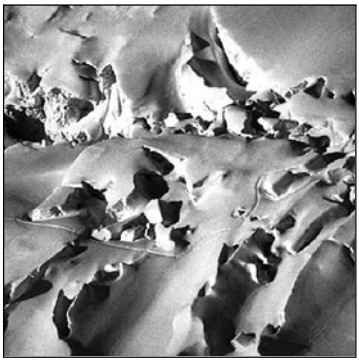
5

5



Analysis

I am Foo with a name, if I could only remember it.

—A programmer of very little brain

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 (Chapter 6, *System Design: Decomposing the System*).

In this chapter, we discuss the analysis activities in more detail. We focus on the identification of objects, their behavior, their relationships, their classification, and their organization. We describe management issues related to analysis in the context of a multi-team development project. Finally, we discuss in more detail analysis issues and trade-offs using the ARENA case study.

5.1 Introduction: An Optical Illusion

In 1915, Rubin exhibited a drawing similar to Figure 5-1 to illustrate the concept of multi-stable images. What do you see? Two faces looking at each other? If you focus more closely on the white area, you can see a vase instead. Once you are able to perceive both shapes individually, it is easier to switch back and forth between the vase and the faces.

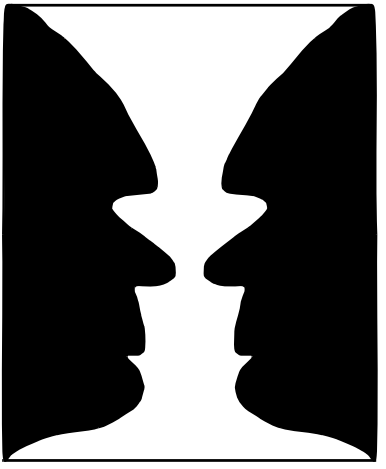


Figure 5-1 Ambiguity: what do you see?

If the drawing in Figure 5-1 had been a requirements specification, which models should you have constructed? Specifications, like multi-stable images, contain ambiguities caused by the inaccuracies inherent to natural language and by the assumptions of the specification authors. For example, a quantity specified without a unit is ambiguous (e.g., the "Feet or Miles?" example in Section 4.1), a time without time zone is ambiguous (e.g., scheduling a phone call between different countries).

Formalization helps identify areas of ambiguity as well as inconsistencies and omissions in a requirements specification. Once developers identify problems with the specification, they address them by eliciting more information from the users and the client. Requirements elicitation and analysis are iterative and incremental activities that occur concurrently.

5.2 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 5-2). This formalization leads to new insights and the discovery of errors in the

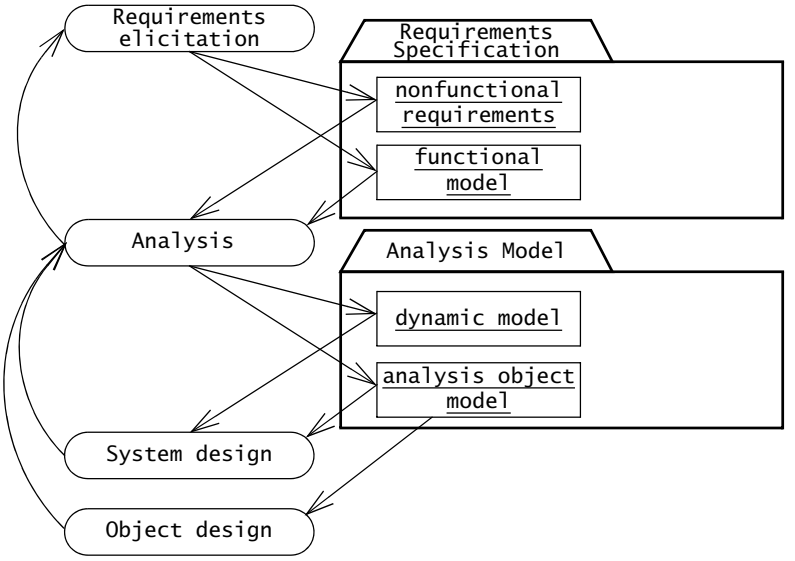


Figure 5-2 Products of requirements elicitation and analysis (UML activity diagram).

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 a 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 5-3). 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 are 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.

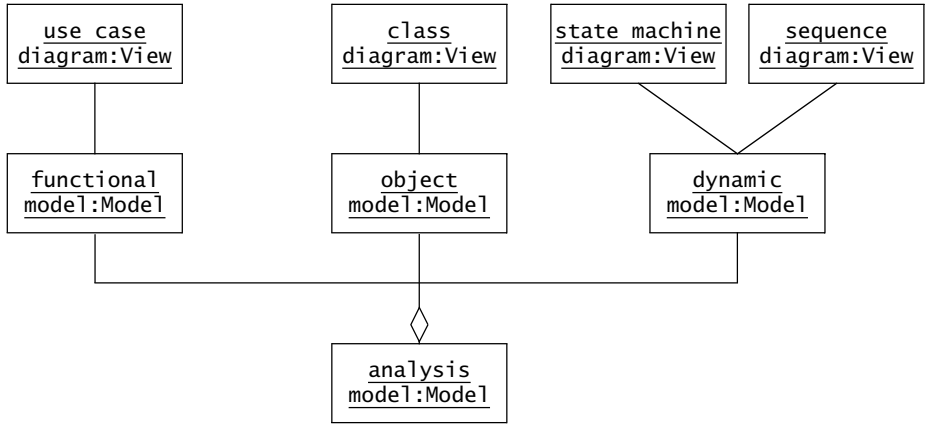


Figure 5-3 The analysis model is composed of the functional model, the object model, and the dynamic model. In UML, the functional model is represented with use case diagrams, the object model with class diagrams, and the dynamic model with state machine and sequence diagrams.

5.3 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 (Section 5.3.1)
- Entity, Boundary, and Control Objects (Section 5.3.2)
- Generalization and Specialization (Section 5.3.3).

5.3.1 Analysis 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`, `SessionManager`, `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 5-4 depicts good and bad examples of analysis objects for the `SatWatch` example.

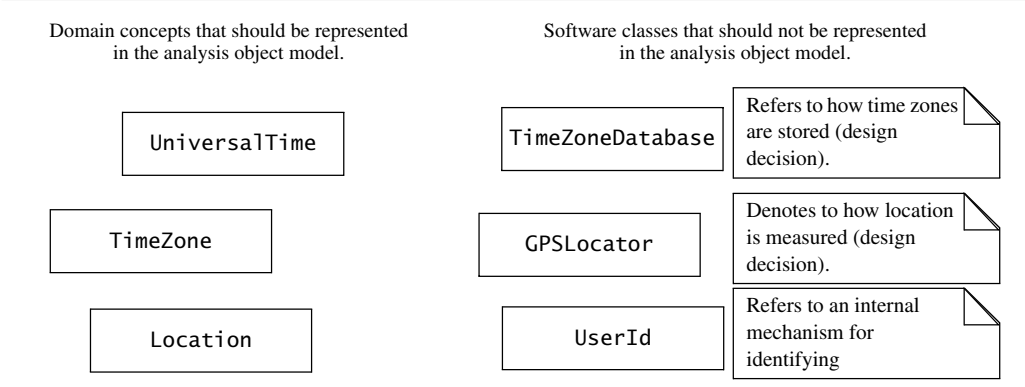


Figure 5-4 Examples and counterexamples of classes in the analysis object model of `SatWatch`.

5.3.2 Entity, Boundary, and Control Objects

The analysis object model consists of entity, boundary, and control objects [Jacobson et al., 1999]. **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; `ChangeDateControl` 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 `LCDDisplay`. 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 5-5, we attach the «control» stereotype to the ChangeDateControl 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.

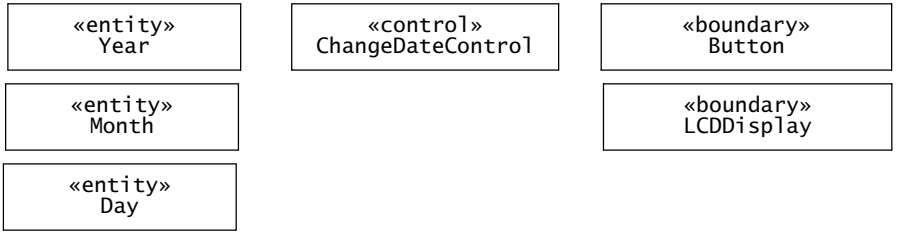


Figure 5-5 Analysis classes for the 2Bwatch example.

5.3.3 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 5-6), 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

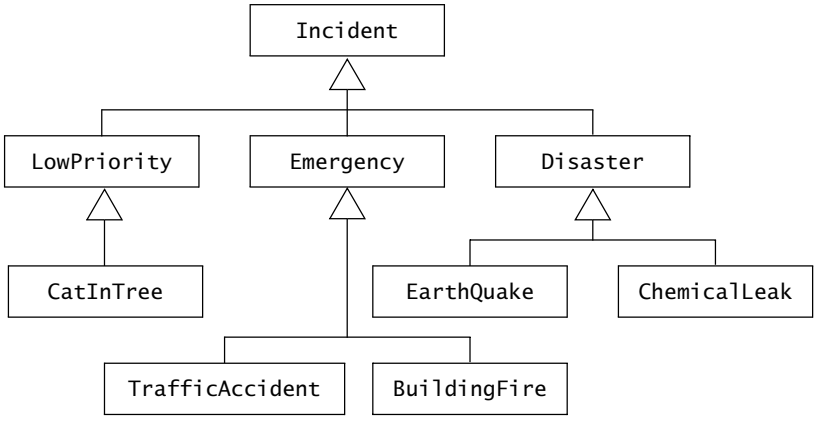


Figure 5-6 An example of a generalization hierarchy (UML class diagram). The top of the hierarchy represents the most general concept, whereas the bottom nodes represent the most specialized concepts.

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`.

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.

5.4 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 (Section 5.4.1)
- Identifying Boundary Objects (Section 5.4.2)
- Identifying Control Objects (Section 5.4.3)
- Mapping Use Cases to Objects with Sequence Diagrams (Section 5.4.4)
- Modeling Interactions among Objects with CRC Cards (Section 5.4.5)
- Identifying Associations (Section 5.4.6)
- Identifying Aggregates (Section 5.4.7)
- Identifying Attributes (Section 5.4.8)
- Modeling State-Dependent Behavior of Individual Objects (Section 5.4.9)

- Modeling Inheritance Relationships (Section 5.4.10)
- Reviewing the Analysis Model (Section 5.4.11).

We illustrate each activity by focusing on the `ReportEmergency` use case of `FRIEND` described in Chapter 4, *Requirements Elicitation*. These activities are guided by heuristics. The quality of their outcome depends on the experience of the developer in applying these heuristics and methods. The methods and heuristics presented in this section are adapted from [De Marco, 1978], [Jacobson et al., 1999], [Rumbaugh et al., 1991], and [Wirfs-Brock et al., 1990].

5.4.1 Identifying Entity Objects

Participating objects (see Section 4.4.6) form the basis of the analysis model. As described in Chapter 4, *Requirements Elicitation*, participating objects are found by examining each use case and identifying candidate objects. Natural language analysis [Abbott, 1983] 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 5-1 provides examples of such mappings by examining the `ReportEmergency` use case (Figure 5-7).

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

Table 5-1 Abbott's heuristics for mapping parts of speech to model components [Abbott, 1983].

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

Use case name	`ReportEmergency`
Entry condition	1. The `FieldOfficer` activates the "Report Emergency" function of her terminal.
Flow of events	2. `FRIEND` responds by presenting a form to the officer. The form includes an emergency type menu (general emergency, fire, transportation), a location, incident description, resource request, and hazardous material fields. 3. The `FieldOfficer` completes the form by specifying minimally the emergency type and description fields. The `FieldOfficer` may also describe possible responses to the emergency situation and request specific resources. Once the form is completed, the `FieldOfficer` submits the form by pressing the "Send Report" button, at which point, the `Dispatcher` is notified. 4. The `Dispatcher` reviews the information submitted by the `FieldOfficer` and creates an `Incident` in the database by invoking the `OpenIncident` use case. All the information contained in the `FieldOfficer`'s form is automatically included in the incident. The `Dispatcher` selects a response by allocating resources to the incident (with the `AllocateResources` use case) and acknowledges the emergency report by sending a `FRIENDgram` to the `FieldOfficer`.
Exit condition	5. The `FieldOfficer` receives the acknowledgment and the selected response.

Figure 5-7 An example of use case, `ReportEmergency` (one-column format).

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. 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 (see Section 5.4.11).

For example, after a first examination of the `ReportEmergency` use case (Figure 5-7), we 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 5-2 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`.

5.4.2 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.

We find the boundary objects of Table 5-3 by examining the `ReportEmergency` use case.

Table 5-3 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.

Note that the IncidentForm is not explicitly mentioned anywhere in the ReportEmergency use case. We identified this object by observing that the Dispatcher needs an interface to view the emergency report submitted by the FieldOfficer and to send back an acknowledgment. The terms used for describing the boundary objects in the analysis model should follow the user terminology, even if it is tempting to use terms from the implementation domain.

We have made progress toward describing the system. We now have included the interface between the actor and the system. We are, however, still missing some significant pieces of the description, such as the order in which the interactions between the actors and the system occur. In the next section, we describe the identification of control objects.

5.4.3 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, we 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 5-4).

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.

In modeling the ReportEmergency use case, we modeled the same functionality by using entity, boundary, and control objects. By shifting from the event flow perspective to a structural perspective, we increased the level of detail of the description and selected standard terms to refer to the main entities of the application domain and the system. In the next section, we

Table 5-4 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`.

construct a sequence diagram using the `ReportEmergency` use case and the objects we discovered to ensure the completeness of our model.

5.4.4 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.

In this section, we model the sequence of interactions among objects needed to realize the use case. Figures 5-8 through 5-10 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 5-8 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 5-8, 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 5-8 an object of class ReportEmergencyForm is created when object of ReportEmergencyControl sends the «create» message and is destroyed once the EmergencyReportForm has been submitted.

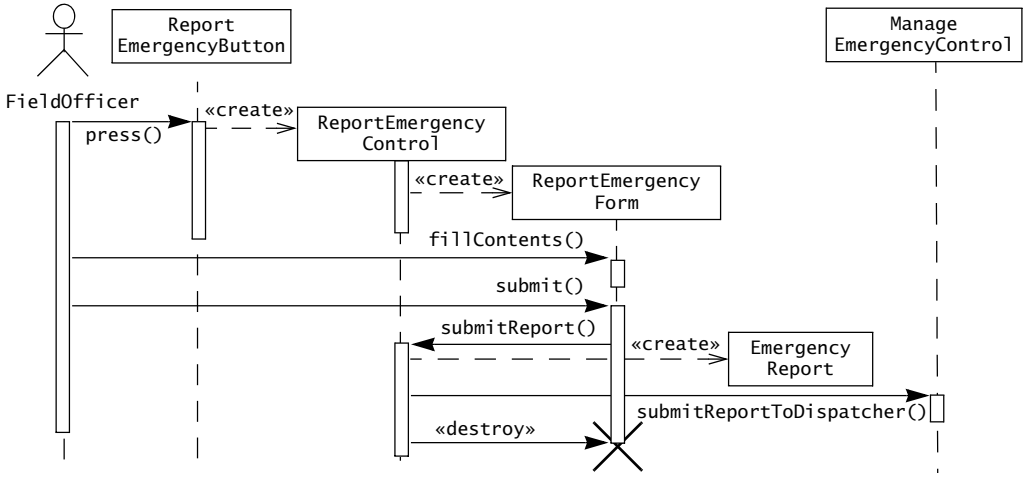


Figure 5-8 Sequence diagram for the ReportEmergency use case.

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).

In Figure 5-9, we discover the entity object Acknowledgment that we forgot during our initial examination of the ReportEmergency use case (in Table 5-2). The Acknowledgment object is different from an AcknowledgmentNotice: Acknowledgment holds the information associated with an Acknowledgment and is created before the AcknowledgmentNotice boundary object. When describing the Acknowledgment object, we also realize that the original ReportEmergency use case (described in Figure 5-7) is incomplete. It only mentions the existence of an Acknowledgment and does not describe the information associated with it. In this case, developers need clarification from the client to define what information is needed in the

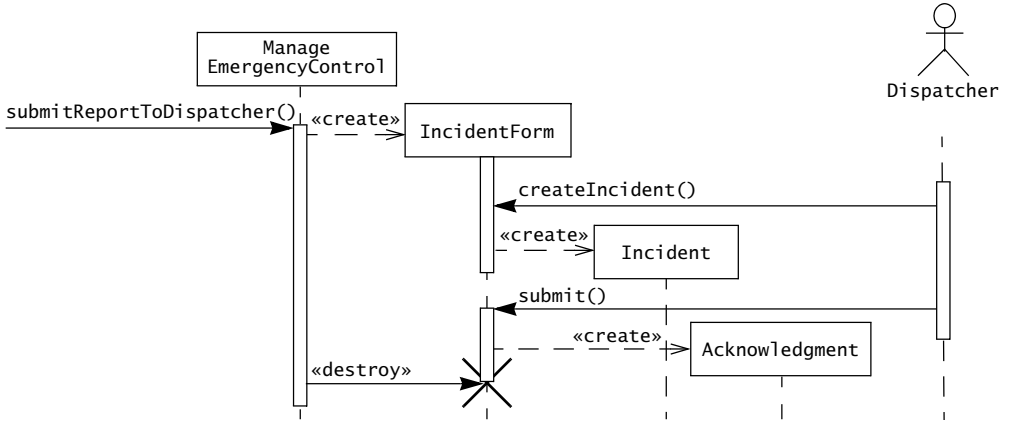


Figure 5-9 Sequence diagram for the `ReportEmergency` use case (continued from Figure 5-8).

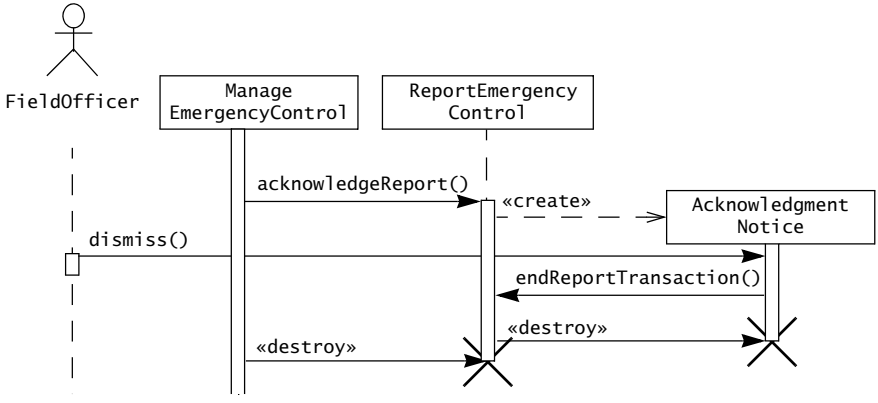


Figure 5-10 Sequence diagram for the `ReportEmergency` use case (continued from Figure 5-9).

`Acknowledgment`. After obtaining such clarification, the `Acknowledgment` object is added to the analysis model (Table 5-5), and the `ReportEmergency` use case is clarified to include the additional information (Figure 5-11).

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.

Use case name	ReportEmergency
Entry condition	1. The FieldOfficer activates the "Report Emergency" function of her terminal.
Flow of events	2. FRIEND responds by presenting a form to the officer. The form includes an emergency type menu (general emergency, fire, transportation), a location, incident description, resource request, and hazardous material fields. 3. The FieldOfficer completes the form by specifying minimally the emergency type and description fields. The FieldOfficer may also describe possible responses to the emergency situation and request specific resources. Once the form is completed, the FieldOfficer submits the form by pressing the "Send Report" button, at which point the Dispatcher is notified. 4. The Dispatcher reviews the information submitted by the FieldOfficer and creates an Incident in the database by invoking the OpenIncident use case. All the information contained in the FieldOfficer's form is automatically included in the Incident. The Dispatcher selects a response by allocating resources to the Incident (with the AllocateResources use case) and acknowledges the emergency report by sending a FRIENDgram to the FieldOfficer. **The Acknowledgment indicates to the FieldOfficer that the EmergencyReport was received, an Incident created, and resources allocated to the Incident. The Acknowledgment includes the resources (e.g., a fire truck) and their estimated arrival time.**
Exit condition	5. The FieldOfficer receives the Acknowledgment and the selected response.

Figure 5-11 Refined ReportEmergency use case. The discovery and addition of the Acknowledgment object to the analysis model revealed that the original ReportEmergency use case did not accurately describe the information associated with Acknowledgments. The refinements are indicated in **boldface**.

Table 5-5 Acknowledgment object for the ReportEmergency use case.

Acknowledgment	Response of a dispatcher to a FieldOfficer's EmergencyReport. By sending an Acknowledgment, the Dispatcher communicates to the FieldOfficer that she has received the EmergencyReport, created an Incident, and assigned resources to it. The Acknowledgment contains the assigned resources and their estimated arrival time.

Sharing operations across use cases allows developers to remove redundancies in the requirements specification and to improve its consistency. Note that clarity should always be given precedence to eliminating redundancy. Fragmenting behavior across many operations unnecessarily complicates the requirements specification.

In analysis, sequence diagrams are used to help identify new participating objects and missing behavior. Because sequence diagrams focus on high-level behavior, implementation

issues such as performance should not be addressed at this point. Given that building interaction diagrams can be time consuming, developers should focus on problematic or underspecified functionality first. Drawing interaction diagrams for parts of the system that are simple or well defined might not look like a good investment of analysis resources, but it should also be done to avoid overlooking some key decisions.

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.

5.4.5 Modeling Interactions among Objects with CRC Cards

An alternative for identifying interactions among objects are **CRC cards** [Beck & Cunningham, 1989]. CRC cards (CRC stands for class, responsibilities, and collaborators) were initially introduced as a tool for teaching object-oriented concepts to novices and to experienced developers unfamiliar with object-orientation. Each class is represented with an index card (called the CRC card). The name of the class is depicted on the top, its responsibilities in the left column, and the names of the classes it needs to accomplish its responsibilities are depicted in the right column. Figure 5-12 depicts two cards for the `ReportEmergencyControl` and the `Incident` classes.

CRC cards can be used during modeling sessions with teams. Participants, typically a mix of developers and application domain experts, go through a scenario and identify the classes that are involved in realizing the scenario. One card per instance is put on the table. Responsibilities

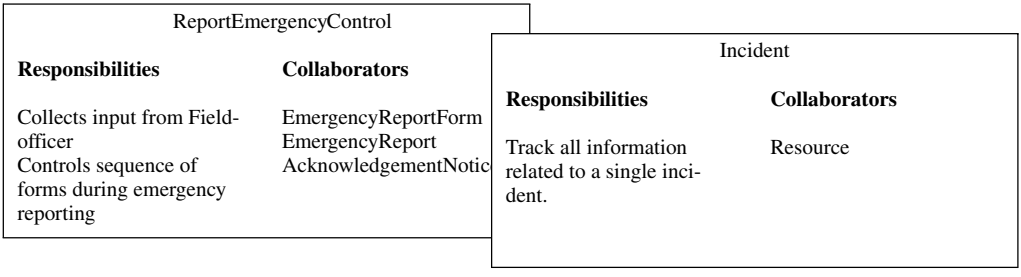


Figure 5-12 Examples of CRC cards for the `ReportEmergencyControl` and the `Incident` classes.

are then assigned to each class as the scenario unfolds and participants negotiate the responsibilities of each object. The collaborators column is filled as the dependencies with other cards are identified. Cards are modified or pushed to the side as new alternatives are explored. Cards are never thrown away, because building blocks for past alternatives can be reused when new ideas are put on the table.

CRC cards and sequence diagrams are two different representations for supporting the same type of activity. Sequence diagrams are a better tool for a single modeler or for documenting a sequence of interactions, because they are more precise and compact. CRC cards are a better tool for a group of developers refining and iterating over an object structure during a brainstorming session, because they are easier to create and to modify.

5.4.6 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. In Section 5.4.8, we discuss the use of class diagrams for representing object attributes.

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 (e.g., `Writes` in Figure 5-13). Association names are optional and need not be unique globally.

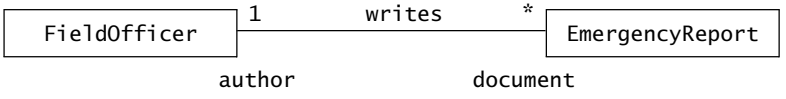


Figure 5-13 An example of association between the `EmergencyReport` and the `FieldOfficer` classes.

- 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 5-1), 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.

The object model will initially include too many associations if developers include all associations identified after examining verb phrases. In Figure 5-14, for example, we identify two relationships: the first between an `Incident` and the `EmergencyReport` that triggered its creation; the second between the `Incident` and the reporting `FieldOfficer`. Given that the `EmergencyReport` and `FieldOfficer` already have an association modeling authorship, the association between `Incident` and `FieldOfficer` is not necessary. Adding unnecessary associations complicates the model, leading to incomprehensible models and redundant information.

Most entity objects have an identifying characteristic used by the actors to access them. `FieldOfficers` and `Dispatchers` have a badge number. `Incidents` and `Reports` are assigned numbers and are archived by date. Once the analysis model includes most classes and associations, the developers should go through each class and check how it is identified by the actors and in which context. For example, are `FieldOfficer` badge numbers unique across the universe? Across a city? A police station? If they are unique across cities, can the `FRIEND` system know about `FieldOfficers` from more than one city? This approach can be formalized by examining each individual class and identifying the sequence of associations that need to be traversed to access a specific instance of that class.

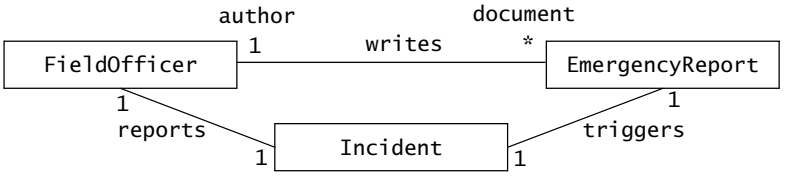


Figure 5-14 Eliminating redundant association. The receipt of an `EmergencyReport` triggers the creation of an `Incident` by a `Dispatcher`. Given that the `EmergencyReport` has an association with the `FieldOfficer` that wrote it, it is not necessary to keep an association between `FieldOfficer` and `Incident`.

5.4.7 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 5-15). 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.

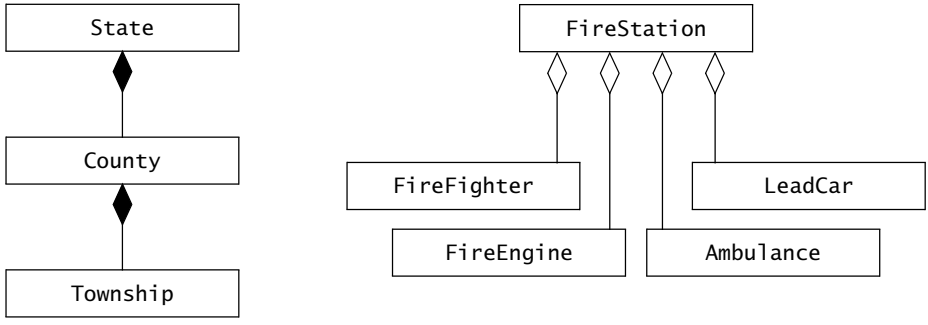


Figure 5-15 Examples of aggregations and compositions (UML class diagram). A `State` is composed of many `Counties`, which in turn is composed of many `Townships`. A `FireStation` includes `FireFighters`, `FireEngines`, `Ambulances`, and a `LeadCar`.

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 5-15, `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.

5.4.8 Identifying Attributes

Attributes are properties of individual objects. For example, an `EmergencyReport`, as described in Table 5-2, 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.

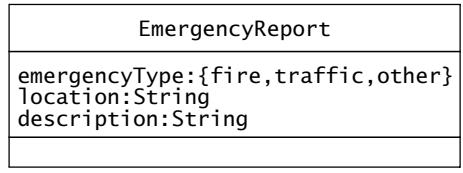


Figure 5-16 Attributes of the `EmergencyReport` class.

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.

Attributes can be identified using Abbott's heuristics (see Table 5-1). In particular, a noun phrase followed by a possessive phrase (e.g., the description of an emergency) or an adjective phrase (e.g., the emergency description) should be examined. In the case of entity objects, any property that must be stored by the system is a candidate attribute.

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[a]

- 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.

a. Adapted from [Rumbaugh et al., 1991].

5.4.9 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 5-17 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.

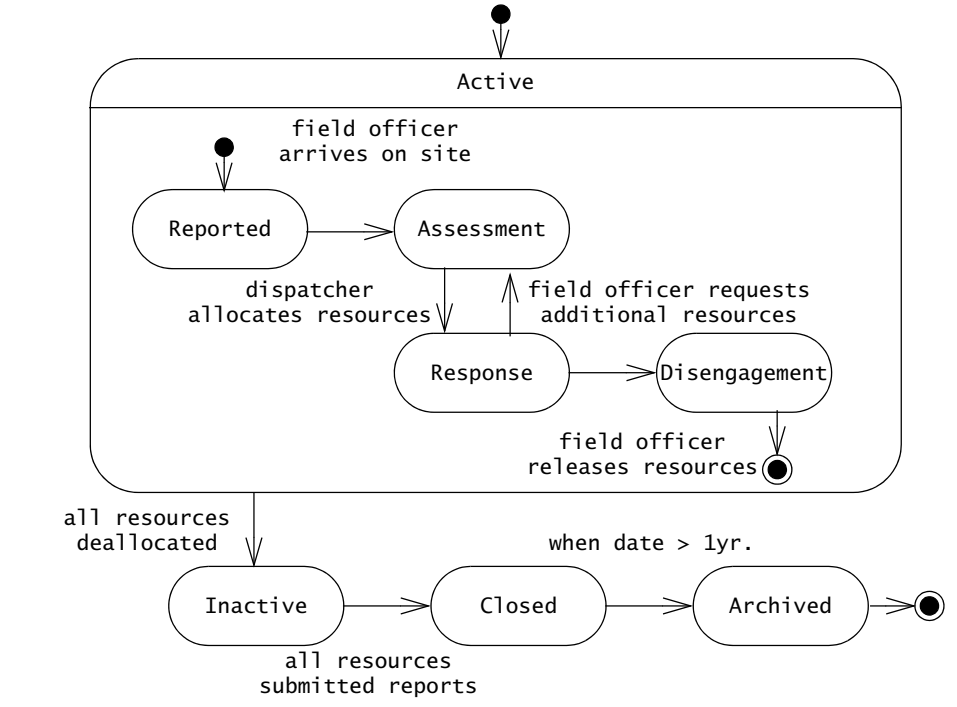


Figure 5-17 UML state machine for `Incident`.

5.4.10 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 5-18).

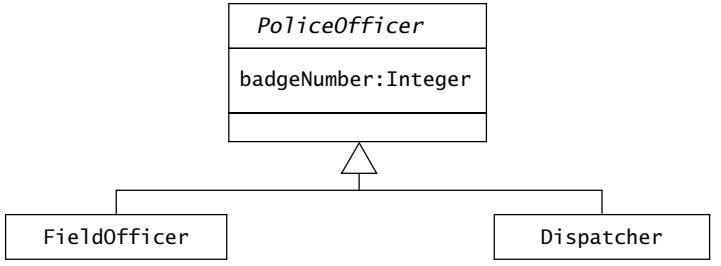


Figure 5-18 An example of inheritance relationship (UML class diagram).

5.4.11 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. Below are example questions adapted from [Jacobson et al., 1999] and [Rumbaugh et al., 1991].

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 *complete*:

- For each object: Is it needed by any use case? In which use case is it created? modified? destroyed? Can it be accessed from a boundary object?

- For each attribute: When is it set? What is its type? Should it be a qualifier?
- For each association: When is it traversed? Why was the specific multiplicity chosen? Can associations with one-to-many and many-to-many multiplicities be qualified?
- For each control object: Does it have the necessary associations to access the objects participating in its corresponding use case?

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?

5.4.12 Analysis Summary

The requirements elicitation activity is highly iterative and incremental. Chunks of functionality are sketched and proposed to the users and the client. The client adds requirements, criticizes existing functionality, and modifies existing requirements. The developers investigate nonfunctional requirements through prototyping and technology studies and challenge each proposed requirement. Initially, requirements elicitation resembles a brainstorming activity. As the description of the system grows and the requirements become more concrete, developers need to extend and modify the analysis model in a more orderly manner to manage the complexity of information.

Figure 5-19 depicts a typical sequence of the analysis activities. The users, developers, and client are involved in developing an initial use case model. They identify a number of concepts and build a glossary of participating objects. These first two activities were discussed in the previous chapter. The remaining activities were covered in this section. The developers classify the participating objects into entity, boundary, and control objects (in *Define entity objects*, Section 5.4.1, *Define boundary objects*, Section 5.4.2, and *Define control objects*, Section 5.4.3). These activities occur in a tight loop until most of the functionality of the system has been identified as use cases with names and brief descriptions. Then the developers construct sequence diagrams to identify any missing objects (*Define interactions*, Section 5.4.4). When all entity objects have been named and briefly described, the analysis model should remain fairly stable as it is refined.

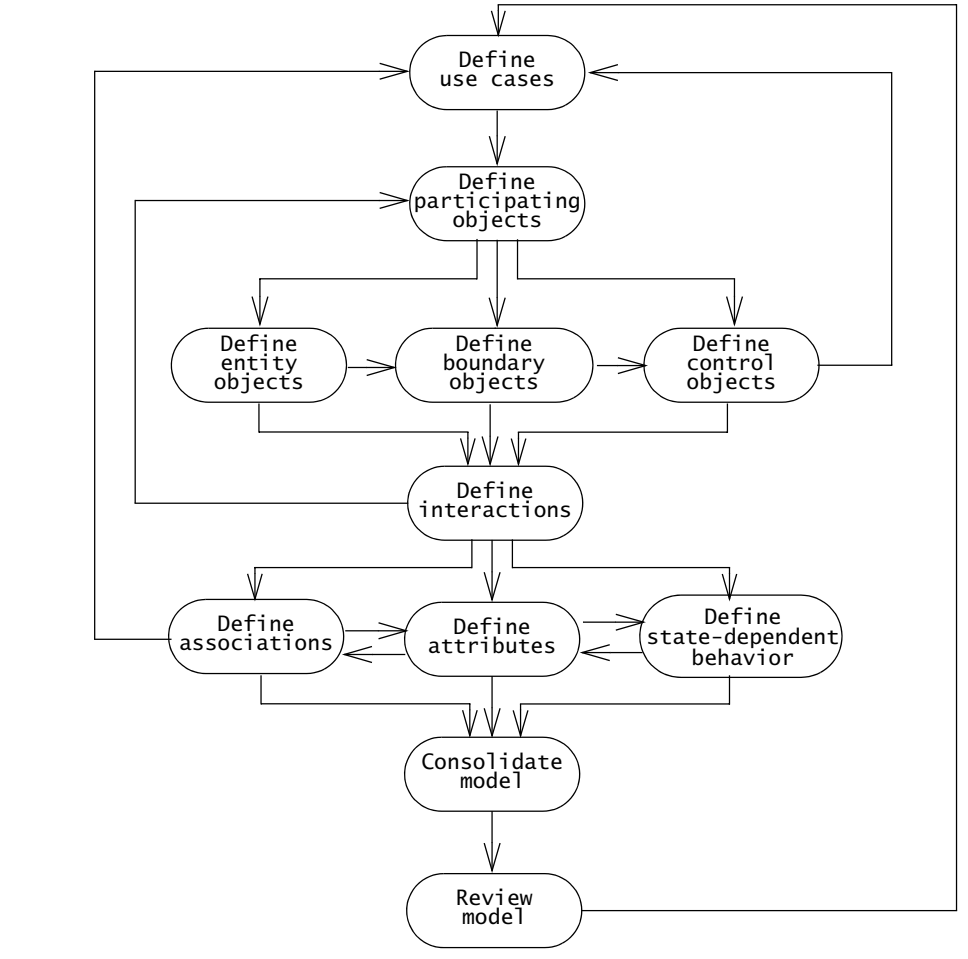


Figure 5-19 Analysis activities (UML activities diagram).

Define associations (Section 5.4.6), *Define attributes* (Section 5.4.8) and *Define state-dependent behavior* (Section 5.4.9) constitute the refinement of the analysis model. These three activities occur in a tight loop during which the state of the objects and their associations are extracted from the sequence diagrams and detailed. The use cases are then modified to account for any changes in functionality. This phase may lead to the identification of an additional chunk of functionality in the form of additional use cases. The overall process is then repeated incrementally for these new use cases.

During *Consolidate model* (Section 5.4.10), the developers solidify the model by introducing qualifiers and generalization relationships and suppressing redundancies. During *Review model* (Section 5.4.11), the client, users, and developers examine the model for

correctness, consistency, completeness, and realism. The project schedule should plan for multiple reviews to ensure high-quality requirements and to provide opportunities to learn the requirements activity. However, once the model reaches the point where most modifications are cosmetic, system design should proceed. There will come a point during requirements where no more problems can be anticipated without further information from prototyping, usability studies, technology surveys, or system design. Getting every detail right becomes a wasteful exercise: some of these details will become irrelevant by the next change. Management should recognize this point and initiate the next phase in the project.

5.5 Managing Analysis

In this section, we discuss issues related to managing the analysis activities in a multi-team development project. The primary challenge in managing the requirements in such a project is to maintain consistency while using so many resources. In the end, the requirements analysis document should describe a single coherent system understandable to a single person.

We first describe a document template that can be used to document the results of analysis (Section 5.5.1). Next, we describe the role assignment to analysis (Section 5.5.2). We then address communication issues during analysis. Next, we address management issues related to the iterative and incremental nature of requirements (Section 5.5.4).

5.5.1 Documenting Analysis

As we saw in the previous chapter, the requirements elicitation and analysis activities are documented in the Requirements Analysis Document (RAD, Figure 5-20). RAD Sections 1 through 3.5.2 have already been written during requirements elicitation. During analysis, we revise these sections as ambiguities and new functionality are discovered. The main effort, however, focuses on writing the sections documenting the analysis object model (RAD Sections 3.5.3 and 3.5.4).

RAD Section 3.5.3, *Object models*, documents in detail all the objects we identified, their attributes, and, when we used sequence diagrams, operations. As each object is described with textual definitions, relationships among objects are illustrated with class diagrams.

RAD Section 3.5.4, *Dynamic models*, documents the behavior of the object model in terms of state machine diagrams and sequence diagrams. Although this information is redundant with the use case model, dynamic models enable us to represent more precisely complex behaviors, including use cases involving many actors.

The RAD, 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 5-20 Overview outline of the Requirements Analysis Document (RAD). See Figure 4-16 for a detailed outline.

5.5.2 Assigning Responsibilities

Analysis requires the participation of a wide range of individuals. The target user provides application domain knowledge. The client funds the project and coordinates the user side of the effort. The analyst elicits application domain knowledge and formalizes it. Developers provide feedback on feasibility and cost. The project manager coordinates the effort on the development side. For large systems, many users, analysts, and developers may be involved, introducing additional challenges during for integration and communication requirements of the project. These challenges can be met by assigning well-defined roles and scopes to individuals. There are three main types of roles: generation of information, integration, and review.

- The **end user** is the application domain expert who generates information about the current system, the environment of the future system, and the tasks it should support. Each user corresponds to one or more actors and helps identify their associated use cases.
- The **client**, an integration role, defines the scope of the system based on user requirements. Different users may have different views of the system, either because they will benefit from different parts of the system (e.g., a dispatcher vs. a field officer) or because the users have different opinions or expectations about the future system. The client serves as an integrator of application domain information and resolves inconsistencies in user expectations.

- The **analyst** is the application domain expert who models the current system and generates information about the future system. Each analyst is initially responsible for detailing one or more use cases. For a set of use cases, the analysis will identify a number of objects, their associations, and their attributes using the techniques outlined in Section 5.4. The analyst is typically a developer with broad application domain knowledge.
- The **architect**, an integration role, unifies the use case and object models from a system point of view. Different analysts may have different styles of modeling and different views of the parts of the systems for which they are not responsible. Although analysts work together and will most likely resolve differences as they progress through analysis, the role of the architect is necessary to provide a system philosophy and to identify omissions in the requirements.
- The **document editor** is responsible for the low-level integration of the document and for the overall format of the document and its index.
- The **configuration manager** is responsible for maintaining a revision history of the document as well as traceability information relating the RAD with other documents (such as the System Design Document; see Chapter 6, *System Design: Decomposing the System*).
- The **reviewer** validates the RAD for correctness, completeness, consistency, and clarity. Users, clients, developers, or other individuals may become reviewers during requirements validation. Individuals that have not yet been involved in the development represent excellent reviewers, because they are more able to identify ambiguities and areas that need clarification.

The size of the system determines the number of different users and analysts that are needed to elicit and model the requirements. In all cases, there should be one integrating role on the client side and one on the development side. In the end, the requirements, however large the system, should be understandable by a single individual knowledgeable in the application domain.

5.5.3 Communicating about Analysis

The task of communicating information is most challenging during requirements elicitation and analysis. Contributing factors include

- *Different backgrounds of participants*. Users, clients, and developers have different domains of expertise and use different vocabularies to describe the same concepts.
- *Different expectations of stakeholders*. Users, clients, and managements have different objectives when defining the system. Users want a system that supports their current work processes, with no interference or threat to their current position (e.g., an improved system often translates into the elimination of current positions). The client

wants to maximize return on investment. Management wants to deliver the system on time. Different expectations and different stakes in the project can lead to a reluctance to share information and to report problems in a timely manner.

- *New teams*. Requirements elicitation and analysis often marks the beginning of a new project. This translates into new participants and new team assignments, and, thus, into a ramp-up period during which team members must learn to work together.
- *Evolving system*. When a new system is developed from scratch, terms and concepts related to the new system are in flux during most of the analysis and the system design. A term may have a different meaning tomorrow.

No requirements method or communication mechanism can address problems related to internal politics and information hiding. Conflicting objectives and competition will always be part of large development projects. A few simple guidelines, however, can help in managing the complexity of conflicting views of the system:

- *Define clear territories*. Defining roles as described in Section 5.5.2 is part of this activity. This also includes the definition of private and public discussion forums. For example, each team may have a discussion database as described in Chapter 3, *Project Organization and Communication*, and discussion with the client is done on a separate client database. The client should not have access to the internal database. Similarly, developers should not interfere with client/user internal politics.
- *Define clear objectives and success criteria.* The codefinition of clear, measurable, and verifiable objectives and success criteria by both the client and the developers facilitates the resolution of conflicts. Note that defining a clear and verifiable objective is a nontrivial task, given that it is easier to leave objectives open-ended. The objectives and the success criteria of the project should be documented in Section 1.3 of the RAD.
- *Brainstorm*. Putting all the stakeholders in the same room and to quickly generate solutions and definitions can remove many barriers in communication. Conducting reviews as a reciprocal activity (i.e., reviewing deliverables from both the client and the developers during the same session) has a similar effect.

Brainstorming, and more generally the cooperative development of requirements, can lead to the definition of shared, ad hoc notations for supporting the communication. Storyboards, user interface sketches, and high-level dataflow diagrams often appear spontaneously. As the information about the application domain and the new system accrue, it is critical that a precise and structured notation be used. In UML, developers employ use cases and scenarios for communicating with the client and the users, and use object diagrams, sequence diagrams, and state machines to communicate with other developers (see Sections 4.4 and 5.4). Moreover, the latest release of the requirements should be available to all participants. Maintaining a live

online version of the requirements analysis document with an up-to-date change history facilitates the timely propagation of changes across the project.

5.5.4 Iterating over the Analysis Model

Analysis occurs iteratively and incrementally, often in parallel with other development activities such as system design and implementation. Note, however, that the unrestricted modification and extension of the analysis model can only result in chaos, especially when a large number of participants are involved. Iterations and increments must be carefully managed and requests for changes tracked once the requirements are baselined. The requirements activity can be viewed as several steps (brainstorming, solidification, maturity) converging toward a stable model.

Brainstorming

Before any other development activity is initiated, requirements is a brainstorming process. Everything—concepts and the terms used to refer to them—changes. The objective of a brainstorming process is to generate as many ideas as possible without necessarily organizing them. During this stage, iterations are rapid and far reaching.

Solidification

Once the client and the developers converge on a common idea, define the boundaries of the system, and agree on a set of standard terms, solidification starts. Functionality is organized into groups of use cases with their corresponding interfaces. Groups of functionality are allocated to different teams that are responsible for detailing their corresponding use cases. During this stage, iterations are rapid but localized.

Maturity

Changes at the higher level are still possible but more difficult, and thus, are made more carefully. Each team is responsible for the use cases and object models related to the functionality they have been assigned. A cross-functional team, the architecture team, made of representatives of each team, is responsible for ensuring the integration of the requirements (e.g., naming).

Once the client signs off on the requirements, modification to the analysis model should address omissions and errors. Developers, in particular the architecture team, need to ensure that the consistency of the model is not compromised. The requirements model is under configuration management and changes should be propagated to existing design models. Iterations are slow and often localized.

The number of features and functions of a system will always increase with time. Each change, however, can threaten the integrity of the system. The risk of introducing more problems with late changes results from the loss of information in the project. The dependencies across functions are not all captured; many assumptions may be implicit and forgotten by the time the

change is made. Often the change responds to a problem, in which case there is a lot of pressure to implement it, resulting in only a superficial examination of the consequence of the change. When new features and functions are added to the system, they should be challenged with the following questions: Were they requested by the client? Are they necessary, or are they embellishments? Should they be part of a separate, focused utility program instead of part of the base system? Are the changes core requirements or optional features? What is the impact of the changes to existing functions in terms of consistency, interface, reliability?

When changes are necessary, the client and developer define the scope of the change and its desired outcome and change the analysis model. Given that a complete analysis model exists for the system, specifying new functionality is easier (although implementing it is more difficult).

5.5.5 Client Sign-Off

The client sign-off represents the acceptance of the analysis model (as documented by the requirements analysis document) by the client. The client and the developers converge on a single idea and agree about the functions and features that the system will have. In addition, they agree on:

- a list of priorities
- a revision process
- a list of criteria that will be used to accept or reject the system
- a schedule and a budget.

Prioritizing system functions allows the developers to understand better the client's expectations. In its simplest form, it allows developers to separate bells and whistles from essential features. It also allows developers to deliver the system in incremental chunks: essential functions are delivered first, additional chunks are delivered depending on the evaluation of the previous chunk. Even if the system is to be delivered as a single, complete package, prioritizing functions enables the client to communicate clearly what is important to her and where the emphasis of the development should be. Figure 5-21 provides an example of a priority scheme.

Each function shall be assigned one of the following priorities

- **High priority**—A high-priority feature must be demonstrated successfully during client acceptance.
- **Medium priority**—A medium-priority feature must be taken into account in the system design and the object design. It will be implemented and demonstrated in the second iteration of the system development.
- **Low priority**—A low-priority feature illustrates how the system can be extended in the longer term.

Figure 5-21 An example of a priority scheme for requirements.

After the client sign off, the requirements are baselined and are used for refining the cost estimate of the project. Requirements continue to change after the sign-off, but these changes are subject to a more formal revision process. The requirements change, whether because of errors, omissions, changes in the operating environment, changes in the application domain, or changes in technology. Defining a revision process up front encourages changes to be communicated across the project and reduces the number of surprises in the later stages. Note that a change process need not be bureaucratic or require excessive overhead. It can be as simple as naming a person responsible for receiving change requests, approving changes, and tracking their implementation. Figure 5-22 depicts a more complex example in which changes are designed

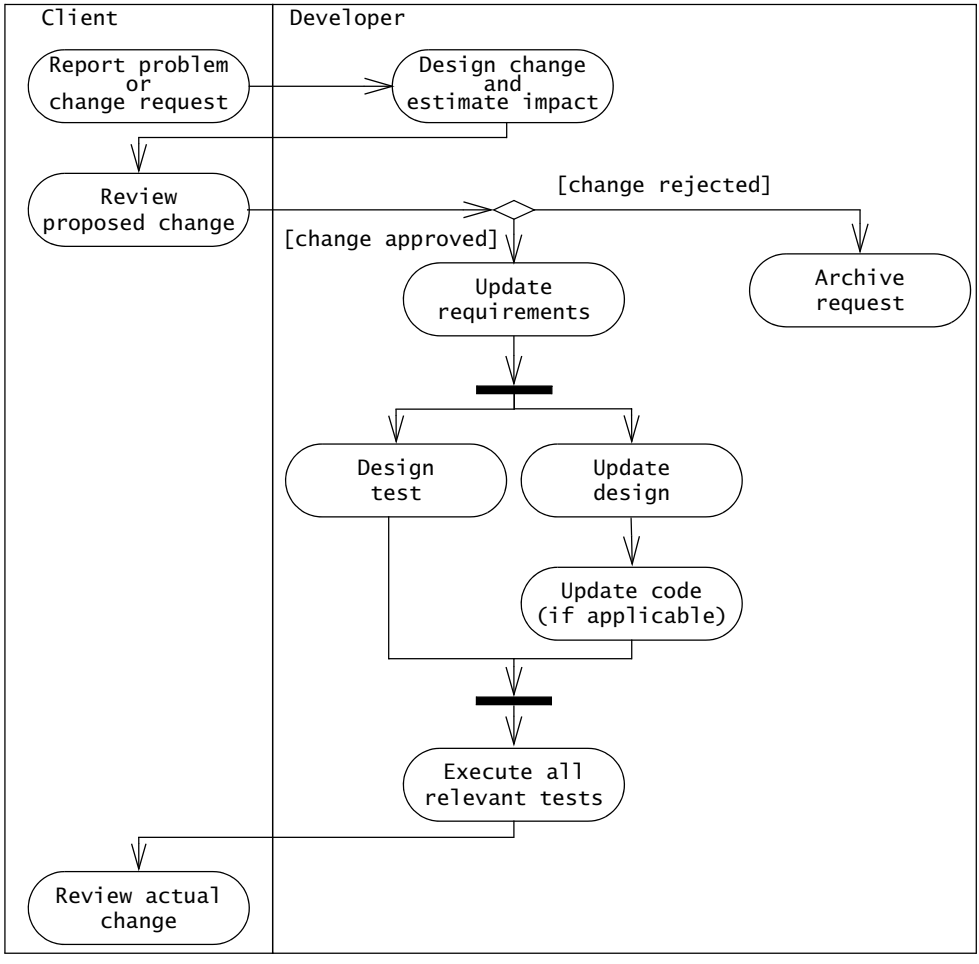


Figure 5-22 An example of a revision process (UML activity diagram).

and reviewed by the client before they are implemented in the system. In all cases, acknowledging that requirements cannot be frozen (but only baselined) will benefit the project.

The list of acceptance criteria is revised prior to sign-off. The requirements elicitation and analysis activity clarifies many aspects of the system, including the nonfunctional requirements with which the system should comply and the relative importance of each function. By restating the acceptance criteria at sign-off, the client ensures that the developers are updated about any changes in client expectations.

The budget and schedule are revisited after the analysis model becomes stable. We describe in Chapter 14, *Project Management*, issues related to cost estimation.

Whether the client sign-off is a contractual agreement or whether the project is already governed by a prior contract, it is an important milestone in the project. It represents the convergence of client and developer on a single set of functional definitions of the system and a single set of expectations. The acceptance of the requirements analysis document is more critical than any other document, given that many activities depend on the analysis model.

5.6 ARENA Case Study

In this section, we apply the concepts and methods described in this chapter to the `ARENA` system. We start with the use case model and the glossary developed in the previous chapter. We identify participating entity, boundary, and control objects, and refine them by adding attributes and associations to the analysis object model. Finally, we identify inheritance relationships and consolidate the analysis object model. In this section, we focus primarily on the `AnnounceTournament` use case.

5.6.1 Identifying Entity Objects

Entity objects represent concepts in the application domain that are tracked by the system. We use the glossary produced during elicitation as a starting point for identifying entity objects in `ARENA`. We identify additional entity objects and their attributes by applying Abbott's heuristics on the use cases. We initially focus only on noun phrases that denote concepts of the application domain. Figure 5-23 depicts the `AnnounceTournament` use case with the first occurrence of noun phrases we identified in **bold**.

Note that we identify entity objects corresponding to actors in the use case model. Actors are concepts in the application domain and are relevant to the system (e.g., for access control or for documenting responsibilities or authorship). In `ARENA`, each legitimate `LeagueOwner` is represented with an object that is used to store data specific to that `LeagueOwner`, such as her contact information, the leagues that she manages, and so on.

Note, also, that not all noun phrases we identified correspond to classes. For example, `name of a tournament` is a noun phrase referring to an attribute of the `Tournament` class. `List of Advertisers` is an association, in this case, between the `League` class and the `Advertiser` class. We can use a few simple heuristics to distinguish between noun phrases that correspond to objects, attributes, and associations:

Name	AnnounceTournament
Flow of events	1. The **LeagueOwner** requests the creation of a **tournament**. 2. The system checks if the LeagueOwner has exceeded the **number of tournaments** in the **league** or in the **arena**. If not, the system presents the LeagueOwner with a form. 3. The LeagueOwner specifies a **name**, **application start and end dates** during which Players can apply to the tournament, **start and end dates** for conducting the tournament, and a **maximum number of Players**. 4. The system asks the LeagueOwner whether an exclusive sponsorship should be sought and, if yes, presents a **list of Advertisers** who expressed the desire to be **exclusive sponsors**. 5. If the LeagueOwner decides to seek an exclusive sponsor, he selects a subset of the **names** of the **proposed sponsors**. 6. The system notifies the selected sponsors about the upcoming tournament and the **flat fee** for exclusive sponsorships. 7. The system communicates their **answers** to the LeagueOwner. 8. If there are interested sponsors, the LeagueOwner selects one of them. 9. The system records the **name** of the exclusive sponsor and charges the flat fee for sponsorships to the **Advertiser's account**. From now on, all **advertisement banners** associated with the tournament are provided by the exclusive sponsor only. 10. If no sponsors were selected (either because no Advertisers were interested or the LeagueOwner did not select any), the advertisement banners are selected at random and charged to each Advertiser's account on a per unit basis. 11. Once the sponsorship issues is closed, the system prompts the LeagueOwner with a **list of groups of Players, Spectators, and Advertisers** that could be interested in the new tournament. 12. The LeagueOwner selects which groups to notify. 13. The system creates a home page in the arena for the tournament. This page is used as an entry point to the tournament (e.g., to provide interested Players with a form to apply for the tournament, and to interest Spectators into watching **matches**). 14. At the application start date, the system notifies each interested user by sending them a link to the main tournament page. The Players can then apply for the tournament with the ApplyForTournament use case until the application end date.

Figure 5-23 Applying Abbott's heuristics for identifying entity objects in the AnnounceTournament use case. The first occurrence of a noun phrase is emphasized in **bold**.

- *Attributes are properties.* Attributes represent a single property of an object. They represent a partial aspect of an object and are incomplete. For example, the name of an `Advertiser` is an attribute that identifies an `Advertiser`. However, it does not include other relevant information about the `Advertiser` (e.g., her current account balance, the type of banners she advertises, etc.) that are represented by other attributes or associations of the `Advertiser` class.
- *Attributes have simple types.* Attributes are properties that often have types such as a number (e.g., maximum number of `Tournaments`), string (e.g., the name of an `Advertiser`), dates (e.g., the application start and end date of a `Tournament`). Properties such as an address, a social security number, and a vehicle identification number are also usually considered as simple types (and hence represented as attributes) because users treat those as simple, atomic concepts. Complex concepts are represented as objects that are related to other objects with associations. For example, an `Account` is an object that is related to the corresponding `Advertiser` and can include a balance, a history of transactions, a credit limit, and other similar properties.
- *Nouns referring to collections are associations, often with implicit ends*. Lists, groups, tables, and sets are represented by associations. For example, `ARENA` prompts the `LeagueOwner` with a list of `Advertisers` that are potentially interested in exclusive sponsorships. This concept can be represented with an association between the `Arena` class and the `Advertiser` class, denoting which `Advertisers` are interested in exclusive sponsorships. Often, the association end is implicit. For example, when sponsorship issues are closed, `ARENA` prompts the `LeagueOwner` with a list of groups of `Players`, `Spectators`, and `Advertisers`. We identify a new class, `InterestGroup`, representing collections of users interested in new events about a league or a game. Then, we identify an association between the `Arena` class and the `InterestGroup` class (corresponding to the word "list") representing all `InterestGroups`. Then, we identify an association between the `InterestGroup` class and the `Player`, `Spectator`, and `Advertisers` classes (corresponding to the word "group"). Finally, we identify additional associations originating from the `InterestGroup` class to other classes representing the interest of the users in the `InterestGroup` (i.e., `League`, `Game`).

Table 5-6 lists the entity objects, their attributes, and their associations that we identified so far from the `AnnounceTournament` use case. We attach the attributes and associations to their relevant classes and write definitions for new classes. Writing definitions has several purposes. First, a name is not specific enough for all stakeholders to share the same understanding about the concept. For example, terms such as `Game` and `Match` can be interchanged in many contexts. In `ARENA`, however, they refer to distinct concepts (i.e., a `Game` represents a set of rules enforced by a piece of software, a `Match` represent a competition among a set of `Players`). Second, objects identified during analysis correspond also to terms in the glossary we started during elicitation. Stakeholders use the glossary throughout development to resolve ambiguities and

establish a standard terminology. Writing short definitions as classes are identified is the best way to prevent ambiguities and misunderstandings. Postponing the writing of definitions results in loss of information and in incomplete definitions.

Table 5-6 Entity objects participating in the `AnnounceTournament` use case identified from noun phrases in the use case. "(?)" denote areas of uncertainty that lead to the questions in Figure 5-24.

Entity Object	Attributes & Associations	Definition
`Account`	• balance • history of charges (?) • history of payments (?)	An `Account` represents the amount currently owed by an `Advertiser`, a history of charges, and payments.
`Advertiser`	• name • leagues of interest for exclusive sponsorships (?) • sponsored tournaments • account	Actor interested in displaying advertisement banners during the `Matches`.
`Advertisement`	• associated game (?)	Image provided by an `Advertiser` for display during matches.
`Arena`	• max number of tournaments • flat fee for sponsorships (?) • leagues *(implied)* • interest groups *(implied)*	An instantiation of the `ARENA` system.
`Game`		A `Game` is a competition among a number of `Players` that is conducted according to a set of rules. In `ARENA`, the term `Game` refers to a piece of software that enforces the set of rules, tracks the progress of each `Player`, and decides the winner.
`InterestGroup`	• list of players, spectators, or advertisers • games and leagues of interests *(implied)*	`InterestGroups` are lists of users in the `ARENA` which share an interest (e.g, for a game or a league). `InterestGroups` are used as mailing lists for notifying potential actors of new events.
`League`	• max number of tournament • game	A `League` represents a community for running `Tournaments`. A `League` is associated with a specific `Game` and `TournamentStyle`. `Players` registered with the `League` accumulate points according to the `ExpertRating` of the `League`.

Table 5-6 *Continued.*

Entity Object	Attributes & Associations	Definition
`LeagueOwner`	• name *(implied)*	The actor creating a `League` and responsible for organizing `Tournaments` within the `League`.
`Match`	• tournament • players	A `Match` is a contest between two or more `Players` within the scope of a `Game`. The outcome of a `Match` can be a single winner and a set of losers or a tie (in which their are no winners or losers). Some `TournamentStyles` may disallow ties.
`Player`	• name *(implied)*	
`Tournament`	• name • application start date • application end date • play start date • play end date • max number of players • exclusive sponsor	A `Tournament` is a series of `Matches` among a set of `Players`. `Tournaments` end with a single winner. The way `Players` accumulate points and `Matches` are scheduled is dictated by the `League` in which the `Tournament` is organized.

The identification of entity objects and their related attributes usually triggers additional questions for the client. For example, when we identify implicit attributes and associations, we should double-check with the client to confirm whether our intuition was correct. In other cases, the ends of an association are ambiguous. We collect all the questions generated by the identification of objects and go back to the client (or the domain expert). Figure 5-24 depicts the questions we have after identifying entity objects participating in the `AnnounceTournament` use case.

Questions for the ARENA client

- What information should be recorded in the advertisers' accounts? For example, should a complete log of the display of each advertisement banner be recorded?
- Do advertisers express the interest for exclusive sponsorships for specific leagues or for the complete arena?
- Should advertisement banners be associated to games (to enable a more intelligent selection of banners when there is no exclusive sponsorship)?
- Does the flat fee for exclusive sponsorship vary across leagues or tournaments?

Figure 5-24 Questions triggered by the identification of entity objects.

5.6.2 Identifying Boundary Objects

Boundary objects represent the interface between the system and the actors. They are identified from the use cases and usually represent the user interface at a coarse level. Do not represent layout information or user interface details such as menus and buttons. User interface mock-ups are much better suited for this type of information. Instead, boundary objects represent concepts such as windows, forms, or hardware artifacts such as workstations. This enables stakeholders to visualize where functionality is available in the system.

Abbott's heuristics do not identify many boundary objects, as they are often left implicit initially. Instead, we scan the `AnnounceTournament` use case (Figure 5-23) and identify where information is exchanged between the actors and the system. We focus both on forms in which actors provide information to the system (e.g., the form used by the `LeagueOwner` to create a `Tournament`) and on notices in which the system provides information to the actors (e.g., a notice received by `Advertisers` requesting sponsorship). As with other objects, we briefly define each class as we identify it. Table 5-7 depicts the boundary objects we identified for `AnnounceTournament` with their definitions. Figure 5-25 depicts our additional questions.

Note that `AnnounceTournament` is a relatively complex use case involving several actors. This yields a relatively large number boundary objects. In practice, a use case can have as few as a single boundary object to represent the interface between the initiating actor and the system. In all cases, however, each use case should have at least one participating boundary object (possibly shared with other use cases).

Table 5-7 Boundary objects participating in the `AnnounceTournament` use case.

Boundary Object	Definition
`TournamentForm`	Form used by the `LeagueOwner` to specify the properties of a `Tournament` during creation or editing.
`RequestSponsorshipForm`	Form used by the `LeagueOwner` to request sponsorships from interested `Advertisers`.
`SponsorshipRequest`	Notice received by `Advertisers` requesting sponsorship.
`SponsorshipReply`	Notice received by `LeagueOwner` indicating whether an `Advertiser` wants the exclusive sponsorship of the tournament.
`SelectExclusiveSponsorForm`	Form used by the `LeagueOwner` to close the sponsorship issue.
`NotifyInterestGroupsForm`	Form used by the `LeagueOwner` to notify interested users.
`InterestGroupNotice`	Notice received by interested users about the creation of a new `Tournament`.

More questions for the ARENA client

- What should we do about sponsors who do not answer?
- How should we advertise a new tournament if there are no relevant interest groups?
- How should users be notified (e.g., E-mail, cell phone, `ARENA` notice box)?

Figure 5-25 Questions triggered by the identification of boundary objects.

5.6.3 Identifying Control Objects

Control objects represent the coordination among boundary and entity objects. In the common case, a single control object is created at the beginning of the use case and accumulates all the information needed to complete the use case. The control object is then destroyed with the completion of the use case.

In `AnnounceTournament`, we identify a single control object called `AnnounceTournamentControl`, which is responsible for sending and collecting notices to `Advertisers`, checking resource availability, and, finally, notifying interested users. Note that, in the general case, several control objects could participate in the same use case, if, for example, there are alternate flows of events to be coordinated, multiple workstations operating asynchronously, or if some control information survives the completion of the use case.

5.6.4 Modeling Interactions Among Objects

We have identified a number of entity, boundary, and control objects participating in the `AnnounceTournament` use case. Along the way, we also identified some of their attributes and associations. We represent these objects in a sequence diagram, depicting the interactions that occur during the use case to identify additional associations and attributes.

In the sequence diagram, we arrange the objects we identified along the top row. We place left-most the initiating actor (i.e., `LeagueOwner`), followed by the boundary object responsible for initiating the use case (i.e., `TournamentForm`), followed by the main control object (i.e., `AnnounceTournamentControl`), and the entity objects (i.e., `Arena`, `League`, and `Tournament`). Any other participating actors and their corresponding boundary objects are on the right of the diagram. We split the sequence diagram associated with `AnnounceTournament` into three figures for space reasons. Figure 5-26 depicts the sequence of interactions leading to the creation of a tournament. Figure 5-27 depicts the workflow for requesting and selecting an exclusive sponsor. Figure 5-28 focuses on the notification of interest groups.

The sequence diagram in Figure 5-26 is straightforward. The `LeagueOwner` requests the creation of the tournament and specifies its initial parameter (e.g., name, maximum number of players). The `AnnounceTournamentControl` instance is created and, if resources allow, a `Tournament` entity instance is created.

The sequence diagram in Figure 5-27 is more interesting as it leads to the identification of additional associations and attributes. When requesting sponsorships, the control object must

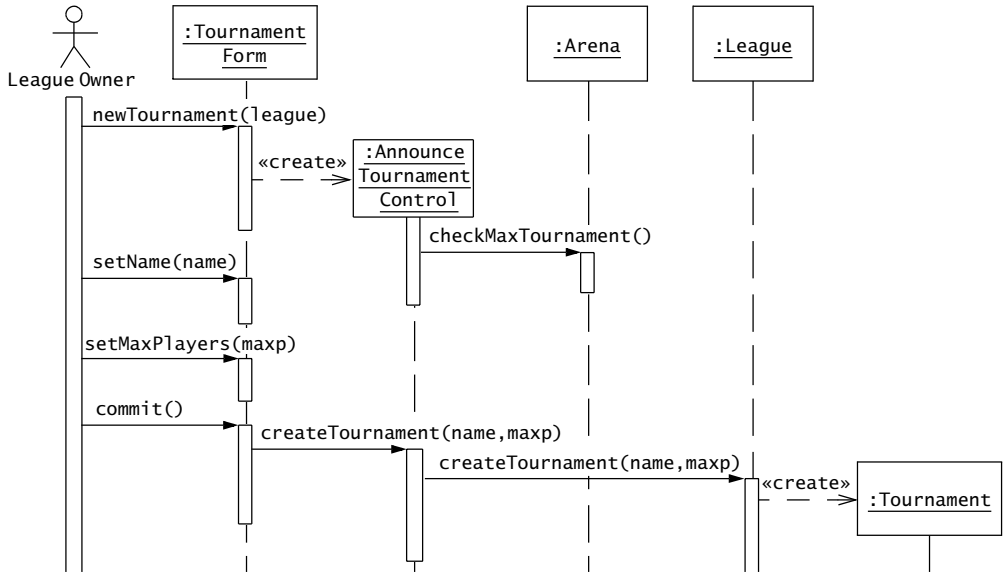


Figure 5-26 UML sequence diagram for `AnnounceTournament`, tournament creation workflow.

first obtain a list of interested sponsors. It requests it from the `Arena` class, which maintains the list of interested sponsors. This entails that the `Arena` class maintains at all times the list of all `Advertisers`, so that it can return this list to the `AnnounceTournamentControl` object (or control objects for other use cases that require the list of all `Advertisers`). To notify an `Advertiser`, we also may need contact information, such as E-mail address, or we may need to create a mailbox for notices within `ARENA`. Consequently, we add an contact attribute to the `Advertiser` class, which initially stores the E-mail address of the `Advertiser` until further devices are supported. Anticipating similar needs for other actors, we also add contact attributes to the `LeagueOwner` and `Player` classes.

When constructing the sequence diagram for notifying interest groups (Figure 5-28), we realize that the use case does not specify how the selected sponsor is notified. Consequently, we add a step in the use case to notify all sponsors who replied about the sponsorship decisions before interest groups are notified. This requires the identification of a new boundary object, `SponsorNotice`. The rest of the interaction does not yield any new discovery, as we already anticipated the need for the `InterestGroup` and the `InterestGroupNotice` classes.

5.6.5 Reviewing and Consolidating the Analysis Model

Now that we have identified most participating objects, their associations, and their attributes, we draw UML class diagrams documenting the results of our analysis so far. As we have identified many objects, we use several class diagrams to depict the analysis object model. We

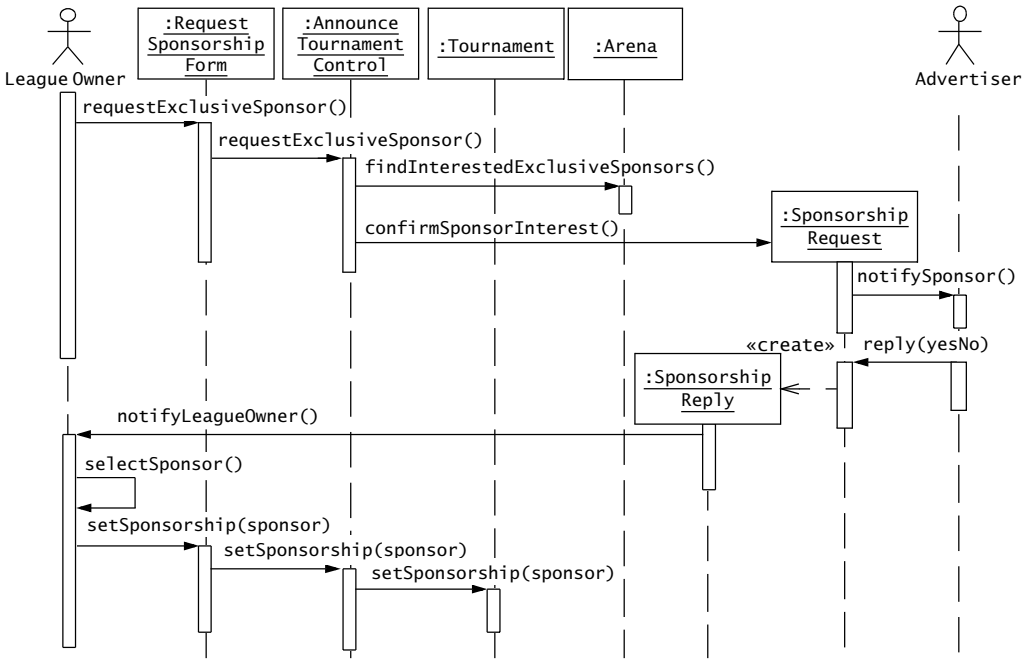


Figure 5-27 UML sequence diagram for `AnnounceTournament` use case, sponsorship workflow.

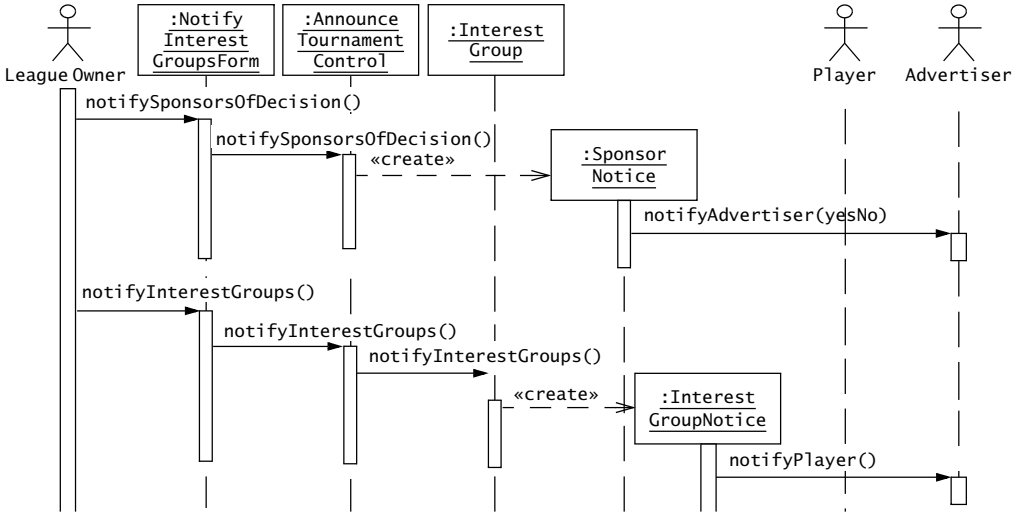


Figure 5-28 UML sequence diagram for `AnnounceTournament` use case, interest group workflow.

use these class diagrams as a visual index into the glossary we developed. Although we should not expect the client or the users to be able to review class diagrams, we can use class diagrams for generating more questions for interviews with the client.

We first focus on the entity objects, since these need to be carefully reviewed by the client as they represent application domain concepts (Figure 5-29). Note that we use the `Arena` class as a root object in the system; the `Arena` class represents a specific instantiation. For example, given an instantiation, it is possible to get a list of all `InterestGroups`, `Advertisers`, `LeagueOwners`, `Games`, and `TournamentStyles` by querying the `Arena` class. Moreover, note that objects are not shared among instantiations. For example, `LeagueOwners` belong to exactly one instantiation of the system. If a user is a `LeagueOwner` in several `ARENA` instantiations of the system, she holds a `LeagueOwner` account in each instantiation. We make these type of choices during analysis based on our interpretation of the problem statement, based on our experience, and based on resources available to build in the system. In all cases, these decisions need to be reviewed and confirmed by the client.

Next, we draw a class diagram depicting the inheritance hierarchies (Figure 5-30). Although UML allows inheritance relationships and associations to coexist in the same diagram, it is good practice during analysis to draw two separate diagrams to depict each type of

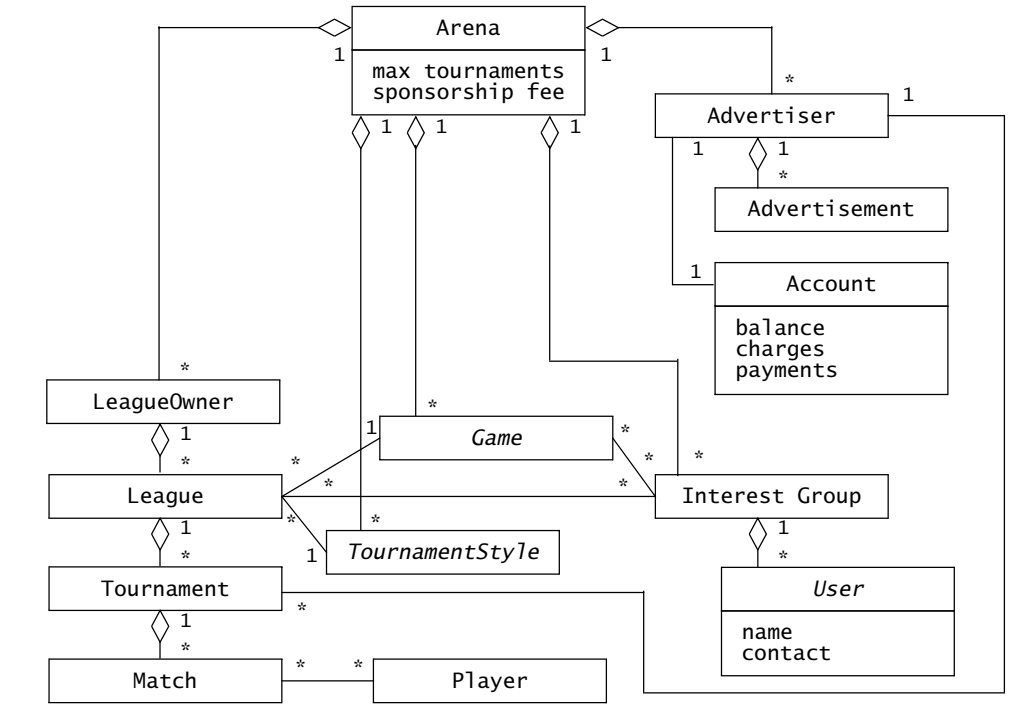


Figure 5-29 Entity objects identified after analyzing the `AnnounceTournament` use case.

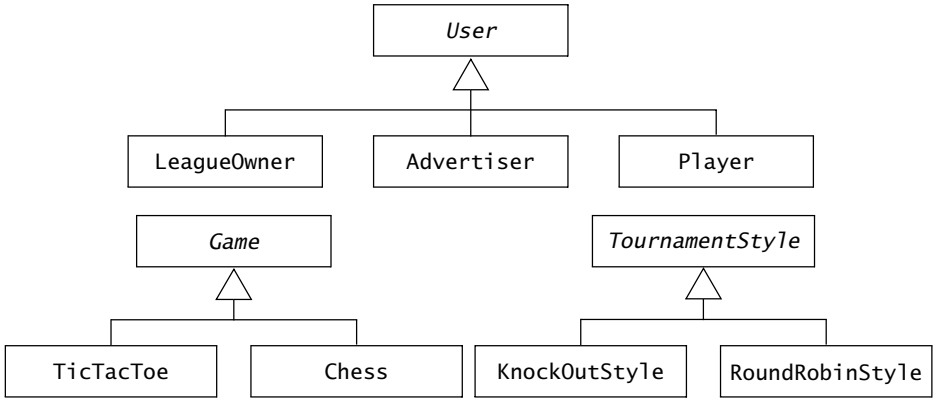


Figure 5-30 Inheritance hierarchy among entity objects of the `AnnounceTournament` use case.

relationship. First, the UML symbols used to denote each type are similar and easily confused. Second, analysts usually focus on inheritance and associations at different times. We will see later, in Chapters 6 through 10, that this is not the case during system design and object design, where it is often necessary to consider both relationships to understand how different classes are related.

Figure 5-30 shows three inheritance hierarchies. First, we identified an abstract class *`User`* through generalization. This enables us to treat common attributes of various users in a more general fashion, including contact information and registration procedures. Note that in the problem statement and in the use cases, we already used the term "user," so we are simply formalizing a concept that was already in use. We identified two other inheritance hierarchies, *`Game`* and *`TournamentStyle`*, identified through specialization to provide examples for both concepts and to provide traceability to the problem statement. The `TicTacToe` and the `Chess` classes are concrete specializations of *`Game`* that embody rules for the games called "tic tac toe" and "chess," respectively. The `KnockOutStyle` and the `RoundRobinStyle` classes are concrete specializations of the *`TournamentStyle`* that provide algorithms for assigning `Players` to knock-out tournaments (in which players need to win to remain in the tournament) and round robin tournaments (in which each player plays all other players exactly once), respectively.

Finally, we draw a class diagram that depicts the associations among the boundary, control, and selected entity objects associated with the use case (Figure 5-31). To generate this diagram from the sequence diagrams, we draw the equivalent communication diagram, with the control object to the left, the boundary objects in the center, and the entity objects on the right. We then replace the iterations with associations, where necessary, so that the objects in the workflow can carry send messages to objects depicted in the sequence diagrams. We then add navigation to the associations to denote the direction of the dependencies: control and boundary

objects usually know about each other, but entity objects do not depend on any control or boundary objects.

Whereas the class diagram in Figure 5-29 focused primarily on the relationships among application domain concepts, the class diagram of Figure 5-31 focuses on the concepts associated with workflow of the use case at a coarse level. The control object acts as the glue among boundary and entity objects, since it represents the coordination and the sequencing among the forms and notices. As indicated in the sequence diagrams in Figures 5-26 through 5-28, the control object also creates several of the boundary objects. The class diagram in Figure 5-31 provides a summary of the objects participating in the use case and the associations traversed during the use case. However, the sequence diagrams provide the complete sequencing information of the workflow.

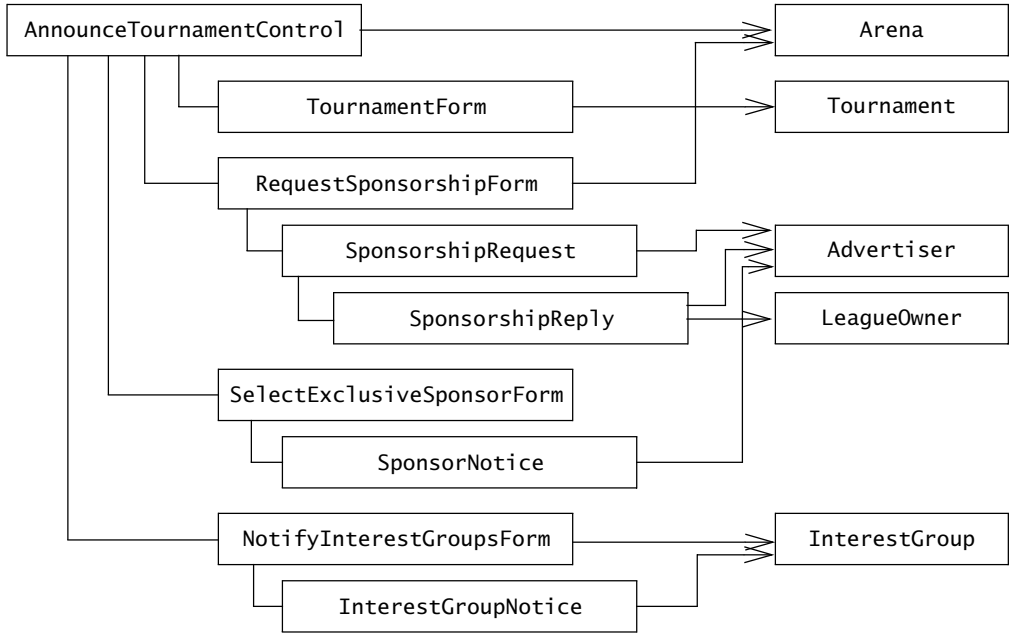


Figure 5-31 Associations among boundary, control, and selected entity objects participating in the `AnnounceTournament` use case.

5.6.6 Lessons Learned

In this section, we developed the part of the analysis object model relevant to the `AnnounceTournament` use case of `ARENA`. We started by identifying entity objects using Abbott's heuristics, then identified boundary and control objects, and used sequence diagrams to find additional associations, attributes, and objects. Finally, we consolidated the object model and depicted it with a series of class diagrams. We learned that:

- Identifying objects, their attributes and associations, takes many iterations, often with the client.
- Object identification uses many sources, including the problem statement, use case model, the glossary, and the event flows of the use cases.
- A nontrivial use case can require many sequence diagrams and several class diagrams. It is unrealistic to represent all discovered objects in a single diagram. Instead, each diagram serves a specific purpose—for example, depicting associations among entity objects, or depicting associations among participating objects in one use case.
- Key deliverables, such as the glossary, should be kept up to date as the analysis model is revised. Others, such as sequence diagrams, can be redone later if necessary. Maintaining consistency at all times, however, is unrealistic.
- There are many different ways to model the same application domain or the same system, based on the personal style and experience of the analyst. This calls for developing style guides and conventions within a project, so that all analysts can communicate effectively.

5.7 Further Readings

The classification of analysis objects into entity, boundary, and control objects has been made popular by the Objectory method [Jacobson et al., 1992]. These concepts originated from the model/view/controller (MVC) paradigm used in the Smalltalk-80 environment and also found their way into the Java Swing user interface framework [JFC, 2009].

CRC cards were introduced by Beck and Cunningham for teaching object-oriented thinking to novices and experienced developers in an OOPSLA paper entitled *A Laboratory For Teaching Object-Oriented Thinking* [Beck & Cunningham, 1989]. CRC cards are used extensively in the responsibility-driven design method from Wirfs-Brock [Wirfs-Brock et al., 1990].

Object-oriented analysis and design has evolved from many different sets of heuristics and terminologies. Modeling, like programming, is a craft, and requires much experience and willingness to make mistakes (hence the importance of client and user feedback). *Object-Oriented Modeling and Design* [Rumbaugh et al., 1991] provides an excellent guide to novices for class modeling. A more recent book, *Applying UML and Patterns* [Larman, 2005], provides a comprehensive treatment of object-oriented analysis and design, including use case modeling and reusing design patterns. For dynamic modeling with state machines, *Doing Hard Time: Using Object Oriented Programming and Software Patterns in Real Time Applications* [Douglass, 1999] provides detailed information and modeling heuristics on the topic.

5.8 Exercises

5-1 Consider a file system with a graphical user interface, such as Macintosh's Finder, Microsoft's Windows Explorer, or Linux's KDE. The following objects were identified from a use case describing how to copy a file from a floppy disk to a hard disk: `File`, `Icon`, `TrashCan`, `Folder`, `Disk`, `Pointer`. Specify which are entity objects, which are boundary objects, and which are control objects.

5-2 Assuming the same file system as before, consider a scenario consisting of selecting a `File` on a floppy, dragging it to `Folder` and releasing the mouse. Identify and define at least one control object associated with this scenario.

5-3 Arrange the objects listed in Exercises 5-1 and 5-2 horizontally on a sequence diagram, the boundary objects to the left, then the control object you identified, and finally, the entity objects. Draw the sequence of interactions resulting from dropping the file into a folder. For now, ignore the exceptional cases.

5-4 Examining the sequence diagram you produced in Exercise 5-3, identify the associations between these objects.

5-5 Identify the attributes of each object that are relevant to this scenario (copying a file from a floppy disk to a hard disk). Also consider the exception cases "There is already a file with that name in the folder" and "There is no more space on disk."

5-6 Consider the object model in Figure 5-32 (adapted from [Jackson, 1995]):

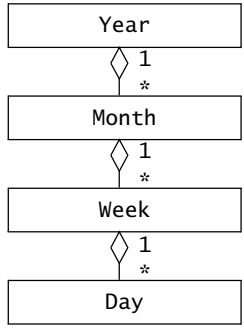


Figure 5-32 A naive model of the Gregorian calendar (UML class diagram).

Given your knowledge of the Gregorian calendar, list all the problems with this model. Modify it to correct each of them.

5-7 Consider the object model of Figure 5-32. Using association multiplicity only, can you modify the model such that a developer unfamiliar with the Gregorian calendar could deduce the number of days in each month? Identify additional classes if necessary.

5-8 Consider a traffic light system at a four-way crossroads (two roads intersecting at right angles). Assume the simplest algorithm for cycling through the lights (e.g., all traffic on one road is allowed to go through the crossroad, while the other traffic is stopped). Identify the states of this system and draw a state machine describing them. Remember that each individual traffic light has three states (green, yellow, and red).

5-9 From the sequence diagram Figure 2-34, draw the corresponding class diagram. Hint: Start with the participating objects in the sequence diagram.

5-10 Consider the addition of a nonfunctional requirement stipulating that the effort needed by `Advertisers` to obtain exclusive sponsorships should be minimized. Change the `AnnounceTournament` (Figure 5-23) and the `ManageAdvertisements` use case (solution of Exercise 4-12) so that the `Advertiser` can specify preferences in her profile so that exclusive sponsorships can be decided automatically by the system.

5-11 Identify and write definitions for any additional entity, boundary, and control objects participating in the `AnnounceTournament` use case that were introduced by realizing the change specified in Exercise 5-10.

5-12 Update the class diagrams of Figure 5-29 and Figure 5-31 to include the new objects you identified in Exercise 5-11.

5-13 Draw a state machine describing the behavior of the `AnnounceTournamentControl` object based on the sequence diagrams of Figures 5-26 through 5-28. Treat the sending and receiving of each notice as an event that triggers a change of state.

References

[Abbott, 1983] R. Abbott, "Program design by informal English descriptions," *Communications of the ACM*, Vol. 26, No. 11, 1983.

[Beck & Cunningham, 1989] K. Beck & W. Cunningham, "A laboratory for teaching object-oriented thinking," *OOPSLA'89 Conference Proceedings*, New Orleans, LA, Oct. 1–6, 1989.

[De Marco, 1978] T. De Marco, *Structured Analysis and System Specification*, Yourdon, New York, 1978.

[Douglass, 1999] B. P. Douglass, *Doing Hard Time: Using Object Oriented Programming and Software Patterns in Real Time Applications*, Addison-Wesley, Reading, MA, 1999.

[Jackson, 1995] M. Jackson, *Software Requirements & Specifications: A Lexicon of Practice, Principles and Prejudices*, Addison-Wesley, Reading, MA, 1995.

[Jacobson et al., 1992] I. Jacobson, M. Christerson, P. Jonsson, & G. Overgaard, *Object-Oriented Software Engineering—A Use Case Driven Approach,* Addison-Wesley, Reading, MA, 1992.

[Jacobson et al., 1999] I. Jacobson, G. Booch, & J. Rumbaugh, *The Unified Software Development Process*, Addison-Wesley, Reading, MA, 1999.

[JFC, 2009] *Java Foundation Classes*, JDK Documentation, Javasoft, 2009.

[Larman, 2005] C. Larman, *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design*, 3rd ed., Prentice Hall, Upper Saddle River, NJ, 2005.

[Rumbaugh et al., 1991] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, & W. Lorensen, *Object-Oriented Modeling and Design*, Prentice Hall, Englewood Cliffs, NJ, 1991.

[Wirfs-Brock et al., 1990] R. Wirfs-Brock, B. Wilkerson, & L. Wiener, *Designing Object-Oriented Software*, Prentice Hall, Englewood Cliffs, NJ, 1990.