Object-Oriented Modeling and Design

James Rumbaugh
Michael Blaha
William Premerlani
Frederick Eddy
William Lorensen

General Electric Research and Development Center
Schenectady, New York

Prentice-Hall International, Inc.

a computer operating system might contain modules for process control, device control, file maintenance, and memory management. Modules may vary in size.

Each association should generally be shown on a single sheet, but some classes must be shown more than once to connect different sheets. Look for cut points among the classes: a class that is the sole connection between two otherwise disconnected parts of the object network. Such a class forms the bridge between two sheets or modules. For example, in a file management system, a file is the cut point between the directory structure and the file contents. If a single cut point cannot be found, try to minimize the number of bridge classes. Try to choose modules to reduce the number of crossovers in the object diagram. With a little care, most object diagrams can be drawn as planar graphs, without crossing lines.

A "star" pattern is frequently useful for organizing modules: a single core module contains the top-level structure of high-level classes. Other modules expand each high-level class into a generalization hierarchy and add associations to additional low-level classes.

Reuse a module from a previous design if possible, but avoid forcing a fit. Reuse is easiest when part of the problem domain matches a previous problem. If the new problem is similar to a previous problem but different, the original design may have to be extended to encompass both problems. Use your judgment about whether this is better than building a new design.

ATM example. The model we have presented is small and would not require breakdown into modules, but it could serve as a core for a more detailed model. The modules might be:

- tellers—cashier, entry station, cashier station, ATM
- accounts—account, cash card, card authorization, customer, transaction, update, cashier transaction, remote transaction
- banks—consortium, bank

Each module could add details: The account module could contain varieties of transactions, information about customers, interest payments and fees; the banks module could contain information about branches, addresses, and cost allocations.

8.5 DYNAMIC MODELING

The dynamic model shows the time-dependent behavior of the system and the objects in it. Begin dynamic analysis by looking for events—externally-visible stimuli and responses. Then summarize permissible event sequences for each object with a state diagram. Algorithm execution is not relevant during analysis if there are no externally-visible manifestations; algorithms are part of implementation.

The dynamic model is insignificant for a purely static data repository, such as a database. The dynamic model is important for interactive systems. For most problems, logical correctness depends on the sequences of interactions, not the exact times of interactions. Real-time systems, however, do have specific timing requirements on interactions that must be considered during analysis. We do not address real-time analysis in this book.

First prepare scenarios of typical dialogs. Even though these scenarios may not cover every contingency, they at least ensure that common interactions are not overlooked. Extract events from the scenarios. It is usually best to identify events first and then assign each event to its target object. Organize the sequences of events and states into a state diagram. Finally compare state diagrams for different objects to make sure the events exchanged by them match. The resulting set of state diagrams constitute the dynamic model.

In summary, the following steps are performed in constructing a dynamic model:

- Prepare scenarios of typical interaction sequences [8.5.1]
- Identify events between objects [8.5.3]
- Prepare an event trace for each scenario [8.5.3]
- Build a state diagram [8.5.4]
- Match events between objects to verify consistency [8.5.5]

8.5.1 Preparing a Scenario

Prepare one or more typical dialogs between user and system to get a feel for expected system behavior. These scenarios show the major interactions, external display formats, and information exchanges. Approach the dynamic model by scenarios, rather than trying to write down the general model directly, to ensure that important steps are not overlooked and that the overall flow of the interaction is smooth and correct.

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 format. 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 what order to ask for them. The problem statement may specify needed information but leaves open the manner in which it is obtained. In many applications, gathering input is a major task or sometimes the only major task. The dynamic model is crucial in such applications.

First prepare scenarios for "normal" cases, interactions without any unusual inputs or error conditions. Then consider "special" cases, such as omitted input sequences, maximum and minimum values, and repeated values. Then consider user error cases, including invalid values and failures to respond. For many interactive applications, error handling is the most difficult part of the implementation. 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.

A *scenario* is a sequence of events. 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 parameters of the event. 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—a pure signal. Anytime information is input to the system or output from the system, an event occurs.

For each event, identify the actor (system, user, or other external agent) that caused the event and the parameters of the event. The screen layout or output format generally doesn't affect the logic of the interaction or the values exchanged. Don't worry about output formats for the initial dynamic model; describe output formats during refinement of the model.

ATM example. Figure 8.15 shows a normal ATM scenario. Figure 8.16 shows a scenario with exceptions.

The ATM asks the user to insert a card; the user inserts a cash card.
The ATM accepts the card and reads its serial number.
The ATM requests the password; the user enters "1234."
The ATM verifies the serial number and password with the consortium; the consortium checks it with bank "39" and notifies the ATM of acceptance.
The ATM asks the user to select the kind of transaction (withdrawal, deposit, transfer, query); the user selects withdrawal.
The ATM asks for the amount of cash; the user enters $100.
The ATM verifies that the amount is within predefined policy limits and asks the consortium to process the transaction; the consortium passes the request to the bank, which eventually confirms success and returns the new account balance.
The ATM dispenses cash and asks the user to take it; the user takes the cash.
The ATM asks whether the user wants to continue; the user indicates no.
The ATM prints a receipt, ejects the card, and asks the user to take them; the user takes the receipt and the card.
The ATM asks a user to insert a card.

Figure 8.15 Normal ATM scenario

The ATM asks the user to insert a card; the user inserts a cash card.
The ATM swallows the card and reads its serial number.
The ATM requests the password; the user enters "9999."
The ATM verifies the serial number and password with the consortium, which rejects it after consulting the appropriate bank.
The ATM indicates a bad password and asks the user to renter it; the user enters "1234" which the ATM successfully verifies with the consortium.
The ATM asks the user to select the kind of transaction; the user selects withdrawal.
The ATM asks for the amount of cash; the user has a change of mind and hits "cancel."
The ATM ejects the card and asks the user to take it; the user takes it.
The ATM asks a user to insert a card.

Figure 8.16 ATM scenario with exceptions

Many other variations can be described: The user fails to insert the card in time, the ATM can't read the card, the card has expired, the transaction is inappropriate to the account, the amount is invalid, the machine is out of cash or paper, the remote communication lines are down, or the transaction is rejected because of suspicious patterns of usage on the card.

Additional scenarios should be written for administrative parts of the ATM system, such as authorizing new cards, opening accounts, adding banks to the consortium, and obtaining transaction logs. We will not explore these aspects.

8.5.2 Interface Formats

Most interactions can be separated into two parts: application logic and the user interface. The analysis should concentrate first 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. The dynamic model captures the control logic of the application.

It is hard to evaluate a user interface without actually testing it. Often the interface can be mocked up so that users can try it. Application logic can often be simulated with dummy procedures. Decoupling application logic from the user interface allows the "look and feel" of the user interface to be evaluated while the application is under development.

Figure 8.17 shows a possible ATM layout. Its exact details are not important at this point, even less the wording of the messages. The important thing is the information exchanged, in whatever form. For example, don't worry about the sequence of keystrokes needed to enter a password; treat "enter password" as an atomic event. Nevertheless, it is good to draw at least one possible illustration of each interaction format, to help make sure nothing important is forgotten.

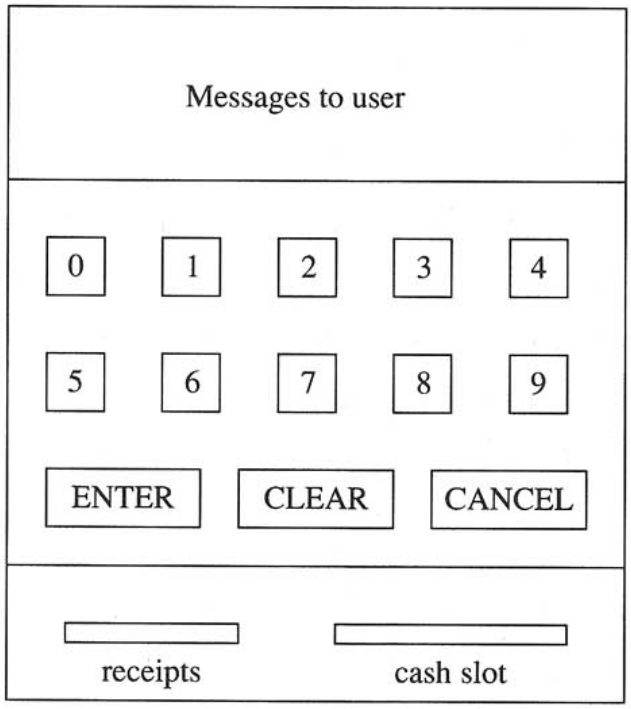


Figure 8.17 Format of ATM interface

8.5.3 Identifying Events

Examine the scenarios to identify all external events. Events include all signals, inputs, decisions, interrupts, transitions, and actions to or from users or external devices. Internal computation steps are not events, except for decision points that interact with the external world. Use scenarios to find normal events, but don't forget error conditions and unusual events.

An action by an object that transmits information is an event. For example, *enter password* is an event sent from external agent *User* to application object *ATM*. Most object-to-object interactions and operations correspond to events. For example, *insert card* is an event sent from *User* to *ATM*. Some information flows are implicit. For example, *dispense cash* is an event from *ATM* to *User*.

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 class, since the password value does not affect the flow of control. Similarly *dispense cash* is also an event class, since the amount of cash dispensed does not affect the flow of control. Events that affect the flow of control should be distinguished. *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. For example, 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, would probably be considered a distinct event, since the application would respond differently. The distinction depends on the application. You may have to construct the state diagram before you can classify all events; some distinctions between events may have no effect on behavior and can be ignored.

Allocate each type of event to the object classes that send it and receive it. The event is an output event for the sender and an input event for the receiver. Sometimes an object sends an event to itself, in which case the event is both an output and an input for the same class.

Show each scenario as an *event trace*—an ordered list of events between different objects assigned to columns in a table. Figure 8.18 shows an event trace for an ATM scenario. 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 trace, you can see the events that directly affect a particular object. Only these events can appear in the state diagram for the object.

Show the events between a group of classes (such as a module) on an *event flow diagram*, as in Figure 8.19. This diagram summarizes events between classes, without regard for sequence. Include events from all scenarios, including error events. The event flow diagram is a dynamic counterpart to an object diagram. Paths in the object diagram show possible information flows; paths in the event flow diagram show possible control flows.

8.5.4 Building a State Diagram

Prepare a state diagram for each object class with nontrivial dynamic behavior, showing the events the object receives and sends. Every scenario or event trace corresponds to a path through the state diagram. Each branch in control flow is represented by a state with more than one exit transition.

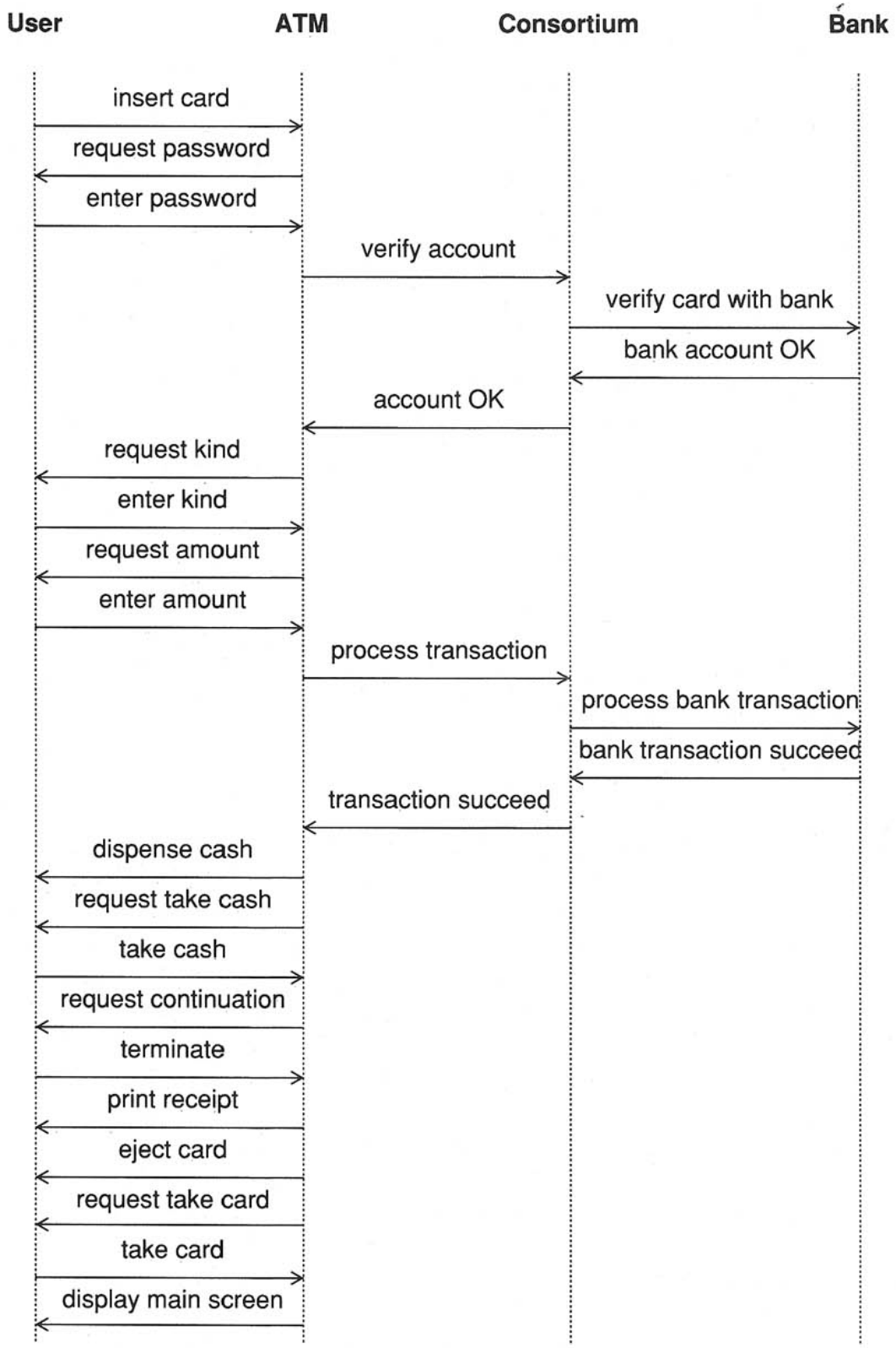


Figure 8.18 Event trace for ATM scenario

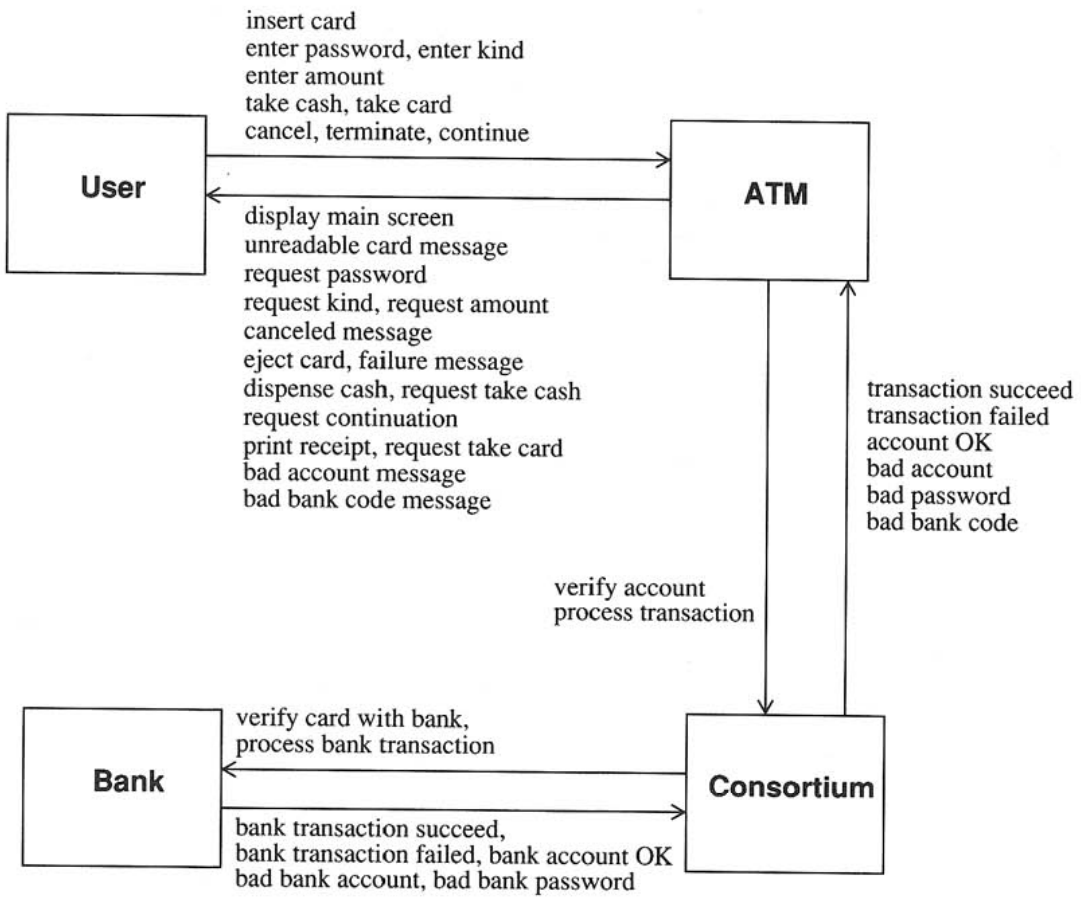


Figure 8.19 Event flow diagram for ATM example

Start with the event trace diagrams that affect the class being modeled. Pick a trace showing a typical interaction and only consider the events affecting a single object. Arrange the events into a path whose arcs are labeled by the input and output events found along one column in the trace. 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. The initial diagram will be a sequence of events and states. If the scenario can be repeated indefinitely, close the path in the state diagram.

Now find loops within the diagram. If a sequence of events can be repeated indefinitely, then they form a loop. Replace finite sequences of events with loops when possible. 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.

Now merge other scenarios into the state diagram. Find the point in each scenario where it diverges from previous scenarios. 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 states and scenarios, you may think of other possible events that can occur at each state; add them to the state diagram as well.

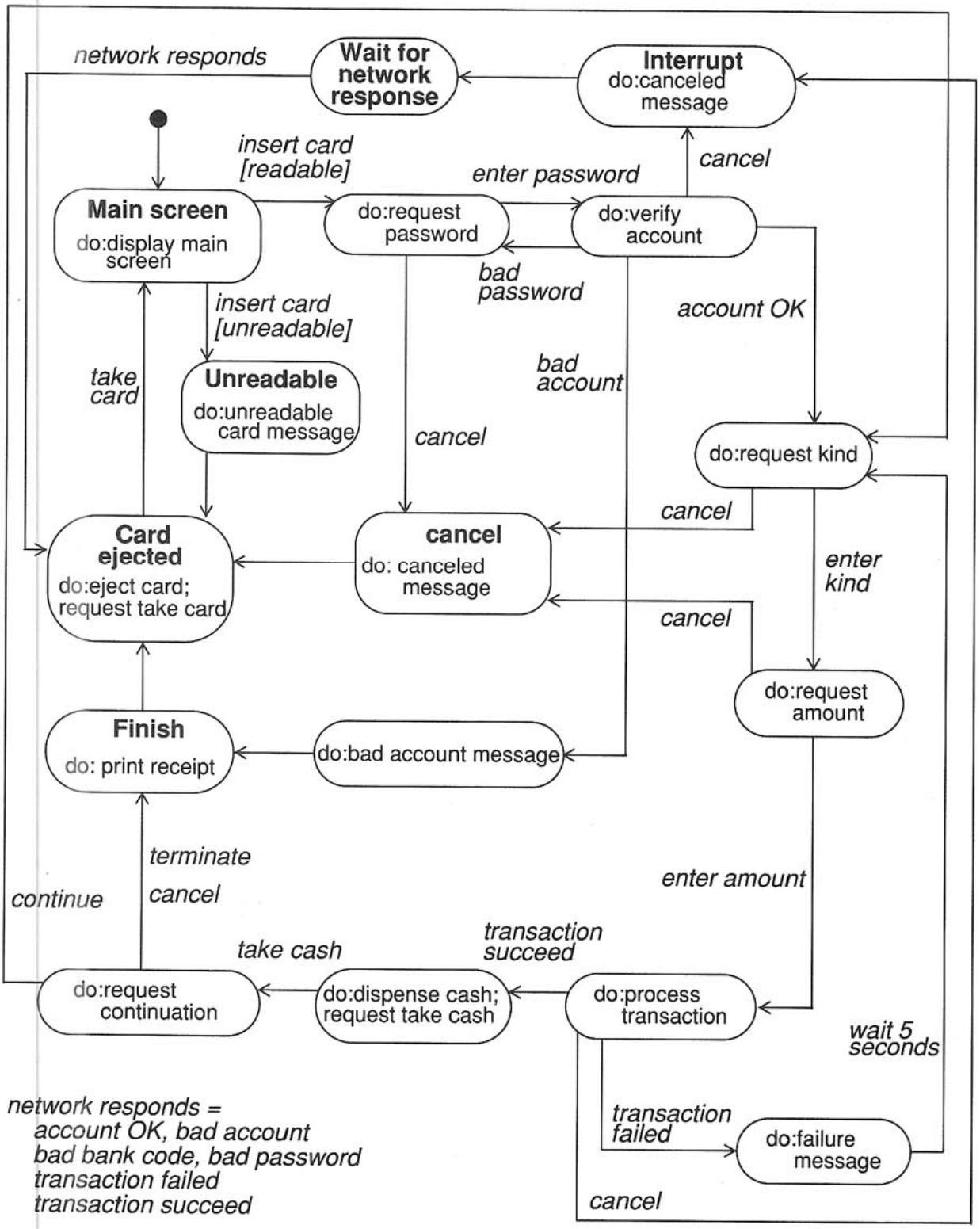


Figure 8.20 State diagram for class *ATM*

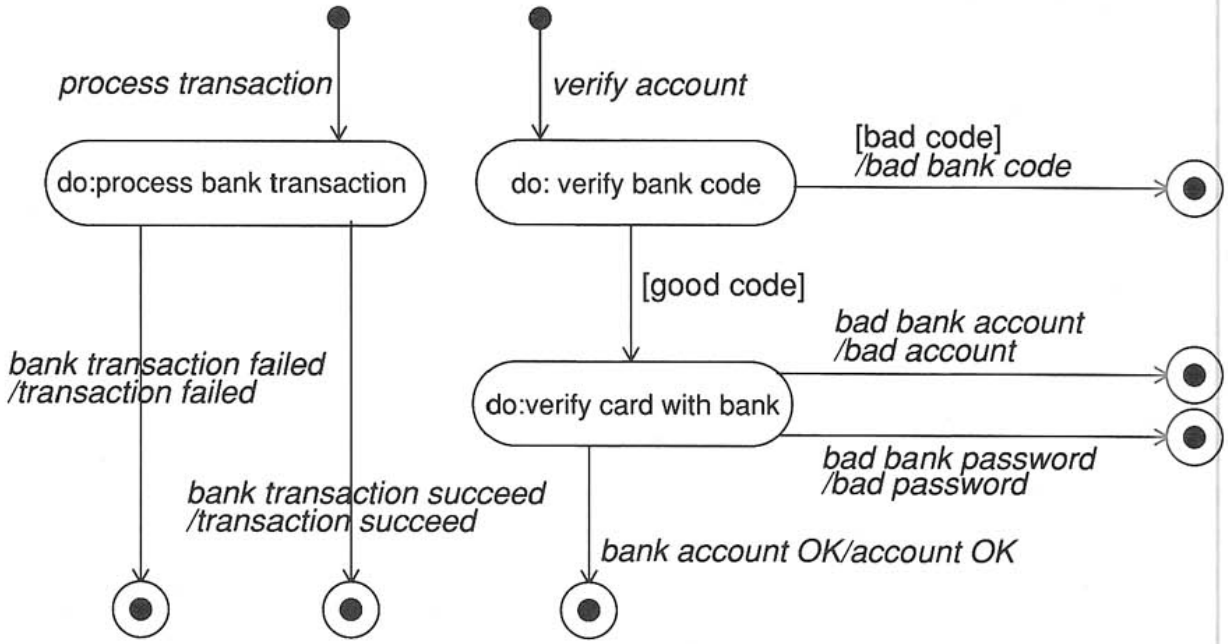


Figure 8.21 State diagram for class *Consortium*

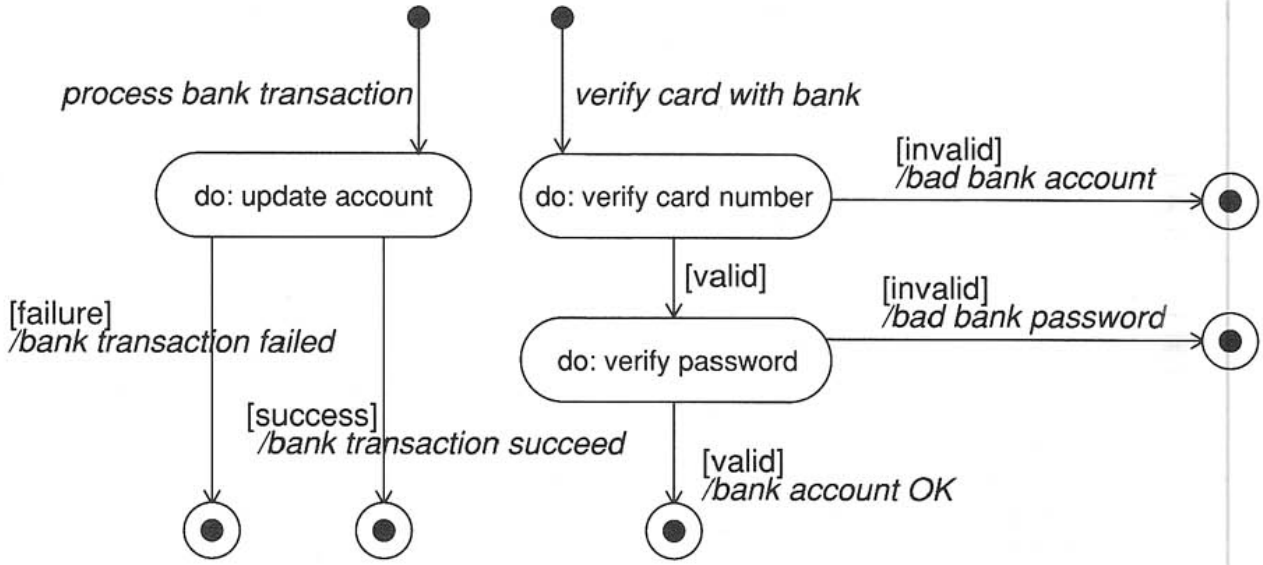


Figure 8.22 State diagram for class *Bank*

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 your 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 which 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 value of 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 boundary cases and special cases, as discussed in Section 8.5.1. 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 an object of the class in each of its states. You can use the state diagram to suggest new scenarios by considering how some event not already handled should affect the state of the object. Posing "What if" questions is a good way to test completeness and error-handling capabilities of a class (and should be repeated at the module and system levels as well).

If there are complex interactions with independent inputs, you can organize the dynamic model using a nested state diagram, as described in Chapter 5. Otherwise a flat state diagram is usually adequate.

Repeat the above process of building state diagrams for each class of objects. Concentrate on classes with important interactions.

Not all classes need a state diagram. Many objects respond to input events independently of their past history, or capture all important history as parameters that do not affect control. Such objects may receive and send events. List the input events for each object and the output events sent in response to each input event, but there will be no further state structure.

Eventually you may be able to write down state diagrams directly without preparing event traces. A few scenarios are usually helpful, in any case.

ATM example. Objects *ATM, Cashier station, Consortium,* and *Bank* are actors that exchange events. Objects *Cash card, Transaction,* and *Account* are passive objects that are acted on and do not exchange events. The customer and cashier are actors, but their interactions

with the entry stations are already shown; the customer and cashier objects are external to the system and need not be implemented within it anyway. Figure 8.20 shows the state machine for the ATM. Figure 8.21 shows the state diagram for the consortium. Many copies of the diagram may be active concurrently; each diagram corresponds to one transaction. Figure 8.22 shows the state machine for the bank. Again, each diagram corresponds to one transaction. These state diagrams are all simplistic, especially with regard to their error handling. For example, Figure 8.20 makes no provision for failure of the network communication link. In such a case, an ATM would be expected to eject the customer's card; we do not show this behavior. The state machine for *Cashier station* has been omitted to save space; it is similar to *ATM*.

8.5.5 Matching Events Between Objects

Check for completeness and consistency at the system level when the state diagrams for each class are complete. 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. The set of state diagrams for object classes with important dynamic behavior constitute the dynamic model for an application.

ATM example. An account can potentially be accessed concurrently by more than one machine. Access to an account needs to be controlled to ensure that only one update at a time is applied.

Examination of the state diagrams shows that *bad bank code* is sent by consortium but not received by ATM. It needs to be added, followed by action *print bad bank code* and a transition to *card ejected*.

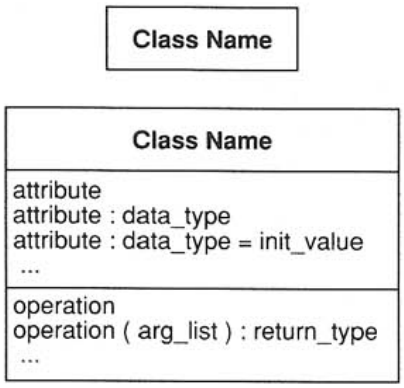
8.6 FUNCTIONAL MODELING

The functional model shows how values are computed, without regard for sequencing, decisions, or object structure. The functional model shows which values depend on which other values and the functions that relate them. Data flow diagrams are useful for showing functional dependencies. Functions are expressed in various ways, including natural language, mathematical equations, and pseudocode.

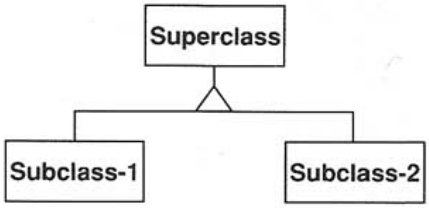
The processes on a data flow diagram correspond to activities or actions in the state diagrams of the classes. The flows on a data flow diagram correspond to objects or attribute values in an object diagram. It is best to construct the functional model after the object and dynamic models.

Object Model Notation
Basic Concepts

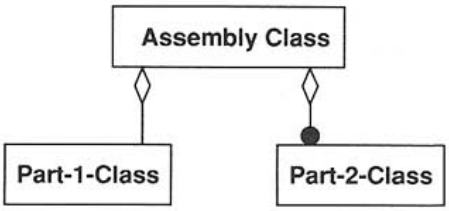
Class:



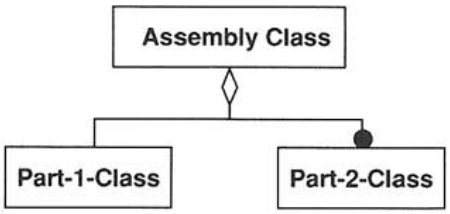
Generalization (Inheritance):



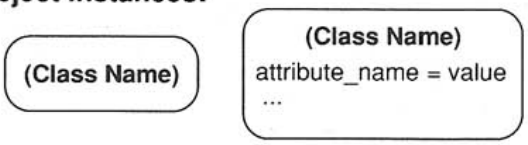
Aggregation:



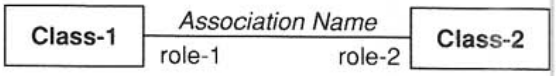
Aggregation (alternate form):



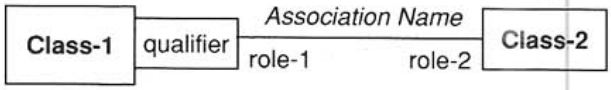
Object Instances:



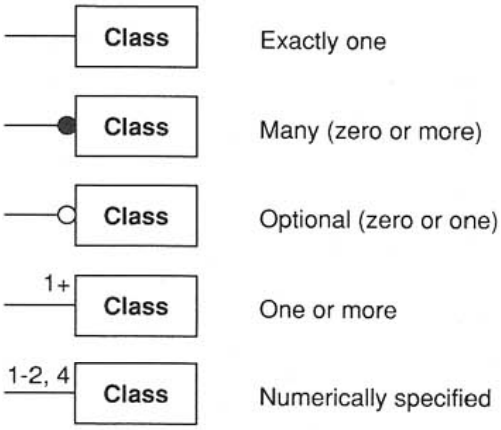
Association:



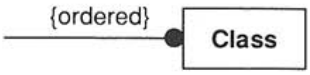
Qualified Association:



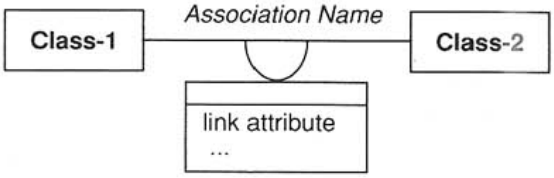
Multiplicity of Associations:



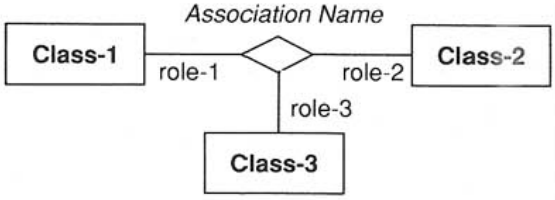
Ordering:



Link Attribute:



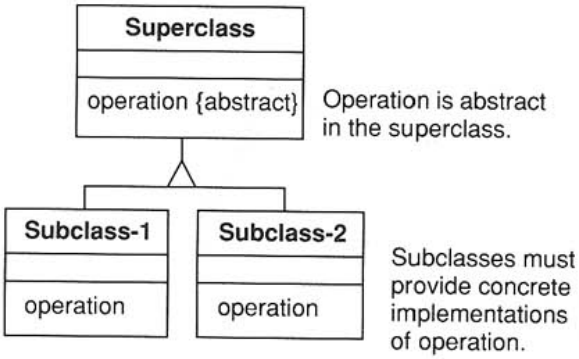
Ternary Association:



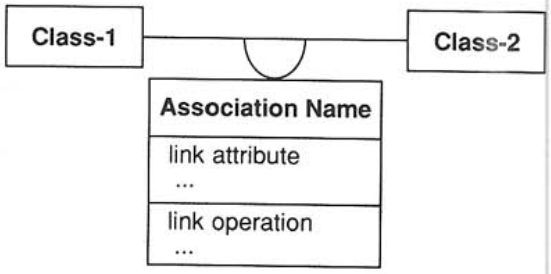
Instantiation Relationship:

Object Model Notation
Advanced Concepts

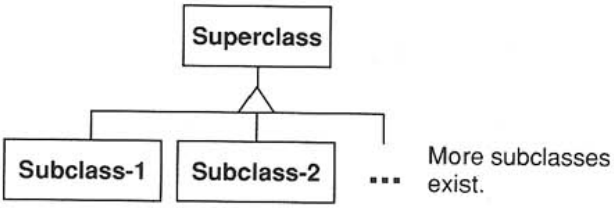
Abstract Operation:

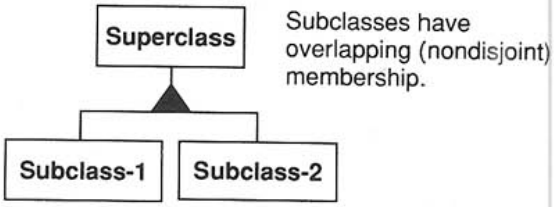


Association as Class:

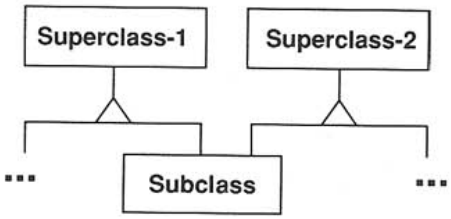


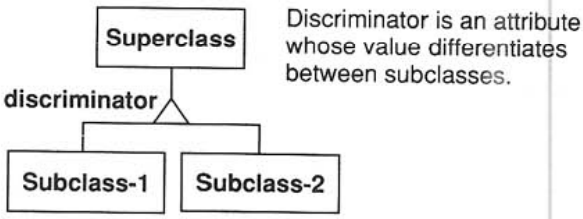
Generalization Properties:



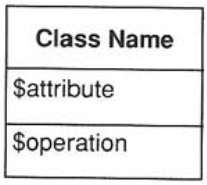


Multiple Inheritance:

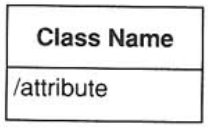




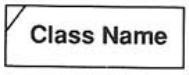
Class Attributes and Class Operations:



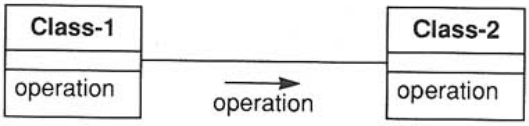
Derived Attribute:



Derived Class:

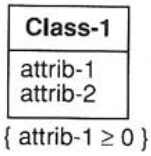
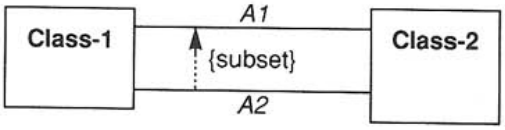


Propagation of Operations:



Derived Association:

Constraints on Objects:



Constraint between Associations: