7. Requirements Analysis

"Details count."

- **Peter Weinberger, Bell Labs**

Requirements analysis results into a model of the system that aims to be correct, complete, consistent, and verifiable. Developers formalize the system specification produced during requirements elicitation and examine in more detail boundary conditions and exceptional cases. Developers correct and clarify the system specification if any errors or ambiguities are found. The client and the user may be involved in this process, especially when the system specification needs to be changed and additional information gathered.

In object-oriented analysis, developers build an analysis model describing the application domain. For example, the analysis model of a watch describes time according to the watch (e.g., 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 (e.g., How does the watch owner resets the time? How does the watch owner resets the day of the week?). Developers use the analysis model, together with nonfunctional requirements, to establish the architecture of the system during high-level design (see Chapter 8, *System Design*).

In this chapter, we discuss in more detail requirements analysis. We focus on the identification of objects, their behavior, their relationships, their classification, and their organization. We review briefly non object-oriented analysis presentations and methods. Finally, we describe management issues related to requirements analysis in the context of a 40 person project such as PROSE.

7.1. Introduction: an optical illusion

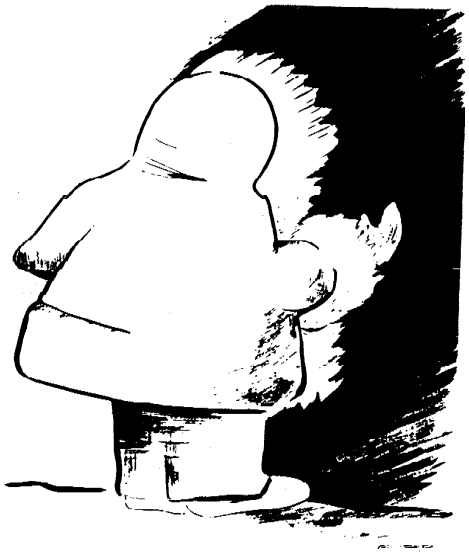

FIGURE 80. Ambiguity: what is this?

Consider Figure 80: what do you see? An eskimo peering into a cave? In Figure 81, the class diagram on the left would then be the corresponding analysis model. You could spend a lot of effort detailing this object model to account for the details in the drawing of Figure 80. However, instead of seeing an eskimo, you might have seen an indian head, in which case the object model in the right of Figure 81 is what you would have derived from this description.

If the drawing in Figure 80 had been a system specification, which models should you have constructed? The eskimo or the indian head? Formalization indicates areas of ambiguity and omissions in the system specification. Developers address ambiguities and omissions by eliciting more information from the users and the client. Requirements elicitation and requirements analysis are iterative and incremental activities.

Requirements analysis focuses on producing a model of the system (hereafter the analysis model) that is correct, complete, consistent, and verifiable. Requirements analysis is different from requirements elicitation in that developers focus on structuring and formalizing the information gathered from users (Figure 82). Although the analysis model may not be not be understandable to the users and the client, it helps the developers to verify the consistency and completeness of the requirements. Requirements elicitation and

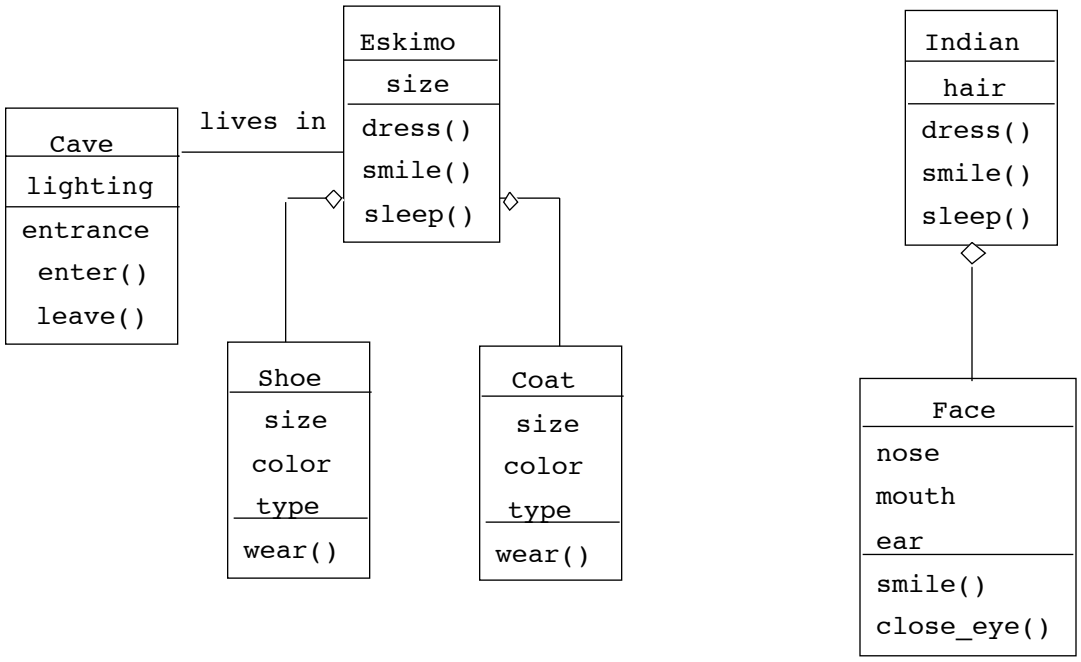

FIGURE 81. Eskimo and indian analysis models of drawing Figure 80.

analysis are often conducted iteratively and incrementally, enabling developers to gather more information from the users in case omissions or ambiguities are found.

There is a natural tendency from 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 avoid confrontation with reality or among peers. Unfortunately, difficult decisions will need to be made eventually, often at higher cost once the development of the system has started and intrinsic problems are discovered during testing, or worse, during user evaluation. Translating a system specification into a formal (or semi-formal) model forces difficult issues to be identified and resolved early in the process.

In the previous chapter, we described how to elicit requirements from the users and describe them as use cases and scenarios using UML notations. In this chapter, we describe how to identify objects from use cases, identify their behavior with sequence diagrams, model their associations with class diagrams, and their individual behavior with state charts (Section 7.3). The object model produced during requirements analysis constitutes the analysis model. The class diagrams, sequence diagrams, and statechart diagrams constitute

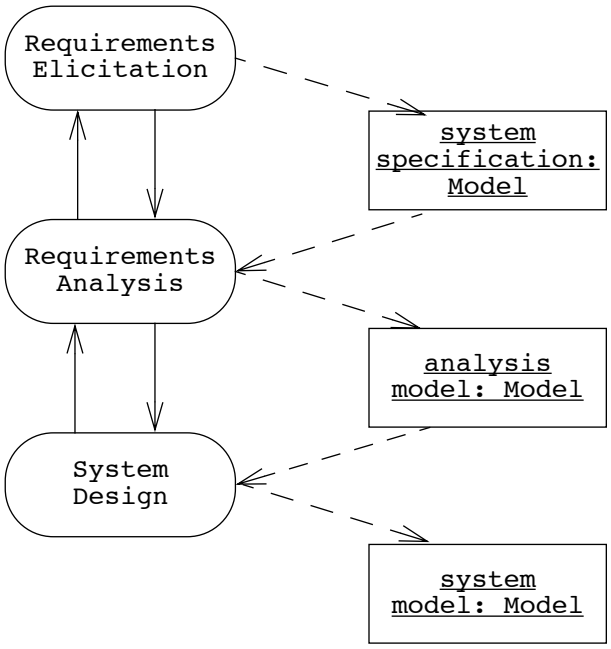

FIGURE 82. Products of requirements elicitation and requirements analysis (UML activity diagram).

different views of the analysis model (see Figure 83). We also survey other notations that are used during requirements analysis (Section 7.3) and illustrate how to manage object-oriented requirements analysis (Section 7.4).

7.2. Analysis models and views

In this section, we survey notations for building analysis models. We first describe the notations used throughout the book, UML sequence diagrams (Section 7.2.1) and UML statechart diagrams (Section 7.2.2). Sequence diagrams describe a pattern of interactions among a number of objects. Statecharts describe the behavior of a single object. In Section 7.3, we discuss the process of transforming a set of use cases into an object model, sequence diagrams and statecharts to describe its behavior and class diagrams to describe its structure. The UML class diagram notation that we use to describe object models has been introduced in Chapter 2, *Introduction to UML*.

Many other notations have been proposed and used, of which we describe dataflow diagrams, decision methods, and Z to illustrate the broader range of perspectives. Dataflow

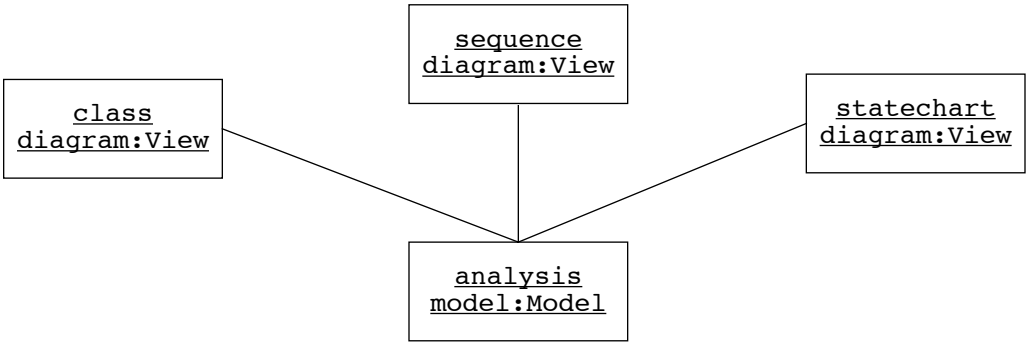

FIGURE 83. Class diagrams, sequence diagrams, and statechart diagrams are three different views of the analysis model.

diagrams (Section 7.2.3) are still a popular notation used for in the development of data processing systems. They were introduced in the Structured Analysis method by De Marco [De Marco, 1978]. Dataflow diagrams depict a system in terms of data pipelines and data processing stages. Decision tables (Section 7.2.4) are used to represent behavior that depends on a complex combinations of several conditions. Decision tables are useful for specifying state driven behavior precisely and provide a compact alternative to state diagrams. Z schemas (Section 7.2.5) enable the analyst to specify a system using a formal notation. This allows a system to be defined precisely without resorting to pseudo code or a programming language. Formal specifications tend to be compact and much more accurate than traditional methods, at the cost of additional training and resources. In development projects were the correct functioning of a system is critical (e.g., a railroad traffic control system), such investment in resources is justifiable.

7.2.1. UML sequence diagrams

Sequence diagrams describe a pattern of interactions among a set of objects. An object interacts with another object by sending ***messages***. The reception of a message by an object triggers the execution of an operation which in turn may send messages to other objects.

Arguments may be passed along with a message and are bound to the parameters of the executing operation.

UML definitions related to sequence diagrams:

- ***Sequence diagram -*** A diagram that shows object interactions arranged in time sequence. In particular, it shows the objects participating in the interaction and the sequence of messages exchanged. Unlike a collaboration diagram, a sequence diagram includes time sequences but does not include object relationships. A sequence diagram can exist in a generic form (describes all possible scenarios) and in an instance form (describes one actual scenario). Sequence diagrams and collaboration diagrams express similar information, but show it in different ways (…).
- ***Message*** - A communication between objects that conveys information with the expectation that activity will ensue. The receipt of a message is normally considered an event.
- ***Argument*** - A specific value corresponding to a parameter. Synonym: actual parameter. Contrast: parameter.

For example, let us consider the case of a digital watch with two buttons (hereafter 2Bwatch). Setting the time on 2Bwatch requires the user to first press both buttons simultaneously, after which 2Bwatch enters the set time mode. In the set time mode, 2Bwatch blinks the number being changed (e.g., the hours, the minutes, or the seconds, day, month, year). Initially, when the user enters the set time mode, the hours are blinking. If the user presses the first button, the next number will blink (e.g, if the hours are blinking and the user presses the first button, the hours will stop blinking and the minutes will start blinking. If the user presses the second button, the blinking number will be incremented by one unit. If the blinking number reaches the end of its range, it is reset to the beginning of its range (e.g., assume the minutes are blinking and its current value is 59, its new value will be set to 0 if the user presses the second button). The user exits the set time mode by pressing both buttons simultaneously. Figure 84 depicts a sequence diagram for the case of a user setting his 2Bwatch one minute ahead.

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

Sequence diagrams are used to describe use cases (i.e., all possible interactions) and scenarios (i.e., one possible interaction, as in Figure 84). Usually, sequence diagrams are drawn for a prototypical case to discover new operations and missing attributes during the design phase. When describing all possible interactions, sequence diagrams also provide notations for conditionals and iterators. A condition on a message send is denoted by an expression in brackets before the message name (see `op1` and `op2` in Figure 85). If the

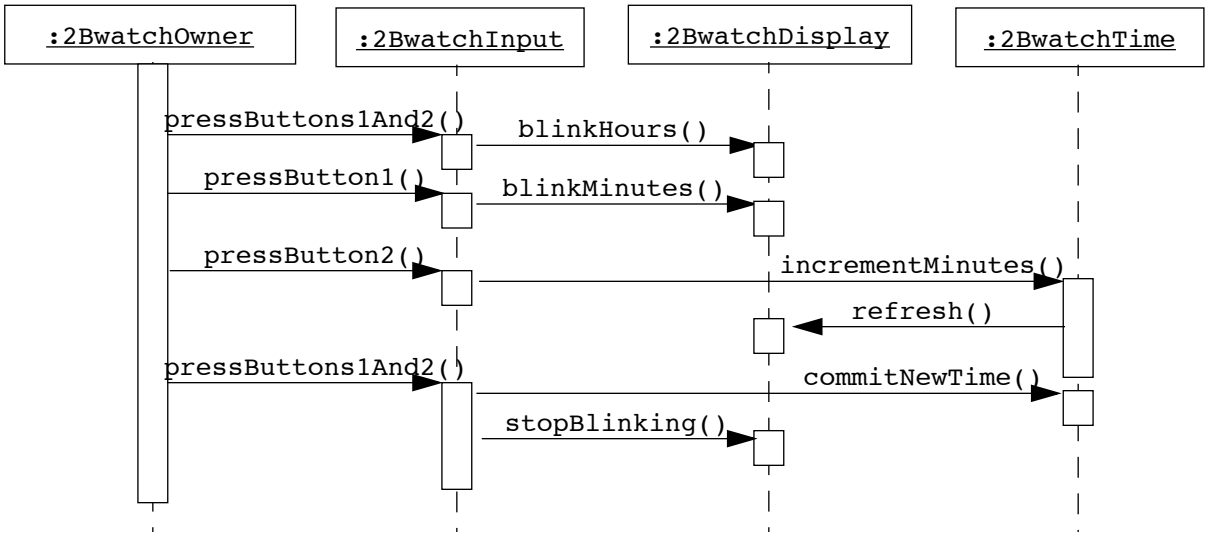

FIGURE 84. Example of sequence diagrams: setting the time on 2Bwatch.

expression is true, the message is sent. Repetitive invocation of a message is denoted by a * before the message name (see `op3` in Figure 85)

7.2.2. UML statechart diagrams

A ***statechart*** is a notation provided by UML to describe the sequence of states an object or an interaction goes through in response to external events. Statecharts are extensions of the traditional flat state machines model. On the one hand, statecharts provide notations for nesting states and state machines (i.e., a state can be described by a state machine). On the other hand, statecharts provide notations for binding transitions with message sends and conditions on objects. UML statecharts were inspired by Harel's statecharts [Harel, 1987]. A statechart is equivalent to a traditional Mealy or Moore state machine.

A state is represented by a rounded rectangle. A transition is represented by stick arrows relating two states. States are labeled with their name and are optionally expanded. A small

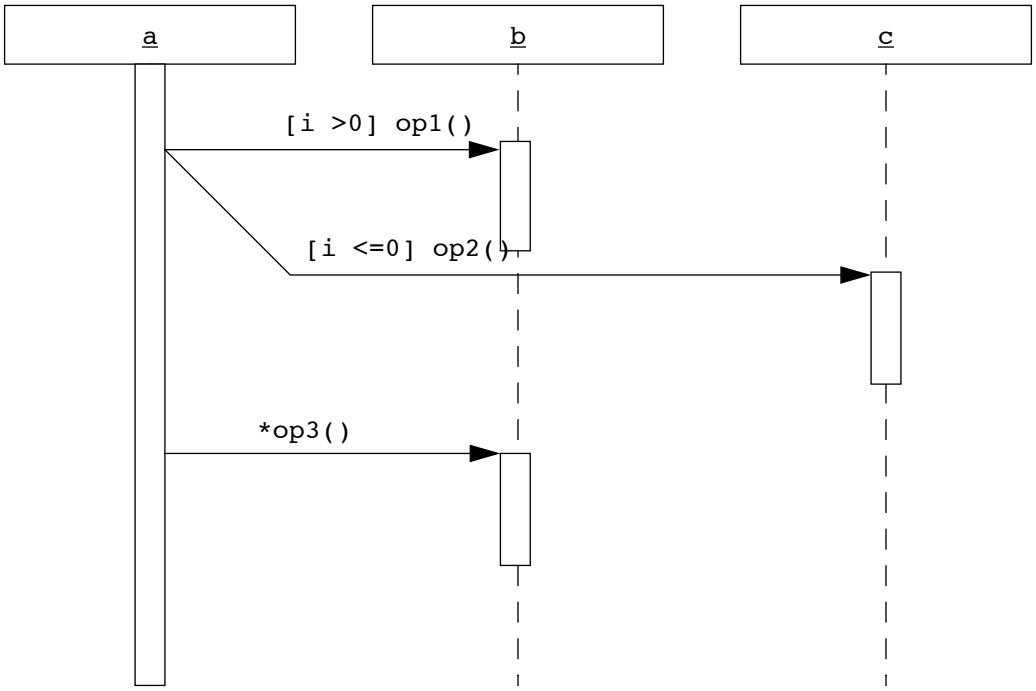

FIGURE 85. Examples of conditions and iterators in sequence diagrams.

solid black circle indicates the initial state. A circle surrounding a small solid black circle indicates a final state.

UML definitions related to statechart diagrams:

- ***Statechart-*** a diagram that shows a state machine.
- ***State machine*** - A behavior that specifies the sequences of states that an object or an interaction goes through during its life in response to events, together with its responses and actions.
- ***State*** - A condition or situation during the life of an object during which it satisfies some condition, performs some activity, or waits for some event. Contrast: state [OMA].
- ***Transition*** - A relationship between two states indicating that an object in the first state will perform certain specified actions and enter the second state when a specified event occurs and specified conditions are satisfied. On such a change of state the transition is said to fire.
- ***Action*** - The specification of an executable statement that forms an abstraction of a computational procedure. An action results in a change in the state of the model, and is realized by sending a message to an object or modifying a value of an attribute.

For example, Figure 86 displays a statechart for the 2Bwatch example for which we constructed a sequence diagram (see Figure 84). At the highest level of abstraction, 2Bwatch has two states, `MeasureTime` and `SetTime`. 2Bwatch changes states when the user presses and releases both buttons simultaneously. When 2Bwatch is first powered, it is in the `SetTime` state. This is indicated by the small solid black circle which represents the initial state. A final state would have been depicted with a circle surrounding a small solid black circle.

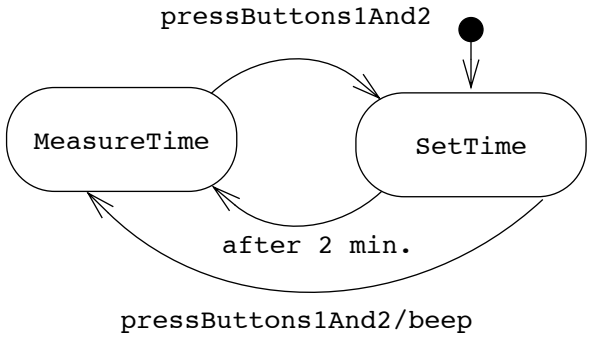

FIGURE 86. Statechart diagram for `2Bwatch` set time function.

Transitions model changes of state triggered by external events, conditions, or time. For example in Figure 86, there are three transitions: two are triggered by the `pressButton1And2` event and the other one is triggered by the passage of time (**`after`** `2 minutes`). Transitions may have actions associated with them. For example in Figure 86, 2Bwatch beeps when the user correctly switches back to the `MeasureTime` state. An action can be realized by one or more operations.

The statechart diagram in Figure 86 does not represent the details of measuring or setting the time. These details have been abstracted away from the toplevel statechart diagram and can be modeled separately using either internal transitions or a nested statechart. Internal transitions (Figure 87)are transitions that remain within a single state. They can also have actions associated with them. Entry and exit are displayed as an internal transition given that their actions do not depend on the originating and destination states.

Nested statecharts (Figure 88) can be used instead of internal transitions. In Figure 88, the current number is modeled as nested states, while actions corresponding to modifying the current number are still modeled using internal transitions. Note that each state could be modeled as a nested statechart (e.g., the `BlinkHour` statechart would have twenty four sub

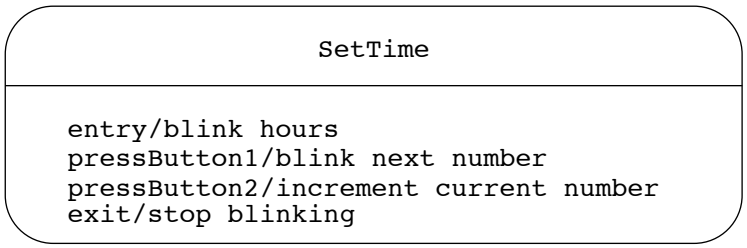

FIGURE 87. Internal transitions associated with the `SetTime` state.

states which correspond to the hours in the day, transitions between these states would correspond to pressing the second button).

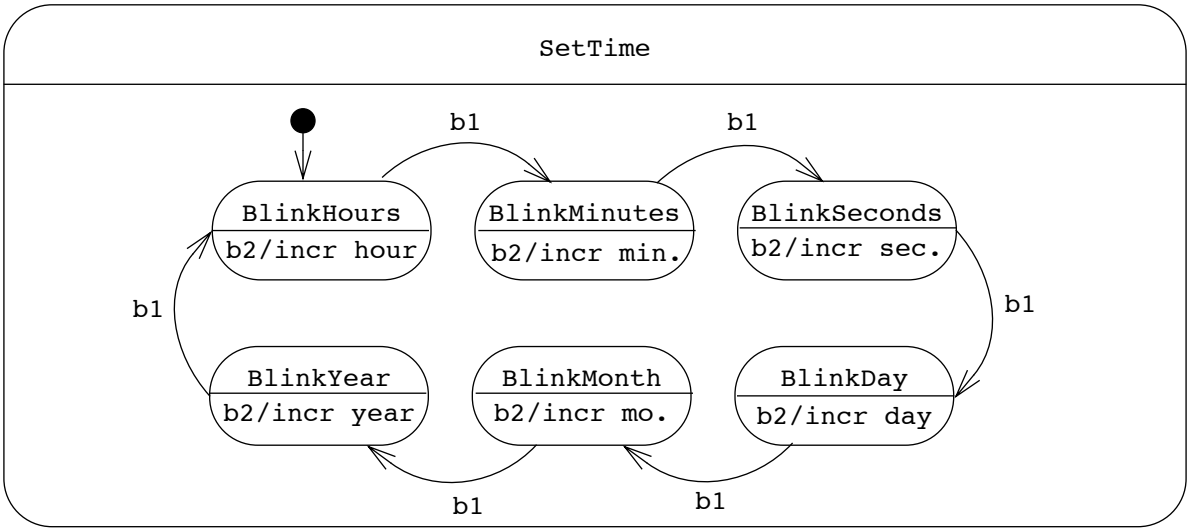

FIGURE 88. Refined statechart associated with the `SetTime` state.

7.2.3. Dataflow diagrams

The Structured Analysis method, developed by De Marco [De Marco, 1978], is a requirements analysis method most useful for data processing applications. The system is described as a set of data flows and processes that create, transform, store, or consume data. This method has been widely used and successful at taming the complexity of large

systems. Data flow diagrams, the principal notation used in Structured Analysis, have been used in other methods and form the basis for UML activity diagrams (see Chapter 2, *Introduction to UML*). We do not use the data flow notation in this book. We present, however, the original data flow notation here, given its popularity and legacy.

The basic elements of data flow diagrams are:

- data flows, denoted by arrows,
- processes, denoted by circles,
- files, denoted by horizontal lines, and
- data sources and sinks, denoted by boxes.

Figure 89 is an example of data flow diagram for the SatWatch described in Section 6.1. `GFS` is a data source. `WatchOwner` is a data sink. `Inquire Location`, `Determine Time Zone`, and `Compute Local Time` are processes. `Time Zone Table` and `GMT Time` are files.

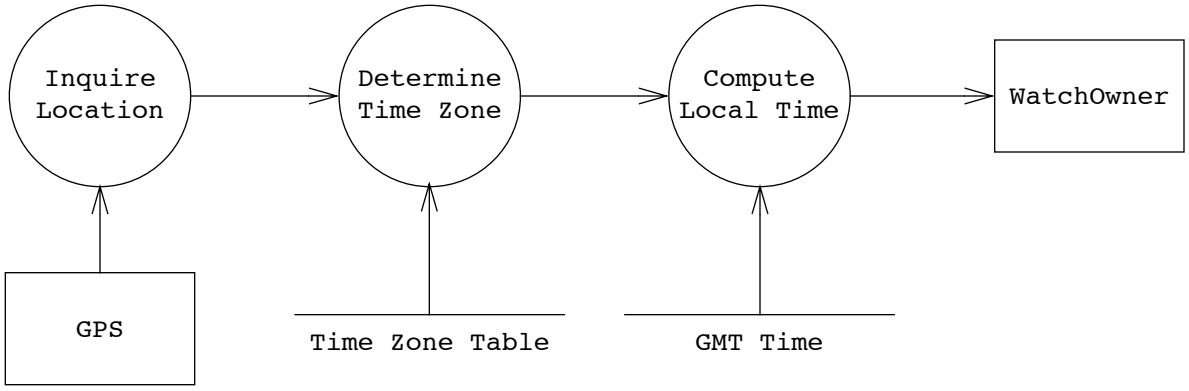

FIGURE 89. Data flow diagram for SatWatch. The watch receives its current location from GPS, compute its time zone from internal tables, computes the local time from the GMT time, and displays the local time for the user to read.

Data flow diagrams are hierarchical. Each process can be described in turn by a data flow diagram, enabling a top down presentation of the system. At the lowest level of detail, each process, file, data source, data sink, and data flow is described in natural language.

7.2.4. Decision tables

A decision table [De Marco, 1978] is a convenient notation for expressing decisions that depend on a complex or large number of conditions. Consider for example the rules that determine the number of days in a month. January, March, May, July, August, October, and December have 31 days. April, June, September, and November have 30 days. The number of days in February depends on the year: in leap years, February has 29 days, and in non leap years, 28. A year is leap if it is divisible by 4. If a year is a century, it is not a leap year, unless it is divisible by 400. Is February 29, 2000 a legitimate date?

		Rules					
Conditions	Month is 1, 3, 5, 7, 8, 10, or 12	Y	N	N	N	N	N
	Month is 4, 6, 9, or 11	N	Y	N	N	N	N
	Month is 2	N	N	Y	Y	Y	Y
	Year is divisible by 4			N	Y	Y	Y
	Year is divisible by 100			N	N	Y	Y
	Year is divisible by 400			N	N	N	Y
Action	Compute number of days of month	31	30	28	29	28	29

FIGURE 90. Decision table for determining the number of days given a month and a year.

Figure 90 depicts the corresponding decision table. The upper left cells of the table represent the conditions of interest. The upper right cells represent all legal values of the individual conditions. Note for example, that all values of the leap year conditions have been omitted for all months except for February. The lower left cells of the table contain the actions of interest. In this rather artificial example, we examine only a single action, `Compute the number of days in a month.` The lower right cells describe the result of the action given each set of values. A column regrouping a set of values and a result is called a rule.

A decision table makes it easier to select a rule given a set of conditions and compute an action. (e.g., February 2000 correspond to the last rule). Decision tables can be used to specify a range of behaviors, including processes in a data flow diagram, operations in an object model, the behavior of a digital circuit, or the number of days of a given month.

7.2.5. Z schemas

Z ([Spivey, 1989] and [Wordsworth, 1992]) is a notation for developing formal specifications. Formal specifications use mathematical symbols and rules to describe a system precisely such that its properties can be inferred and theorems about them proven. A Z specification is

a sequence of formal schemas interleaved with informal text. Schemas are used to describe invariants on the state of the system and the operations on the state. Figure 91 is a small specification describing a birthday book.

A birthday book consists of records peoples birthday. People are recorded by their name.

BirthdayBook

$$known : \mathbb{P}\ NAME$$
$$birthday : NAME \pfun DATE$$

$$known = \mathrm{dom}\ birthday$$

The `AddBirthday` operation adds a new birthday to the book given a new name and a date. If the name is not already in the book, it is recorded with the specified date.

AddBirthday

$$\Delta BirthdayBook$$
$$name? : NAME$$
$$date? : DATE$$

$$name? \notin known$$
$$birthday' = birthday \cup \{ name? \mapsto date? \}$$

The `Remind` operation generates a set of names corresponding to the poeple whose birthday match the specified date.

Remind

$$\Xi BirthdayBook$$
$$today? : DATE$$
$$cards! : \mathbb{P}\ NAME$$

$$cards! = \{ n : known \mid birthday(n) = today? \}$$

FIGURE 91. Z specification for a birthday book (adapted from [Spivey, 1989])

The *BirthdayBook* schema describes the state: a birthday book is a map from a set of names to a set of dates. The domain of the map is called the *known* set of persons.

The *AddBirthday* schema describes the operation `AddBirthday` in terms of the state before and after the operation. The declaration Δ*BirthdayBook* is equivalent to the declaration of four variables: *birthday* represents the map of users to birthdays before the new name is added, *birthday'* represents the map after the addition. Similarly, *known* and *known'* represent the set of known names before and after the operation. *name?* and *date?* are input parameters to the function. The expression The expression $name? \notin known$ is a precondition: the operation is executed only when the name is not in the known set of persons. The behavior

of the system in the case where name is already known is specified by another schema not shown here.

The *Remind* schema describes the operation `Remind`. Unlike `AddBirthday`, `Remind` does not modify the state of the birthday book. The declaration Ξ*BirthdayBook* is equivalent to the declaration Δ*BirthdayBook* and the two additional conditions: *known = known'* and *birthday = birthday'*. Also, the *Remind* operation returns its results in the form of a set called *cards!*

7.3. From use cases to objects

The analysis model consists of entity, interface, and control objects [Objectory, 1993]. ***Entity*** objects represent the persistent information tracked by the system. ***Interface*** objects represent the interactions between the actors and the system.***Control*** objects represent the tasks that are performed by the user and supported by the system. In the 2Bwatch example, `Button_Interface` and `LCDDisplay_Interface` are interface objects, `ChangeDate_Control` is a control object that represents the process of changing the date by pressing combinations of buttons. `Year`, `Month`, `Day` are entity objects.[1]

Modeling the system using these three types of objects has several advantages. First, it provides developers with simple heuristics to distinguish similar, but different, concepts. For example, the time that is tracked by a watch has different properties that the display that depicts the time. Differentiating between interface 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_Interface`. Second, the three object type approach results into smaller and more specialized objects. Third, the three object type approach leads to models that are more resilient to change: the interface to the system (represented by the interface objects) is more likely to change than its basic functionality (represented by the entity and control objects).

In this section, we describe how to construct an analysis model from the scenarios and use cases produced during requirements elicitation. Section 7.3.1 describes how to find entity objects. Section 7.3.2 describes how to represent system interfaces with interface objects. Section 7.3.3 describes control objects. Section 7.3.4 describes how to map use cases to the objects we identified and how to find missing objects. Section 7.3.5 focuses on the

1. 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, the `ChangeDate` class would be stereotyped with `<<control>>`. We prefer to use naming conventions for clarity and recommend to distinguish the three different types of objects on a syntactical basis: Interface objects have the suffix `_Interface` appended to their name; control objects have the suffix `_Control` appended to their name; entity objects do not have any suffix appended to their name.

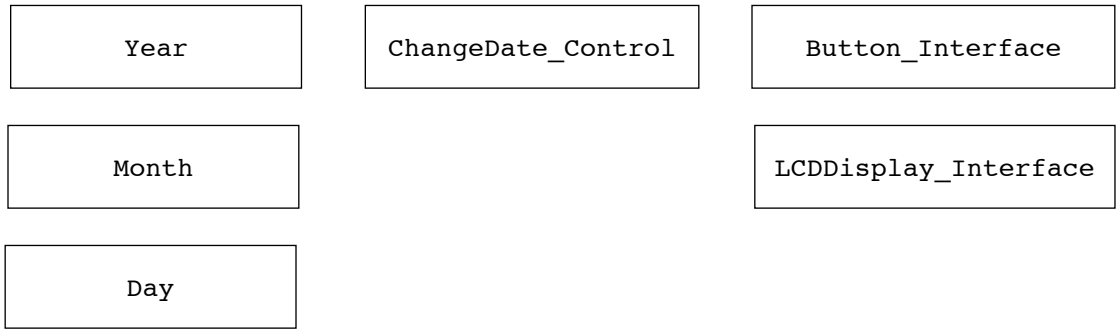

FIGURE 92. Analysis classes for the 2Bwatch example.

identification of associations among the objects. Section 7.3.6 focuses on the identification of object attributes. Section 7.3.8 describes how to model the behavior of individual objects with statecharts. Section 7.3.9 describes the elimination of redundancy by using generalization relationships. Section 7.3.10 discusses the activities that need to be performed when reviewing the analysis model. We illustrate each activity by focusing on the `ReportEmergency` use case example described in Chapter 6, *Requirements Elicitation*. These activities are mostly guided by heuristics. The quality of their outcome depends on the experience of the developer in applying these heuristics and methods. We adapted the methods and heuristics presented in this section from [Objectory, 1993], [Rumbaugh et al., 1991], [Booch, 1994], and [Wirfs-Brock et al., 1990].

7.3.1. Identifying entity objects

Participating objects (see Section 6.2.6) form the basis of the analysis model. As described in Chapter 6, *Requirements Elicitation*, participating objects are found by examining each use case and identifying candidate objects. Natural language analysis is an intuitive set of heuristics for identifying objects, attributes, and associations from a system 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 29 provides examples of such mappings by examining the `ReportEmergency` use case Figure 93.

Natural language analysis has the advantage of putting the focus 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 risk to be equally as imprecise. This limitation can be addressed by rephrasing and

Table 29 Abbott's heuristics for mapping parts of speech to model components [Abbott, 1983]

Part of speech	Model component	Examples
Proper noun	Object	Alice
Improper noun	Class	FieldOfficer
Doing verb	Operation	creates, submits, selects
Being verb	Inheritance	is a kind of
Having verb	Aggregation	has, consists of, includes
Modal verb	Constraints	Must be
Adjective	Attribute	incident description

clarifying the system specification as objects are identified and terms standardized. A second limitation of natural language analysis is that there are many more nouns than relevant classes. Many nouns correspond to attributes or synonyms for other nouns. Sorting through all the nouns for a large system specification is a time consuming activity. In general, Abbott's rules work well for generating a list of initial candidate objects from short descriptions, such as a scenario or a use case. The following heuristics can be used in conjunction with Abbott's rules:

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 keep track of (e.g., `FieldOfficer`, `Dispatcher`, `Resource`),
- Real world processes and procedures that the system needs to keep track of (e.g., `EmergencyOperationsPlan`),
- Use cases (e.g., `ReportEmergency`),
- Data sources or sinks (e.g., `Printer`),
- *Always* use the user's terms.

Developers name and briefly describe the objects, their attributes, and their responsibilities as they are identified. Uniquely naming objects promotes a standard terminology. Describing objects, even briefly, allows developers to clarify the concepts they are using 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 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, the description of each object should be as detailed as necessary once the analysis model is finalized (see Section 7.3.10).

For example, after a first examination of the `ReportEmergency` use case (Figure 93), we identify the objects `Dispatcher`, `EmergencyReport`, `FieldOfficer`, and `Incident`.

Use case name	`ReportEmergency`
Entry condition	1. The `FieldOfficer` activates the "Report Emergency" function of her terminal.
Description	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` fills the form, by specifying minimally the emergency type and description fields. The `FieldOfficer` may also describes 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 `AllocateResource` 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 93. An example of use case: `ReportEmergency`.

The definition of these objects leads to the initial analysis model described in Table 30

Note that the `EmergencyReport` object is not mentioned explicitly by name in the `ReportEmergency` use case. Step 3. 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`.

Table 30 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 a emergency level, a type (fire, road accident, or other), a location, and a description.
`FieldOfficer`	Police or fire officer on duty. A FieldOfficer can be allocated to at most one Incident. 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.

Note that the above object model is far from being a complete description of the system implementing the `ReportEmergency` use case. In the next section, we describe the identification interface objects.

7.3.2. Identifying interface objects

Interface objects model the system interface with the actors. In each use case, each actor interacts at least through one interface object. The interface object collects the information from the actor and translates it into an interface neutral form that can be used by the control and entity objects.

Interface objects model the user interface at a coarse level. They need not describe in detail the visual aspects of the user interface. First, developers can do this more easily with user interface sketches and mock-ups. Second, the design of the user interface design will continue to evolve even after the functional specification of the system becomes stable.

Updating the analysis model every time a visual change is made to the interface is time consuming and does not yield any substantial benefit.

Heuristics for identifying interface objects:

- Identify forms and windows the users needs to enter data into the system (e.g., `EmergencyReport_Interface`, `ReportEmergencyButton_Interface`),
- Identify notices and messages the system uses to respond to the user (e.g., `Acknowledgment_Interface`),
- Do not model the visual aspects of the interface with interface objects (user mock-ups are better suited for that),
- *Always* use the user's terms for describing interfaces, not the terms from the implementation technology.

We find the following interface objects by examining the `ReportEmergency` use case (see Table 31).

Table 31 Interface objects for the ReportEmergency use case

`Acknowledgment_Interface`	Notice used for displaying the `Dispatcher's` acknowledgment to the `FieldOfficer`.
`DispatcherStation_Interface`	Computer used by the `Dispatcher`.
`EmergencyButton_Interface`	Button used by a `FieldOfficer` to initiate the `ReportEmergency` use case.
`EmergencyReport_Interface`	Form used for the input of the `EmergencyReport`. This form is presented to the `FieldOfficer` on the `FieldOfficerStation_Interface` when the "Report Emergency" function is selected. The `EmergencyReport_Interface` contains fields for specifying all attributes of an emergency report and a button (or other control) for submitting the form once it is completed.
`FieldOfficerStation_Interface`	Portable computer used by the `FieldOfficer`.
`Incident_Interface`	Form used for the creation of Incidents. This form is presented to the `Dispatcher` on the `DispatcherStation_Interface` when the `EmergencyReport` is received. The `Dispatcher` also uses this form to allocate resources and to acknowledge the `FieldOfficer`'s report.

Note that the `Incident_Interface` is not explicitly mentioned anywhere in the `ReportEmergency` use case. We identified this object by observing that the `Dispatcher` needs an interface both to view the emergency report submitted by the `FieldOfficer` and

to send back an acknowledgment. The terms used for describing the interface 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 towards 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.

7.3.3. Identifying control objects

Control objects are responsible for coordinating interface and entity objects. There is often a close relationship 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 interface 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.

Heuristics for identifying control objects:

- Identify one control object per use case or more if the use case is complex and can be divided into shorter flows of events,
- Identify one control object per actor in the use case,
- The life span of a control object should be 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 may not have a well define entry and exit condition.

Initially, we model the control flow of the `ReportEmergency` use case with a control object for each actor (Table 32).

The decision to model the control flow of the `ReportEmergency` use case with two control objects stems from the knowledge that the `FieldOfficerStation_Interface` and the `DispatcherStation_Interface` 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.

In modeling the `ReportEmergency` use case, we modeled the same functionality using entity, interface, 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

Table 32 Control objects for the ReportEmergency use case

`ReportEmergency_Control`	Manages the report emergency reporting function on the `FieldOfficerStation_Interface`. This object is created when the `FieldOfficer` selects the "Report Emergency" button. It then creates an `EmergencyReport_Interface` and presents it to the `FieldOfficer`. Upon submission of the form, this object then collects the information form 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_Interface`. When the acknowledgment is received, the `ReportEmergency_Control` object creates an `Acknowledgment_Interface` and displays it to the `FieldOfficer`.
`ManageEmergency_Control`	Manages the report emergency reporting function on the `DispatcherStation_Interface`. 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, the ManageEmergencyCtr object forwards the acknowledgment to the FieldOfficerStation_Interface.

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

7.3.4. Modeling interactions between objects: sequence diagrams

A sequence diagram shows how the behavior of a use case (or of a scenario) is distributed among its participating objects. Sequence diagrams are usually not a good medium for communication with the user. They represent, however, another shift in perspective and allow the developers to find missing objects or grey areas in the system specification.

In this section, we model the sequence of interactions among objects needed to realize the use case. Figures 94, 95, and 96 are sequence diagrams associated with the `ReportEmergency` use case. The columns of a sequence diagram represent the objects which participates in the use case. The left most column is the actor who initiates the use case. Horizontal arrows across columns represent messages, or stimuli, which are sent from one object to the other. For example, the first arrow in Figure 94 represents the `press` message sent by a `FieldOfficer` to an `EmergencyReportButton`. The receipt of a message triggers the activation of an operation represented by a hollow rectangle from which other messages may originate. In Figure 94, the operation triggered by the `press` message sends a create

message to the `ReportEmergency_Control` class. An operation can be thought of as a service that the object provides to other objects.

Heuristics for drawing sequence diagrams:

- The first column should correspond to the actor who initiated the use case,
- The second column should be an interface 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 interface objects initiating use cases,
- Interface objects are created by control objects,
- Entity objects are accessed by control and interface objects,
- Entity objects *never* access interface or control objects, this makes it easier to share entity objects across use cases.

In general the second column of a sequence diagram represents the interface object with which the actor interacts to initiate the use case (e.g., `EmergencyReportButton`). The third column is a control object which manages the rest of the use case (e.g., `ReportEmergency_Control`). From then on, the control object creates other interface objects and may interact with other control objects as well (in this case, the `ManageEmergency_Control` object).

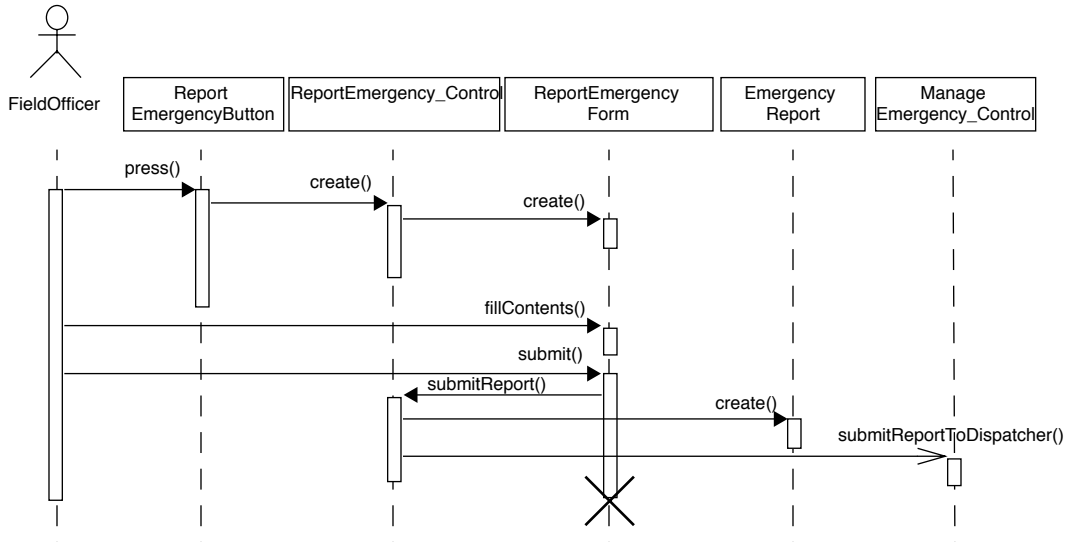

FIGURE 94. Sequence diagram for the `ReportEmergency` use case (initiation from the `FieldOfficerStation_Interface` side).

In Figure 95, we discover the entity object `Acknowledgment` that we forgot during our initial examination of the `ReportEmergency` use case. `Acknowledgment` is different from an `Acknowledgment_Interface`: it holds the information associated with an Acknowledgment and it pre-exists the `Acknowledgment_Interface` interface object. When describing the `Acknowledgment` object, we also realize that the original `ReportEmergency` use case 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 needs to appear in the `Acknowledgment`. After obtaining such clarification, the `Acknowledgment` object is added to the analysis model and the `ReportEmergency` use case is clarified to include the additional information.

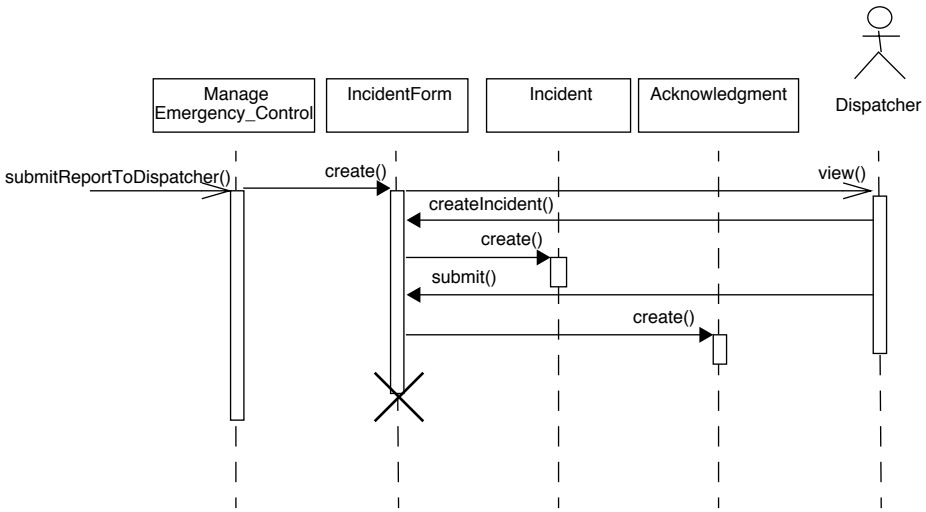

FIGURE 95. Sequence diagram for the `ReportEmergency` use case (`DispatcherStation_Interface`).

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. In other terms, we assign to each object responsibilities 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. In other terms, if an operation appears in more than one sequence diagram, its behavior should be the same.

Sharing operations across use cases allows developers to remove redundancies in the system specification and to improve its consistency. Note that clarity should always be

given precedence to eliminating redundancy. Fragmenting behavior across many operations obfuscates the system specification.

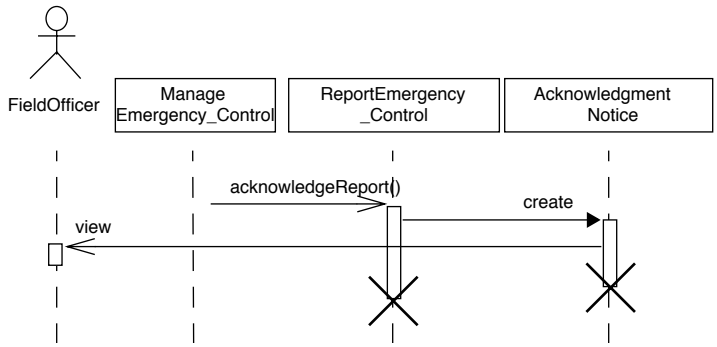

FIGURE 96. Sequence diagram for the `ReportEmergency` use case (acknowledgment on the `FieldOfficerStation_Interface`).

In requirements analysis, sequence diagrams are used to help identify new participating objects and missing behavior. They focus on high-level behavior, and thus, implementation issues such as performance should not be addressed at this point. The architecture of the system and the complete distribution of behavior across objects will be completed during system design. 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 which are simple or well defined is not be a good investment of analysis resources.

7.3.5. Identifying associations

While sequence diagrams allow developers to represent interactions among objects over time, class diagrams allow developers to describe the spatial connectivity of objects. We described the UML class diagram notation in Chapter 2, *Introduction to UML* and used it throughout the book to represent various project artifacts (e.g., processes, activities, deliverables). In this section, we discuss the use of class diagrams for representing associations among objects. In Section 7.3.5, we discuss the use of class diagrams for representing object attributes.

An association shows a dependency between two or more classes. For example, a `FieldOfficer` writes an `EmergencyReport` (see Figure 97). 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 (e.g., an EmergencyReport is created by exactly one FieldOfficer or Dispatcher).

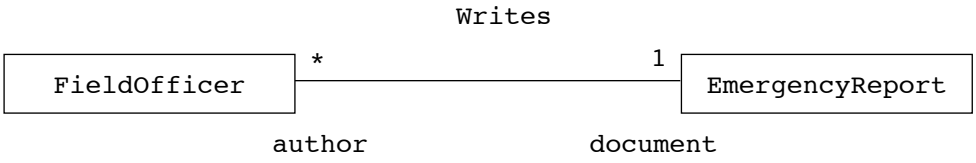

FIGURE 97. An example of association between the EmergencyReport and the FieldOfficer classes.

Associations have several properties:

- a ***name***, to describe the association between the two classes (e.g., Writes in Figure 97). Association names are optional and need not be unique globally.
- 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 ***cardinality*** 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 29), 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 assigned to each end. Association names need not be unique.

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.

The object model will initially include too many associations if developers include all associations identified after examining verb phrases. In Figure 98, for example, we

identified a relationship between an `Incident` and the `EmergencyReport` that triggered its creation, and the `Incident` and the reporting `FieldOfficer`. Given the `EmergencyReport` and `FieldOfficer` already have an association modeling authorship, the association between `Incident` and `FieldOfficer` is not necessary.

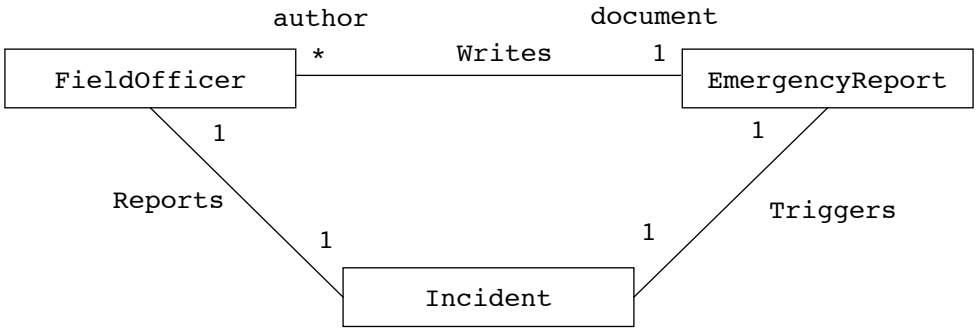

FIGURE 98. Eliminating redundant association. The receipt of an `EmergencyReport` triggers the creation of an `Incident` by a `Dispatcher`. Given that the `EmergencyReport` has an association with the `FieldOfficer` that wrote it, it is not necessary to keep an association between `FieldOfficer` and `Incident`.

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.

7.3.6. Identifying attributes

Attributes are properties of individual objects. For example, an `EmergencyReport`, as described in Table 30, has an emergency type, a location, and a description property (see Figure 99). 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, every `FieldOfficer` have a social security number which is not relevant to the emergency information system.

Instead, `FieldOfficers` are identified by badge number, represented by the `badgeNumber` property.

Properties that are represented by objects are not attributes. For example, every `EmergencyReport` has an author 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.

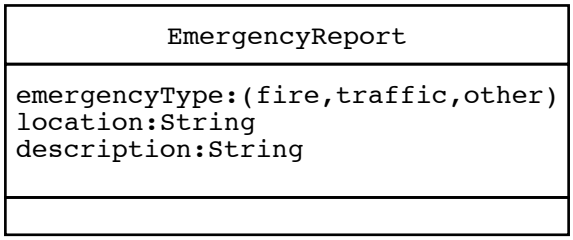

FIGURE 99. Attributes of the `EmergencyReport` class.

Attributes minimally 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). These attributes should not be both called `type` to avoid confusion,
- 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 which can take one of three values: `fire`, `traffic`, `other`.

Attributes can be identified using Abbotts heuristics (see Table 29). In particular, noun phrases followed by a possessive phrases (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 needs to 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

representing 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 attributes of entity object.
- Describe each attribute.
- Do not represent identifiers (e.g., names) as attributes, use qualifiers instead (see Section 7.3.5).
- Do not represent an attribute as an object, use an association instead (see Section 7.3.5).
- Do not waste time describing fine details before the object structure is stable.

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

7.3.7. Identifying qualifiers

A qualified association relates two classes and a qualifier. A qualifier is an attribute of the association that partitions the targets of the association into exclusive subsets. For example, consider a hierarchical file system in which each file belongs to exactly one directory. Assume that each file is uniquely identified by a name in the context of a directory. In Figure 100, the top class diagram represents the association between `Directory` and the `File` without qualification. The bottom diagram represents the same association with a qualifier, filename. The addition of the qualifier and the `1` multiplicity on the left side indicate that the `filename` uniquely identifies the `File` in a given `Directory`. The `0…1` multiplicity on the right side indicates that, given a legitimate filename, there may or may not be a file associated with it.

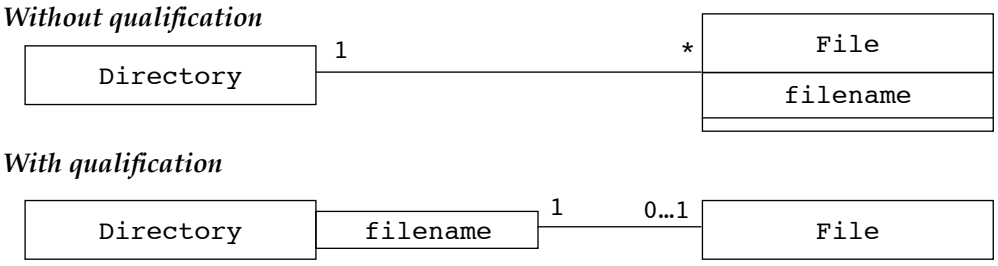

FIGURE 100. Example of qualified association.

Adding qualifiers usually reduces the multiplicity of the association and adds information to the model. Developers should examine each association that has a one to many or many

to many multiplicity and check if a qualifier is missing. Often, these associations can be qualified with an attribute of the target class, (e.g., the `filename` attribute in Figure 100).

Identifying qualifiers and searching for identifying attributes can reveal missing classes. Usually, objects are identified by a name or an attribute in some context. Names are rarely unique globally. For example, a `FieldOfficer` is identified by badge number in the context of a `City`. `City` is a new class with an association to `FieldOfficer` qualified by the badge number of a `FieldOfficer` (see Figure 101).

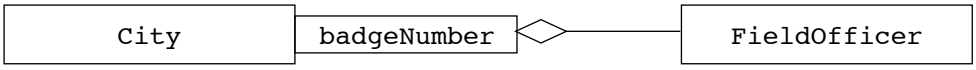

FIGURE 101. An example of qualified association. The `City` class was identified when examining identifying attributes of the `FieldOfficer` class.

7.3.8. Modeling the non trivial behavior of individual objects

Sequence diagrams (see Sections 7.2.1 and 7.3.4) are used to distribute behavior across objects and identifying operations. Sequence diagrams represent the behavior of the system from the perspective of a single use case. Statecharts (see Section 7.2.2) represent behavior from the perspective of a single object. Viewing behavior from the perspective of each object enables the developer, on the one hand, to identify missing use cases, and, on the other hand, to build a more formal description of the behavior of the object. Note that it is not necessary to build statecharts for every class in the system. Only the statecharts of objects with an extended lifespan and with non trivial behavior are worth constructing.

Figure 102 displays a statechart for the `Incident` class. The examination of this statechart may help the developer check if there are use cases for documenting, closing, and archiving `Incidents`. Note that Figure 102 is a high level statechart and does not model the state changes an `Incident` goes through while it is active (e.g., when resources are assigned to it). Such behavior can be modeled by associating a nested statechart with the `Active` state.

7.3.9. Modeling generalization 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`

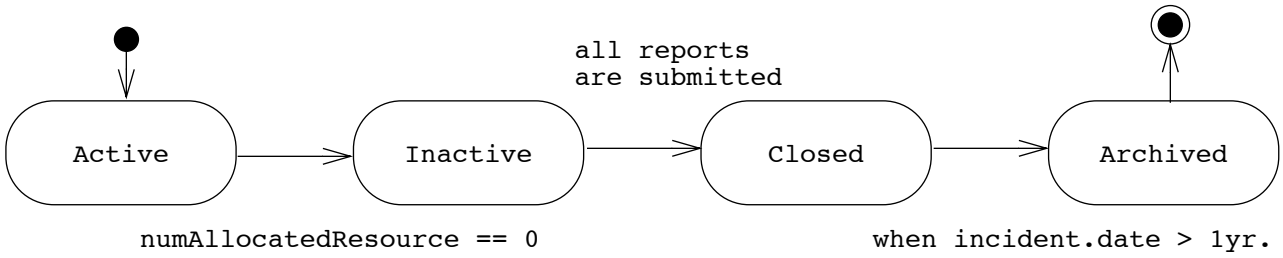

FIGURE 102. Statechart for `Incident`

who are assigned different functions. To model explicitly this similarity, we introduce a `PoliceOfficer` class from which the `FieldOfficer` and `Dispatcher` classes inherit (see Figure 103).

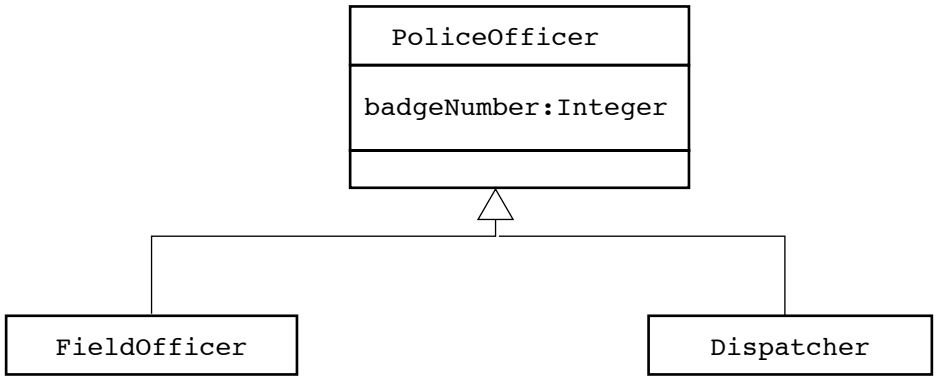

FIGURE 103. An example of inheritance relationship

7.3.10. Reviewing the analysis model

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

Once the analysis model becomes stable (i.e., when the number of changes to the model are minimal and the scope of the changes localized), the analysis model is reviewed, first by the developers (i.e., internal reviews), then jointly by the developers and the client. The review can be facilitated by a check list or a list of questions. Below are example questions adapted from [Objectory, 1993] and [Rumbaugh et al., 1991].

The following questions should be asked to ensure that the system 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 interface 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?
- Are the start up and the shut down phases of the system described?
- Are the administration functions of the system described?

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 an interface 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 phenomena?
- Are all entities described at the same level of detail?
- 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 there any studies or prototypes built to ensure there feasibility?

- Can the performance and reliability requirements be met? Were these requirements verified by any prototypes running on the selected hardware?

7.3.11. Requirements analysis summary

The requirements process is highly iterative and incremental. Chunks of functionality are sketched and proposed to the users and the client. The client adds additional requirements, criticize existing functionality, and modifies existing requirements. The developers investigate nonfunctional requirements through prototyping and technology studies and challenge each pseudo 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 in order to manage the complexity of information.

Figure 104 depicts a typical sequence of the requirements analysis activities we described in this chapter. The users, developers, and client are involved in the *Defining use cases* and develop an initial use case model. They identify a number of concepts and built a glossary of participating objects. The developers then classify these objects into entity, interface, and control objects (in *Defining entity objects*, Section 7.3.1, *Defining interface objects*, Section 7.3.2, and *Defining control objects*, Section 7.3.3). 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 (*Defining interactions*, Section 7.3.4). Once all entity objects have been named and briefly described, the analysis model should remain fairly stable as it is refined.

Defining interesting behavior (Section 7.3.8), *Defining attributes* (Section 7.3.6), and *Defining associations* (Section 7.3.5) constitute the refinement of the analysis model. These three activities represent 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 *Consolidating requirements* (Section 7.3.7 and Section 7.3.9), the developers solidify the model by introducing qualifiers, generalization relationships, and suppressing redundancies. During *Reviewing requirements* (Section 7.3.10), the client, users, and developers examine the model for correctness, consistency, completeness, and realism. The project schedule should plan for multiple reviews to ensure a high quality of the requirements and allow space for learning the requirements process. However, once the model reaches the point where most modifications are cosmetic, system design should proceed. There will be 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.

7.4. Managing requirements

In this section we discuss issues related to managing the requirements analysis activities in the context of a 40 person project such as PROSE. 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 requirements analysis (Section 7.4.1). Next, we describe the role assignment to requirements analysis (Section 7.4.2). Next, we address communication issues during requirements analysis. Next, we address management issues related to the iterative and incremental nature of requirements (Section 7.4.4).

7.4.1. Documenting the requirements

The product of the requirements elicitation and requirements analysis activities is the Requirements Analysis Document (RAD). It completely describes the system in terms of functional and nonfunctional requirements and often serves as a contractual basis between the client and the developers. The audience for the RAD includes the client, the users, the project management, the system analysts (i.e., the developers who participate in the

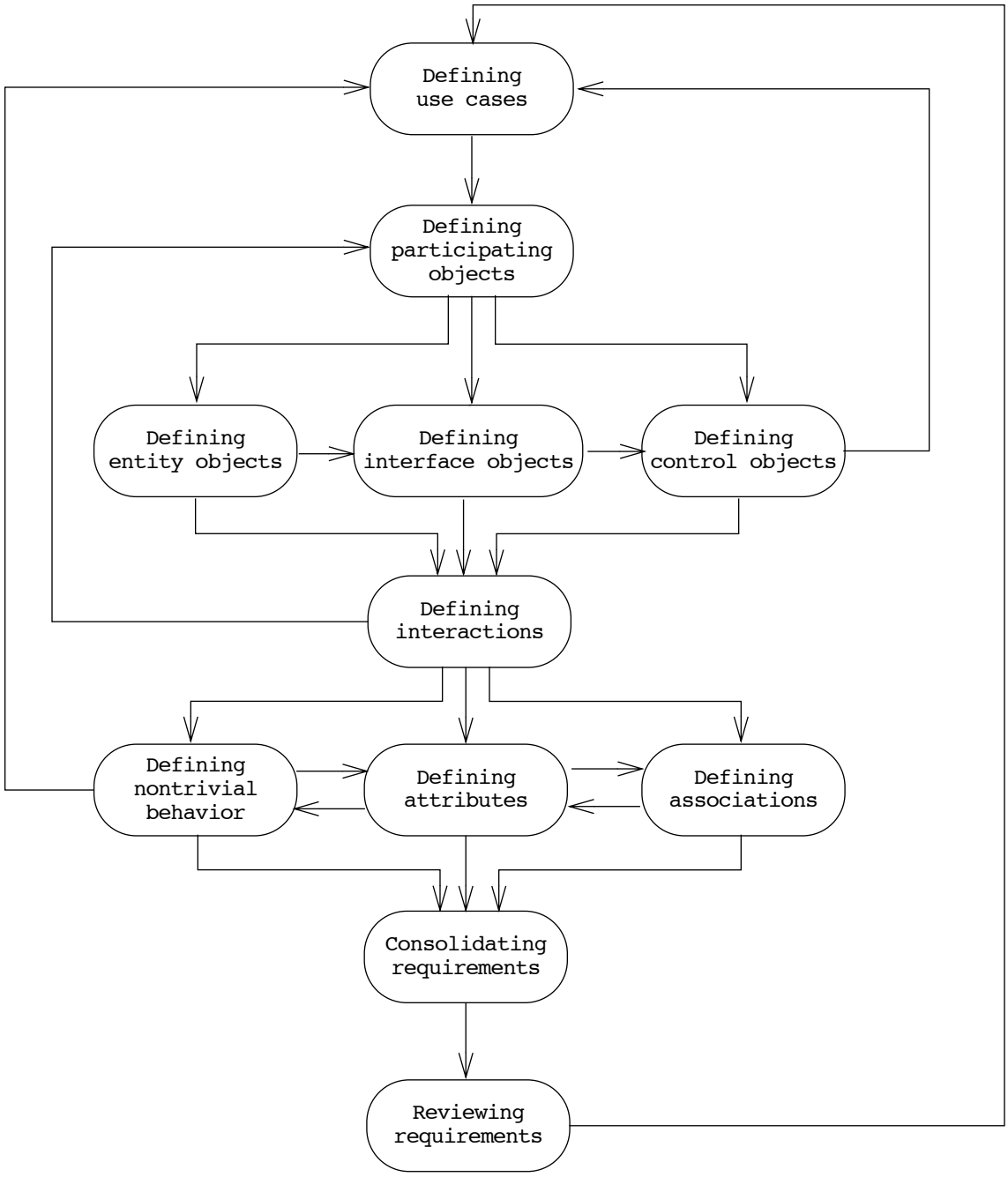

FIGURE 104. Requirements analysis activities (UML activities diagram). As depicted in this figure, requirements analysis is iterative and incremental.

requirements) and the system designers (i.e., the developers who participate in the system design). The following template is an example of RAD for PROSE:

Requirements Analysis Document (RAD)

Revision history

1. Introduction
    - 1.1 Purpose of the system
    - 1.2 Scope of the system
    - 1.3 Objectives and success criteria of the project
    - 1.4 Definitions, acronyms, and abbreviations
    - 1.5 References
    - 1.6 Overview

2. Current system

3. Proposed system
    - 3.1 Overview
    - 3.2 Functional requirements
    - 3.3 Nonfunctional requirements
        - 3.3.1 User interface and human factors
        - 3.3.2 Documentation
        - 3.3.3 Hardware consideration
        - 3.3.4 Performance characteristics
        - 3.3.5 Error handling and extreme conditions
        - 3.3.6 System interfacing
        - 3.3.7 Quality issues
        - 3.3.8 System modifications
        - 3.3.9 Physical environment
        - 3.3.10 Security issues
        - 3.3.11 Resource issues
    - 3.4 Pseudo requirements
    - 3.5 System models
        - 3.5.1 Scenarios
        - 3.5.2 Use case model
        - 3.5.3 Object model
            - 3.5.3.1 Data dictionary
            - 3.5.3.2 Class diagrams
        - 3.5.4 Dynamic models
        - 3.5.5 User interface - navigational paths and screen mock-ups

Appendixes

Index

The first section of the RAD is an introduction. Its purpose is to provide a brief overview of the function of the system and the reasons for its development, its scope, and references to the development context (e.g., related problem statement, references to existing systems,

feasibility studies). The introduction also includes the objectives and success criteria of the project.

The second section, *Current system,* describes the current state of affairs. If the new system will replace an existing system, this section describes the functionality and the problems of the current system. Otherwise, this section describes how the tasks supported by the new system are accomplished now. For example, in the case of SatWatch, the user currently resets her watch whenever she travels across a time zone. Due to the manual nature of this operation, the user occasionally sets the wrong time. The SatWatch will continually ensure accurate time within its lifetime In the case of FRIEND, the current system is paper based: dispatchers keep track of resource assignments by filling forms. Communication between dispatchers and field officers is radio-based. The current system entails a higher documentation and management cost that the FRIEND system attempts to reduce.

The third section, *Proposed system,* documents the requirements elicitation and the requirements analysis model of the new system. It is divided into five subsections:

- *Overview* presents a functional overview of the system.
- *Functional requirements* describes in natural language the high level functionality of the system.
- *Nonfunctional requirements* describes user level requirements that are not directly related to functionality. This includes performance, security, modifiability, error handling, hardware platform, physical environment.
- *Pseudo requirements* describes design constraints imposed by the client.
- *System models* describes the scenarios, use cases, object model, and dynamic models we discussed in the previous sections. This section contains the complete functional specification of the system, including mock-ups and navigational charts illustrating the user interface of the system.

The RAD should be written after the analysis model is stable, that is, when the number of modifications to the requirements is minimal. The RAD, however, will be updated throughout the process when specification problems are discovered or when the scope of the system is changed. The RAD, once published, will be under configuration management. The revision history section of the RAD will provide a history of changes as a list of author responsible for the change, date of change, and brief description of the change.

7.4.2. Assigning responsibilities

Requirements 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, analysis, 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 ***user*** is the application domain expert. She 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 help 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 development domain expert. She models the current system and generates information about the future system. Each analysis 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 7.3.
- 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 which they are not responsible for. Although analysts work together and will most likely resolve differences as they progress through requirements analysis, the role of the architect is necessary to provide a system philosophy and identify omissions in the requirements.
- The ***document editor*** is responsible for the low level integration of the document. The document editor is responsible for the formatting, index, and ensures the overall terminology used in the document is consistent.
- 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 8, *System Design*).
- The ***reviewer*** validates the RAD for correctness, completeness, consistency, realism, verifiability, and traceability. Users, clients, developers, or other individuals may become reviewers during requirements validation. Individuals that have not yet been involved in the process represent excellent reviewers since 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.

7.4.3. Communicating about requirements analysis

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

- *Different background 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 requirements 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 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 requirements analysis and the system design. A term today may have a different meaning tomorrow.

No requirements method or communication mechanism can address problems related with 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 7.4.2 is part of this process. This also includes the definition of private and public discussion forums. For example, each team may have a discussion bboard as described in Chapter 5, *Project Communication* and discussion with the client is done on a separate client bboard. The client should not have access to the internal bboards. Similarly, developers should not interfere with client/user internal politics.
- *Define clear objectives and success criteria.* The co-definition 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 non trivial 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 have them generate quickly solutions and definitions can remove many barriers in the 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, adhoc 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, object diagrams, sequence diagrams, and statecharts to communicate with other developers (see Sections 6.2 and 7.3). Moreover, the latest release of the requirements should be available to all participants. Maintaining a live on-line version of the requirements analysis document with an up to date change history facilitates the timely propagation of changes across the project.

7.4.4. Iterating over the analysis model

Requirements analysis occurs iteratively and incrementally, often in parallel with other development processes 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 need to be carefully managed and requests for changes tracked.

Before any other development process is initiated, requirements elicitation 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.

Once the client and the developers converge on a common idea, defined the boundaries of the system, and agreed on a set of standard terms, requirements analysis 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. Changes at the higher level are still possible but are more difficult, and thus, 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 representative of each team, is responsible for ensuring the integration of the requirements (e.g., naming).

Once the client signs off the requirements, modification to the requirements 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 is due to 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 into 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: Are they necessary or are they embellishments? Were they requested by the client? Should they be part of a separate, focused utility program instead of part of the base system? What are 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, 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).

7.4.5. 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 converged on a single idea and agreed 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, and
- 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 of the system. In general, it allows developers to deliver the system in incremental chunks: essential functions are delivered first, additional chunks are delivered depending on the evaluation of the first chunk. Even if the system is to be delivered as a single, complete package, prioritizing functions enables the client to clearly communicate

what is important to her and where the emphasis of the development should be. Figure 105 provides an example of 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 and detailed designs. It will be implemented and demonstrated with the second increment of the system.
- ***Low priority*** - A low-priority feature illustrates how the system could be extended in the longer term.

FIGURE 105. Example of priority scheme for requirements.

A revision process enables the client and developer to define how changes in the requirements are to be dealt after the sign-off. The requirements will change, either 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 long term. 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 106 depicts a much more complex example in which changes are designed and reviewed by the client before they are implemented in the system. In all cases, acknowledging that requirements cannot be frozen will benefit the project.

The list of acceptance criteria is revised at sign-off. The requirements elicitation and requirements analysis process clarifies many aspects of the system, including the nonfunctional requirements with which the system should comply and the relative importance of each function. By re-stating 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 described in Chapter 4, *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, the client sign-off is an important milestone in the project. It represents the convergence of client and developer on a single functional definition 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.

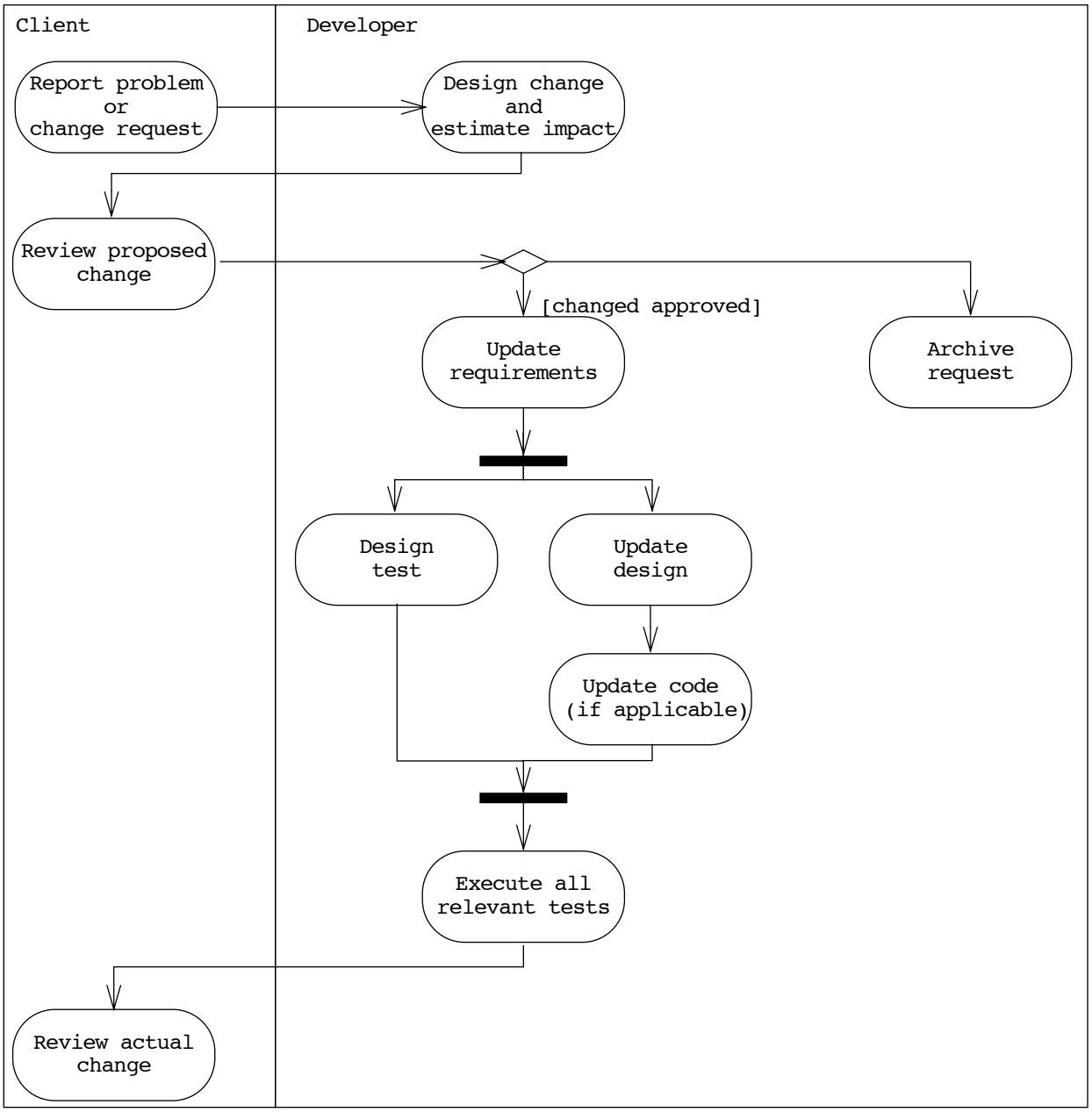

FIGURE 106. Example of revision process (UML activity diagram).

7.5. Exercises

1. Reverse engineering the `SetTime` use case for the 2Bwatch from the sequence diagram of Figure 84.
2. Identify entity, control, and interface objects from this use case using the method presented in this chapter. Include attributes and operations. Modify the use case if you find any ambiguity or inconsistency in this process.
3. The makers of 2Bwatch are now proposing an software upgrade that includes a timer function. Write the use cases `StartTimer`, `StopTimer`, and `ResetTimer`. Remember that 2Bwatch only has two buttons and that present functionality (i.e., `SetTime`) needs to be supported.
4. Update the analysis model developed in Exercise 2. to reflect the additional functionality described in Exercise 3. Were any of the original objects changed?
5. You need to communicate to the original developer the changes you made to the analysis model in Exercise 4. How would you communicate document such changes?

7.6. References

[Abbott, 1983] R. Abbott, "Program Design by Informal English Descriptions," *Communications of the ACM*, vol 26, no. 11, 1983.

[Booch, 1994] G. Booch, *Object-Oriented Analysis and Design with Applications,* Second Edition, Benjamin/Cummings Publishing Company, Inc., Redwood City, CA, 1994.

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

[FRIEND, 1994] FRIEND Project Documentation, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 1992-95.

[Harel, 1987] D. Harel, ˇStatecharts: A Visual Formalism for Complex Systems,ˇ *Science of Computer Programming*, pp. 231-274, 1987.

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

[Jackson, 1995] M. Jackson, Software Requirements & Specifications: a lexicon of practice, principles and prejudices, ACM Press, Addison-Wesley, 1995.

[Objectory, 1993] *Objectory 3.3,* Objective Systems SF AB, Kista, Sweden, 1993.

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

[Spivey, 1989] J. M. Spivey, *The Z Notation, A Reference Manual.* Prentice Hall International

(UK) Ltd., Hemel Hempstead, Hertfordshire, U.K. 1989.

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

[Wordsworth, 1992] J.B. Wordsworth, *Software Development with Z: a Practical Approach to Formal Methods in Software Engineering,* Addison-Wesley, 1992.