Chapter Four: Analysis

Analysis results in a model of the system that aims to be correct, complete, consistent, and unambiguous. Developers formalize the requirements specification produced during requirements elicitation and examine in more detail boundary conditions and exceptional cases. Developers validate, correct and clarify the requirements specification if any errors or ambiguities are found. The client and the user are usually involved in this activity when the requirements specification must be changed and when additional information must be gathered.

In object-oriented analysis, developers build a model describing the application domain. For example, the analysis model of a watch describes how the watch represents time: Does the watch know about leap years? Does it know about the day of the week? Does it know about the phases of the moon? The analysis model is then extended to describe how the actors and the system interact to manipulate the application domain model: How does the watch owner reset the time? How does the watch owner reset the day of the week? Developers use the analysis model, together with nonfunctional requirements, to prepare for the architecture of the system developed during high-level design.

This chapter, discuss the analysis activities in more detail. It focuses on the identification of objects, their behavior, their relationships, their classification, and their organization. This chapter also describes management issues related to analysis in the context of a multi-team development project.

4.1. An Overview of Analysis

Analysis focuses on producing a model of the system, called the analysis model, which is correct, complete, consistent, and verifiable. Analysis is different from requirements elicitation in that developers focus on structuring and formalizing the requirements elicited from users(Figure 4.1).

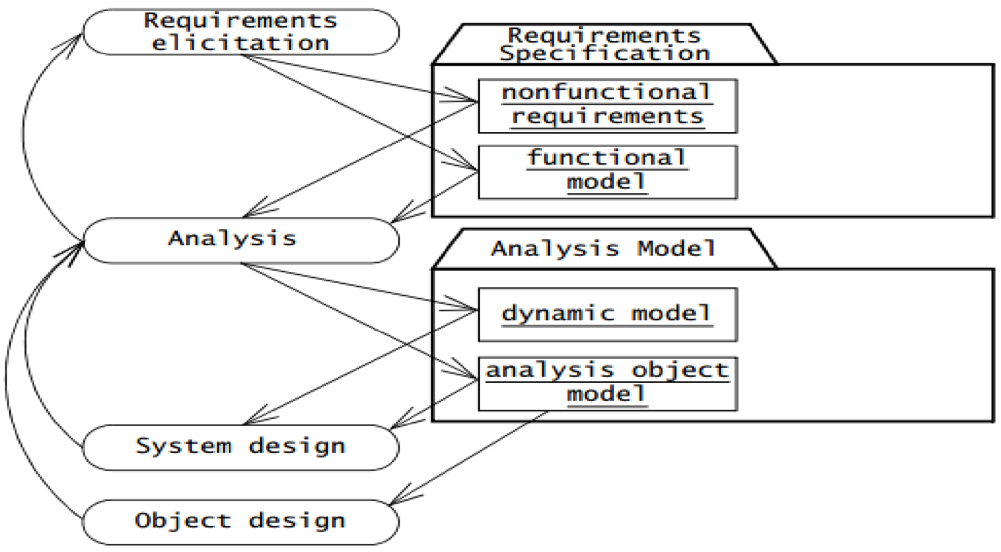


Figure 4-1 Products of requirements elicitation and analysis (UML activity diagram).

This formalization leads to new insights and the discovery of errors in the requirements. As the analysis model may not be understandable to the users and the client, developers need to update the requirements specification to reflect insights gained during analysis, then review the changes with the client and the users. In the end, the requirements, however large, should be understandable by the client and the users.

There is a natural tendency for users and developers to postpone difficult decisions until later in the project. A decision may be difficult because of lack of domain knowledge, lack of technological knowledge, or simply because of disagreements among users and developers. Postponing decisions enables the project to move on smoothly and avoids confrontation with reality or peers. Unfortunately, difficult decisions eventually must be made, often at higher cost when intrinsic problems are discovered during testing, or worse, during user evaluation. Translating a requirements specification into formal or semiformal model forces developers to identify and resolve difficult issues early in the development.

The **analysis model** is composed of three individual models: the **functional model**, represented by use cases and scenarios, the **analysis object model**, represented by class and object diagrams, and the **dynamic model**, represented by state machine and sequence diagrams (Figure 4-2). In the previous chapter, we described how to elicit requirements from the users and describe them as use cases and scenarios. In this chapter, we describe how to refine the functional model and derive the object and the dynamic model. This leads to a more precise and complete specification as details is added to the analysis model. We conclude the chapter by describing management activities related to analysis. In the next section, we define the main concepts of analysis

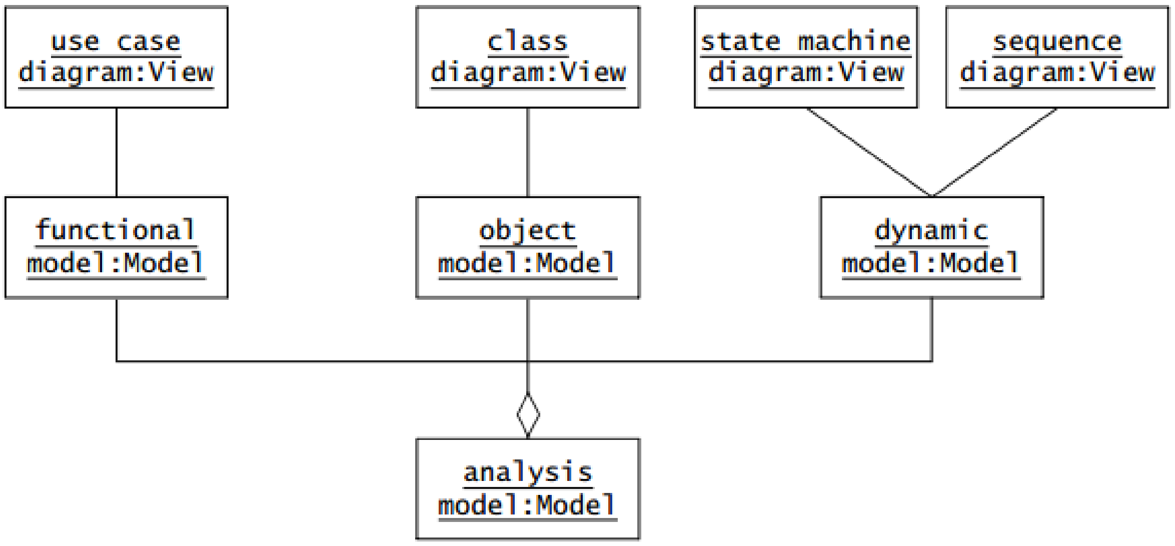


Figure 4-4 The analysis model is composed of the functional model, the object model, and the dynamic model. In UML, the functional model is represented with use case diagrams, the object model with class diagrams, and the dynamic model with state machine and sequence diagrams.

4.3. Analysis Concepts

In this section, we describe the main analysis concepts used in this chapter. In particular, we describe

Analysis Object Models and Dynamic Models
Entity, Boundary, and Control Objects
Generalization and Specialization

4.3.1Analysis Object Models and Dynamic Models

The analysis model represents the system under development from the user's point of view. The **analysis object model** is a part of the analysis model and focuses on the individual concepts that are manipulated by the system, their properties and their relationships. The analysis object model, depicted with UML class diagrams, includes classes, attributes, and operations. The analysis object model is a visual dictionary of the main concepts visible to the user.

The **dynamic model** focuses on the behavior of the system. The dynamic model is depicted with sequence diagrams and with state machines. Sequence diagrams represent the interactions among a set of objects during a single use case. State machines represent the behavior of a single object (or a group of very tightly coupled objects). The dynamic model serves to assign responsibilities to individual classes and, in the process, to identify new classes, associations, and attributes to be added to the analysis object model.

When working with either the analysis object model or the dynamic model, it is essential to remember that these models represent user-level concepts, not actual software classes or components. For example, classes such as Database, Subsystem, Session Manager, Network, should not appear in the analysis model as the user is completely shielded from those concepts. Note that most classes in the analysis object model will correspond to one or more software classes in the source code. However, the software classes will include many more attributes and associations than their analysis counterparts. Consequently, analysis classes should be viewed as high-level abstractions that will be realized in much more detail later. Figure 4-3 depicts good and bad examples of analysis objects for the SatWatch example.

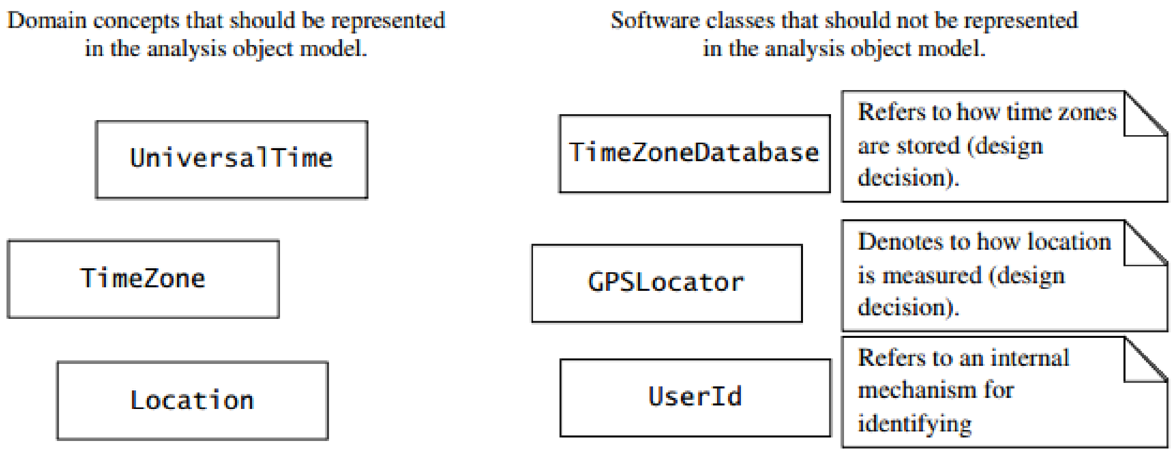

Domain concepts that should be represented in the analysis object model.
Software classes that should not be represented in the analysis object model.
UniversalTime
TimeZone
Location
TimeZoneDatabase
Refers to how time zones are stored (design decision).
GPSLocator
Denotes to how location is measured (design decision).
UserId
Refers to an internal mechanism for identifying

Figure 4-3 Examples and counterexamples of classes in the analysis object model of SatWatch.

4.3.2. Entity, Boundary, and Control Objects

The analysis object model consists of entity, boundary, and control objects. **Entity objects** represent the persistent information tracked by the system. **Boundary objects** represent the interactions between the actors and the system. **Control objects** are in charge of realizing use cases. In the 2Bwatch example, Year, Month, and Day are entity objects; Button and LCDDisplay are boundary objects; Change Date Control is a control object that represents the activity of changing the date by pressing combinations of buttons.

Modeling the system with entity, boundary, and control objects provides developers with simple heuristics to distinguish different, but related concepts. For example, the time that is tracked by a watch has different properties than the display that depicts the time. Differentiating between boundary and entity objects forces that distinction: The time that is tracked by the watch is represented by the Time object. The display is represented by the LCD Display. This approach with three object types results in smaller and more specialized objects. The three object-type approach also leads to models that are more resilient to change: the interface to the system (represented by the boundary objects) is more likely to change than its basic functionality (represented by the entity and control objects). By separating the interface from the basic functionality, we are able to keep most of a model untouched when, for example, the user interface changes, but the entity objects do not.

To distinguish between different types of objects, UML provides the stereotype mechanism to enable the developer to attach such meta-information to modeling elements. For example, in Figure 4-4, attach the «control» stereotype to the Change Date Control object. In addition to stereotypes, we may also use naming conventions for clarity and recommend distinguishing the three different types of objects on a syntactical basis: control objects may have the suffix Control appended to their name; boundary objects may be named to clearly denote an interface feature (e.g., by including the suffix Form, Button, Display, or Boundary); entity objects usually do not have any suffix appended to their name. Another benefit of this naming convention is that the type of the class is represented even when the UML stereotype is not available, for example, when examining only the source code.

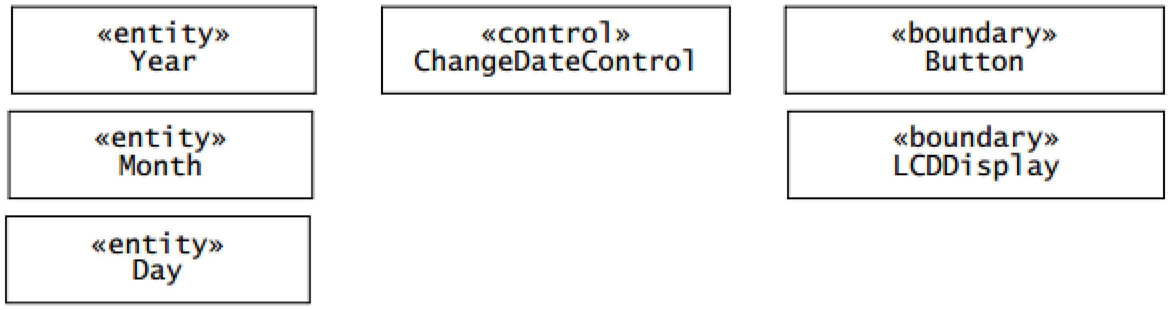


Figure 4.4. Analysis classes for the 2Bwatch example.

4.3. 3. Generalization and Specialization

As we saw in Chapter 2, *Modeling with UML*, **inheritance** enables us to organize concepts into hierarchies. At the top of the hierarchy is a general concept (e.g., an Incident, Figure 4-5), and at the bottom of the hierarchy are the most specialized concepts (e.g., CatInTree, TrafficAccident, BuildingFire, EarthQuake, ChemicalLeak). There may be any number of intermediate levels in between, covering more-or-less generalized concepts (e.g., LowPriorityIncident, Emergency, Disaster). Such hierarchies allow us to refer to many concepts precisely. When we use the term Incident, we mean all instances of all types of Incidents. When we use the term Emergency, we only refer to an Incident that requires an immediate response.

Generalization is the modeling activity that identifies abstract concepts from lower-level ones. For example, assume we are reverse-engineering an emergency management system and discover screens for managing traffic accidents and fires. Noticing common features among these three concepts, we create an abstract concept called Emergency to describe the common (and general) features of traffic accidents and fires.

Specialization is the activity that identifies more specific concepts from a high-level one. For example, assume that we are building an emergency management system from scratch and that we are discussing its functionality with the client. The client first introduces us with the concept of an incident, then describes three types of Incidents: Disasters, which require the collaboration of several agencies, Emergencies, which require immediate handling but can be handled by a single agency, and LowPriorityIncidents, that do not need to be handled if resources are required for other, higher-priority Incidents.

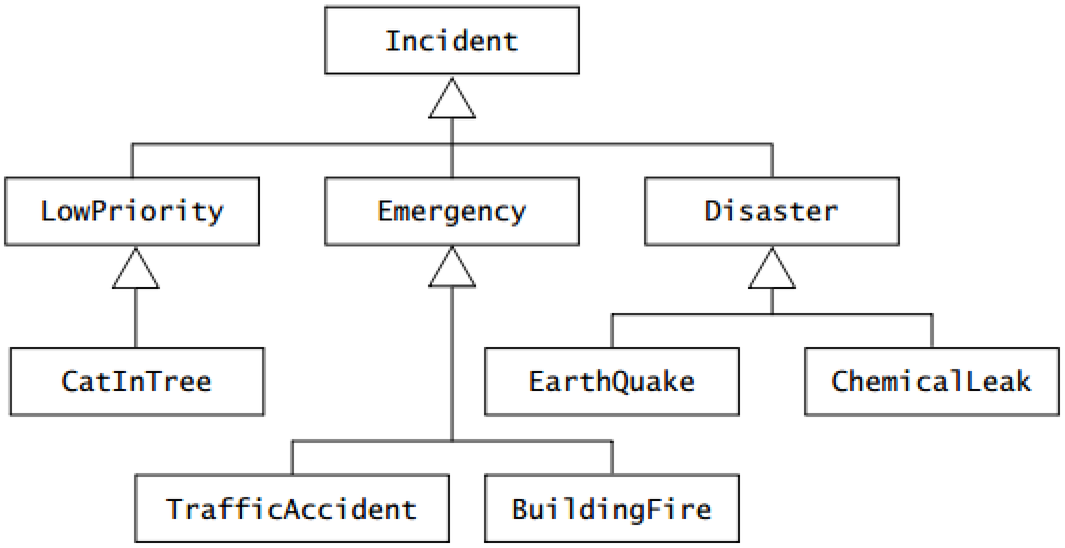


Figure 4-5 An example of a generalization hierarchy (UML class diagram). The top of the hierarchy represents the most general concept, whereas the bottom nodes represent the most specialized concepts.

In both cases, generalization and specialization result in the specification of inheritance relationships between concepts. In some instances, modelers call inheritance relationships **generalization-specialization** relationships. In this book, we use the term “inheritance” to denote the relationship and the terms “generalization” and “specialization” to denote the activities that find inheritance relationships.

4.4. Analysis Activities: From Use Cases to Objects

In this section, we describe the activities that transform the use cases and scenarios produced during requirements elicitation into an analysis model. Analysis activities include:

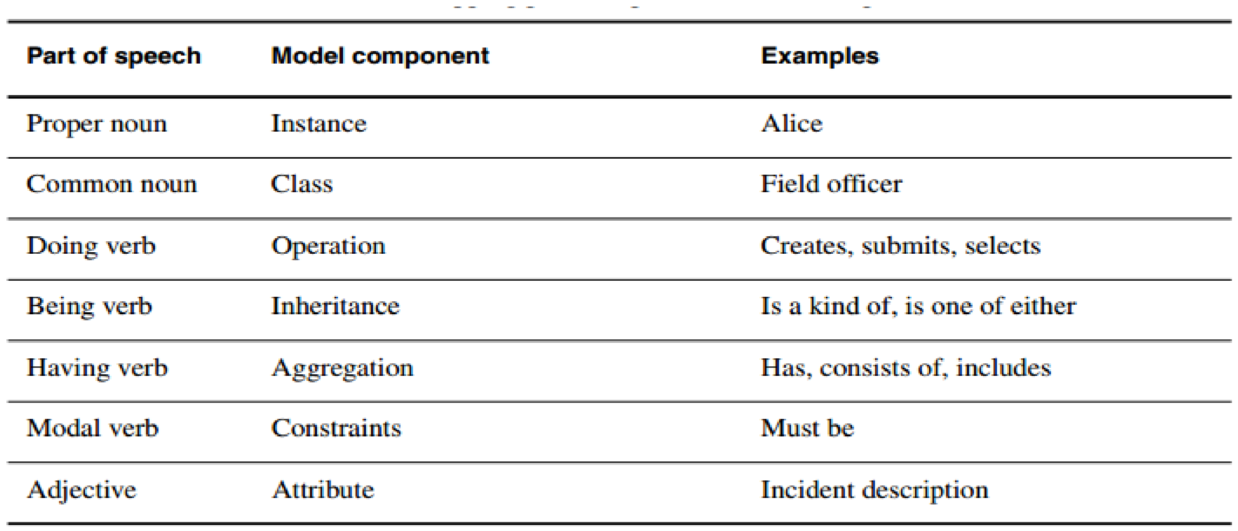
- Identifying Entity Objects
- Identifying Boundary Objects
- Identifying Control Objects
- Mapping Use Cases to Objects with Sequence Diagrams
- Identifying Associations

- Identifying Aggregates
- Identifying Attributes
- Modeling State-Dependent Behavior of Individual Objects
- Modeling Inheritance Relationships
- Reviewing the Analysis Model

4.4.1. Identifying Entity Objects

As described in Chapter 3, *Requirements Elicitation*, participating objects are found by examining each use case and identifying candidate objects. Natural language analysis is an intuitive set of heuristics for identifying objects, attributes, and associations from a requirements specification.
Abbott’s heuristics maps parts of speech (e.g., nouns, having verbs, being verbs, adjectives) to model components (e.g., objects, operations, inheritance relationships, classes). Table 4-1 provides examples of such mappings by examining the ReportEmergency use case (Figure 4-6). Natural language analysis has the advantage of focusing on the users’ terms. However, it suffers from several limitations. First, the quality of the object model depends highly on the style of writing of the analyst (e.g., consistency of terms used, verbification of nouns). Natural language is an imprecise tool, and an object model derived literally from text risks being imprecise. Developers can address this limitation by rephrasing and clarifying the requirements specification as they identify and standardize objects and terms. A second limitation of natural language analysis is that there are many more nouns than relevant classes. Many nouns correspond to attributes or synonyms for other nouns. Sorting through all the nouns for a large requirements specification is a time-consuming activity. In general, Abbott’s heuristics work well for generating a list of initial candidate objects from short descriptions, such as the flow of events of a scenario or a use case.

Table 4-1 Abbott’s heuristics for mapping parts of speech to model components



Part of speech	Model component	Examples
Proper noun	Instance	Alice
Common noun	Class	Field officer
Doing verb	Operation	Creates, submits, selects
Being verb	Inheritance	Is a kind of, is one of either
Having verb	Aggregation	Has, consists of, includes
Modal verb	Constraints	Must be
Adjective	Attribute	Incident description

The following heuristics can be used in conjunction with Abbott's heuristics:

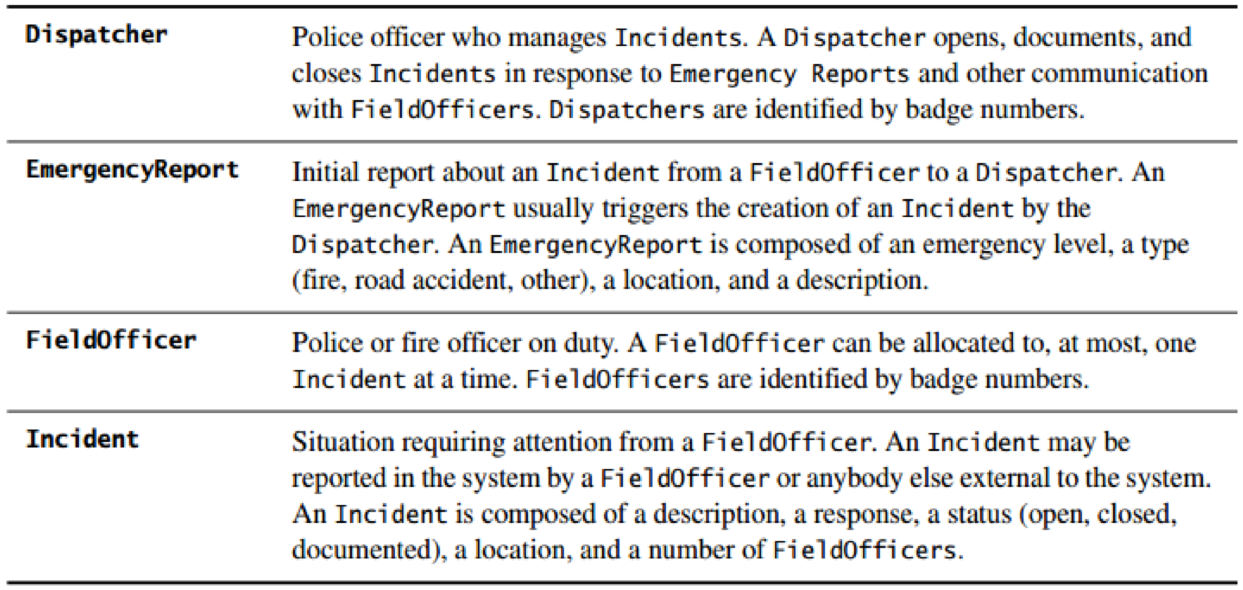
Heuristics for identifying entity objects

- Terms that developers or users need to clarify in order to understand the use case
- Recurring nouns in the use cases (e.g., `Incident`)
- Real-world entities that the system needs to track (e.g., `FieldOfficer`, `Dispatcher`, `Resource`)
- Real-world activities that the system needs to track (e.g., `EmergencyOperationsPlan`)
- Data sources or sinks (e.g., `Printer`).

Developers name and briefly describe the objects, their attributes, and their responsibilities as they are identified. Uniquely naming objects promotes a standard terminology. For entity objects we recommend *always* to start with the names used by end users and application domain specialists. Describing objects, even briefly, allows developers to clarify the concepts they use and avoid misunderstandings (e.g., using one object for two different but related concepts). Developers need not, however, spend a lot of time detailing objects or attributes given that the analysis model is still in flux. Developers should document attributes and responsibilities if they are not obvious; a tentative name and a brief description for each object is sufficient otherwise. There will be plenty of iterations during which objects can be revised. However, once the analysis model is stable, the description of each object should be as detailed as necessary

For example, after a first examination of the ReportEmergency use case , use application domain knowledge and interviews with the users to identify the objects Dispatcher, EmergencyReport, FieldOfficer, and Incident. Note that the EmergencyReport object is not mentioned explicitly by name in the ReportEmergency use case. Step 4 of the use case refers to the emergency report as the "information submitted by the FieldOfficer." After review with the client, we discover that this information is usually referred to as the "emergency report" and decide to name the corresponding object EmergencyReport. The definition of entity objects leads to the initial analysis model described in Table 5-2. Note that this model is far from a complete description of the system implementing the ReportEmergency use case. In the next section, we describe the identification of boundary objects.

Table 4-2 Entity objects for the ReportEmergency use case.



Dispatcher	Police officer who manages Incidents. A Dispatcher opens, documents, and closes Incidents in response to Emergency Reports and other communication with FieldOfficers. Dispatchers are identified by badge numbers.
EmergencyReport	Initial report about an Incident from a FieldOfficer to a Dispatcher. An EmergencyReport usually triggers the creation of an Incident by the Dispatcher. An EmergencyReport is composed of an emergency level, a type (fire, road accident, other), a location, and a description.
FieldOfficer	Police or fire officer on duty. A FieldOfficer can be allocated to, at most, one Incident at a time. FieldOfficers are identified by badge numbers.
Incident	Situation requiring attention from a FieldOfficer. An Incident may be reported in the system by a FieldOfficer or anybody else external to the system. An Incident is composed of a description, a response, a status (open, closed, documented), a location, and a number of FieldOfficers.

4.4.2. Identifying Boundary Objects

Boundary objects represent the system interface with the actors. In each use case, each actor interacts with at least one boundary object. The boundary object collects the information from the actor and translates it into a form that can be used by both entity and control objects.

Boundary objects model the user interface at a coarse level. They do not describe in detail the visual aspects of the user interface. For example, boundary objects such as “menu item” or “scroll bar” are too detailed. First, developers can discuss user interface details more easily with sketches and mock-ups. Second, the design of the user interface continues to evolve as a consequence of usability tests, even after the functional specification of the system becomes stable. Updating the analysis model for every user interface change is time consuming and does not yield any substantial benefit.

Heuristics for identifying boundary objects

- Identify user interface controls that the user needs to initiate the use case (e.g., ReportEmergencyButton).
- Identify forms the users needs to enter data into the system (e.g., EmergencyReportForm).
- Identify notices and messages the system uses to respond to the user (e.g., AcknowledgmentNotice).
- When multiple actors are involved in a use case, identify actor terminals (e.g., DispatcherStation) to refer to the user interface under consideration.

- Do not model the visual aspects of the interface with boundary objects (user mock-ups are better suited for that).
- *Always* use the end user's terms for describing interfaces; do not use terms from the solution or implementation domains.

Table 4-3 Boundary objects for the ReportEmergency use case.

AcknowledgmentNotice	Notice used for displaying the Dispatcher's acknowledgment to the FieldOfficer.
DispatcherStation	Computer used by the Dispatcher.
ReportEmergencyButton	Button used by a FieldOfficer to initiate the ReportEmergency use case.
EmergencyReportForm	Form used for the input of the ReportEmergency. This form is presented to the FieldOfficer on the FieldOfficerStation when the "Report Emergency" function is selected. The EmergencyReportForm contains fields for specifying all attributes of an emergency report and a button (or other control) for submitting the completed form.
FieldOfficerStation	Mobile computer used by the FieldOfficer.
IncidentForm	Form used for the creation of Incidents. This form is presented to the Dispatcher on the DispatcherStation when the EmergencyReport is received. The Dispatcher also uses this form to allocate resources and to acknowledge the FieldOfficer's report.

4.4.3. Identifying Control Objects

Control objects are responsible for coordinating boundary and entity objects. Control objects usually do not have a concrete counterpart in the real world. Often a close relationship exists between a use case and a control object; a control object is usually created at the beginning of a use case and ceases to exist at its end. It is responsible for collecting information from the boundary objects and dispatching it to entity objects. For example, control objects describe the behavior associated with the sequencing of forms, undo and history queues, and dispatching information in a distributed system.

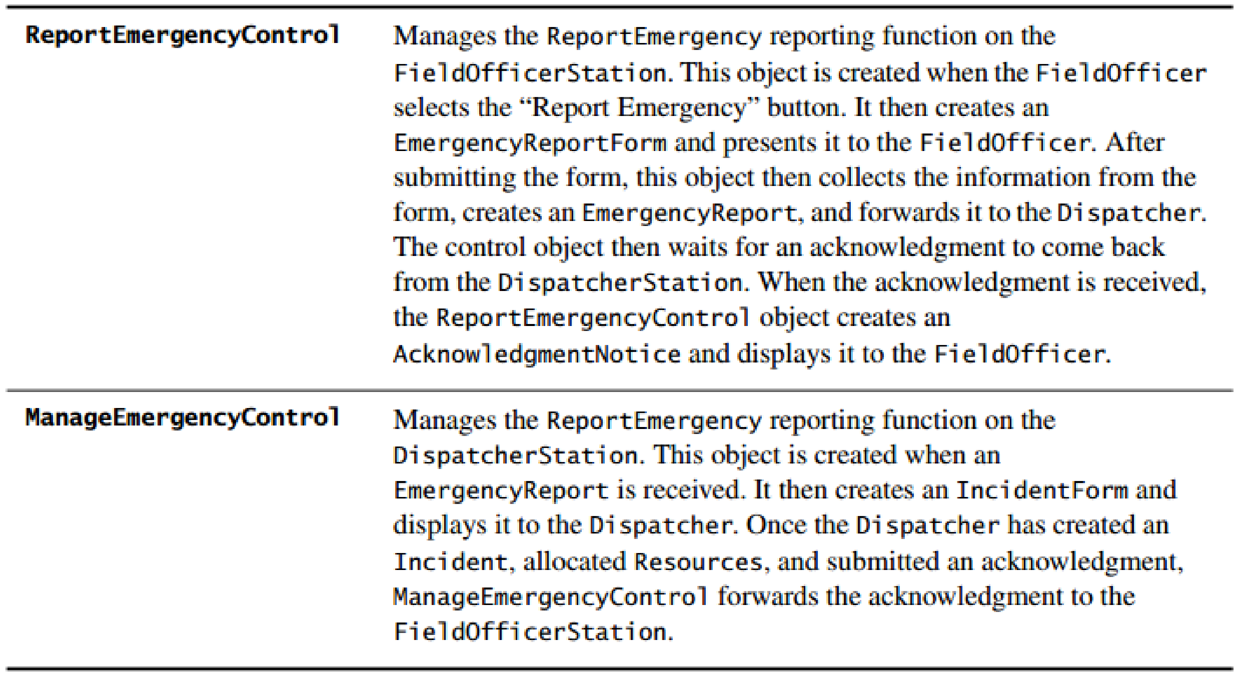
Initially, the developers model the control flow of the ReportEmergency use case with a control object for each actor: ReportEmergencyControl for the FieldOfficer and ManageEmergency-Control for the Dispatcher, respectively (Table 4-4).

The decision to model the control flow of the ReportEmergency use case with two control objects stems from the knowledge that the FieldOfficerStation and the DispatcherStation are actually two subsystems communicating over an asynchronous link. This decision could have been postponed until the system design activity. On the other hand, making this concept visible in the analysis model allows us to focus on such exception behavior as the loss of communication between both stations.

Heuristics for identifying control objects

- Identify one control object per use case.
- Identify one control object per actor in the use case.
- The life span of a control object should cover the extent of the use case or the extent of a user session. If it is difficult to identify the beginning and the end of a control object activation, the corresponding use case probably does not have well-defined entry and exit conditions.

Table 4-4 Control objects for the ReportEmergency use case.



ReportEmergencyControl	Manages the ReportEmergency reporting function on the FieldOfficerStation. This object is created when the FieldOfficer selects the "Report Emergency" button. It then creates an EmergencyReportForm and presents it to the FieldOfficer. After submitting the form, this object then collects the information from the form, creates an EmergencyReport, and forwards it to the Dispatcher. The control object then waits for an acknowledgment to come back from the DispatcherStation. When the acknowledgment is received, the ReportEmergencyControl object creates an AcknowledgmentNotice and displays it to the FieldOfficer.
ManageEmergencyControl	Manages the ReportEmergency reporting function on the DispatcherStation. This object is created when an EmergencyReport is received. It then creates an IncidentForm and displays it to the Dispatcher. Once the Dispatcher has created an Incident, allocated Resources, and submitted an acknowledgment, ManageEmergencyControl forwards the acknowledgment to the FieldOfficerStation.

4.4.4. Mapping Use Cases to Objects with Sequence Diagrams

A **sequence diagram** ties use cases with objects. It shows how the behavior of a use case (or scenario) is distributed among its participating objects. Sequence diagrams are usually not as good a medium for communication with the user as use cases are, since sequence diagrams require more background about the notation. For computer savvy clients, they are intuitive and can be more precise than use cases. In all cases, however, sequence diagrams represent another shift in perspective and allow the developers to find missing objects or grey areas in the requirements specification.

This section, model the sequence of interactions among objects needed to realize the use case. Figures 4-7 through 4-9 are sequence diagrams associated with the ReportEmergency use case. The columns of a sequence diagram represent the objects that participate in the use case. The left-most column is the actor who initiates the use case. Horizontal arrows across columns represent messages, or stimuli, that are sent from one object to the other. Time proceeds vertically from top to bottom. For example, the first arrow in Figure 4-7 represents the press message sent by a FieldOfficer to an ReportEmergencyButton. The receipt of a message triggers the activation of an operation. The activation is represented by a vertical rectangle from which other messages can originate. The length of the rectangle represents the time the operation is active. In Figure 4-7, the operation triggered by the press message sends a create message to the ReportEmergencyControl class. An operation can be thought of as a service that the object provides to other objects. Sequence diagrams also depict the lifetime of objects. Objects that already exist before the first stimuli in the sequence diagram are depicted at the top of the diagram. Objects that are created during the interaction are depicted with the «create» message pointing to the object. Instances that are destroyed during the interaction have a cross indicating when the object ceases to exist. Between the rectangle representing the object and the cross (or the bottom of the diagram, if the object survives the interaction), a dashed line represents the time span when the object can receive messages. The object cannot receive messages below the cross sign. For example, in Figure 4-7 an object of class ReportEmergencyForm is created when object of ReportEmergencyControl sends the «create» message and is destroyed once the EmergencyReportForm has been submitted.

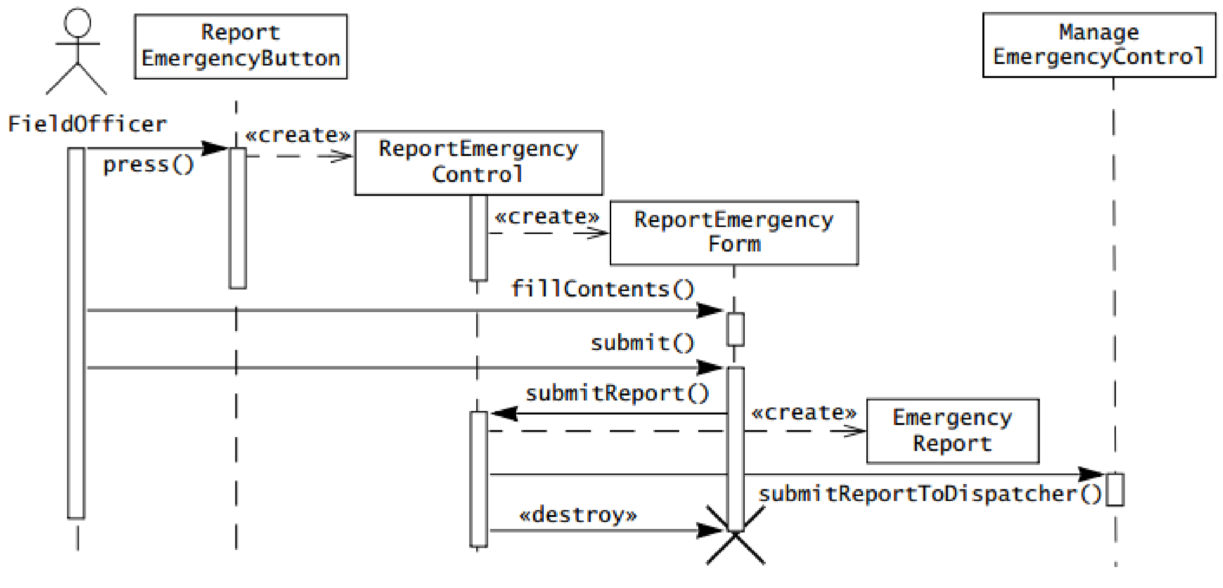


Figure 4-7 Sequence diagram for the ReportEmergency use case.

In general, the second column of a sequence diagram represents the boundary object with which the actor interacts to initiate the use case (e.g., ReportEmergencyButton). The third column is a control object that manages the rest of the use case (e.g., ReportEmergency– Control). From then on, the control object creates other boundary objects and may interact with other control objects as well (e.g., ManageEmergencyControl).

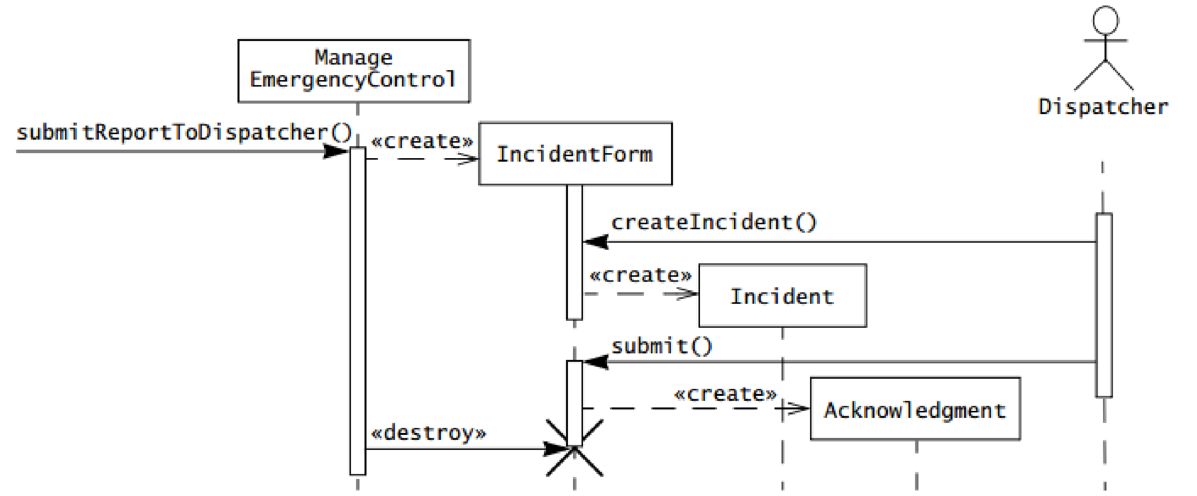


Figure 4-8 Sequence diagram for the ReportEmergency use case (continued from Figure 4-7)

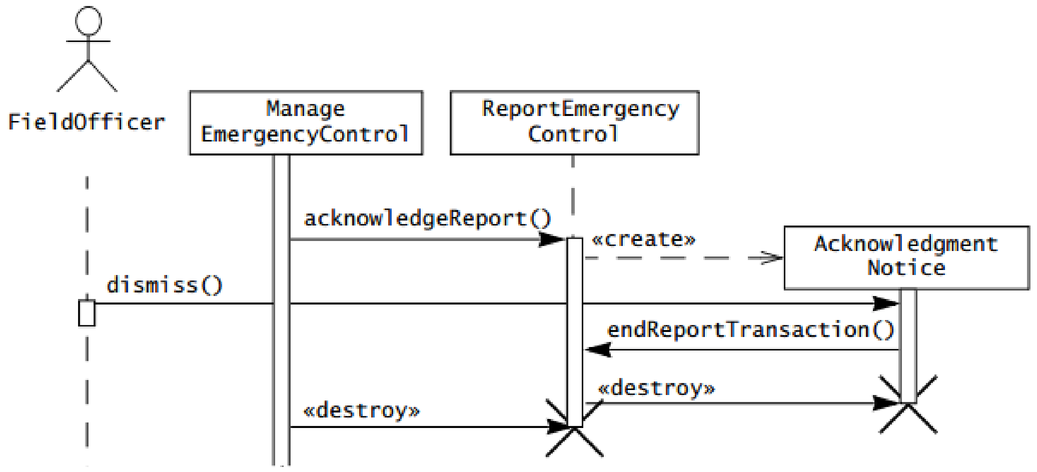


Figure 4-9 Sequence diagram for the ReportEmergency use case (continued from Figure 4-8).

By constructing sequence diagrams, we not only model the order of the interaction among the objects, we also distribute the behavior of the use case. That is, we assign responsibilities to each object in the form of a set of operations. These operations can be shared by any use case in which a given object participates. Note that the definition of an object that is shared across two or more use cases should be identical; that is, if an operation appears in more than one sequence diagram, its behavior should be the same.

Heuristics for drawing sequence diagrams

- The first column should correspond to the actor who initiated the use case.
- The second column should be a boundary object (that the actor used to initiate the use case).
- The third column should be the control object that manages the rest of the use case.
- Control objects are created by boundary objects initiating use cases.
- Boundary objects are created by control objects.
- Entity objects are accessed by control and boundary objects.
- Entity objects *never* access boundary or control objects; this makes it easier to share entity objects across use cases.

4.4.5. Identifying Associations

Whereas sequence diagrams allow developers to represent interactions among objects over time, class diagrams allow developers to describe the interdependencies of objects. We described the UML class diagram notation in Chapter 2, *Modeling with UML*, and use it throughout the book to represent various project artifacts (e.g., activities, deliverables). In this section, we discuss the use of class diagrams for representing associations among objects.

An **association** shows a relationship between two or more classes. For example, a FieldOfficer writes an EmergencyReport (see Figure 5-13). Identifying associations has two advantages. First, it clarifies the analysis model by making relationships between objects explicit (e.g., an EmergencyReport can be created by a FieldOfficer or a Dispatcher). Second, it enables the developer to discover boundary cases associated with links. Boundary cases are exceptions that must be clarified in the model. For example, it is intuitive to assume that most EmergencyReports are written by one FieldOfficer. However, should the system support EmergencyReports written by more than one? Should the system allow for anonymous EmergencyReports? Those questions should be investigated during analysis by discussing them with the client or with end users.

Associations have several properties:

- A **name** to describe the association between the two classes. Association names are optional and need not be unique globally.

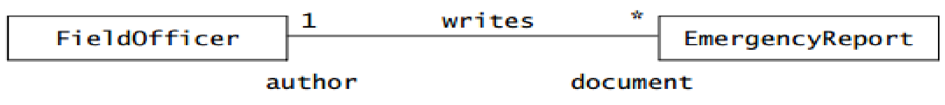


Figure 4-11 An example of association between the EmergencyReport and the FieldOfficer classes.

- A **role** at each end, identifying the function of each class with respect to the associations (e.g., author is the role played by FieldOfficer in the Writes association).
- A **multiplicity** at each end, identifying the possible number of instances (e.g., * indicates a FieldOfficer may write zero or more EmergencyReports, whereas 1 indicates that each EmergencyReport has exactly one FieldOfficer as author).

Initially, the associations between entity objects are the most important, as they reveal more information about the application domain. According to Abbott's heuristics (see Table 4-1), associations can be identified by examining verbs and verb phrases denoting a state (e.g., *has, is part of, manages, reports to, is triggered by, is contained in, talks to, includes*). Every association should be named, and roles should be assigned to each end.

Heuristics for identifying associations

- Examine verb phrases.
- Name associations and roles precisely.
- Use qualifiers as often as possible to identify namespaces and key attributes.
- Eliminate any association that can be derived from other associations.
- Do not worry about multiplicity until the set of associations is stable.
- Too many associations make a model unreadable.

4. 4.6. Identifying Aggregates

Aggregations are special types of associations denoting a whole–part relationship. For example, a FireStation consists of a number of FireFighters, FireEngines, Ambulances, and a LeadCar. A State is composed of a number of Counties that are, in turn, composed of a number of Townships (Figure 4-12). An aggregation is shown as a association with a diamond on the side of the whole part.

There are two types of aggregation, composition and shared. A solid diamond denotes composition. A **composition aggregation** indicates that the existence of the parts depends on the whole. For example, a County is always part of exactly one State, a Township is always part of a County. As political boundaries do not change often, a Township will not be part of or shared with another County (at least, in the life time of the emergency response system).

A hollow diamond denotes a **shared aggregation** relationship, indicating the whole and the part can exist independently. For example, although a FireEngine is part of at most one FireStation at the time, it can be reassigned to a different FireStation during its life time.

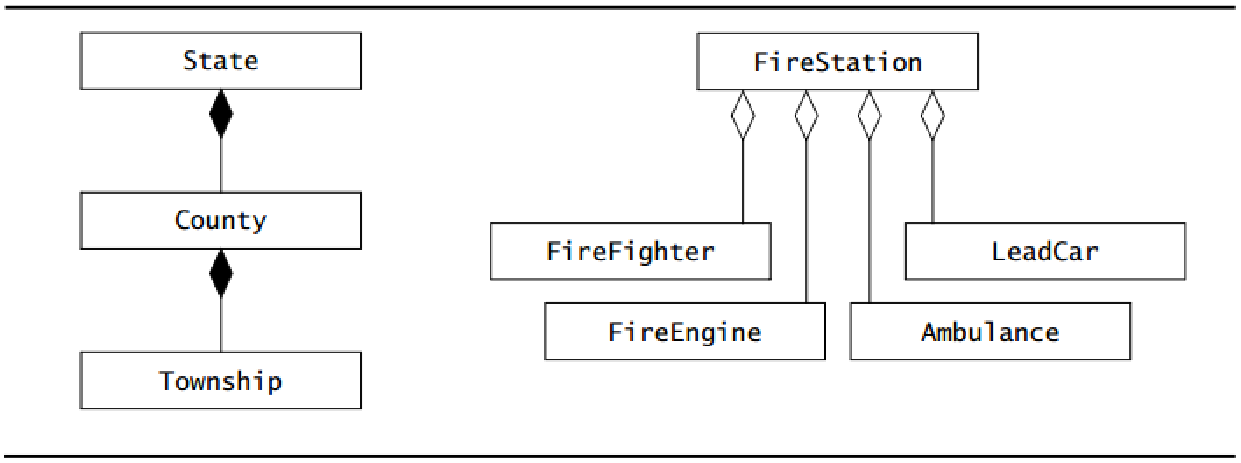


Figure 4-12 Examples of aggregations and compositions (UML class diagram). A State is composed of many Counties, which in turn is composed of many Townships. A FireStation includes FireFighters, FireEngines, Ambulances, and a LeadCar.

Aggregation associations are used in the analysis model to denote whole–part concepts. Aggregation associations add information to the analysis model about how containment concepts in the application domain can be organized in a hierarchy or in a directed graph. Aggregations are often used in the user interface to help the user browse through many instances. For example, in Figure 4-12, FRIEND could offer a tree representation for Dispatchers to find Counties within a State or Townships with a specific County. However, as with many modeling concepts, it is easy to over-structure the model. If you are not sure that the association you are describing is a whole–part concept, it is better to model it as a one-to-many association, and revisit it later when you have a better understanding of the application domain.

4. .4.7. Identifying Attributes

Attributes are properties of individual objects. For example, an EmergencyReport, as described in Table 4-2, has an emergency type, a location, and a description property (see Figure 5-16). These are entered by a FieldOfficer when she reports an emergency and are subsequently tracked by the system. When identifying properties of objects, only the attributes relevant to the system should be considered. For example, each FieldOfficer has a social security number that is not relevant to the emergency information system. Instead, FieldOfficers are identified by badge number, which is represented by the badgeNumber property.

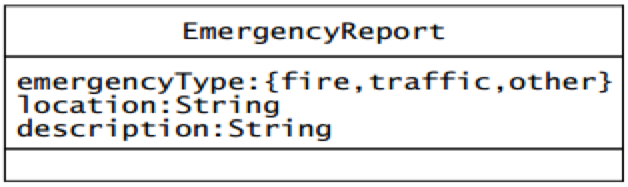


Figure 4-13 Attributes of the EmergencyReport class.

Properties that are represented by objects are not attributes. For example, every EmergencyReport has an author that is represented by an association to the FieldOfficer class. Developers should identify as many associations as possible before identifying attributes to avoid confusing attributes and objects. Attributes have:

- A **name** identifying them within an object. For example, an EmergencyReport may have a reportType attribute and an emergencyType attribute. The reportType describes the kind of report being filed (e.g., initial report, request for resource, final report). The emergencyType describes the type of emergency (e.g., fire, traffic, other). To avoid confusion, these attributes should not both be called type.
- A brief description.
- A **type** describing the legal values it can take. For example, the description attribute of an EmergencyReport is a string. The emergencyType attribute is an enumeration that can take one of three values: fire, traffic, other. Attribute types are based on predefined basic types in UML.

Note that attributes represent the least stable part of the object model. Often, attributes are discovered or added late in the development when the system is evaluated by the users. Unless the added attributes are associated with additional functionality, the added attributes do not entail major changes in the object (and system) structure. For these reasons, the developers need not spend excessive resources in identifying and detailing attributes that represent less important aspects of the system. These attributes can be added later when the analysis model or the user interface sketches are validated.

Heuristics for identifying attributes

- Examine possessive phrases.
- Represent stored state as an attribute of the entity object.
- Describe each attribute.
- Do not represent an attribute as an object; use an association instead (see Section 5.4.6).
- Do not waste time describing fine details before the object structure is stable.

4.4.8. Modeling State-Dependent Behavior of Individual Objects.

Sequence diagrams are used to distribute behavior across objects and to identify operations. Sequence diagrams represent the behavior of the system from the perspective of a single use case. State machine diagrams represent behavior from the perspective of a single object. Viewing behavior from the perspective of each object enables the developer to build a more formal description of the behavior of the object, and consequently, to identify missing use cases. By focusing on individual states, developers may identify new behavior. For example, by examining each transition in the state machine diagram that is triggered by a user action, the developer should be able to identify a flow step in a use case that describes the actor action that triggers the transition. Note that it is not necessary to build state machines for every class in the system. Only objects with an extended lifespan and state-dependent behavior are worth considering. This is almost always the case for control objects, less often for entity objects, and almost never for boundary objects.

Figure 4-14 displays a state machine for the Incident class. The examination of this state machine may help the developer to check if there are use cases for documenting, closing, and archiving Incidents. By further refining each state, the developer can add detail to the different user actions that change the state of an incident. For example, during the Active state of an indicate, FieldOfficers should be able to request new resources, and Dispatchers should be able to allocate resource to existing incidents.

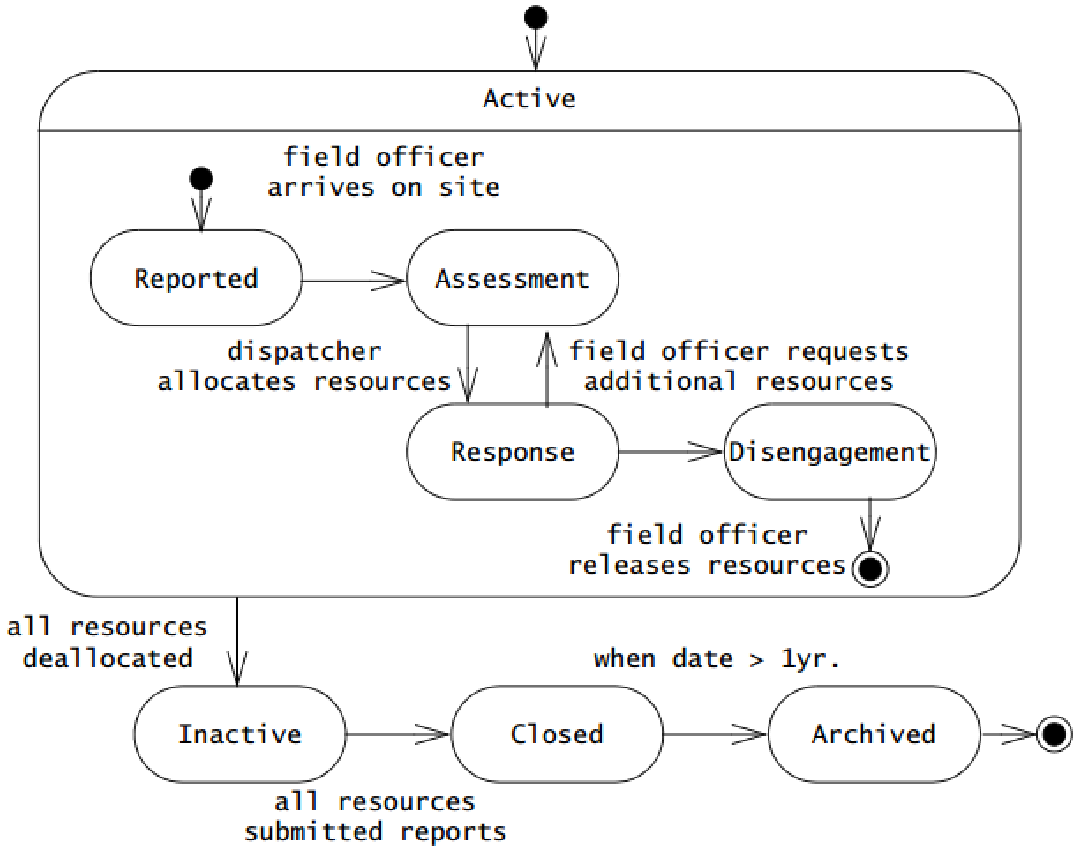

Active
field officer arrives on site
Reported
Assessment
dispatcher allocates resources
field officer requests additional resources
Response
Disengagement
field officer releases resources
all resources deallocated
when date > 1yr.
Inactive
Closed
Archived
all resources submitted reports

Figure 4-14 UML state machine for Incident.

4.4.9.Modeling Inheritance Relationships between Objects

Generalization is used to eliminate redundancy from the analysis model. If two or more classes share attributes or behavior, the similarities are consolidated into a superclass. For example, Dispatchers and FieldOfficers both have a badgeNumber attribute that serves to identify them within a city. FieldOfficers and Dispatchers are both PoliceOfficers who are assigned different functions. To model explicitly this similarity, we introduce an abstract *PoliceOfficer* class from which the FieldOfficer and Dispatcher classes inherit (see Figure 4-15).

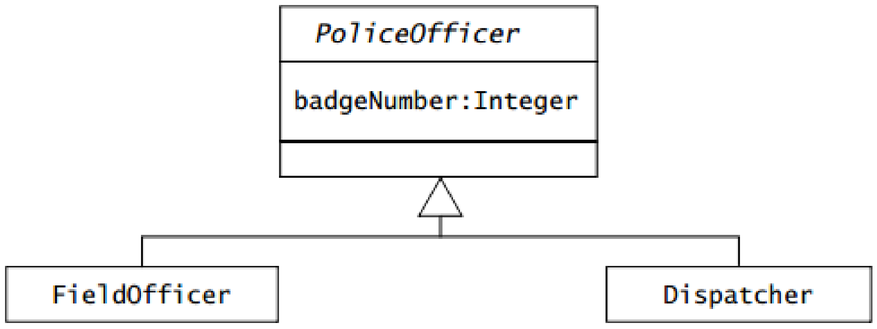


Figure 4-15 An example of inheritance relationship (UML class diagram).

4.4.10. Reviewing the Analysis Model

The analysis model is built incrementally and iteratively. The analysis model is seldom correct or even complete on the first pass. Several iterations with the client and the user are necessary before the analysis model converges toward a correct specification usable by the developers for design and implementation. For example, an omission discovered during analysis will lead to adding or extending a use case in the requirements specification, which may lead to eliciting more information from the user.

Once the number of changes to the model are minimal and the scope of the changes localized, the analysis model becomes stable. Then the analysis model is reviewed, first by the developers (i.e., internal reviews), then jointly by the developers and the client. The goal of the review is to make sure that the requirements specification is correct, complete, consistent, and unambiguous. Moreover, developers and client also review if the requirements are realistic and verifiable. Note that developers should be prepared to discover errors downstream and make changes to the specification. It is, however, a worthwhile investment to catch as many requirements errors upstream as possible. The review can be facilitated by a checklist or a list of questions.

The following questions should be asked to ensure that the model is *correct*:

- Is the glossary of entity objects understandable by the user?
- Do abstract classes correspond to user-level concepts?
- Are all descriptions in accordance with the users' definitions?

- Do all entity and boundary objects have meaningful noun phrases as names?
- Do all use cases and control objects have meaningful verb phrases as names?
- Are all error cases described and handled?

The following questions should be asked to ensure that the model is *consistent*:

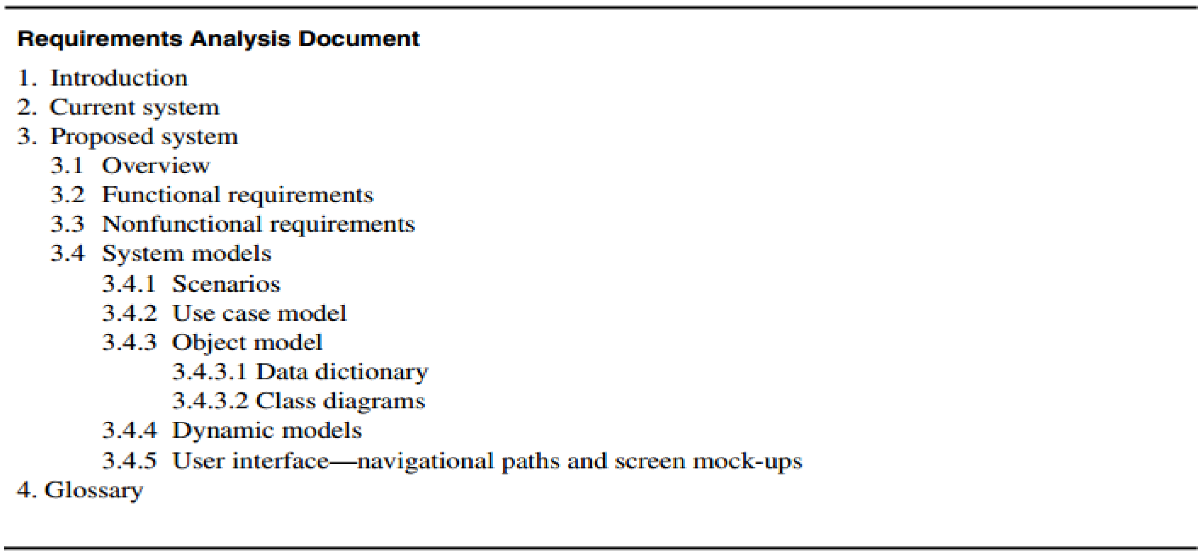
- Are there multiple classes or use cases with the same name?
- Do entities (e.g., use cases, classes, attributes) with similar names denote similar concepts?
- Are there objects with similar attributes and associations that are not in the same generalization hierarchy?

The following questions should be asked to ensure that the system described by the analysis model is *realistic*:

- Are there any novel features in the system? Were any studies or prototypes built to ensure their feasibility?
- Can the performance and reliability requirements be met? Were these requirements verified by any prototypes running on the selected hardware?

4.4.11. Documenting Analysis

The RAD(Requirement Analysis Documenting), once completed and published, will be baselined and put under configuration management. The revision history section of the RAD will provide a history of changes including the author responsible for each change, the date of the change, and brief description of the change.



Requirements Analysis Document

1. Introduction
2. Current system
3. Proposed system
   3.1 Overview
   3.2 Functional requirements
   3.3 Nonfunctional requirements
   3.4 System models
      3.4.1 Scenarios
      3.4.2 Use case model
      3.4.3 Object model
         3.4.3.1 Data dictionary
         3.4.3.2 Class diagrams
      3.4.4 Dynamic models
      3.4.5 User interface—navigational paths and screen mock-ups
4. Glossary

Figure 4-15 Overview outline of the Requirements Analysis Document (RAD). See Figure 4-16 for a detailed outline.