Paul C. Jorgensen and Carl Erickson

Object-Oriented Integration Testing

Object-oriented software development raises important testing issues. Many of these stem from attempts to directly apply theoretical constructs and techniques of traditional software develop ment and testing to object-oriented software. We examine this traditional heritage here, with special emphasis on assumptions and practices that need to be modified or replaced.

We identify five levels of object-oriented testing; four of these map nicely into the commonly accepted unit, integration, and system levels of traditional software testing. (Placement of the remaining level is primarily a management consideration.) We also identify two new testing constructs and a directed graph notation that helps formalize object-oriented integration testing. These are illustrated with an object-oriented formulation of an automated teller machine (ATM) system. The source code (Objective-C) for this system is available from the authors.

We begin with an important distinction: structure vs. behavior. Most of the popular notations used in software development (E/R models, data flow diagrams, structure charts, PDLs, and so on) portray software structure: the components, relationships among these, the interfaces, control and dataflow, and so on. Such information is certainly needed by software developers, but it is only moderately useful to testers. For simple programs, we can understand behavior in terms of structure, but there is a relatively low threshold of program complexity beyond which this derivation becomes untenable. Software testing is fundamentally concerned with behavior (what it does), and not structure (what it is). Customers understand software in terms of its behavior, not its structure. The object-oriented testing constructs we introduce here are deliberately behavioral rather than structural.

To provide a context for object-oriented integration testing, we highlight the traditional software (and system) testing notions that have special implications for object-oriented software testing. Traditional software is

- written in an imperative language
- described by a functional decomposition
- developed in a waterfall life cycle
- separated into three levels of testing

Since these often do not apply directly to object-oriented software, they represent latent assumptions which must be revisited.

Most software developers use an imperative language, in which the order of source statements determines the execution order of compiled object instructions. The familiar languages (Fortran, Cobol, C, Pascal, Ada, and assembly languages) are all imperative, as opposed to declarative languages (e.g., Prolog), in which the source statement order has little to do with execution order. Imperative languages are so widely used (and for so long), they have become "natural" to most programmers. All of structured programming, with the basic control structures of sequence, selection, and repetition, and the single-entry, single-exit precept, is directed at imperative languages.

Imperative languages lend themselves to a rigorous description as a directed graph, or program graph [8], in which nodes are statements (or statement fragments) and edges represent control flow sequence. From this starting point, several graph theory-based testing constructs have been defined: DD-Paths, define/reference nodes, definition clear paths, and program slices, to name a few. These all help the tester give a more accurate description of what is being tested, and all lead to useful test coverage metrics.

In contrast, declarative languages suppress sequentiality, thereby sacrificing the descriptive benefits of directed graphs. (On the other hand, declarative languages are naturally represented by more formal notations, such as the predicate or the lambda calculus, which in turn open possibilities of formal proofs of correctness.)

The event-driven nature of object-oriented systems forces a "declarative spirit" on testing. This is not evident at the unit level (most object-oriented programming languages are imperative), but it is pronounced at the integration and system levels.

Functional decomposition is the natural extension of the systems analysis introduced as a problem-solving technique by the U.S. Army in the 1930s [1]. Known equally often by its synonym, top-down development, functional decomposition can be either prescriptive or descriptive. The prescriptive view (which is enforced in functional languages such as Lisp) demands that software development begins "at the top," and proceeds by subdividing functionality into successively lower levels of detail, resulting in a hierarchy, or functional decomposition tree.

The descriptive view is more tolerant of the way people work, often flitting across levels of abstraction in seemingly random orders [5], and reinforcing analysis with synthesis, a symbiosis found in most other engineering disciplines. The end result is the same: a tree-like decomposition of system functionality into components that exhibit several senses of hierarchy: levels of abstraction, lexical inclusion, information hiding, and corresponding data structures which may have parallel decompositions into various user-defined types.

Functional decomposition has been the mainstay of software development since the 1950s, partly because it fits so well with other hierarchies: organizational structures, program language packaging, hardware packaging (system, frame, rack, card,. . .) and the fan-out of activities in the waterfall model of software development. Despite these reinforcing morphisms, functional decomposition has its vulnerable points. Decomposing a problem so that existing components (to be reused) appear in the tree is tricky, and the decomposition criteria used have an enormous impact on the resulting system. The rival strategy, composition, has been all but lost in the structural revolution.

Functional decomposition has deep implications for testing: first, it emphasizes levels of abstraction (hence, levels of testing), and second, it creates questions of integration order (top-down or bottom-up). Most important (and insidious) is that it stresses structure over behavior.

The well-known waterfall model of software development is sometimes depicted as a "V" in which the development phases (requirements specification, preliminary design, and detailed design) are at levels corresponding to system, integration, and unit testing. The sequential nature of the waterfall model predisposes a bottom-up testing approach in which unit testing produces separately tested components which are eventually integrated to support system testing. The integration portion of this is driven by the functional decomposition tree, where there is another top-down or bottom-up question. Whichever alternative is chosen, it is important to note that the goal is to fit the units together into the functional decomposition tree. Thus the structure of the system is the goal, not the behavior.

Since the mid-1980s, the waterfall model has been critized for several fundamental defects [1]. Most of these pertain to the development side of the model, and to project management considerations, rather than to testing. We believe that the preference of structure over behavior as the goal of integration testing will be recognized as yet another shortcoming of the waterfall model.

The three widely accepted levels of testing-unit, integration, and system need some clarification. There are several definitions of a unit:

control and dataflow, and so on. Such information is certainly needed by software developers, but it is only moderately useful to testers. For simple programs, we can understand behavior in terms of structure, but there is a relatively low threshold of program complexity beyond which this derivation becomes untenable. Software testing is fundamentally concerned with behavior (what it does), and not structure (what it is). Customers understand software in terms of its behavior, not its structure. The object-oriented testing constructs we introduce here are deliberately behavioral rather than structural.

To provide a context for object-oriented integration testing, we highlight the traditional software (and system) testing notions that have special implications for object-oriented software testing. Traditional software is

- written in an imperative language
- described by a functional decomposition
- developed in a waterfall life cycle
- separated into three levels of testing

Since these often do not apply directly to object-oriented software, they represent latent assumptions which must be revisited.

Most software developers use an imperative language, in which the order of source statements determines the execution order of compiled object instructions. The familiar languages (Fortran, Cobol, C, Pascal, Ada, and assembly languages) are all imperative, as opposed to declarative languages (e.g., Prolog), in which the source statement order has little to do with execution order. Imperative languages are so widely used (and for so long), they have become "natural" to most programmers. All of structured programming, with the basic control structures of sequence, selection, and repetition, and the single-entry, single-exit precept, is directed at imperative languages.

Imperative languages lend themselves to a rigorous description as a directed graph, or program graph [8], in which nodes are statements (or statement fragments) and edges represent control flow sequence. From this starting point, several graph theory-based testing constructs have been defined: DD-Paths, define/reference nodes, definition clear paths, and program slices, to name a few. These all help the tester give a more accurate description of what is being tested, and all lead to useful test coverage metrics.

In contrast, declarative languages suppress sequentiality, thereby sacrificing the descriptive benefits of directed graphs. (On the other hand, declarative languages are naturally represented by more formal notations, such as the predicate or the lambda calculus, which in turn open possibilities of formal proofs of correctness.)

The event-driven nature of object-oriented systems forces a "declarative spirit" on testing. This is not evident at the unit level (most object-oriented programming languages are imperative), but it is pronounced at the integration and system levels.

Functional decomposition is the natural extension of the systems analysis introduced as a problem-solving technique by the U.S. Army in the 1930s [1]. Known equally often by its synonym, top-down development, functional decomposition can be either prescriptive or descriptive. The prescriptive view (which is enforced in functional languages such as Lisp) demands that software development begins "at the top," and proceeds by subdividing functionality into successively lower levels of detail, resulting in a hierarchy, or functional decomposition tree.

The descriptive view is more tolerant of the way people work, often flitting across levels of abstraction in seemingly random orders [5], and reinforcing analysis with synthesis, a symbiosis found in most other engineering disciplines. The end result is the same: a tree-like decomposition of system functionality into components that exhibit several senses of hierarchy: levels of abstraction, lexical inclusion, information hiding, and corresponding data structures which may have parallel decompositions into various user-defined types.

Functional decomposition has been the mainstay of software development since the 1950s, partly because it fits so well with other hierarchies: organizational structures, program language packaging, hardware packaging (system, frame, rack, card,. .) and the fan-out of activities in the waterfall model of software development. Despite these reinforcing morphisms, functional decomposition has its vulnerable points. Decomposing a problem so that existing components (to be reused) appear in the tree is tricky, and the decomposition criteria used have an enormous impact on the resulting system. The rival strategy, composition, has been all but lost in the structural revolution.

Functional decomposition has deep implications for testing: first, it emphasizes levels of abstraction (hence, levels of testing), and second, it creates questions of integration order (top-down or bottom-up). Most important (and insidious) is that it stresses structure over behavior.

The well-known waterfall model of software development is sometimes depicted as a "V" in which the development phases (requirements specification, preliminary design, and detailed design) are at levels corresponding to system, integration, and unit testing. The sequential nature of the waterfall model predisposes a bottom-up testing approach in which unit testing produces separately tested components which are eventually integrated to support system testing. The integration portion of this is driven by the functional decomposition tree, where there is another top-down or bottom-up question. Whichever alternative is chosen, it is important to note that the goal is to lit the units together into the functional decomposition tree. Thus the structure of the system is the goal, not the behavior.

Since the mid-1980s, the waterfall model has been critized for several fundamental defects [1]. Most of these pertain to the development side of the model, and to project management considerations, rather than to testing. We believe that the preference of structure over behavior as the goal of integration testing will be recognized as yet another shortcoming of the waterfall model.

The three widely accepted levels of testing-unit, integration, and system need some clarification. There are several definitions of a unit:

- a single, cohesive function
- a function which, when coded, fits on one page
- the smallest separately compilable segment of code
- the amount of code that can be written in 4 to 40 hours
- a task in a work breakdown structure
- code that is assigned to one person
- code that one person designs, codes, and tests in a three-month period

Curiously, many organizations that specifically conduct unit testing have not chosen their definition of a software unit. However defined, a unit is tested "by itself," with adjacent software units being replaced by stubs and drivers to emulate inputs and outputs. The goal of unit testing is to verify that, taken by itself, the unit functions correctly. (Another view is to see how the unit functions, assuming everything else is perfect.)

Once units are separately tested, they are integrated together. Integration testing is the least well understood of the three levels. Part of this can be seen in the symmetries with the waterfall phases: unit testing with detailed design, integration testing with preliminary design, and system testing with requirements specification. These symmetries are comfortable in that the basis for test case identification is clear. Of these, the unit level is best understood (both in terms of detailed design and unit testing), followed by the system level. The "leftovers" are given to preliminary design and integration testing.

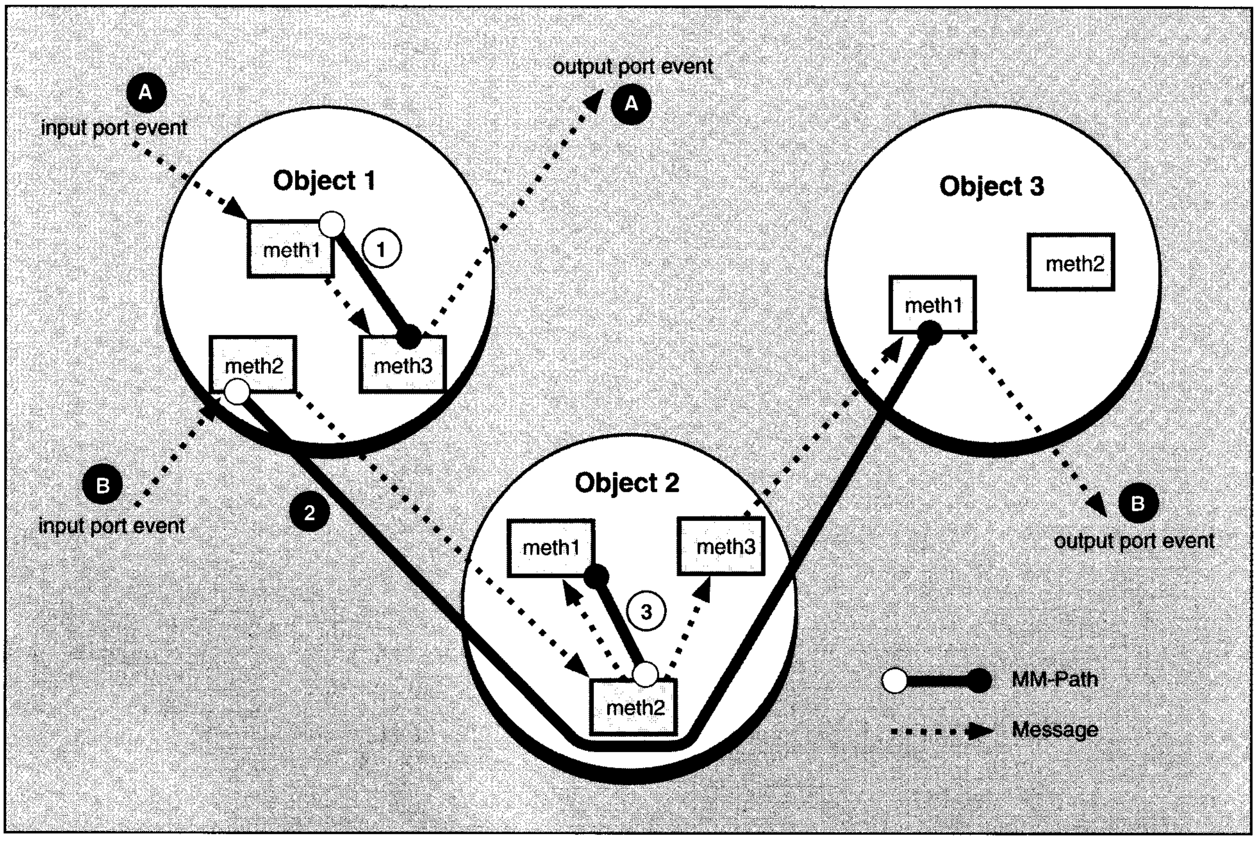
Since the mid-1970s, various module interconnect languages have been proposed [3] as descriptions of the information to be produced by preliminary design. In general, this includes the levels of functional decomposition and the major interfaces among components at these levels. This forces the goal of integration testing to address these primarily structural considerations. Here are some frequently used views:

- the gradual replacement of stubs and drivers by separately tested units
- pairwise integration, in which each unit is integrated with its adjacent units
- bottom-up integration guided by the functional decomposition tree
- top-down integration guided by the functional decomposition tree
- "big bang" integration where all units are thrown together at once

The common thread (and deficiency) among these possibilities is that they stress structure and interfaces, rather than behavior. They all presume that correct behavior is guaranteed by correct structure and interfaces.

Where does integration testing end and system testing begin? Distinctions based on waterfall phases beg the question, because it is equally difficult to decide where requirements specification ends and preliminary design begins. We offer an explicit distinction: system testing is

Figure 1. **Directed-graph representation of the object network. Three Method-Message Paths (MM-Paths 1.2, and 3), and two Atomic System Functions (ASFs 4 and 13) are shown.**



The **declarative** aspect **of** object-oriented **software lies** primarily **in its** event-driven nature. *Dynamic binding also creates an indefiniteness that resembles that of declarative programs.*

conducted exclusively in terms of inputs and outputs that are visible at the port boundary of a system. A system tester can only have access to those port events that are available to the customer/user. In contrast, integration testing can access memory events and conditions that are invisible at the system level. Another place to see this demarcation is when a system is developed on one platform to he delivered on a different target platform. System testing can only occur on the target platform, while integration testing could occur on the development machine.

To the extent that object-oriented software is declarative, much of the descriptive power of graph theory-based structural testing techniques will not he applicable. Within an object, individual methods remain imperative. All object-oriented languages return control to the calling object when a message is "finished." (We consider a message to he the combination of a receiver object, a method name, and, optionally, method arguments.) The declarative aspect of object-oriented software lies primarily in its event-driven nature. Dynamic binding also creates an indefiniteness that resembles that of declarative programs.

Because the concept of a main program is minimized, there is no clearly defined integration structure. Thus there is no decomposition tree to impose the question of integration testing order of objects. We see this as an advantage for object-oriented integration testing; it is no longer natural to focus on structural testing orders.

The shift to composition (especially when reuse occurs) adds another dimension of difficulty to object-oriented software testing: it is impossible to ever know the full set of "adjacent" objects with which a given object may he composed. Taken by themselves, two objects may be correct; yet when they are composed, errors might result. We are reminded of M.C. Rscher's paradoxical drawings which center on deliberate errors of composition. The usual response from the object-oriented community is that if the units (objects) are carefully defined and tested, any composition will work. This was the hope of information hiding as a decomposition criterion in traditional software development. We know from experience that this fails. We know also that unit testing can never reveal integration-level problems.

Object-oriented software development, especially in terms of composition and reuse, usually occurs in a non-waterfall development life cycle; most commonly one based on rapid prototyping, perhaps in conjunction with an incremental approach. The rival models (of waterfall) all have composition as their fundamental underlying strategy, and all make no presumptions about the completeness goal that was so central to waterfall-based practice. We expect to see movement in the direction of operational specification, likely beginning with some form of an executable specification. When requirements specifications are expressed in this way, they create a new problem: the need to make a dynamic-to-static transition. An essentially dynamic, executable specification must somehow lead to static implementation components. This is difficult with traditional languages; the transition is eased by the inherent dynamism of the object-oriented paradigm.

The final implication of traditional software development is that the levels of testing need clarification for object-oriented software. Two levels are clear: object methods are units, and object-oriented unit testing is simply the testing of these methods. Traditional functional and structural testing techniques are fully applicable to this level. At the system level, thread-based testing is completely compatible with object-oriented software. The notion of a thread [4] is a natural construct for system-level resting. Here are several views of a thread:

- a sequence of machine instructions
- a sequence of source instructions
- a scenario of normal usage
- a system-level test case
- a stimulus/response pair (per [2])
- the behavior that results from a sequence of system-level inputs
- an interleaved sequence of system inputs (stimuli) and outputs (responses)
- a sequence of transitions in a state machine description of the system

Threads exist independently of their potential representations. We can interpret a thread to he a sequence of method executions linked by messages in the object network. This will follow from the constructs.

Constructs for Object-Oriented Integration Testing

Taken together, the implications of traditional testing for object-oriented integration testing require an appropriate construct for the integration level. This construct should he compatible with composition, avoid the inappropriate structure-based goals of traditional integration testing, support the declarative aspect of object integration, and he clearly distinct from the unit- and system-level constructs.

We postulate five distinct levels of object-oriented testing:

- a method
- message quiescence
- event quiescence
- thread testing
- thread interaction testing

Taken together; **the implications of traditional testing for object-oriented integration testing** *require an appropriate construct for the integration level.*

An individual method is programmed in an imperative language and performs a single, cohesive function. As such, it corresponds to the unit level of traditional software testing, and both the traditional functional and structural techniques are applicable. As noted earlier, both thread and thread interaction testing are at the system level. To address the two remaining levels, we note that for both cases, method executions are linked by messages, and quiessence provides natural endpoints. This is shown by the object network in Figure 1, in which nodes (rectangles) are methods and edges (dashed lines) are messages. Objects (circles) are not directly represented in the graph; they show related collections of methods.

Definition: A Method/Message Path (MM-Path) is a sequence of method executions linked by messages.

An MM-Path starts with a method and ends when it reaches a method which does not issue any messages of its own. In terms of an executing process, we call this point message *quiescence*. Since MM-Paths are composed of linked method-message pairs in an object network, they interleave and branch off from other MM-Paths. We chose this name to be similar to the DD-Path (decision-to-decision path) construct of traditional structured unit testing; MM-Paths provide analogous descriptive capabilities to object-oriented integration testing. Figure 1 shows three MM-Paths (labeled 1, 2, and 3).

The second construct reflects the event-driven nature of object-oriented software. Execution of object-oriented software begins with an event, which we refer to as a port input event. This system-level input triggers the method-message sequence of an MM-Path. This initial MM-Path may trigger other MM-Paths. Finally, the sequence of MM-Paths should end with some system-level response (a port output event). When such a sequence ends, the system is quiescent, that is, the system is waiting for another port input event that initiates further processing. This fits well with the notion of a reactive system [6] that responds to events in its environment, and with the notion of a stimulus/response pair that is central to the SREM requirements specification technique [2]. Stimulus/response pairs are threads that begin with a stimulus (a port input event), traverse one or more MM-Paths, and culminate with one of several possible port output events. In the case of event-driven, GUI applications, poorly written software may not provide feedback for a user-induced input event, in which case the ending port event is null.

Definition. An Atomic System Function (ASF) is an input port event, followed by a set of MM-Paths, and terminated by an output port event.

An atomic system function is an elemental function visible at the system level. As such, ASFs constitute the point at which integration and system testing meet, which results in a more seamless flow between these two forms of testing. The output port event which defines the end of an ASF may have different values (including null) for multiple executions of the same ASF. Figure 1 shows two ASFs (labeled A and B at the start and stop points). ASF A is composed of a single MM-Path (1). ASF B is composed of MM-Paths 2 and 3.

Example

As a concrete example of the object-oriented testing constructs we have proposed, consider an automated teller machine (ATM) system. All ATM systems must deal with the entry of a customer's personal identification number (PIN), which is known only by the central bank and the customer. The customer's ATM card is encoded with a personal account number (PAN) and is read by the card reader device in the ATM to obtain an expected PIN from the bank. A customer has three chances to enter the correct PIN. Once a correct PIN has been entered, the user sees a screen requesting the transaction type. Otherwise a screen advises the customer that the ATM card will not be returned, and no access to ATM functions is provided.

The following steps occur after the user enters a card:

1. A screen requesting PIN entry is displayed
2. An interleaved sequence of digit key touches with audible and visual feedback
3. The possibility of cancellation by the customer before the full PIN is entered
4. Interdigit time-outs, followed by screens asking if the user needs more time
5. Entry of a yes/no response to the time-out screen
6. A system disposition (valid PIN entered or card retained)

A finite-state machine (FSM) description of PIN entry (to appear in [7]) contains an upper-level FSM with 8 states, 10 transitions, and 4 paths. Three of these states are decomposed to a lower-level FSM that contains 9 states, 18 transitions, and 14 paths, resulting in a cyclomatic complexity of 13.

Classes for ATM PIN Entry. We have implemented an ATM simulator on NEXTSTEP using Objective C. We use this system as a means to ground our work in real code, and as an illustration of our object-oriented testing constructs. The class hierarchy of the ATM simulator is shown in Figure 2, which shows only the classes for the problem domain; we used the standard NeXT AppKit classes for the graphical interface objects.

Identifying MM-Paths. Consider

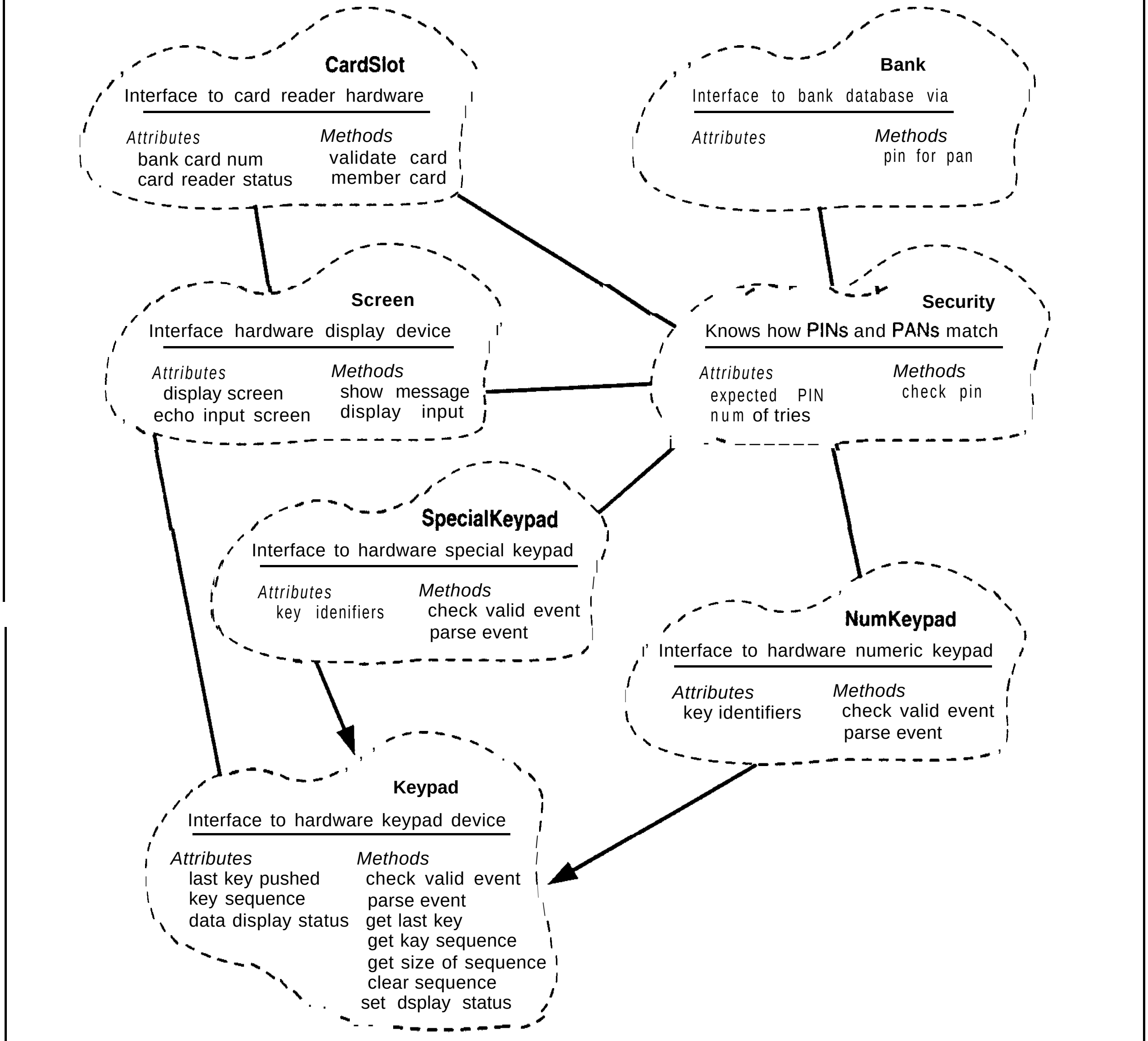


Figure 2. The class hierarchy for the ATM simulator

the following four sequences of behavior visible at the system level of the ATM:

1. Entry of a digit
2. Entry of a PIN
3. A simple transaction: PIN entry, select transaction type, present account details. conduct the operation, report results
4. An ATM session, containing two or more simple transactions

Digit entry (behavior sequence 1) is an example of a minimal MM-Path (see Figure 3). It begins with a port input event (key touch) which acts as a message to the NumKeypad:getKeyEvents method. It completes (reaches message quiescence) with a message to the parseKeyEvent method to decode the key.

PIN Entry (behavior sequence 2) is an example of an atomic system function. It is composed of six MM-Paths, an input port event, and several possible output port events. Figure 4 shows a mainline portion, in which the correct PIN is entered on the first attempt; several error cases are not shown. The objects in these MM-Paths know about the length of a PIN, the number of bad entry attempts, the PAN/PIN for an account, which hank cards are members of the ATMs network, and so forth. For clarity. Figure 4 has been simplified by removing the Timer object, and hence the PIN entry time-out. The MM-Path components of this ASF are listed in terms of Object:method sets.

It is instructive to consider where the longest MM-Path in this ASF should end: Screen:showMessage or NumKeypad:parseKeyEvent? The definition of the ASF makes this choice unambiguous, since ASFs are required to end in output port events. If the NumKeypad:parseKeyEvent were chosen as the end of the PIN Entry ASF, the system would not be in a quiescent state at the end of the ASF. Figure 5 summarizes the PIN Entry ASF.

The simple transaction (behavior sequence 3) is a thread (which can be seen as a sequence of ASFs), and the ATM session (behavior sequence 4) is a sequence of several threads. As an organization for testing, the ASF level focuses on interaction among objects: the thread level entails interactions among ASFs; and the final level stresses thread interaction.

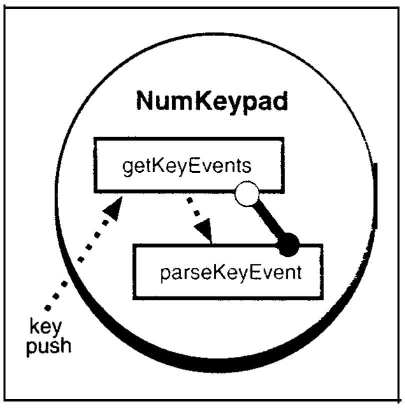


Figure 3. The Digit Entry MM-Path

Examples of Errors. Integration testing of the ATM simulator revealed several errors which would not have been found with unit testing. The error described here occurs as an interaction among methods of a single class.

Objective C classes have initialization methods to properly initialize instance variables defined in the class. Since every class inherits from at least one superclass (the Object class), the initialization method of a class should first invoke the initialization code it inherits, then perform the initialization specific to the class, thus ensuring that initialization occurs in the order of inheritance. The error that integration testing discovered was a lack of invocation of the superclass init method in the NumKeypad init method. As a result, the integer variable hideData was not initialized properly. This variable is used by the keypad driver to determine whether or not to send keystroke values to the screen, or just to send a symbol (such as an asterisk) indicating that a key was pressed. The init method in Key-

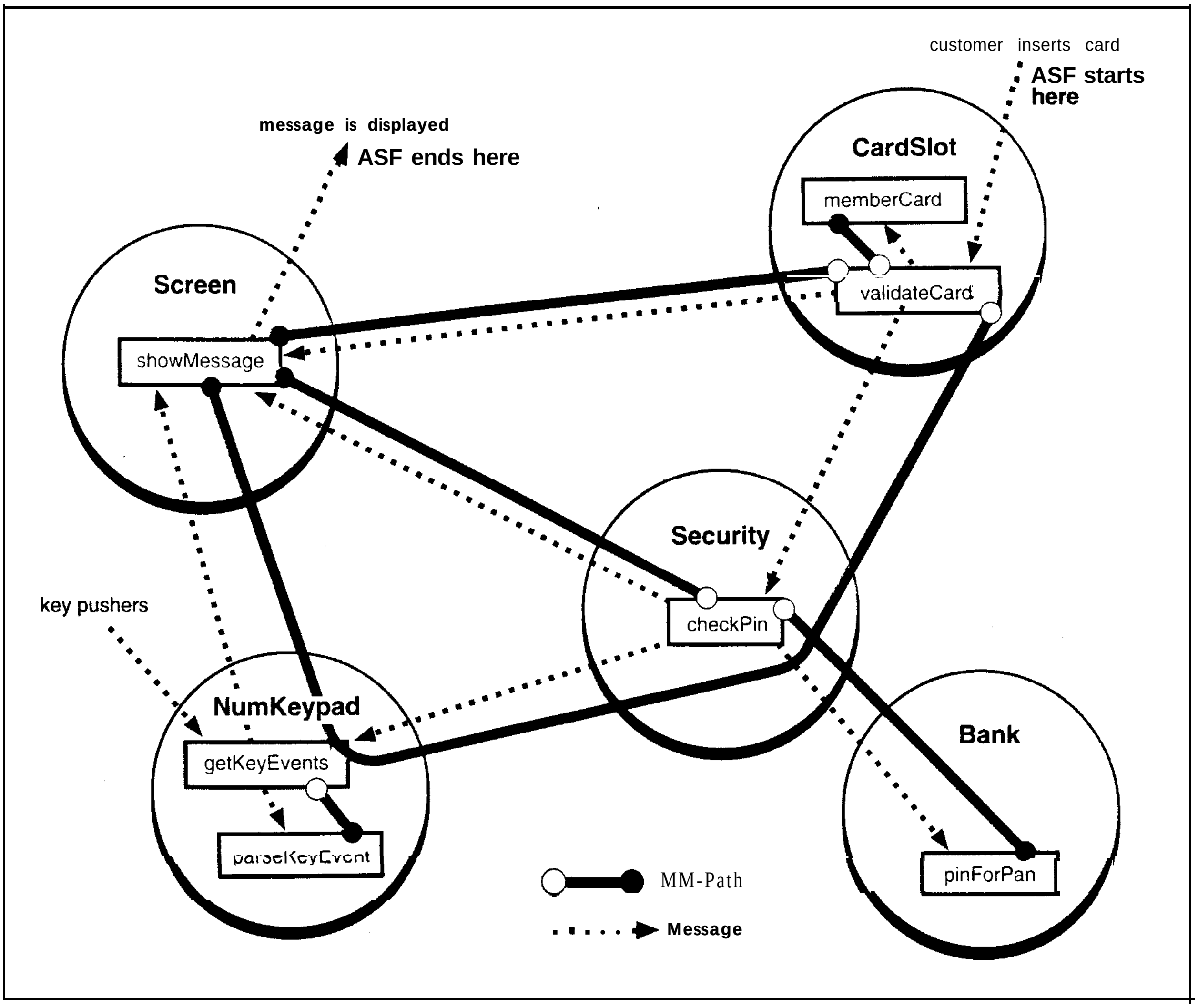
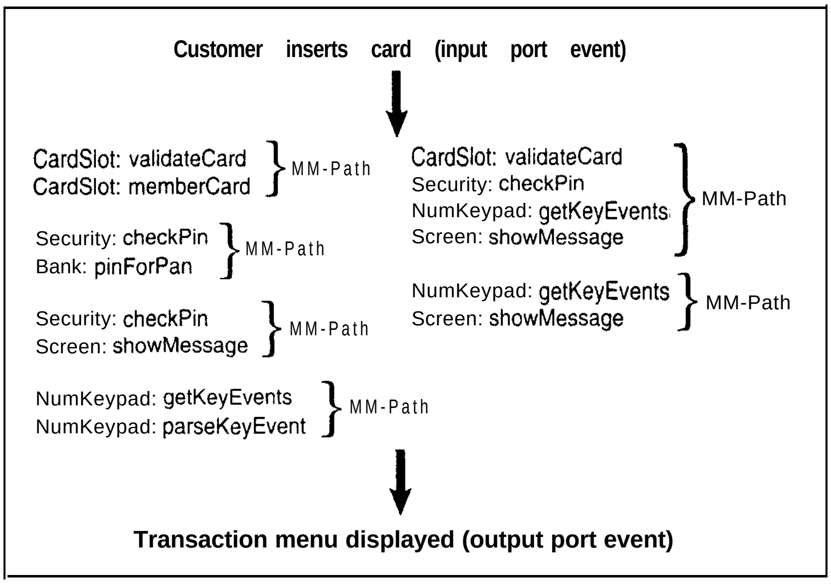


Figure 4. The PIN Entry Atomic System Function



pad sets hideData to TRUE. Since this code was not being invoked for NumKeypad, hideData was not properly initialized. The value it had by chance on allocation was 0, the same as FALSE. When we implemented the PIN Entry portion of the ATM, allocating a NumKeypad object to manage PIN entry, the error of not properly initializing the NumKeypad class was discovered, since the digits were echoed to the screen. Unit testing the init method of NumKeypad did not (in fact, could not) reveal the error because the error was the absence of a message call. Integration testing a NumKeypad object with a Screen and a Security object revealed the error.

Observations

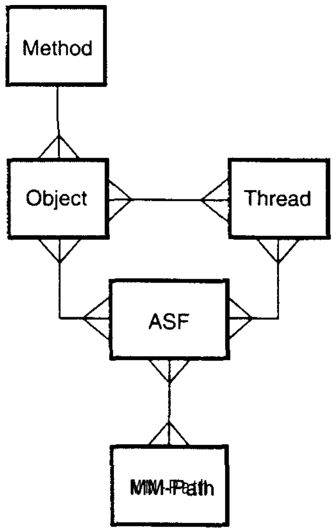
The new constructs defined here result in a unified view of object-oriented testing, with fairly seamless transitions across the five levels discussed earlier. We wish to clarify some of our observations about this formulation. In Figure 6, the constructs of interest are entities in an E/R diagram. The first observation is that many-to-many relationships dominate.

An object may be involved in many threads, and threads entail many objects. Similarly, an object may be involved with many atomic system functions, and an ASF may entail many objects. These two mappings guarantee that objects are integrated,

Figure **5. PIN Entry ASF showing all included MM-Paths**

and furthermore, the integration is grounded in behavioral rather than structural, considerations. One of the pitfalls of structural testing is the problem of infeasible paths. We might expect similar infeasible connections if objects were integrated with structural criteria. So far, our constructs have avoided the problem of structurally possible and behaviorally impossible paths.

Figure 6. **An E/R** model **for** the **constructs of object-oriented testing**



Degenerate cases are sometimes instructive. In our usage of these constructs, we have noted the following:

- The shortest MM-Path consists of two methods linked by one message.
- The shortest ASF consists of an input port event, a minimal MM-Path, and an output port event.
- An MM-Path is maximal within an ASF, that is, an MM-Path ends either at the point of first message quiessence or with the output port event that ends an ASF.

ASFs sometimes have an initial MM-Path which connects to several MM-Paths that produce different port output events. Such ASFs correspond exactly to stimulus/response pairs. This is a seamless juncture between object-oriented integration testing and system(thread) testing.

The shortest thread consists of one ASF. This degeneracy can be compounded by the possibility of the single ASF consisting of a single MM-Path. In the ATM example, this degeneracy happens with digit entry.

We find that the five levels of object-oriented testing result in distinct, useful testing goals, as well as a bottom-up testing order. The lowest level tests individual methods as standalone functions. Once tested, these become nodes in the object network graph, where separately tested methods are connected by messages. Two levels of object integration are helpful, we test the MM-Paths first, and then test their interaction in an ASF. At the system level, the overlap from ASF testing to thread testing is helpful.

Thread interaction testing is beyond the scope of this article, but we note that such interaction is necessarily with respect to data items. If we took such data to be a mutant form of a message (uncertain destination, no return, but clearly a render and receiver), the notation only needs a slight extension.

Two final observations: the new constructs are directly usable as the basis for test coverage metrics, and they work well with composition. For spatial reasons we deleted description of our solution to the time-out problems in the ATM example. To add timeouts. the composition affects the existing objects, MM-Paths, ASFs, and threads. What appears complex and intimidating turned out to be straightforward. From this limited experience, we conjecture that these constructs will be even more useful for object reuse, and the composition that must occur when a system is maintained. **C**

References

1. Agresti, W.A. *New Paradigms for Software Development*. IEEE Computer Society Press, Washington, D.C., 1986.
2. Alford, M. SKEM at the age of eight: The distributed computing design. *IEEE Comput. 18*, 4 (Apr. 1985), 36-46.
3. DeRemer, F. and Kron, H.H. Programming-in-the-large vs. programming-in-the-small. *IEEE Trans. Softw. Eng. SE-2*, 2 (June 1976).
4. Deutsch, M.S. *Software Verification and Validation: Realistic Project Approaches*. Prentice-Hall, Englewood Cliffs, N.J., 1982.
5. Guindon, K. Knowledge exploited by experts during software systems design. Tech. Rep. STP-032-90, MCC, Austin , Tex. , 1989.
6. Harel, D. and Pneuli, A. On the development of reactive systems. In *Logics and Models of Concurrent Systems*, K.R. Apt, Ed. Springer-Verlag, New York, 19x5, pp. 477-498.
7. Jorgensen, P.C. *The Craft of Software Testing*. CRC Press, Boca Raton, Fla. To he published.
8. Paige, M.R. Program graphs, an algebra, and their implications for programming. ***IEEE** Trans. Softw. Eng.* SE-L, 3 (Sept. 1975).

About the Authors:

PAUL C. JORGENSEN is an associate professor of computer science at Grand Valley State University. He also has a consulting practice which combines his research interests with 20 years of industrial software development experience. email: jorgensp@gvsu.edu

CARL ERICKSON is an assistant professor of computer science at Grand Valley State University. His professional interests include object-oriented and distributed information systems, and computer science education. He is a registered NEXTSTEP developer, and an active NEXTSTEP consultant email: erickson @oak.csis.gvsu.edu

Authors' Present Address Computer Science and Information Systems Department, Grand Valley State University, Allendale, MI 4940 1-9403.