Constructing Operational Specifications

, June 01, 1995
22-27 minutes

Source Code Accompanies This Article. Download It Now.

- opspec.zip

Complementing existing methodologies, Mark and Terry propose an operational specification that can be translated into most existing system-development methodologies.

A straightforward approach that complements formal design methodologies

Mark Coats and Terry Mellon

Mark is a senior software engineer for Motorola and can be contacted at mark_coats@email.mot.com. Terry is president of Software Engineering Excellence and can be reached at mellon@seex.com.

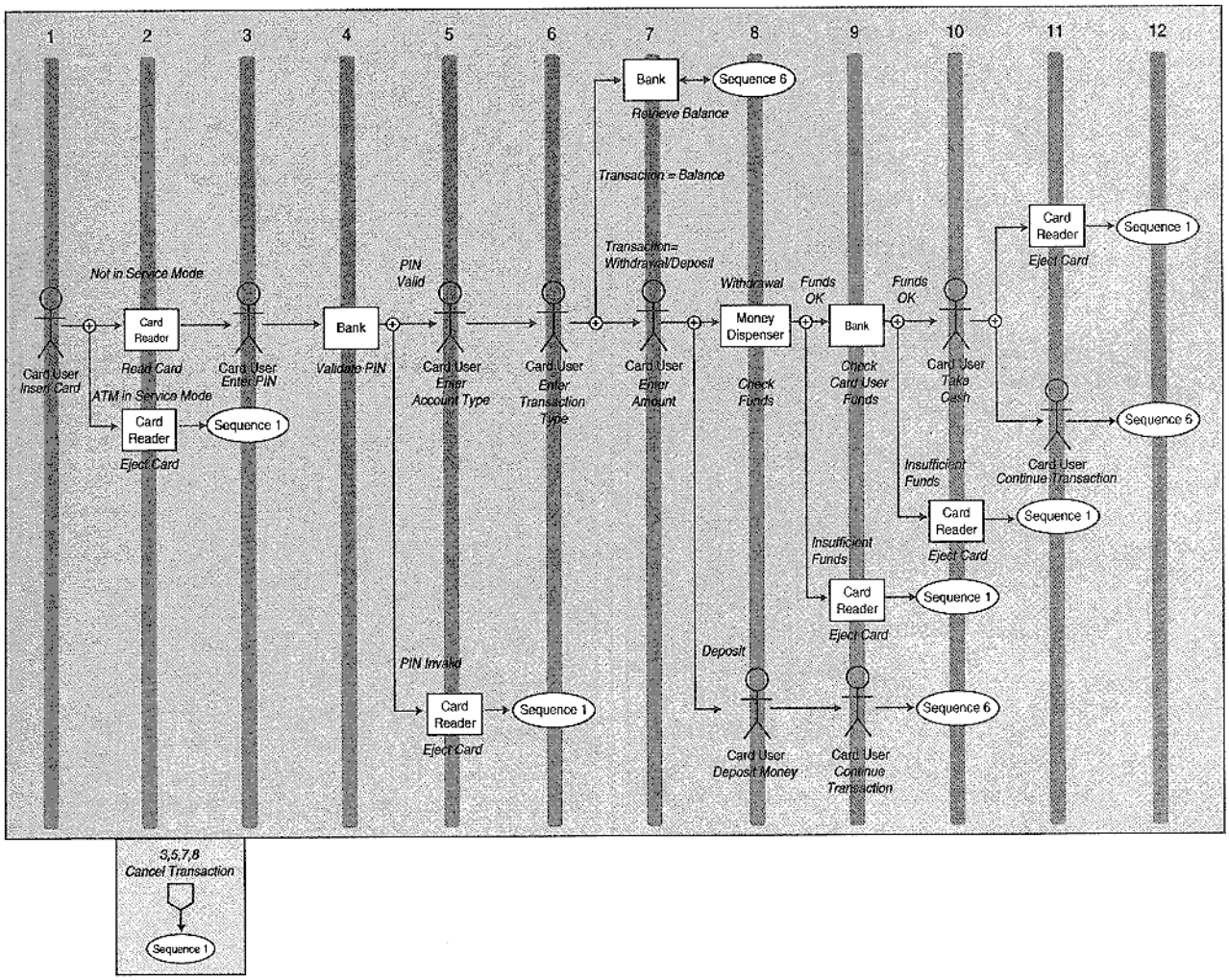
There are a host of successful structured and object-oriented system-development methodologies in use, each with its own strengths and weaknesses. Many weaknesses center around the discovery process in the early stages of development. In this article, we'll present a simple method for initiating and nurturing the discovery process. This method captures system behavior from a user's viewpoint, producing an operational specification that can be translated into most existing system-development methodologies.

This method complements rather than competes with existing methods and is designed to be a "front end" to methodologies such as Shlaer's object-oriented analysis, Rumbaugh's object modeling technique (OMT), and Hatley's structured analysis. Our method allows analysts and designers to more fully understand a system early in the development life cycle, revealing potential problems before analysis or design models are created. The output of this method is a set of diagrams called the "operational specification," intended to be a complete description of a system's desired behavioral operation from a user's point of view. The diagrams in an operational specification contain user events and system responses.

The method was designed to be useful in a "paper and pencil" environment. Our goal was to produce an inexpensive way to capture important operational information quickly and effectively without all the hoopla surrounding CASE. It is intended to be a back-to-basics approach, independent of CASE tools (although tools to support this method should be fairly easy to implement). For the sake of convenience (and in the same spirit of humility as Booch, Shlaer, and others), we'll refer to this as the "Coats-Mellon Operational Specification" (CMOS) methodology.

The Operational Specification

The operational specification is a set of diagrams that specify incoming stimuli and a system's response to these stimuli. The operational specification addresses analysis-phase, system-level behavior only; it does not specify data or functional requirements, design, or implementation information. At the heart of the operational specification is the incoming event. An event is an occurrence at a point in time. The operational specification consists of a set of two types of diagrams that divide behavior up into a set of events and system responses. The models that result from a translation of an operational specification (using methods like OMT and Hatley-Pirbhai) are pure analysis models. To illustrate system-development-method concepts and the production of an operational specification using the CMOS method, we'll use the familiar example of an automated teller machine (ATM) system.



Complete actor-event diagram for ATM system

The CMOS method can be summed up in six steps:

1. Create an actor diagram.
2. Create an actor-inheritance diagram.
3. Create an event-category diagram.
4. Create an actor-event diagram.
5. Create a system-response diagram.
6. Validate the behavior.

The actor and actor-inheritance diagrams help build the actor-event and system-response diagrams that comprise the operational specification. Validation can occur once the operational specification is complete.

The Actor Diagram

The actor diagram is an analysis anchor that crystallizes the operational environment in which the product system will exist. Figure 1 represents the system. Environment entities, or "actors," that interact with the system are represented outside the circle.

Descriptions of the human actors appear just below each stick figure. Nonhuman actors are represented by various boxes containing their descriptions. Each arrow is a summary of *all events* flowing in the direction indicated by the arrow; thus, no labels are used. The actor diagram shows actor-to-actor and system-to-actor interactions. It is important to show actor-to-actor interactions because they often dictate the order of system-to-actor interactions. Actors that can initiate event sequences (event sequences that begin with events that are not caused by another modeled event) are marked with an asterisk. Figure 1 is an actor diagram for an ATM system.

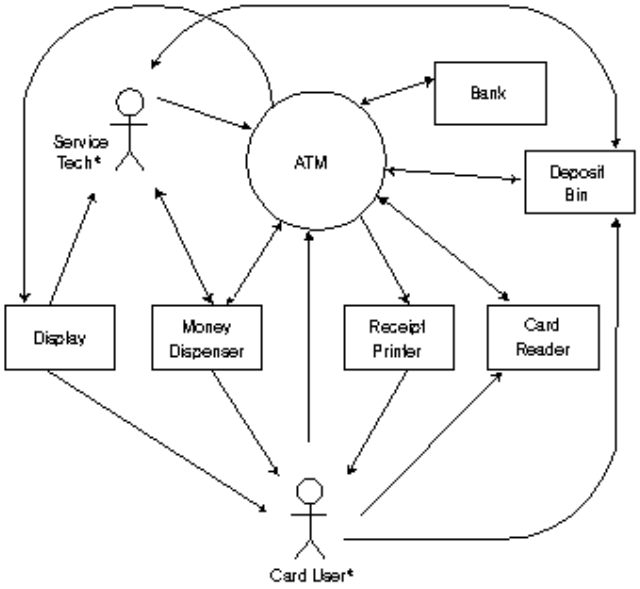


Figure 1: Actor Diagram

The actor diagram records the results of the environment (or "domain"). It is similar to other environment-level diagrams such as Jacobson's actor diagram and Hatley-Pirbhai's context diagram. Creating the actor diagram is a logical first step in any method. The CMOS actor diagram differs from the Jacobson and Hatley-Pirbhai methods in that it shows flows among external entities. This is because the actor diagram is a summary of the sequences of events that flow among actors and between the actors and the system.

The Actor-Inheritance Diagram

The actor-inheritance diagram shows the sharing of event categories among actors. In Figure 1, the Service Tech actor should be able to do anything the Card User actor does. In other words, she inherits all of the Card User's event categories. The Service Tech also has transactions unique from those of the Card User and will therefore include event categories like "replenish money supply" or "run diagnostic." Figure 2(a) is the actor-inheritance diagram for the ATM system.

A more-elaborate actor-inheritance diagram in Figure 2(b) is for a system that implements a metrics-tracking database. Readers can perform the basic functions to read metrics data in the database. Only metric and formula writers are allowed to input metrics or change metric formulas. Area administrators have these abilities, plus additional area administrative functions. Global administrators have complete capability.

Actor-inheritance diagrams may have multiple root actors. Once the actor diagram and the actor-inheritance diagrams are complete, the event-category diagram is constructed.

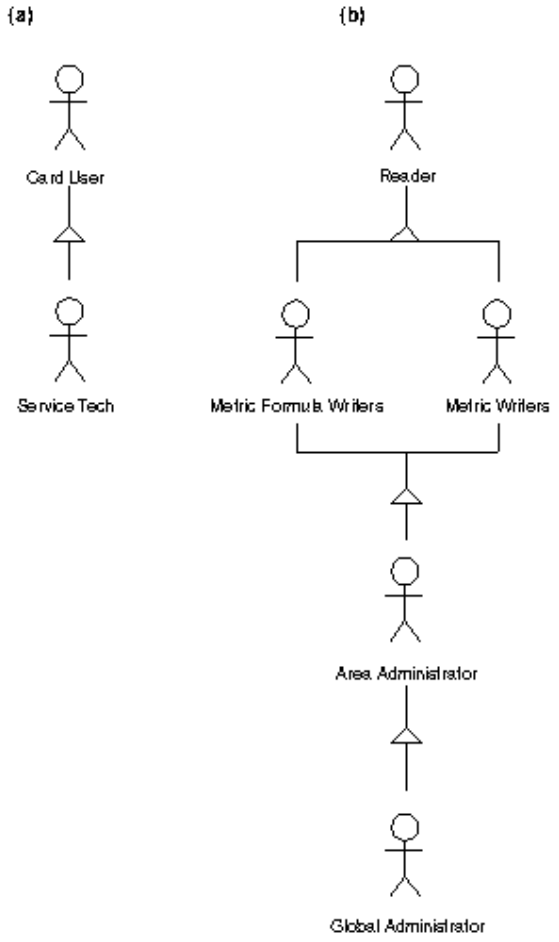


Figure 2: Actor-inheritance diagram

(a) ATM system; (b) metrics collection

The Event-Category Diagram

An event-category diagram is a grouping of related events for an associated actor. This diagram records all possible event categories and their associated events for each actor. Discovery takes place here. At this time, responses to these events are unimportant; instead, think only in terms of each actor's roles and respective responsibilities. Other methods consider actor events and system responses (or "use cases") simultaneously, but CMOS postpones system-response considerations until all actor stimuli have been identified.

Each event is categorized and associated with an actor resulting in one event-category diagram per actor. Figure 3 is the set of event-category diagrams for the ATM; it contains event categories, subcategories, and the categories' respective events. An actor inherits event categories from another actor according to the actor-inheritance diagram. Event-category diagrams look like OMT's object-class diagrams, and they are indeed similar. We chose not to invent new notation unnecessarily. Each box represents an event category that contains events. An event category inherits the categories pointed to by the triangle. There are association lines between the actor and the event

categories for that actor. (An actor can be associated with multiple categories.) Each actor's event categories should be addressed individually, thus the separate diagrams for each actor.

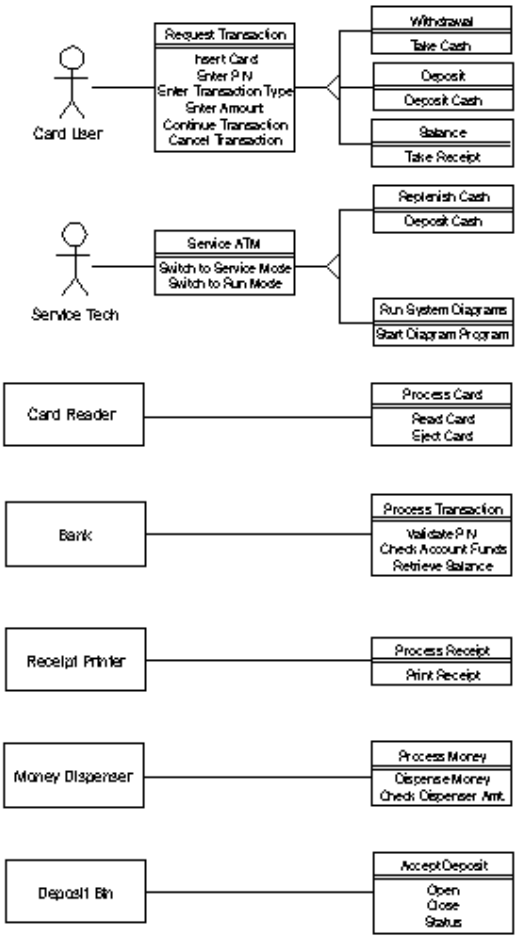


Figure 3: Event-category diagram

The Actor-Event Diagram

The actor-event diagram is one of two types that comprise the operational specification for the system. It will be referenced constantly throughout the development life cycle, so its representation should be easy to read and understand. In the event-category diagram, categories, subcategories, and events are discovered by playing the role of the actor. Once a substantial set of events is captured, the events need to be glued together in sequences. This is done via actor-event diagrams. There will be a separate actor-event diagram for every actor-originated event. Systems can have sequential or concurrent relationships between events.

For example, a Card Reader actor cannot possibly read a card until a Card User actor inserts the card into the reader; the relationship between these two events is sequential. Also, for the Bank to validate a PIN, the events Card User Inserts Card and Card Reader Reads Card must have occurred. Analyzing these events could become extremely complex; thus a diagram is needed. An actor-event diagram captures this information.

Figure 4 is a partial actor-event diagram for the ATM system that describes event flows among actors. (A complete ATM actor-event diagram is shown on page 19. Additional diagrams are provided electronically; see "Availability," page 3.) It shows when each actor-initiated event occurs in relationship to other actor-initiated events. It does not show system responses (this is the role of the system-response diagram). The events are placed on sequence lines. A sequence line is used to

identify which events could possibly concur in the sequence. Sequence lines advance time from left to right and show only sequences, not actual time. A sequence line can also be used to show a return point. This is shown in Figure 5 following the event Card Reader Ejects Card.

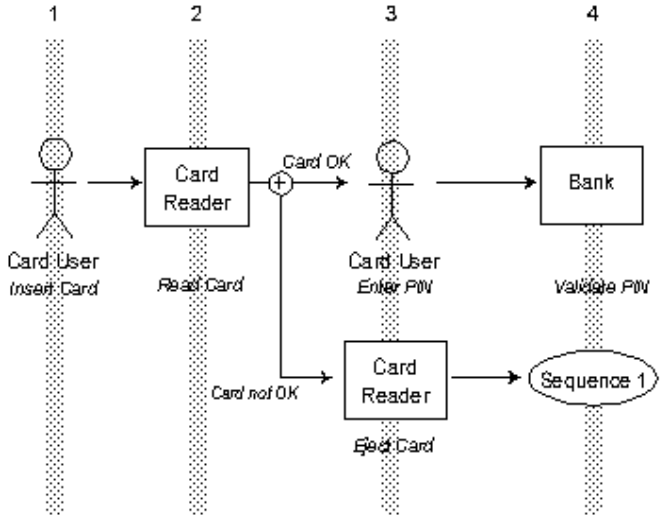


Figure 4: Actor-event diagram for a portion of the ATM system

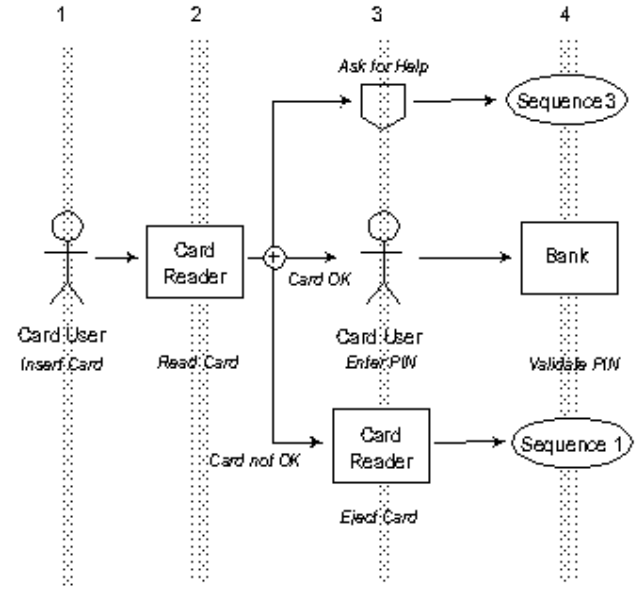


Figure 5: Actor-event diagram showing the link symbol

The next event is one of the set of events located on sequence line 1. The sequence return symbol contains the label of the sequence line to which it returns. A completed actor-event diagram will have sequence return symbols at every leaf; this minimizes the number of possible threads (paths) through the diagram. Once the event flow has reached a sequence return symbol, a thread is complete. It is not important to consider the repeating sequences of a thread since that behavior has already been defined.

Sequence relationships are represented by arrows. A sequence relationship may have a condition attached to it so that the sequence flow can advance to the next event only if the condition has been satisfied. This feature encourages the analyst to consider all possible conditions affecting the flow from one event to the next. A successor event is possible when there is either no condition or the condition is true. At least one successor event must be possible; a plus sign (+) indicates more than one.

A pentagonal link symbol indicates a link to an actor-event subdiagram; see the link symbol "Ask for Help" in Figure 5. At this sequence line, a Card User may ask for help as an alternate to entering his PIN. Figure 6 shows the actor-event subdiagram for Card User Asks for Help. (Subdiagrams are identified by the first event name.) The return symbol in the subdiagram means to return to the link symbol and proceed to the next event.

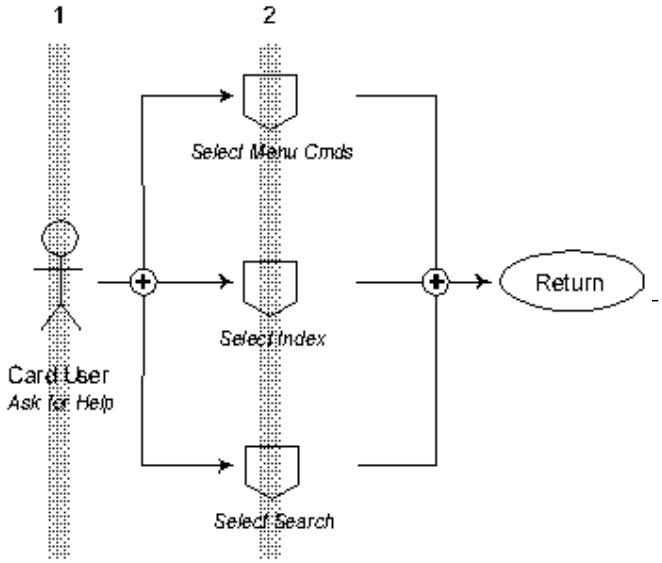


Figure 6: Actor-event subdiagram for Card User Asks for Help

To allow the Ask for Help choice on every sequence line, a box with a link symbol can be attached to the edge of that actor-event diagram. This avoids putting the link symbol on every sequence line. This return symbol indicates a return to the sequence line where the event was invoked. When an event can occur during some but not all sequences (as in Cancel Transaction), the same construct is used with an additional list of the sequence lines upon which the link could occur; see Figure 7. These link events attached to the edge of the actor-event diagram represent implicit, additional event choices for each sequence line specified. Link symbols to actor-event subdiagrams must be used when a selection can create a path longer than one event. The link symbol may also be used to keep diagrams to a manageable size (one page, for instance).

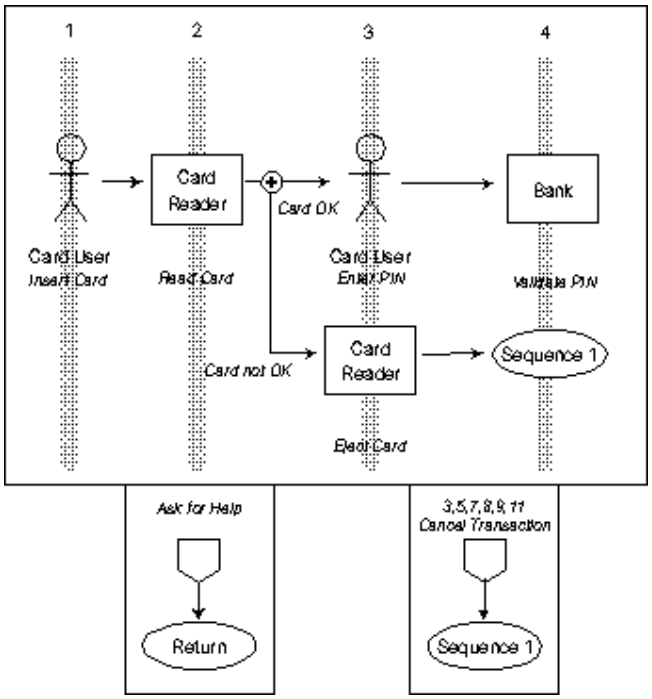


Figure 7: Actor-event diagram showing the return symbol

An ampersand indicates concurrent time relationships between events. The APU startup routine and the IMU startup routine in Figure 8 occur simultaneously; sometime after the payload status is displayed. When used, this construct mandates that the succeeding threads begin concurrently. The succeeding events can either meet up using a "+" or "&". The "+" means that the first preceding event to complete triggers the next event. The remaining (preceding), completing events do not trigger the next event. The "&" means that the completion of both preceding events trigger the next event. Note that the diagram is actor based; there are no objects, only actor events, on the actor-event diagram.

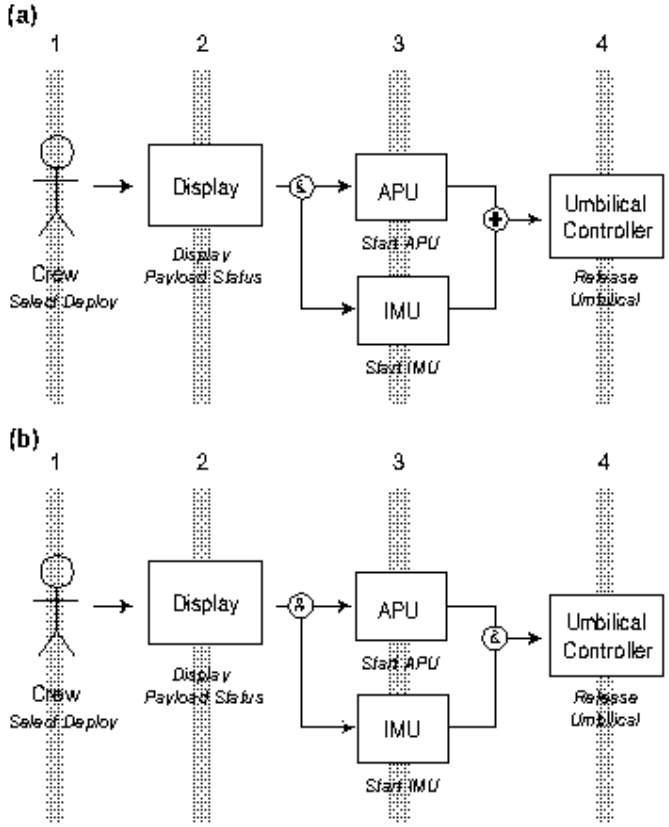


Figure 8: Concurrent example of an actor-event diagram using (a) a plus sign; and (b) an ampersand

The System-Response Diagram

The system-response diagram, along with the actor-event diagram, completes the operational specification by showing the system's responses to each actor event. An actor event usually has a system response, but not always. In Figure 4, a Card User inserts a card into the Card Reader. There is no system response to this event, since the Card Reader is not part of the system. This is shown on the actor diagram where Card User interacts with Card Reader. There is a system response to the event Card Reader Reads Card; see the response diagram in Figure 9(a).

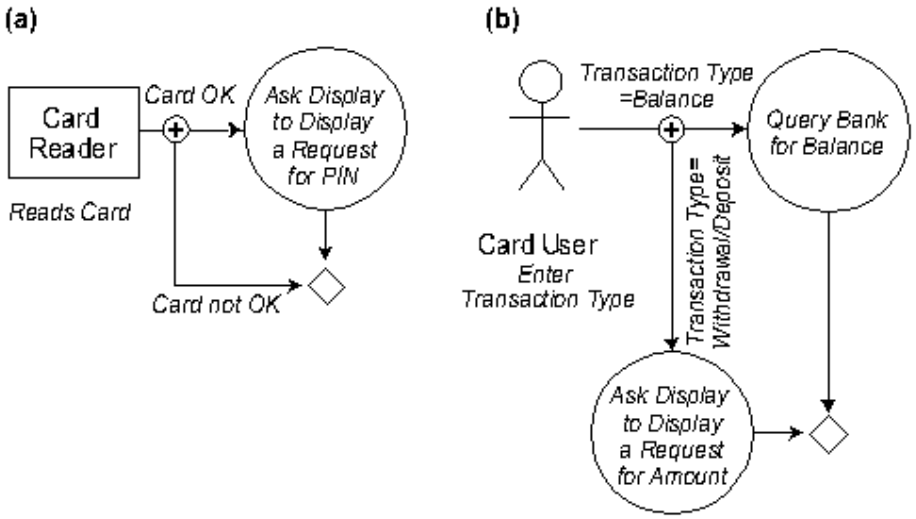


Figure 9: (a) Response diagram for Card Reader Reads Card; (b) response diagram for Card User Enters Transaction Type

Response diagrams begin with the actor event to which the system is responding. They can be trivial or complex, but to manage complexity, they should represent the system response for only one actor event. Response diagrams use circles to describe the system events. (This is consistent with the actor diagram's representation of the system.) The system-response text should be at the level of "responsibilities," as suggested by Wirfs-Brock. The arrows represent sequences with optional conditions, as in the actor-event diagram. The diamond marks the end of the system response, after which only another actor event can occur; this is specified in the actor-event diagram. Figure 9(b) represents the system response to the actor event Card User Enters Transaction Type. (A set of response diagrams for the ATM system is provided electronically.)

Behavior Validation

The set of system-response and actor-event diagrams represent the total operational specification. Once complete, these diagrams can be validated. Each event from an event category can be traced though the actor-event and system-response diagrams until a sequence-return symbol is reached. Each trace to a sequence return symbol is an actor scenario. Traditionally, scenario-based validation and testing produce an overwhelming number of scenarios to validate. The operational specification mitigates this problem by making it easy to see trace patterns, thus allowing better equivalence partitioning. Because of the CMOS's simplicity, customers can sit in on the validation process (tracing events in the specification), so the customer and analysts reach agreement before more-costly development work begins. The operational specification is an ideal peer-review instrument.

Translating to OMT

Entities from CMOS diagrams can be translated to OMT object and dynamic models. The first candidate set of classes for the object model should consist of an interface-object class for each actor (that is, a type of user). Each event sentence in the actor-event diagram and the system-response diagram can suggest classes, attributes, operations, and associations. Noun phrases are potential objects or attributes, verb phases are potential operations, and associations are identified by the sentence structure. For example, the event Ask Display to Display a Request for Amount would produce the portion of the object model shown in Figure 10(a); the portion for Bank Validates PIN is shown in Figure 10(b).

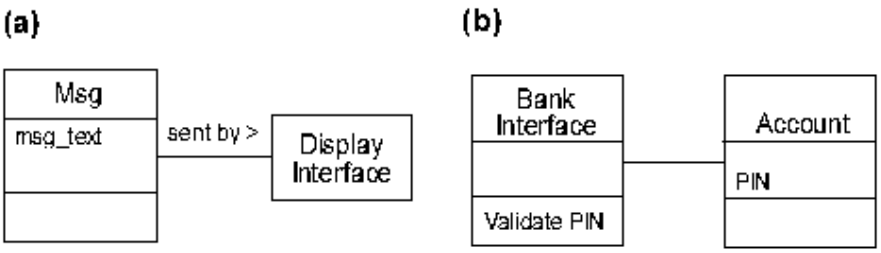


Figure 10: (a) Object model for Ask Display to Display a Request for Amount;

(b) object model for Bank Validates PIN

The dynamic model (a collection of class-level state-transition diagrams) can also be created from the actor-event and system-response diagrams. Events on each diagram are used to directly construct a state-transition model. Figure 11 is a state-transition diagram for the Bank Interface class. Each Bank event becomes an event on the Bank state diagram. States are inserted to receive each event.

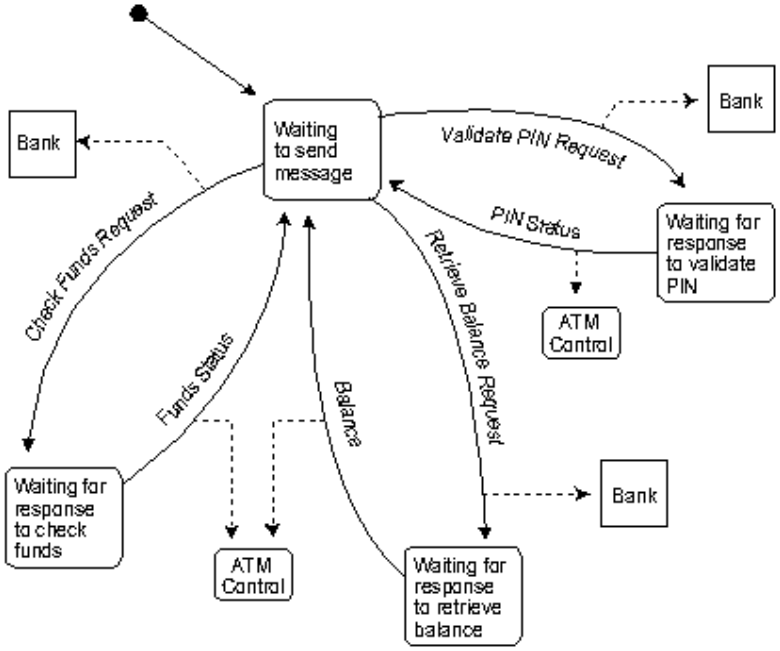


Figure 11: State transition diagram for Bank Interface Class

Translating to Hatley-Pirbhai Structured Analysis

A Hatley-Pirbhai requirements model includes data-context, data-flow (including process specifications, or PSpecs), control-context, and control-flow diagrams (including control specifications, CSpecs). Context diagrams for a system can be derived directly from a CMOS actor diagram. The context diagrams consist of the system and all actors who directly exchange events with the system, along with those exchanged events represented either as data flows or control flows. Guidelines for distinguishing between data and control flows are provided by Hatley-Pirbhai and would be applied here. (The primary guideline is that a flow whose only purpose is to activate or deactivate processes should be modeled as a control flow.) Figures 12 is a context diagram for the ATM.

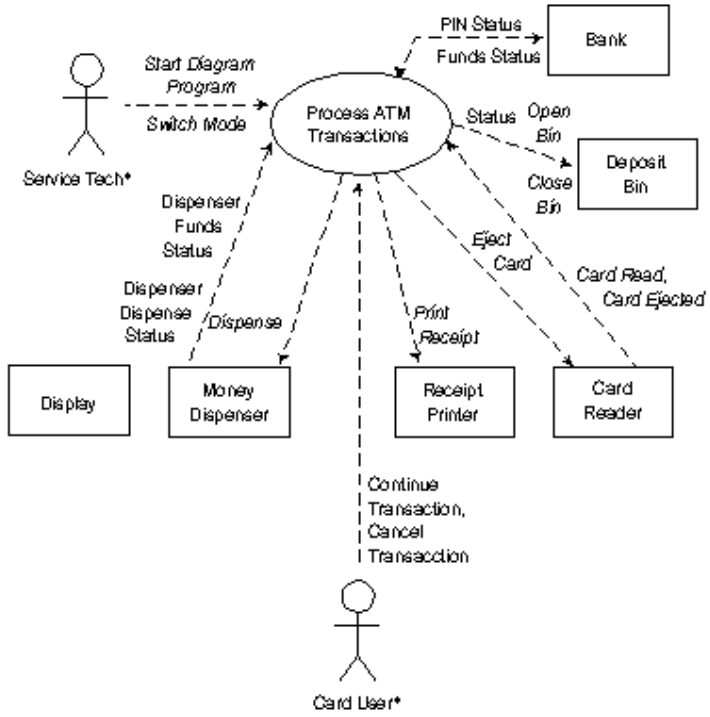


Figure 12: Control-context diagram for the ATM

For identifying the top-level data processes, Hatley-Pirbhai recommends event partitioning, whereby each data flow coming into the system should have its own top-level data process to handle the system's total response to that flow. Hatley-Pirbhai extends this by stating that every control flow coming into the system should flow into a single top-level control process (called "CSpec 0"), which activates or deactivates the top-level data processes. (Recall that data processes also can be activated solely by the arrival of data flows, if appropriate.) In many cases, control flows also flow to CSpecs at lower levels to provide them with finer-grained control.

Event partitioning can be achieved directly from CMOS system-response diagrams. Because each system-response diagram describes the effect of one event flowing into the system, the circles on a system-response diagram identify the system's total response to that flow--collectively, they are a top-level data process. Most real systems respond to so many events that not all the system-response circles can be shown at the top level. In such cases, the circles should be grouped into new, higher-level circles, as was done for the ATM.

An event can be a stimulus with or without associated data. In Hatley-Pirbhai terms, the former is modeled as a control flow and the latter, as a control flow, plus a data flow, or as a data flow alone; see Figures 13 and 14.

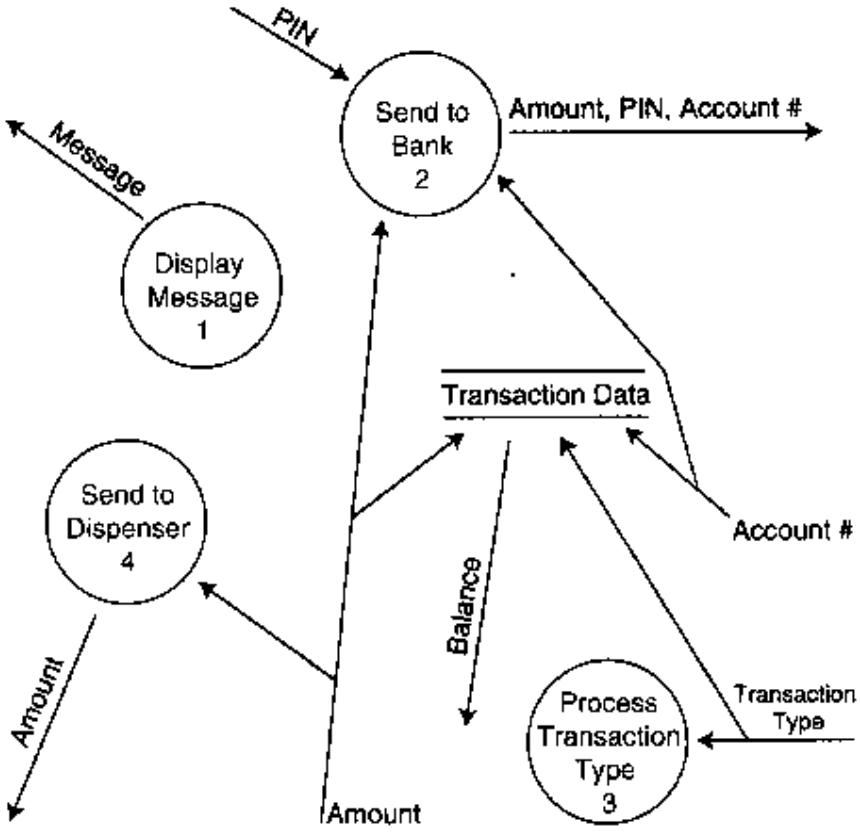


Figure 13: Data-flow diagram 0 for the ATM

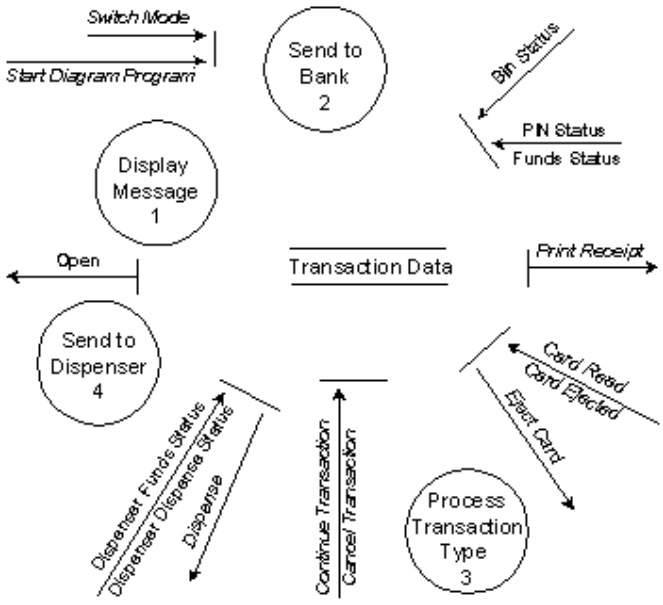


Figure 14: Control-flow diagram 0 for the ATM

CSpec 0 (represented by the bar symbol in Figure 14) would be a state-transition diagram that activated or deactivated the processes on that diagram.

Conclusions

Creating a rich operational specification for a system early in the development cycle reduces the overall cost of system development and maintenance. Systems developed in this manner experience less change, especially later in the development cycle when the cost of change is high. Several successful projects have been developed using the CMOS method. In each case, the goals of the method were realized because a great deal of rich, stable information was captured early in the development process. The specifications allowed a better understanding of the system among customers, analysts, testers, designers, coders, and management. This understanding was realized early in projects before the most costly development work began; and because the completed operational specification is user based, it is easily comprehended by new development personnel and users of the system.

In one experiment, we gave 20 designers the same operational specification. The resulting designs for the specification had only slight differences. This demonstrated that CMOS specifications can communicate a large amount of user-based system knowledge.

Operational specifications can assist in maintenance as well as initial development. System maintenance begins with changes to the operational specification and continues through analysis, design, and code. New capabilities are added in the form of new events or event categories. These, in turn, ripple through the analysis and design models. New capabilities are well understood early in the maintenance life cycle. Operational specifications can be used as a test specification. They directly support "black-box" testing because they are based upon external stimuli.

The information in the operational specification can provide valuable metric data for the purpose of development-cost estimates. These metrics could include the number of actors, event categories, events, threads in the actor-event diagram, system-response events, and so on. By combining this data in different ways and with different formulas, a viable cost-estimation model can also be produced.

CMOS is repeatable and can be incorporated into any development process. Management must be aware, however, that this method may at first appear to decrease productivity. Managers might mistake this for typical unstructured brainstorming. But if management allows the process to unfold, a good deal of time will be reclaimed during the remaining analysis, design, implementation, and especially maintenance phases.

References

Booch, G. "The Booch Method: Notation, Part I and Part II." *Computer Language* (September/October 1992).

Hatley, D.J. and I.A. Pirbhai. *Strategies for Real-Time System Specifications*. New York, NY: Dorset House, 1988.

Hsia, P. et al. "Formal Approach to Scenario Analysis." *IEEE Software* (March 1994).

Jacobson, I. et al. *Object-Oriented Software Engineering: A Use Case Driven Approach*. Reading, MA: Addison-Wesley, 1992.

Krell, B.E. *Developing with Ada: Life-Cycle Methods*. New York, NY: Bantam Books, 1992.

Linger, R.C. "Cleanroom Process Model." *IEEE Software* (March 1994).

McMenamin, S.M. and J.F. Palmer. *Essential Systems Analysis*. Englewood Cliffs, NJ: Prentice-Hall, 1984.

Musa, J. "Operational Profiles in Software-Reliability Engineering." *IEEE Software* (March 1993).

Rumbaugh, J. "Getting Started: Using Use Cases to Capture Requirements." *JOOP* (September 1994).

Rumbaugh, J. et al. *Object-Oriented Modeling and Design*. Englewood Cliffs, NJ: Prentice-Hall, 1991.

Shlaer, S. and S. Mellor. "The Shlaer-Mellor Method." Project Technology, Inc., Technical Report pr.pb.S075, Version 2.0, 1993.

Wirfs-Brock, R. "Designing Scenarios: Making the Case for a Use Case Framework." *The Smalltalk Report* (March 1993).
