.) *MULTIPLICITY*

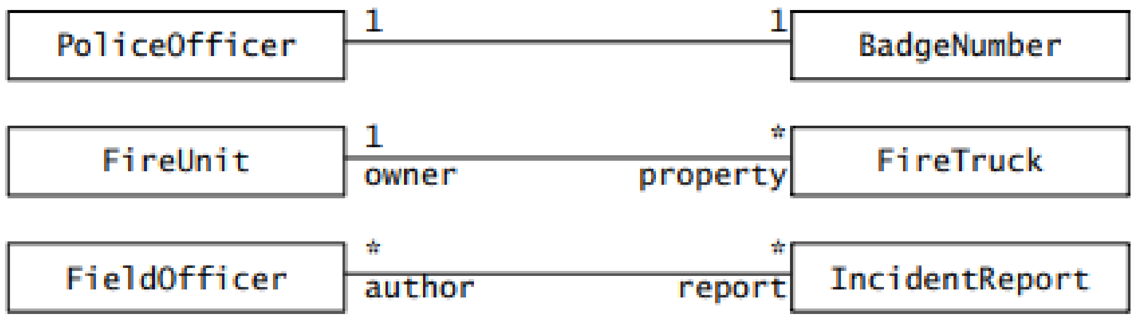


Figure 2-27 Examples of multiplicity (UML class diagram). The association between PoliceOfficer and BadgeNumber is one-to-one. The association between FireUnit and FireTruck is one-to-many. The association between FieldOfficer and IncidentReport is many-to-many.

CLASS DIAGRAMS (CONT.)

QUALIFICATION

- Qualification is a technique for reducing multiplicity by using keys. Associations with a 0..1 or 1 multiplicity are easier to understand than associations with a 0..n or 1..n multiplicity.
- We reduce the multiplicity on the File side by using the filename attribute as a key, also called a qualifier. The relationship between Directory and File is called a qualified association. Reducing multiplicity is always preferable, as the model becomes clearer and fewer cases have to be taken into account.

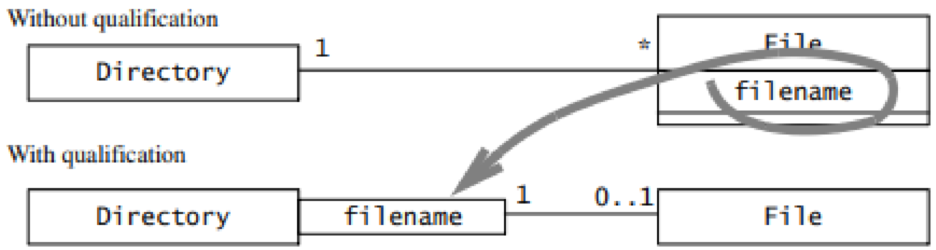


Figure 2-31 Example of how a qualified association reduces multiplicity (UML class diagram). Adding a qualifier clarifies the class diagram and increases the conveyed information. In this case, the model including the qualification denotes that the name of a file is unique within a directory.

CLASS DIAGRAMS (CONT.)

INHERITANCE

- Inheritance is the relationship between a general class and one or more specialized classes. Inheritance enables us to describe all the attributes and operations that are common to a set of classes.

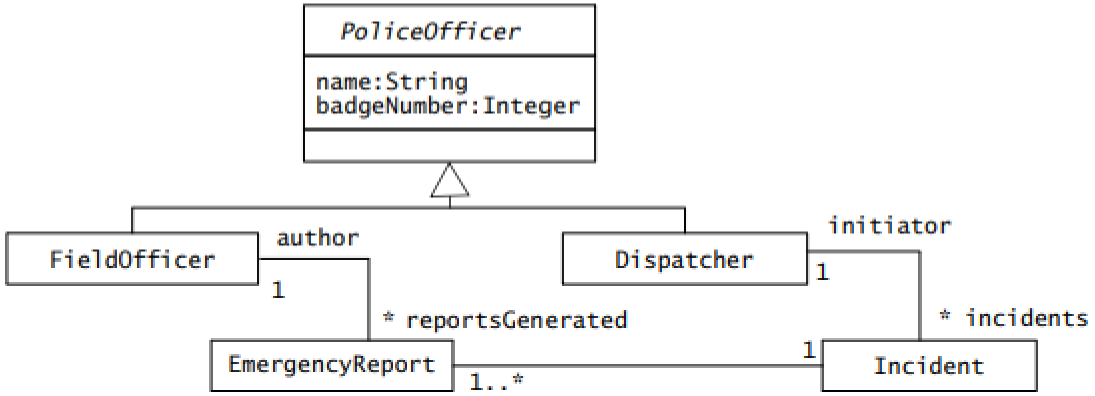


Figure 2-32 An example of an inheritance (UML class diagram). *PoliceOfficer* is an abstract class which defines the common attributes and operations of the FieldOfficer and Dispatcher classes.

INTERACTION DIAGRAMS

- **Interaction diagrams** describe patterns of communication among a set of interacting objects.
- An object interacts with another object by sending messages. The reception of a message by an object triggers the execution of a method, which in turn may send messages to other objects. Arguments may be passed along with a message and are bound to the parameters of the executing method in the receiving object.
- In UML, interaction diagrams can take one of two forms: **sequence diagrams** or **communication diagrams**.
  - Sequence diagrams represent the objects participating in the interaction horizontally and time vertically.
  - Communication diagrams depict the same information as sequence diagrams. Communication diagrams represent the sequence of messages by numbering the interactions.

INTERACTION DIAGRAMS (CONT.)

SEQUENCE DIAGRAM

- **Sequence diagrams represent the objects participating in the interaction horizontally and time vertically.**

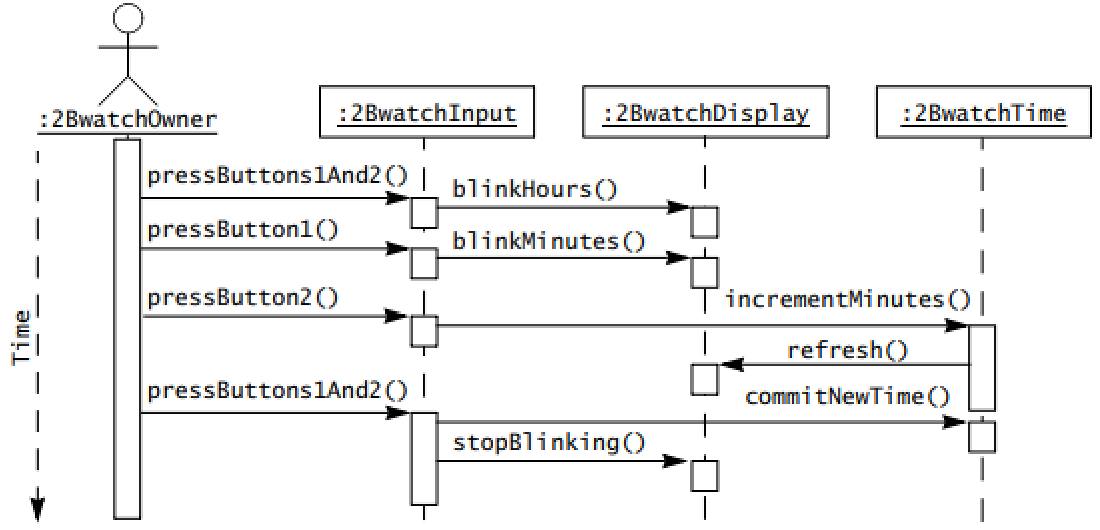


Figure 2-34 Example of a sequence diagram: setting the time on 2Bwatch.

INTERACTION DIAGRAMS (CONT.)

COMMUNICATION DIAGRAM

- Communication diagrams depict the same information as sequence diagrams. Communication diagrams represent the sequence of messages by numbering the interactions.

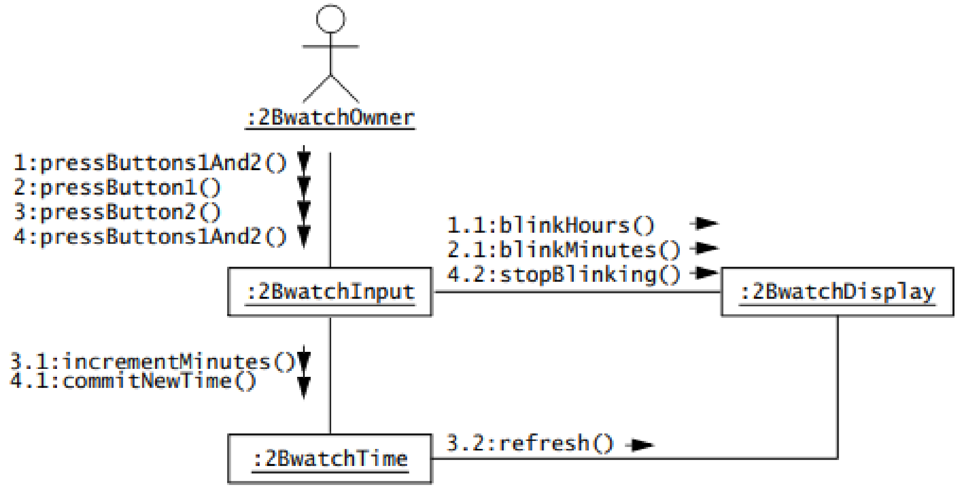


Figure 2-36 Example of a communication diagram: setting the time on 2Bwatch. This diagram represents the same use case as the sequence diagram of Figure 2-34.

STATE MACHINE DIAGRAMS

- UML **state machines** extend the finite state machine model, allowing for nested states and transitions with message sends and conditions.
- Based on Harel's statecharts, UML state machines represent object states in response to external events.
- **States** are conditions satisfied by object attributes, and transitions depict state changes triggered by events, conditions, or time.

- For instance, an Incident object in FRIEND can be in Active, Inactive, Closed, or Archived states. The states are represented by a status attribute with values: Active, Inactive, Closed, and Archived.
- Transitions, such as Active to Inactive, Inactive to Closed, and Closed to Archived, are depicted by open arrows between states. Initial states are marked with a small black circle, and a circle around it denotes a final state.

STATE MACHINE DIAGRAMS (CONT.)

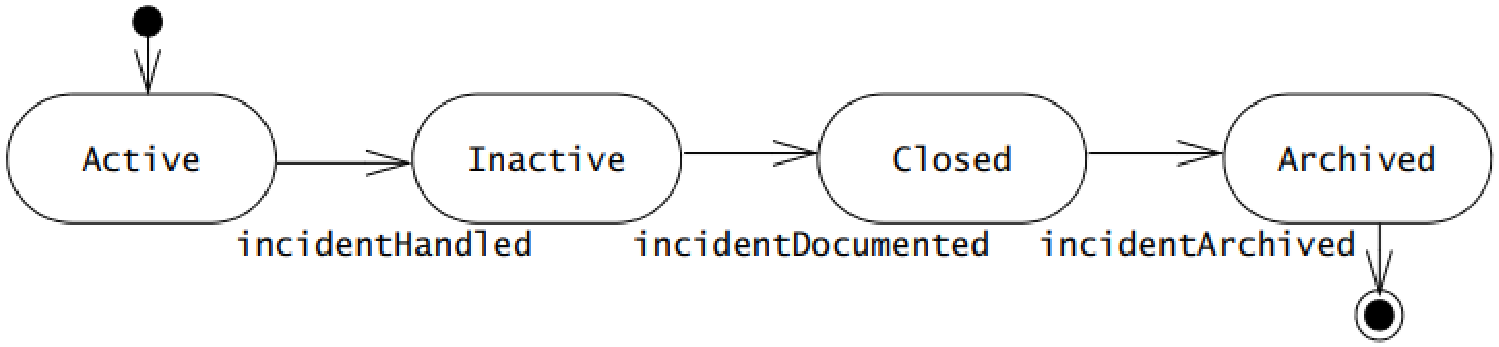


Figure 2-37 A UML state machine diagram for the `Incident` class.

STATE MACHINE DIAGRAMS (CONT.)

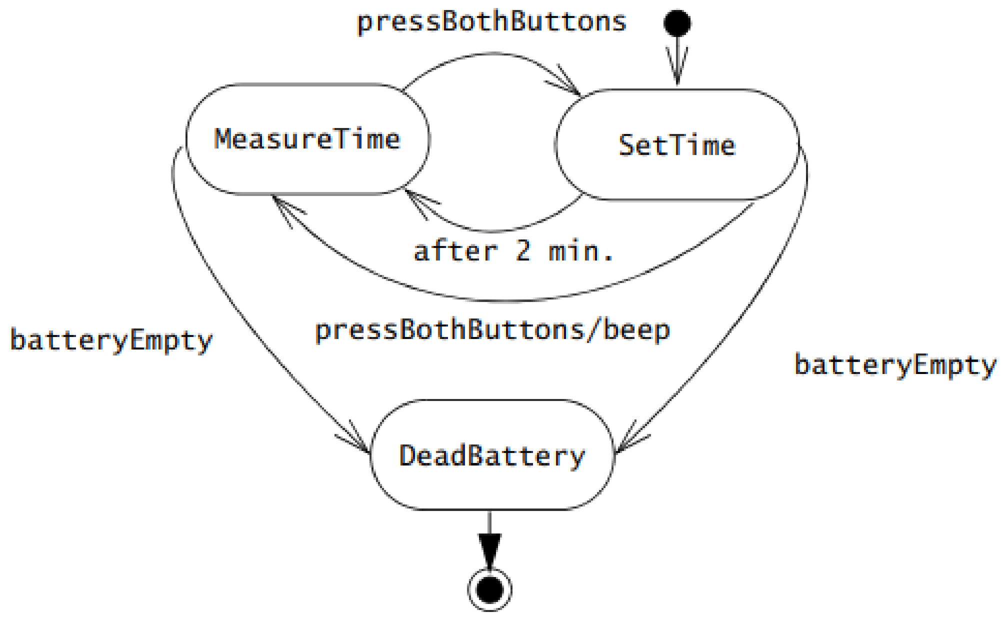


Figure 2-38 State machine diagram for 2Bwatch set time function.

STATE MACHINE DIAGRAMS (CONT.)

- An **internal transition** is a transition that does not leave the state. Internal transitions are triggered by events and can have actions associated with them. However, the firing of an internal transition does not result in the execution of any exit or entry actions.

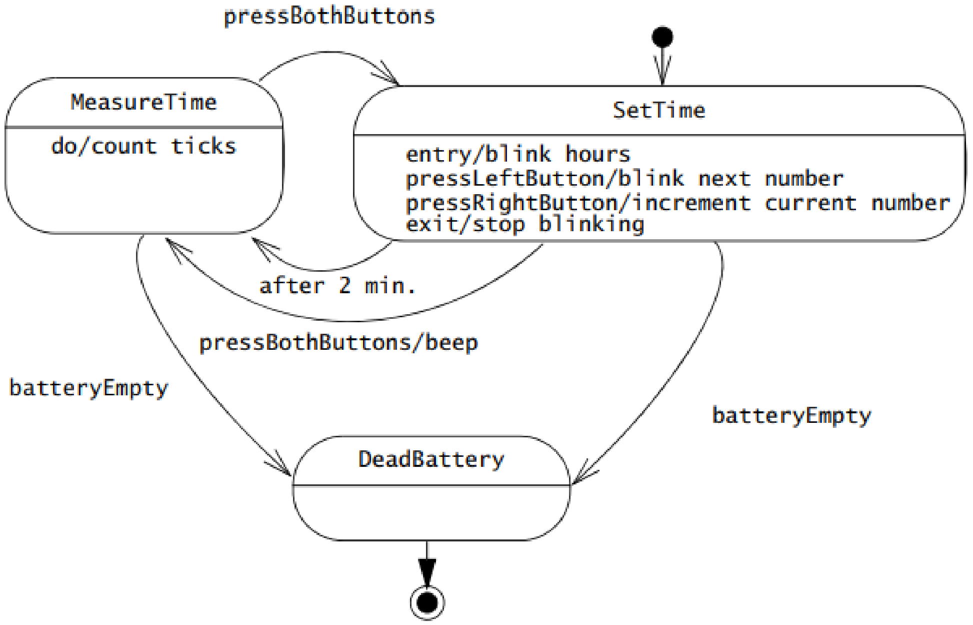


Figure 2-39 Internal transitions associated with the SetTime state (UML state machine diagram).

STATE MACHINE DIAGRAMS (CONT.)

- **Nested state machines** reduce complexity. They can be used instead of internal transitions. In Figure 2-40, the current number is modeled as a nested state, whereas actions corresponding to modifying the current number are modeled using internal transitions.

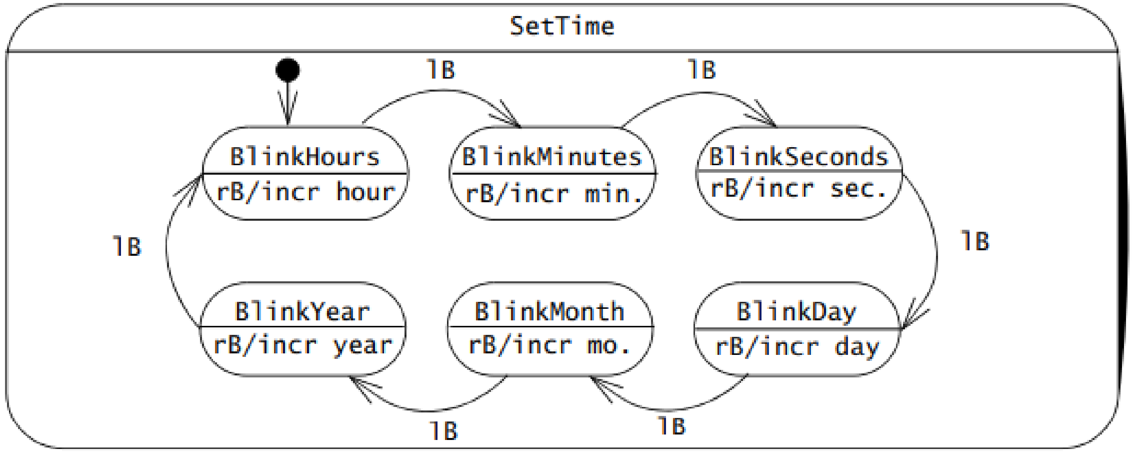


Figure 2-40 Refined state machine associated with the SetTime state (UML state machine diagram). lB and rB correspond to pressing the left and right button, respectively.

ACTIVITY DIAGRAMS

- UML **activity diagrams** represent the sequencing and coordination of lower level behaviors. An activity diagram denotes how a behavior is realized in terms of one or several sequences of activities and the object flows needed for coordinating the activities. Activity diagrams are hierarchical: **an activity** is made out of either an action or a graph of subactivities and their associated object flow.

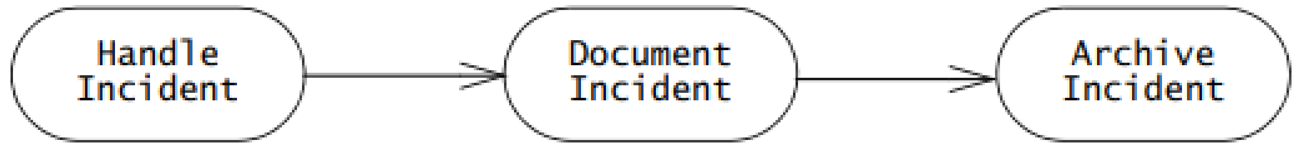


Figure 2-41 A UML activity diagram for Incident. During the action HandleIncident, the Dispatcher receives reports and allocates resources. Once the Incident is closed, the Incident moves to the DocumentIncident activity during which all participating FieldOfficers and Dispatchers document the Incident. Finally, the ArchiveIncident activity represents the archival of the Incident related information onto slow access medium.

ACTIVITY DIAGRAMS (CONT.)

- **Control nodes** coordinate control flows in an activity diagram, providing mechanisms for representing decisions, concurrency, and synchronization.
- The main control nodes we use are
  - Decisions,
  - fork nodes, and
  - join nodes.

ACTIVITY DIAGRAMS (CONT.)

- **Decisions** are branches in the control flow. They denote alternatives 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 open arrows and two or more outgoing arrows.

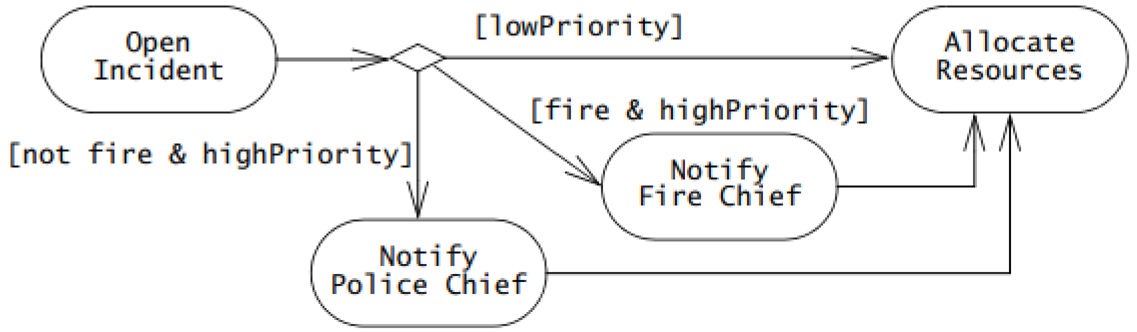


Figure 2-42 Example of decision in the OpenIncident process. If the Incident is a fire and is of high priority, the Dispatcher notifies the FireChief. If it is a high-priority Incident that is not a fire, the PoliceChief is notified. In all cases, the Dispatcher allocates resources to deal with the Incident.

ACTIVITY DIAGRAMS (CONT.)

- **Fork nodes** and **join nodes** represent concurrency. Fork nodes denote the splitting of the flow of control into multiple threads, while join nodes denotes the synchronization of multiple threads and their merging of the flow of control into a single thread.

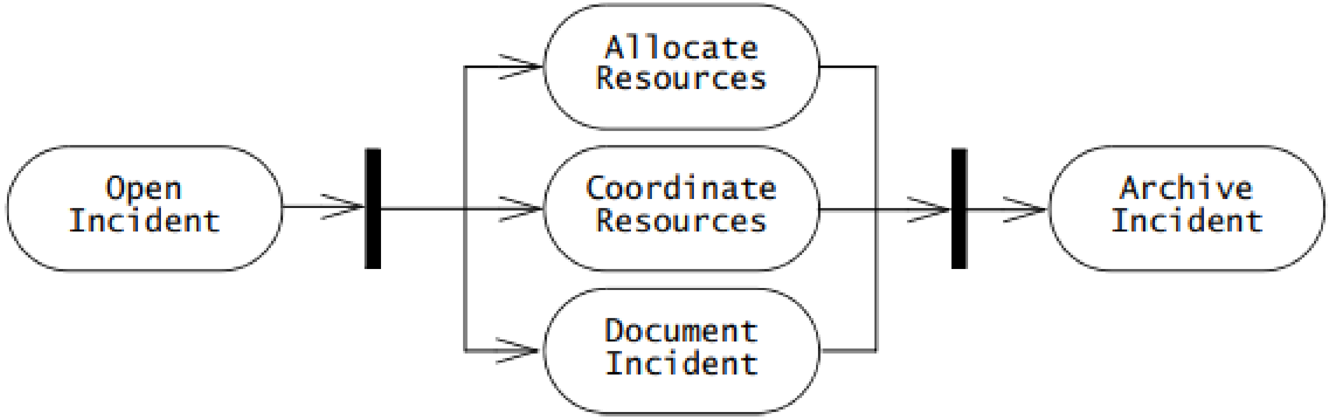


Figure 2-43 An example of fork and join nodes in a UML activity diagram.

ACTIVITY DIAGRAMS (CONT.)

- Activities may be grouped into **swimlanes** (also called activity partitions) to denote the object or subsystem that implements the actions. Swimlanes are represented as rectangles enclosing a group of actions. Transitions may cross swimlanes.

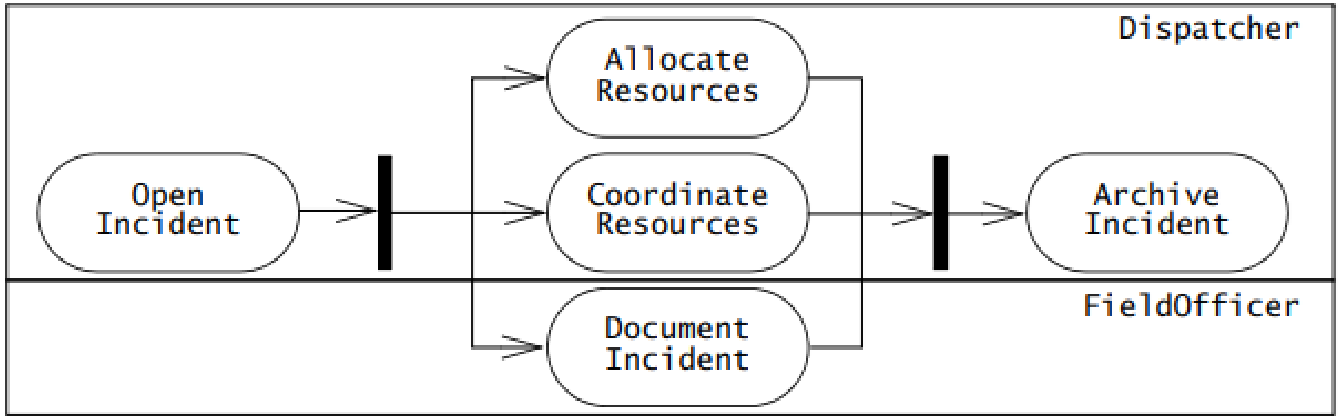


Figure 2-44 An example of swimlanes in a UML activity diagram.

APPLYING ACTIVITY DIAGRAMS

- Activity diagrams provide a task-centric view of the behavior of a set of objects.
- They can be used: for example, to describe sequencing constraints among use cases, sequential activities among a group of objects, or the tasks of a project.

DIAGRAM ORGANIZATION

- 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 or classes, defining scopes of understanding.

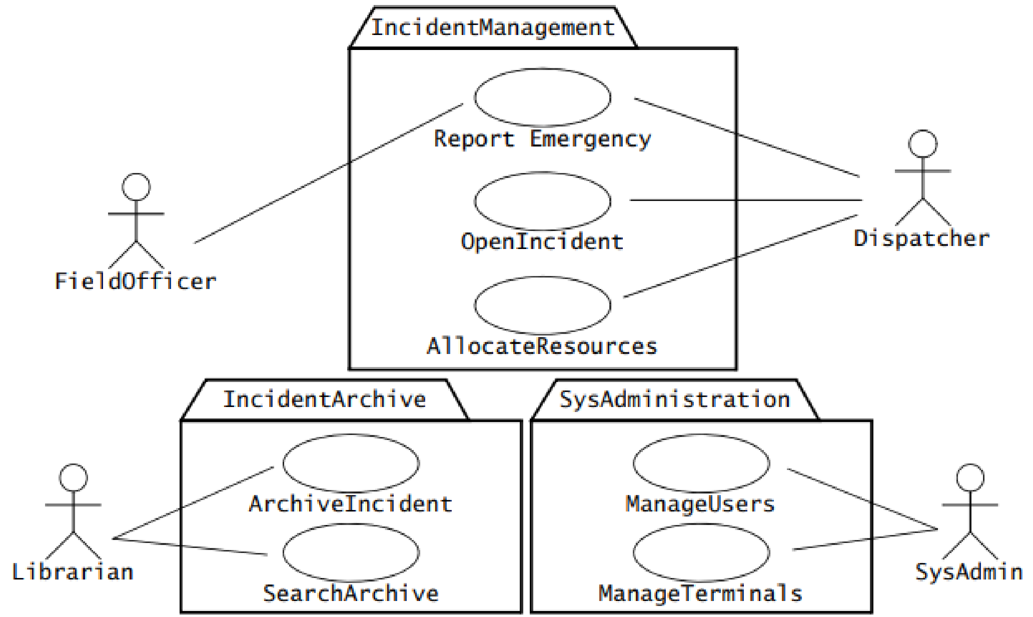


Figure 2-45 Example of packages: use cases of FRIEND organized by actors (UML use case diagram).

DIAGRAM ORGANIZATION (CONT.)

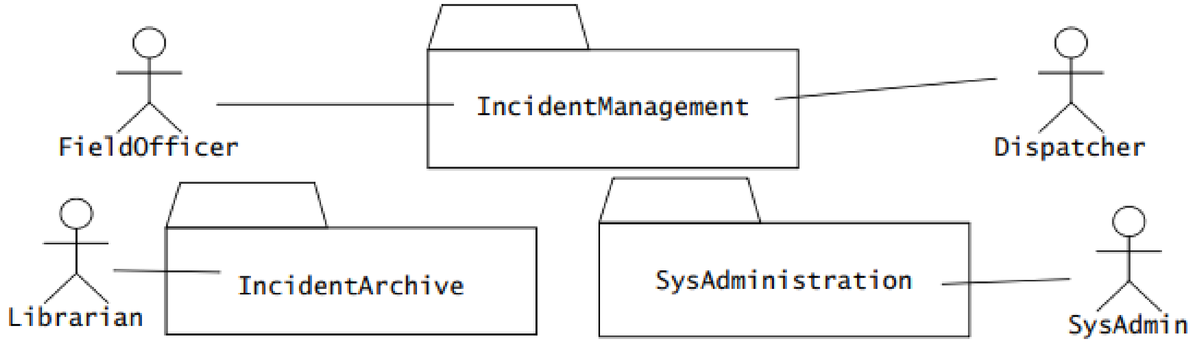


Figure 2-46 Example of packages. This figure displays the same packages as Figure 2-45 except that the details of each packages are suppressed (UML use case diagram).

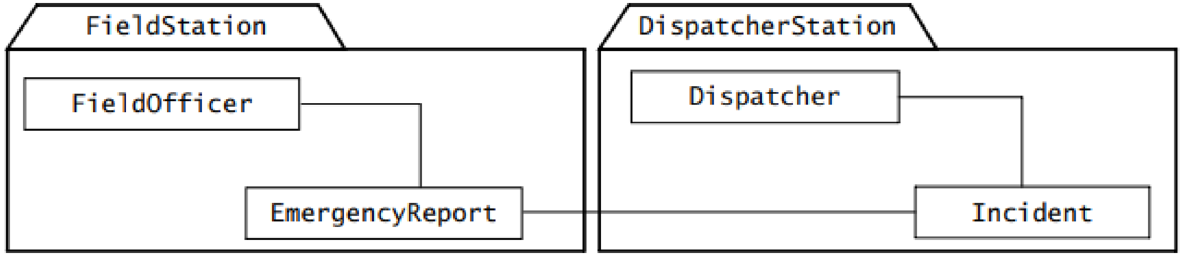


Figure 2-47 Example of packages. The FieldOfficer and EmergencyReport classes are located in the FieldStation package, and the Dispatcher and Incident classes are located on the DispatcherStation package.

DIAGRAM ORGANIZATION (CONT.)

- A note is a comment attached to a diagram. Notes are used by developers for attaching information to models and model elements.

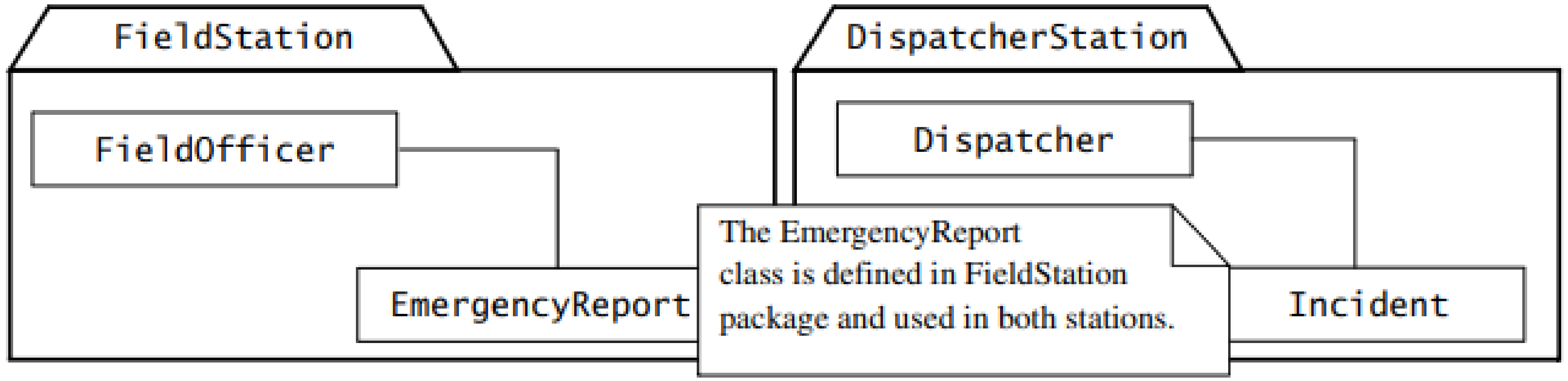

FieldStation
FieldOfficer
EmergencyReport
DispatcherStation
Dispatcher
Incident
The EmergencyReport class is defined in FieldStation package and used in both stations.

Figure 2-48 An example of a note. Notes can be attached to a specific element in a diagram.

DIAGRAM EXTENSIONS

- The goal of the UML designers was 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,
  1. stereotypes and
  2. constraints.

DIAGRAM EXTENSIONS (CONT.)

- **Stereotype:** A stereotype is an extension mechanism that allows developers to classify model elements in UML. A stereotype is represented by string enclosed by guillemets (e.g., «boundary») and attached to the model element to which it applies, such as a class or an association.

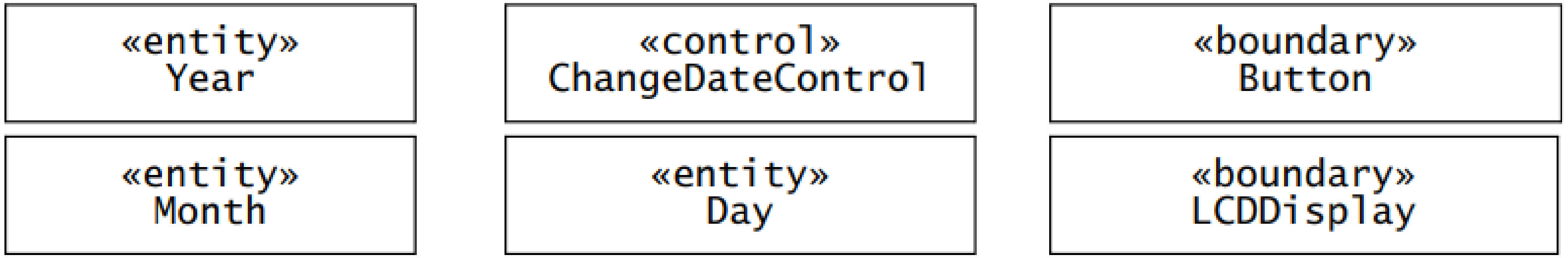


Figure 2-49 Examples of stereotypes (UML class diagram).

DIAGRAM EXTENSIONS (CONT.)

- **Constraint:** A constraint is a rule that is attached to a UML model element restricting its semantics. This allows us to represent phenomena that cannot otherwise be expressed with UML.

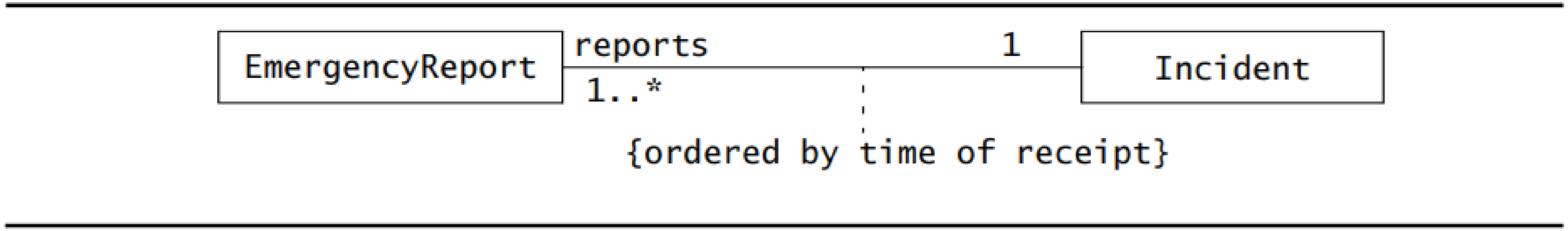


Figure 2-50 An example of constraint (UML class diagram).

THANK YOU