13

Application Analysis

This chapter completes our treatment of analysis by adding major application artifacts to the domain model from the prior chapter. We include these application artifacts in analysis, because they are important, visible to users, and must be approved by them. In general, you cannot find the application classes in the domain itself, but must find them in use cases.

13.1 Application Interaction Model

Most domain models are static and operations are unimportant, because a domain as a whole usually doesn't *do* anything. The focus of domain modeling is on building a model of intrinsic concepts. After completing the domain model we then shift our attention to the details of an application and consider interaction.

Begin interaction modeling by determining the overall boundary of the system. Then identify use cases and flesh them out with scenarios and sequence diagrams. You should also prepare activity diagrams for use cases that are complex or have subtleties. Once you fully understand the use cases, you can organize them with relationships. And finally check against the domain class model to ensure that there are no inconsistencies.

You can construct an application interaction model with the following steps.

- Determine the system boundary. [13.1.1]
- Find actors. [13.1.2]
- Find use cases. [13.1.3]
- Find initial and final events. [13.1.4]
- Prepare normal scenarios. [13.1.5]
- Add variation and exception scenarios. [13.1.6]
- Find external events. [13.1.7]
- Prepare activity diagrams for complex use cases. [13.1.8]

- Organize actors and use cases. [13.1.9]
- Check against the domain class model. [13.1.10]

13.1.1 Determining the System Boundary

You must know the precise scope of an application—the boundary of the system—in order to specify functionality. This means that you must decide what the system includes and, more importantly, what it omits. If the system boundary is drawn correctly, you can treat the system as a black box in its interactions with the outside world—you can regard the system as a single object, whose internal details are hidden and changeable. During analysis, you determine the purpose of the system and the view that it presents to its actors. During design, you can change the internal implementation of the system as long as you maintain the external behavior.

Usually, you should not consider humans as part of a system, unless you are modeling a human organization, such as a business or a government department. Humans are actors that must interact with the system, but their actions are not under the control of the system. However, you must allow for human error in your system.

ATM example. The original problem statement from Chapter 11 says to "design the software to support a computerized banking network including both human cashiers and automatic teller machines..." Now it is important that cashier transactions and ATM transactions be seamless—from the customer's perspective either method of conducting business should yield the same effect on a bank account. However, in commercial practice an ATM application would be separate from a cashier application—an ATM application spans banks while a cashier application is internal to a bank. Both applications would share the same underlying domain model, but each would have its own distinct application model. For this chapter we focus on ATM behavior and ignore cashier details.

13.1.2 Finding Actors

Once you determine the system boundary, you must identify the external objects that interact directly with the system. These are its *actors*. Actors include humans, external devices, and other software systems. The important thing about actors is that they are not under control of the application, and you must consider them to be somewhat unpredictable. That is, even though there may be an expected sequence of behavior by the actors, an application's design should be robust so that it does not crash if an actor fails to behave as expected.

In finding actors, we are not searching for individuals but for archetypical behavior. Each actor represents an idealized user that exercises some subset of the system functionality. Examine each external object to see if it has several distinct faces. An actor is a coherent face presented to the system, and an external object may have more than one actor. It is also possible for different kinds of external objects to play the part of the same actor.

ATM example. A particular person may be both a bank teller and a customer of the same bank. This is an interesting but usually unimportant coincidence—a person approaches the bank in one or the other role at a time. For the ATM application, the actors are *Customer*, *Bank*, and *Consortium*.

13.1.3 Finding Use Cases

For each actor, list the fundamentally different ways in which the actor uses the system. Each of these ways is a *use case*. The use cases partition the functionality of a system into a small number of discrete units, and all system behavior must fall under some use case. You may have trouble deciding where to place some piece of marginal behavior. Keep in mind that there are always borderline cases when making partitions; just make a decision even if it is somewhat arbitrary.

Each use case should represent a kind of service that the system provides—something that provides value to the actor. Try to keep all of the use cases at a similar level of detail. For example, if one use case in a bank is "apply for loan," then another use case should not be "withdraw cash from savings account using ATM." The latter description is much more detailed than the former; a better match would be "make withdrawal." Try to focus on the main goal of the use case and defer implementation choices.

At this point you can draw a preliminary use case diagram. Show the actors and the use cases, and connect actors to use cases. Usually you can associate a use case with the actor that initiates it, but other actors may be involved as well. Don't worry if you overlook some participating actors. They will become apparent when you elaborate the use cases. You should also write a one or two sentence summary for each use case.

ATM example. Figure 13.1 shows the use cases, and the bullets summarize them.

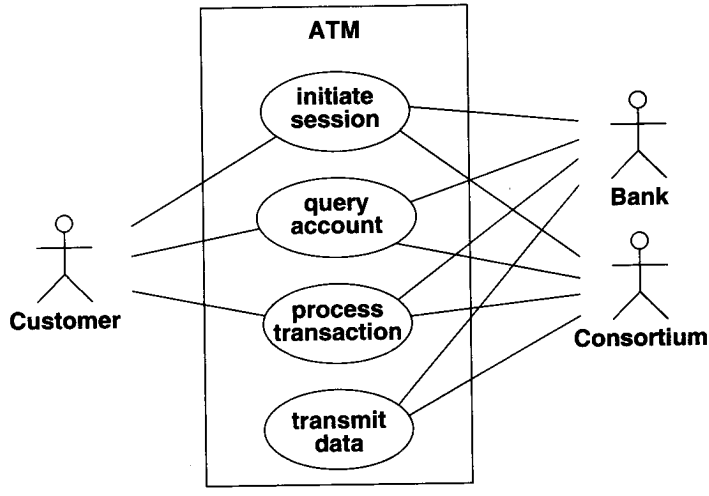

Figure 13.1 Use case diagram for the ATM. Use cases partition the functionality of a system into a small number of discrete units that cover its behavior.

- **Initiate session**. The ATM establishes the identity of the user and makes available a list of accounts and actions.
- **Query account**. The system provides general data for an account, such as the current balance, date of last transaction, and date of mailing for last statement.

- **Process transaction**. The ATM system performs an action that affects an account's balance, such as deposit, withdraw, and transfer. The ATM ensures that all completed transactions are ultimately written to the bank's database.
- **Transmit data**. The ATM uses the consortium's facilities to communicate with the appropriate bank computers.

13.1.4 Finding Initial and Final Events

Use cases partition system functionality into discrete pieces and show the actors that are involved with each piece, but they do not show the behavior clearly. To understand behavior, you must understand the execution sequences that cover each use case. You can start by finding the events that initiate each use case. Determine which actor initiates the use case and define the event that it sends to the system. In many cases, the initial event is a request for the service that the use case provides. In other cases, the initial event is an occurrence that triggers a chain of activity. Give this event a meaningful name, but don't try to determine its exact parameter list at this point.

You should also determine the final event or events and how much to include in each use case. For example, the use case of applying for a loan could continue until the application is submitted, until the loan is granted or rejected, until the money from the loan is delivered, or until the loan is finally paid off and closed. All of these could be reasonable choices. The modeler must define the scope of the use case by defining when it terminates.

ATM example. Here are initial and final events for each use case.

- **Initiate session**. The initial event is the customer's insertion of a cash card. There are two final events: the system keeps the cash card or the system returns the cash card.
- **Query account**. The initial event is a customer's request for account data. The final event is the system's delivery of account data to the customer.
- **Process transaction**. The initial event is the customer's initiation of a transaction. There are two final events: committing or aborting the transaction.
- **Transmit data**. The initial event could be triggered by a customer's request for account data. Another possible initial event could be recovery from a network, power, or another kind of failure. The final event is successful transmission of data.

13.1.5 Preparing Normal Scenarios

For each use case, prepare one or more typical dialogs to get a feel for expected system behavior. These scenarios illustrate the major interactions, external display formats, and information exchanges. A *scenario* is a sequence of events among a set of interacting objects. Think in terms of sample interactions, rather than trying to write down the general case directly. This will help you ensure that important steps are not overlooked and that the overall flow of interaction is smooth and correct.

For most problems, logical correctness depends on the sequences of interactions and not their exact times. (Real-time systems, however, do have specific timing requirements on interactions, but we do not address real-time systems in this book.)

Sometimes the problem statement describes the full interaction sequence, but most of the time you will have to invent (or at least flesh out) the interaction sequence. For example, the ATM problem statement indicates the need to obtain transaction data from the user but is vague about exactly what parameters are needed and in what order to ask for them. During analysis, try to avoid such details. For many applications, the order of gathering input is not crucial and can be deferred to design.

Prepare scenarios for "normal" cases—interactions without any unusual inputs or error conditions. An event occurs whenever information is exchanged between an object in the system and an outside agent, such as a user, a sensor, or another task. The information values exchanged are event parameters. For example, the event *password entered* has the password value as a parameter. Events with no parameters are meaningful and even common. The information in such an event is the fact that it has occurred. For each event, identify the actor (system, user, or other external agent) that caused the event and the parameters of the event.

ATM example. Figure 13.2 shows a normal scenario for each use case.

13.1.6 Adding Variation and Exception Scenarios

After you have prepared typical scenarios, consider "special" cases, such as omitted input, maximum and minimum values, and repeated values. Then consider error cases, including invalid values and failures to respond. For many interactive applications, error handling is the most difficult part of development. If possible, allow the user to abort an operation or roll back to a well-defined starting point at each step. Finally consider various other kinds of interactions that can be overlaid on basic interactions, such as help requests and status queries.

ATM example. Some variations and exceptions follow. We could prepare scenarios for each of these but will not go through the details here. (See the exercises.)

- The ATM can't read the card.
- The card has expired.
- The ATM times out waiting for a response.
- The amount is invalid.
- The machine is out of cash or paper.
- The communication lines are down.
- The transaction is rejected because of suspicious patterns of card usage.

There are additional scenarios for administrative parts of the ATM system, such as authorizing new cards, adding banks to the consortium, and obtaining transaction logs. We will not explore these aspects.

13.1.7 Finding External Events

Examine the scenarios to find all external events—include all inputs, decisions, interrupts, and interactions to or from users or external devices. An event can trigger effects for a target object. Internal computation steps are not events, except for computations that interact with

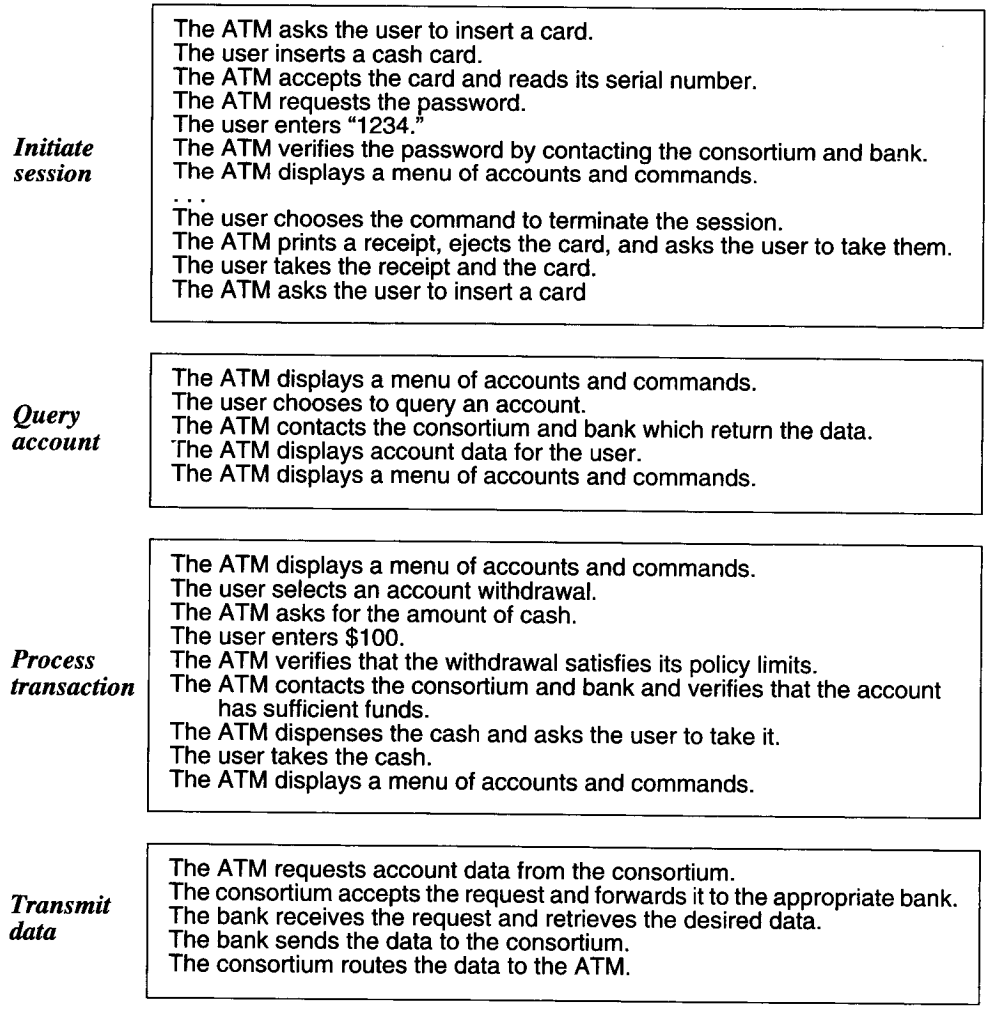

Figure 13.2 Normal ATM scenarios. Prepare one or more scenarios for each use case.

the external world. Use scenarios to find normal events, but don't forget unusual events and error conditions.

A transmittal of information to an object is an event. For example, *enter password* is a message sent from external agent *User* to application object *ATM*. Some information flows are implicit. Many events have parameters.

Group together under a single name events that have the same effect on flow of control, even if their parameter values differ. For example, *enter password* should be an event, whose parameter is the password value. The choice of password value does not affect the flow of

control; therefore events with different password values are all instances of the same kind of event. Similarly, *dispense cash* is also an event, since the amount of cash dispensed does not affect the flow of control. Event instances whose values affect the flow of control should be distinguished as different kinds of events. *Account OK, bad account,* and *bad password* are all different events; don't group them under *card status*.

You must decide when differences in quantitative values are important enough to distinguish as distinct events. For example, the different digits from a keyboard would usually be considered the same event, since the high-level control does not depend on numerical values. Pushing the "enter" key, however, might be considered a distinct event, since an application could treat it differently. The distinction among events depends on the application.

Prepare a sequence diagram for each scenario. A *sequence diagram* shows the participants in an interaction and the sequence of messages among them; each participant is assigned a column in a table. The sequence diagram clearly shows the sender and receiver of each event. If more than one object of the same class participates in the scenario, assign a separate column to each object. By scanning a particular column in the diagram, you can see the events that directly affect a particular object. From the sequence diagrams you can then summarize the events that each class sends and receives.

ATM example. Figure 13.3 shows a sequence diagram for the *process transaction* scenario. Figure 13.4 summarizes events with the arrows indicating the sender and receiver. For brevity, we do not show event parameters in Figure 13.4.

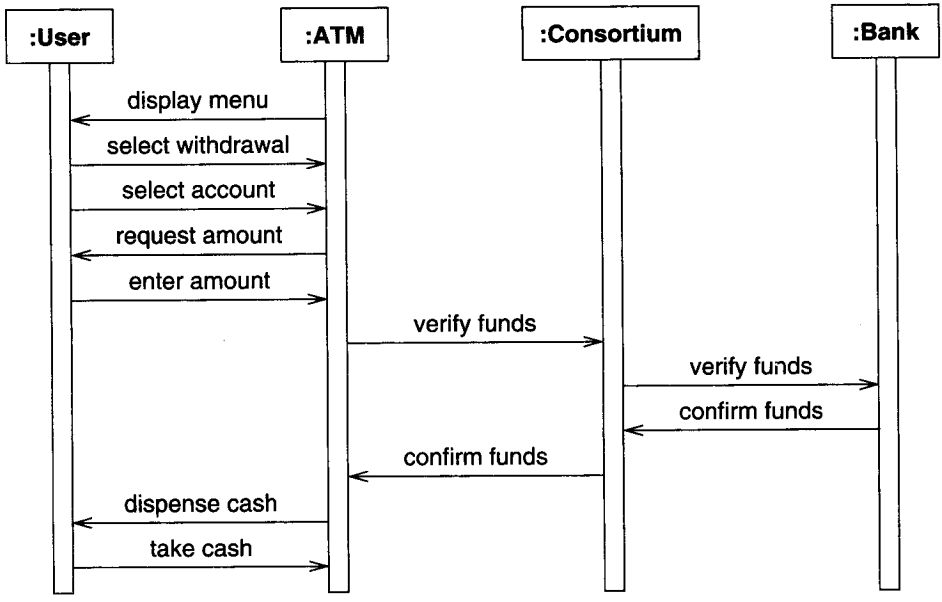

Figure 13.3 Sequence diagram for the *process transaction* scenario. A sequence diagram clearly shows the sender and receiver of each event.

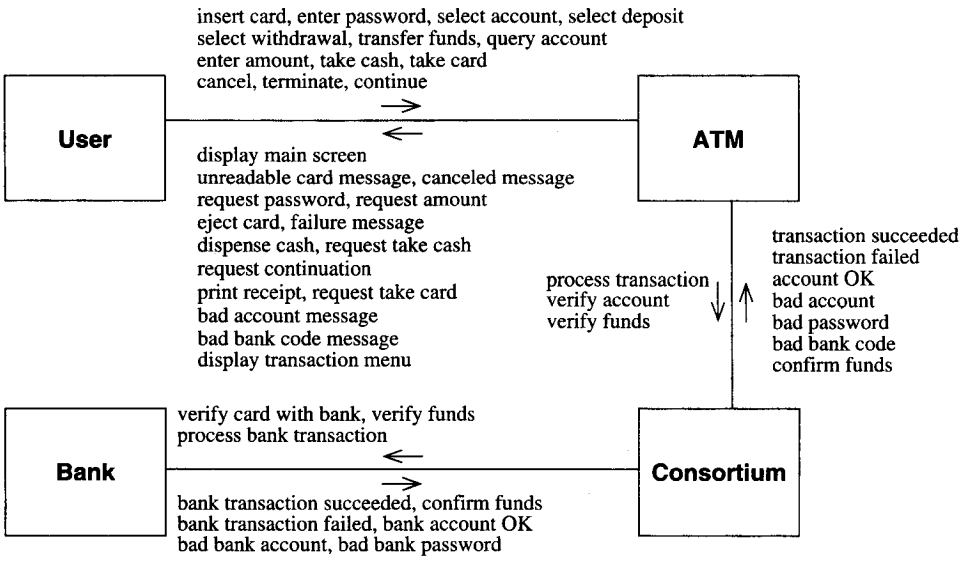

Figure 13.4 Events for the ATM case study. Tally the events in the scenarios and note the classes that send and receive each event.

13.1.8 Preparing Activity Diagrams for Complex Use Cases

Sequence diagrams capture the dialog and interplay between actors, but they do not clearly show alternatives and decisions. For example, you need one sequence diagram for the main flow of interaction and additional sequence diagrams for each error and decision point. Activity diagrams let you consolidate all this behavior by documenting forks and merges in the control flow. It is certainly appropriate to use activity diagrams to document business logic during analysis, but do not use them as an excuse to begin implementation.

ATM example. As Figure 13.5 shows, when the user inserts a card, there are many possible responses. Some responses indicate a possible problem with the card or account; hence the ATM retains the card. Only the successful completion of the tests allows ATM processing to proceed.

13.1.9 Organizing Actors and Use Cases

The next step is to organize use cases with relationships (include, extend, and generalization—see Chapter 8). This is especially helpful for large and complex systems. As with the class and state models, we defer organization until the base use cases are in place. Otherwise, there is too much of a risk of distorting the structure to match preconceived notions.

Similarly, you can also organize actors with generalization. For example, an *Administrator* might be an *Operator* with additional privileges.

ATM example. Figure 13.6 organizes the use cases with the include relationship.

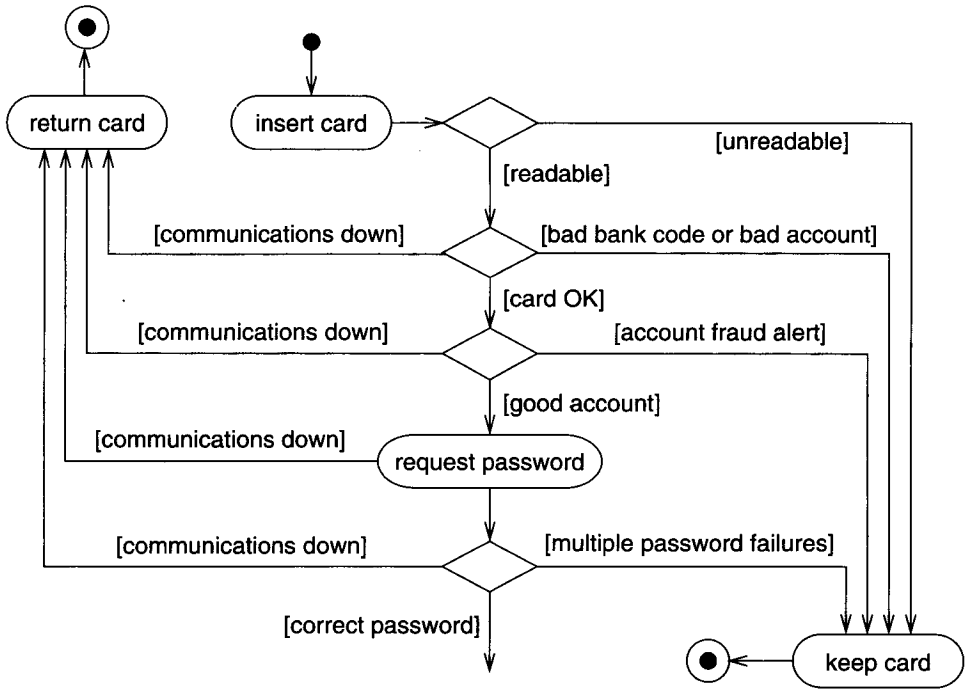

Figure 13.5 Activity diagram for card verification. You can use activity diagrams to document business logic, but do not use them as an excuse to begin premature implementation.

13.1.10 Checking Against the Domain Class Model

At this point, the application and domain models should be mostly consistent. The actors, use cases, and scenarios are all based on classes and concepts from the domain model. Recall that one of the steps in constructing the domain class model is to test access paths. In reality, such testing is a first attempt at use cases.

Cross check the application and domain models to ensure that there are no inconsistencies. Examine the scenarios and make sure that the domain model has all the necessary data. Also make sure that the domain model covers all event parameters.

13.2 Application Class Model

Application classes define the application itself, rather than the real-world objects that the application acts on. Most application classes are computer-oriented and define the way that users perceive the application. You can construct an application class model with the following steps.

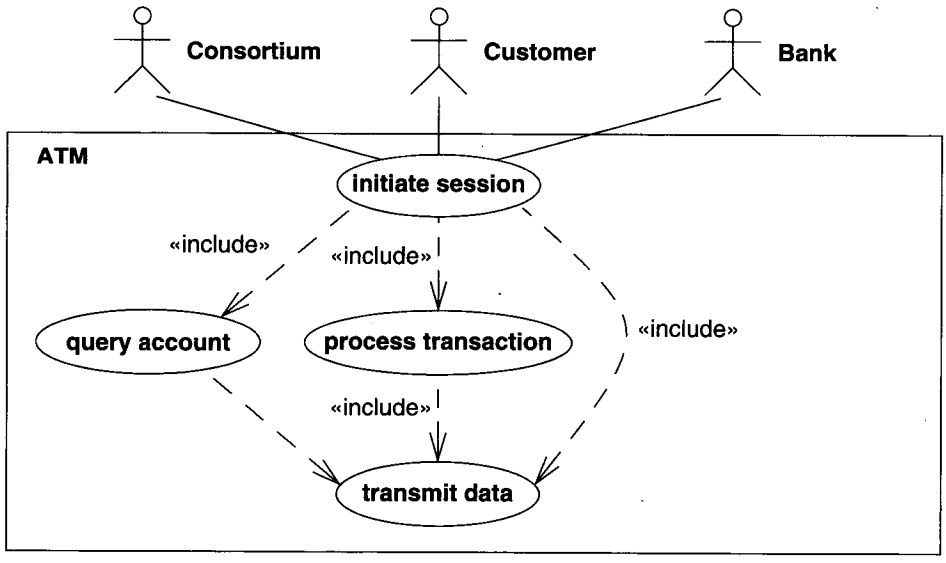

Figure 13.6 Organizing use cases. Once the basic use cases are identified, you can organize them with relationships.

- Specify user interfaces. [13.2.1]
- Define boundary classes. [13.2.2]
- Determine controllers. [13.2.3]
- Check against the interaction model. [13.2.4]

13.2.1 Specifying User Interfaces

Most interactions can be separated into two parts: application logic and the user interface. A ***user interface*** is an object or group of objects that provides the user of a system with a coherent way to access its domain objects, commands, and application options. During analysis the emphasis is on the information flow and control, rather than the presentation format. The same program logic can accept input from command lines, files, mouse buttons, touch panels, physical push buttons, or remote links, if the surface details are carefully isolated.

During analysis treat the user interface at a coarse level of detail. Don't worry about how to input individual pieces of data. Instead, try to determine the commands that the user can perform—a ***command*** is a large-scale request for a service. For example, "make a flight reservation" and "find matches for a phrase in a database" would be commands. The format of inputting the information for the commands and invoking them is relatively easy to change, so work on defining the commands first.

Nevertheless, it is acceptable to sketch out a sample interface to help you visualize the operation of an application and see if anything important has been forgotten. You may also

want to mock up the interface so that users can try it. Dummy procedures can simulate application logic. Decoupling application logic from the user interface lets you evaluate the "look and feel" of the user interface while the application is under development.

ATM example. Figure 13.7 shows a possible ATM layout. Its exact details are not important at this point. The important thing is the information exchanged.

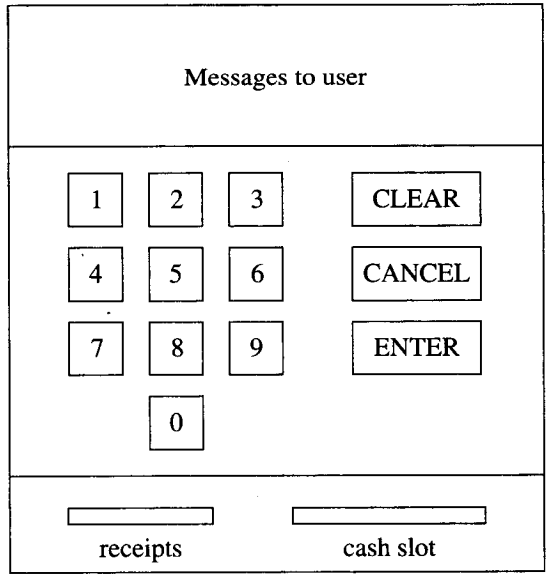

Figure 13.7 Format of ATM interface. Sometimes a sample interface can help you visualize the operation of an application.

13.2.2 Defining Boundary Classes

A system must be able to operate with and accept information from external sources, but it should not have its internal structure dictated by them. It is often helpful to define boundary classes to isolate the inside of a system from the external world. A ***boundary class*** is a class that provides a staging area for communications between a system and an external source. A boundary class understands the format of one or more external sources and converts information for transmission to and from the internal system.

ATM example. It would be helpful to define boundary classes (*CashCardBoundary*, *AccountBoundary*) to encapsulate the communication between the ATM and the consortium. This interface will increase flexibility and make it easier to support additional consortiums.

13.2.3 Determining Controllers

A ***controller*** is an active object that manages control within an application. It receives signals from the outside world or from objects within the system, reacts to them, invokes operations

on the objects in the system, and sends signals to the outside world. A controller is a piece of reified behavior captured in the form of an object—behavior that can be manipulated and transformed more easily than plain code. At the heart of most applications are one or more controllers that sequence the behavior of the application.

Most of the work in designing a controller is in modeling its state diagram. In the application class model, however, you should capture the existence of the controllers in a system, the control information that each one maintains, and the associations from the controllers to other objects in the system.

ATM example. It is apparent from the scenarios in Figure 13.2 that the ATM has two major control loops. The outer loop verifies customers and accounts. The inner loop services transactions. Each of these loops could most naturally be handled with a controller.

13.2.4 Checking Against the Interaction Model

As you build the application class model, go over the use cases and think about how they would work. For example, if a user sends a command to the application, the parameters of the command must come from some user-interface object. The requesting of the command itself must come from some controller object. When the domain and application class models are in place, you should be able to simulate a use case with the classes. Think in terms of navigation of the models, as we discussed in Chapter 3. This manual simulation helps to establish that all the pieces are in place.

ATM example. Figure 13.8 shows a preliminary application class model and the domain classes with which it interacts. There are two interfaces—one for users and the other for communicating with the consortium. The application model just has stubs for these classes, because it is not clear how to elaborate them at this time.

Note that the boundary classes "flatten" the data structure and combine information from multiple domain classes. For simplicity, it is desirable to minimize the number of boundary classes and their relationships.

The *TransactionController* handles both queries on accounts and the processing of transactions. The *SessionController* manages *ATMsessions*, each of which services a customer. Each *ATMsession* may or may not have a valid *CashCard* and *Account*. The *SessionController* has a status of *ready*, *impaired* (such as out of paper or cash but still able to operate for some functions), or *down* (such as a communications failure). There is a log of *ControllerProblems* and the specific problem type (bad card reader, out of paper, out of cash, communication lines down, etc.).

13.3 Application State Model

The application state model focuses on application classes and augments the domain state model. Application classes are more likely to have important temporal behavior than domain classes.

First identify application classes with multiple states and use the interaction model to find events for these classes. Then organize permissible event sequences for each class with

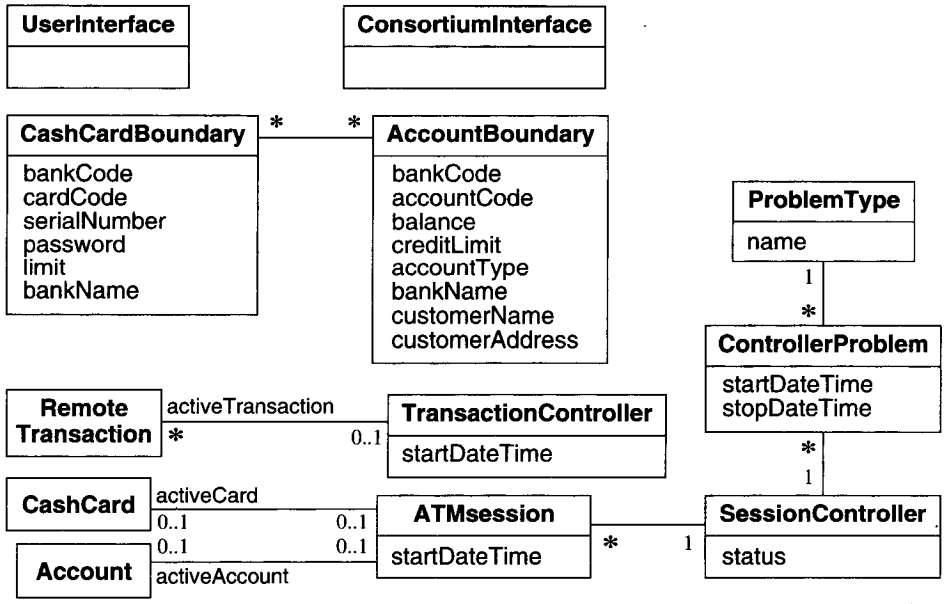

Figure 13.8 ATM application class model. Application classes augment the domain classes and are necessary for development.

a state diagram. Next, check the various state diagrams to make sure that common events match. And finally check the state diagrams against the class and interaction models to ensure consistency.

You can construct an application state model with the following steps.

- Determine application classes with states. [13.3.1]
- Find events. [13.3.2]
- Build state diagrams. [13.3.3]
- Check against other state diagrams. [13.3.4]
- Check against the class model. [13.3.5]
- Check against the interaction model. [13.3.6]

13.3.1 Determining Application Classes with States

The application class model adds computer-oriented classes that are prominent to users and important to the operation of an application. Consider each application class and determine which ones have multiple states. User interface classes and controller classes are good candidates for state models. In contrast, boundary classes tend to be static and used for staging data import and export—consequently they are less likely to involve a state model.

ATM example. The user interface classes do not seem to have any substance. This is probably because our understanding of the user interface is incomplete at this point in devel-

opment. The boundary classes also lack state behavior. However, the controllers do have important states that we will elaborate.

13.3.2 Finding Events

For the application interaction model, you prepared a number of scenarios. Now study those scenarios and extract events. Even though the scenarios may not cover every contingency, they ensure that you do not overlook common interactions and they highlight the major events.

Note the contrast between the domain and application processes for state models. With the domain model, first we find states and then we find events. That is because the domain model focuses on data—significant groupings of data form states that are subject to events. With the application model, in contrast, first we find events and then we determine states. The application model's early attention to events is a consequence of the emphasis on behavior— use cases are elaborated with scenarios that reveal events.

ATM example. We revisit the scenarios from the application interaction model. Some events are: *insert card*, *enter password*, *end session*, and *take card*.

13.3.3 Building State Diagrams

The next step is to build a state diagram for each application class with temporal behavior. Choose one of these classes and consider a sequence diagram. Arrange the events involving the class into a path whose arcs are labeled by the events. The interval between any two events is a state. Give each state a name, if a name is meaningful, but don't bother if it is not. Now merge other sequence diagrams into the state diagram. The initial state diagram will be a sequence of events and states. Every scenario or sequence diagram corresponds to a path through the state diagram.

Now find loops within the diagram. If a sequence of events can be repeated indefinitely, then they form a loop. In a loop, the first state and the last state are identical. If the object "remembers" that it has traversed a loop, then the two states are not really identical, and a simple loop is incorrect. At least one state in a loop must have multiple transactions leaving it or the loop will never terminate.

Once you have found the loops, merge other sequence diagrams into the state diagram. Find the point in each sequence diagram where it diverges from previous ones. This point corresponds to an existing state in the diagram. Attach the new event sequence to the existing state as an alternative path. While examining sequence diagrams, you may think of other possible events that can occur at each state; add them to the state diagram as well.

The hardest thing is deciding at which state an alternate path rejoins the existing diagram. Two paths join at a state if the object "forgets" which one was taken. In many cases, it is obvious from knowledge of the application that two states are identical. For example, inserting two nickels into a vending machine is equivalent to inserting one dime.

Beware of two paths that appear identical but can be distinguished under some circumstances. For example, some systems repeat the input sequence if the user makes an error entering information but give up after a certain number of failures. The repeat sequence is almost the same except that it remembers the past failures. The difference can be glossed

over by adding a parameter, such as *number of failures,* to remember information. At least one transition must depend on the parameter.

The judicious use of parameters and conditional transitions can simplify state diagrams considerably but at the cost of mixing together state information and data. State diagrams with too much data dependency can be confusing and counterintuitive. Another alternative is to partition a state diagram into two concurrent subdiagrams, using one subdiagram for the main line and the other for the distinguishing information. For example, a subdiagram to allow for one user failure might have states *No error* and *One error.*

After normal events have been considered, add variation and exception cases. Consider events that occur at awkward times—for example, a request to cancel a transaction after it has been submitted for processing. In cases when the user (or other external agent) may fail to respond promptly and some resource must be reclaimed, a *time-out* event can be generated after a given interval. Handling user errors cleanly often requires more thought and code than the normal case. Error handling often complicates an otherwise clean and compact program structure, but it must be done.

You are finished with the state diagram of a class when the diagram covers all scenarios and the diagram handles all events that can affect a state. You can use the state diagram to suggest new scenarios by considering how some event not already handled should affect a state. Posing "what if" questions is a good way to test for completeness and error-handling capabilities.

If there are complex interactions with independent inputs, you can use a nested state diagram, as Chapter 6 describes. Otherwise a flat state diagram suffices. Repeat the above process of building state diagrams for each class that has time-dependent behavior.

ATM example. Figure 13.9 shows the state diagram for the *SessionController.* The middle of the diagram has the main behavior of processing the card and password. A communications failure can interrupt processing at any time. The ATM returns the card upon a communications failure, but keeps it if there are any suspicious circumstances. After finishing transactions, receipt printing occurs in parallel to card ejection, and the user can take the receipt and card in any order.

Figure 13.10 and Figure 13.11 show the state diagram for the *TransactionController* that is spawned by the *SessionController.* (See the exercises for the other subdiagrams of Figure 13.10.) We have separated the *TransactionController* and the *SessionController* because their purposes are much different—the *SessionController* focuses on verifying users, while the *TransactionController* services account inquiries and balance changes.

13.3.4 Checking Against Other State Diagrams

Check the state diagrams of each class for completeness and consistency. Every event should have a sender and a receiver, occasionally the same object. States without predecessors or successors are suspicious; make sure they represent starting or termination points of the interaction sequence. Follow the effects of an input event from object to object through the system to make sure that they match the scenarios. Objects are inherently concurrent; beware of synchronization errors where an input occurs at an awkward time. Make sure that corresponding events on different state diagrams are consistent.

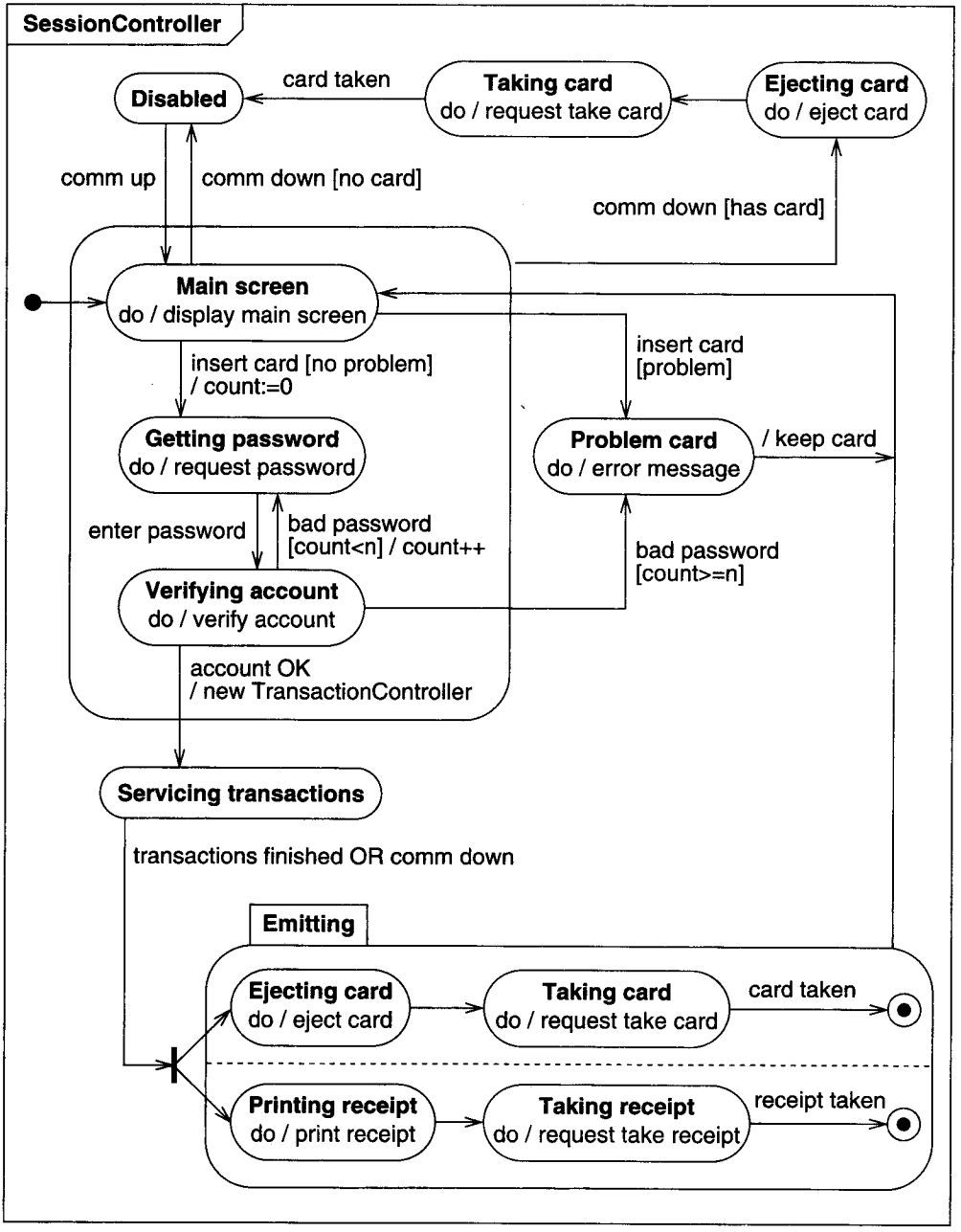

Figure 13.9 State diagram for *SessionController*. Build a state diagram for each application class with temporal behavior.

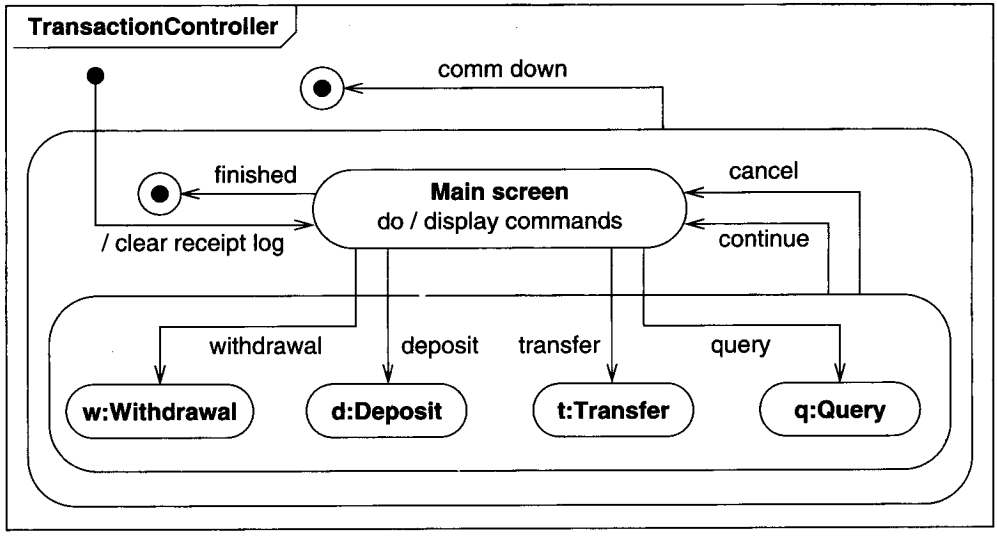

Figure 13.10 State diagram for *TransactionController*. Obtain information from the scenarios of the interaction model.

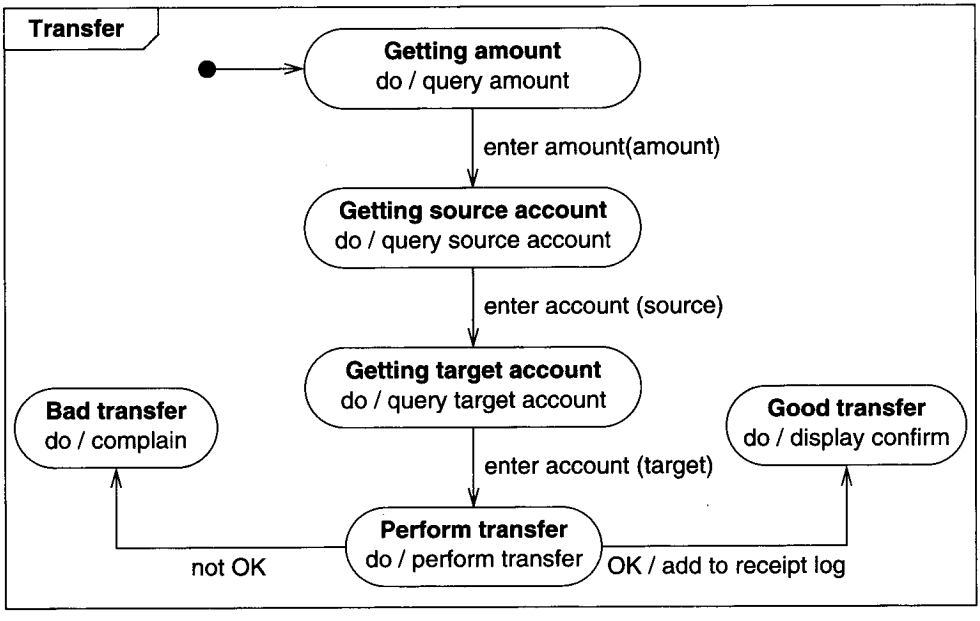

Figure 13.11 State diagram for *Transfer*. This diagram elaborates the *Transfer* state in Figure 13.10.

ATM example. The *SessionController* initiates the *TransactionController*, and the termination of the *TransactionController* causes the *SessionController* to resume.

13.3.5 Checking Against the Class Model

Similarly, make sure that the state diagrams are consistent with the domain and application class models.

ATM example. Multiple ATMs can potentially concurrently access an account. Account access needs to be controlled to ensure that only one update at a time is applied. We will not resolve the details here.

13.3.6 Checking Against the Interaction Model

When the state model is ready, go back and check it against the scenarios of the interaction model. Simulate each behavior sequence by hand and verify that the state diagram gives the correct behavior. If an error is discovered, change either the state diagram or the scenarios. Sometimes a state diagram will uncover irregularities in the scenarios, so don't assume that the scenarios are always correct.

Then take the state model and trace out legitimate paths. These represent additional scenarios. Ask yourself whether they make sense. If not, then modify the state diagram. Often, however, you will discover useful behavior that you had not considered before. The mark of a good design is the discovery of unexpected information that follows from the design, properties that appear meaningful (and often seem obvious) once they are observed.

ATM example. As best as we can tell right now, the state diagrams are sound and consistent with the scenarios.

13.4 Adding Operations

Our style of object-oriented analysis places much less emphasis on defining operations than the traditional programming-based methodologies. We de-emphasize operations because the list of potentially useful operations is open-ended and it is difficult to know when to stop adding them. Operations arise from the following sources, and you should add major operations now. Chapter 15 discusses detailed operations.

13.4.1 Operations from the Class Model

The reading and writing of attribute values and association links are implied by the class model, and you need not show them explicitly. During analysis all attributes and associations are assumed to be accessible.

13.4.2 Operations from Use Cases

Most of the complex functionality of a system comes from its use cases. During the construction of the interaction model, use cases lead to activities. Many of these correspond to operations on the class model.

ATM example. *Consortium* has the activity *verifyBankCode,* and *Bank* has the activity *verifyPassword.* You could implement Figure 13.5 with the operation *verifyCashCard* on class *ATM.*

13.4.3 Shopping-List Operations

Sometimes the real-world behavior of classes suggests operations. Meyer [Meyer-97] calls this a "shopping list," because the operations are not dependent on a particular application but are meaningful in their own right. Shopping-list operations provide an opportunity to broaden a class definition beyond the narrow needs of the immediate problem.

ATM example. Shopping-list operations include:

- account.close()
- bank.createSavingsAccount(customer): account
- bank.createCheckingAccount(customer): account
- bank.createCashCardAuth(customer): cashCardAuthorization
- cashCardAuthorization.addAccount (account)
- cashCardAuthorization.removeAccount (account)
- cashCardAuthorization.close()

13.4.4 Simplifying Operations

Examine the class model for similar operations and variations in form on a single operation. Try to broaden the definition of an operation to encompass such variations and special cases. Use inheritance where possible to reduce the number of distinct operations. Introduce new superclasses as needed to simplify the operations, provided that the new superclasses are not forced and unnatural. Locate each operation at the correct level within the class hierarchy. A result of this refinement is often fewer, more powerful operations that are nevertheless simpler to specify than the original operations, because they are more uniform and general.

ATM example. The ATM example does not require simplification. Figure 13.12 adds some operations to the ATM domain class model from Chapter 12.

13.5 Chapter Summary

The purpose of analysis is to understand the problem so that a correct design can be constructed. A good analysis captures the essential features of the problem without introducing implementation artifacts that prematurely restrict design decisions.

There are two phases of analysis—domain and application. Domain analysis captures general knowledge about an application. Domain analysis involves class models and sometimes state models but seldom has an interaction model. In contrast, application analysis focuses on major application artifacts that are important, visible to users, and must be approved by them. The interaction model dominates application analysis, but the class and state models are also important.