1993

Box-Structured Methods for Systems-Development with Objects

Hevner A. R

Harlan D. Mills

Box-structured methods for systems development with objects

by A. R. Hevner
H. D. Mills

Box structures provide a rigorous and systematic process for performing systems development with objects. Box structures represent data abstractions as objects in three system views and combine the advantages of structured development with the advantages of object orientation. As data abstractions become more complex, the box structure usage hierarchy allows stepwise refinement of the system design with referential transparency and verification at every step. An integrated development environment based on box structures supports flexible object-based systems development patterns. We present a classic example of object-based systems development using box structures.

System and software development organizations face difficult decisions when selecting development methodologies. Complex development projects require formal methods for the intellectual control of the process and the resulting system product. After many years of striving to achieve the proven benefits of structured analysis and design methods (e.g., Structured Analysis and Structured Design,[1] Jackson System Development,[2] and Information Engineering[3]), development organizations must now consider the important advantages of object-oriented development methods.

We propose that the decision between structured development methods and object-oriented methods is not a choice of one or the other. With the right conceptual representations and development processes, the advantages of structured development and objects can be integrated into a formal development methodology. In this paper, we discuss the use of *box structures* as a bridge to support the integration of structured concepts and object-oriented concepts.

Object orientation (i.e., the object-oriented approach) is receiving a great deal of attention as a promising approach for the analysis and design of complex information systems. For many system applications, it is very natural to view the system environment as a collection of identifiable objects that collaborate to achieve a desired behavior. Recent research and development in object orientation has led to a number of methods and techniques to support object-oriented systems development. Three principal areas have been studied: object-oriented analysis, object-oriented design, and object-oriented programming.

Object-oriented analysis (OOA) applies object orientation to the initial stages of the systems development process, specifically the analysis of desired or existing system behavior. Prominent works in this area include Bailin's use of objects for requirements specification,[4] Ward's extension of structured analysis to support objects,[5] Coad and Yourdon's comprehensive framework for understanding object-oriented analysis,[6] and Shlaer and Mellor's text on data modeling in objects.[7]

Object-oriented design (OOD) produces a formal specification of the desired system behavior in terms of objects and their interactions. Various graphical and syntactic representations have been proposed to support an OOD system specification. In addition, processes for developing and evolving the object-oriented designs have been defined. The best known OOD methods include Booch's design method,[8] Seidewitz and Stark's method,[9] Meyer's approach for software construction as defined in the Eiffel programming system,[10] and Coad and Yourdon's methods.[11]

Object-oriented programming (OOP) languages, such as Smalltalk, Object Pascal, C++, and CLOS directly support the implementation of an object-oriented design. Other languages, such as Ada, provide limited support for certain object-oriented features such as inheritance and are collectively named "object-based" languages.[12]

A systematic process for object-oriented development should provide a seamless development environment that supports the complete systems development process. Recent research projects have defined object-oriented system development life-cycle processes,[13] including the *object modeling technique* from General Electric Co.[14] and the *responsibility driven design* from Tektronix, Inc.[15]

In recent years development organizations have made large investments in areas such as training experience, and computer-aided software engineering (CASE) tools for the support of structured development methods. The question arises as to whether there is a way to integrate the advantages of object orientation in this existing development infrastructure. Several proposals have been made to use the structured analysis results from data flow diagrams as a basis for object-oriented design (e.g., see Reference 5). A number of problems exist with these proposals.

First, there is a serious gap between data flow diagrams and object-oriented designs. Block diagrams coalesce separate uses of system objects into single nodes and coalesce the separate usage relations among the objects into single arcs between nodes. Thus, such diagrams irreversibly summarize separate transactions that need to be identified in good object-oriented designs.[16]

Second, there is no systematic means of intellectual control over the hierarchical growth of a complex system. There is little clear discipline or order to the discovery, design, and implementation of objects. In particular, the discovery of embedded objects (i.e., objects within objects) and of inheritance opportunities is not addressed.

Third, the approach depends on the heuristic invention of objects from a data flow perspective. There is no formal, mathematical basis for evaluating the correctness or quality of design decisions. Object-oriented designs are often presented as *faits accomplis* from data flow diagrams skipping important analytic steps. In small problems, this may be possible. But in larger ones, it becomes difficult to determine if the leap was inspired or flawed. As complex as large problems are, and as numerous the design alternatives, it is risky business to accept the discontinuity between data flows and object stimuli and responses without a lot of engineering analysis.

Finally, the design and implementation of the transformational functions that tie together objects are left as exercises for the programmer once the objects are completed. Programmers who are not involved in the design process may not understand the intentions of the design and may produce an incorrect system implementation.

Many of these problems arise because of the widely held misconception that top-down functional decomposition found in structured methods is inappropriate and even contradictory to an object-oriented development process. Instead, it is our premise that, with the correct representations and techniques, the advantages of both system decomposition and object composition can be combined into a rigorous systems development with object orientation.

What is needed is a comprehensive process framework and integrated environment to support systems development with objects from initial requirements analysis through system implementation. The objective of this paper is to present box structures as integrating components for object-based structured systems development. Box structures support a rigorous, yet practical, set of methods for the development of systems.[16–18] Box structure methods have been used successfully on numerous systems development projects both internal and external to IBM. (See Reference 19 for examples.) This paper presents an overview of the box structure theory, shows that box structures are, in fact, formal representations of objects by demonstrating that box structures support essential features of objects, and presents an integrated object-based systems development environment with box structures. Good use of box structure operations provides the flexibility to perform needed systems development tasks. Finally, these ideas are applied, by means of an example, to the development of a classic Master File-Transaction File processing system.

Box structure theory

Box-structured systems development is a stepwise refinement and verification process that produces a system design. Such a system design is defined by a hierarchy of small design steps that permit the immediate verification of their correctness. Three basic principles underlie the box-structured design process:[16]

1. All data to be defined and stored in the design are hidden in data abstractions.
2. All processing is defined by sequential and concurrent uses of data abstractions.
3. Each use of a data abstraction in the system occupies a distinct place in the usage hierarchy of the system.

Box structure methods define a single data abstraction in three forms in order to isolate the creative design steps involved in building the abstraction. The *black box* gives an external description of data abstraction behavior in terms of a mathematical function from stimulus histories to responses. The black box is the most abstract description of system behavior and can be considered as a requirements statement for the system or subsystem. The *state box* includes a designed state and an internal black box that transforms the stimulus and an initial state into the response and a new state. The state is designed from an analysis of the required stimulus histories and responses for the system. Finally, the *clear box* replaces the internal black box with the designed sequential or concurrent usage of other black boxes as subsystems. These new black boxes are expanded at the next level of the system box structure usage hierarchy into state box and clear box forms.

Box structures have underlying mathematical foundations that permit the analysis and design to be applied to larger systems of arbitrary size. These foundations are based on sets and functions that can be described in mathematical notation for small systems or subsystems or in well-structured natural language in a given context in larger systems. In any case, a black box is defined by a mathematical function from histories of stimuli to the next response. Let S be the set of possible stimuli, and R be the set of possible responses of a system or subsystem. In illustration, an airlines reservation system, with many thousands of concurrent users, will accept their stimuli sequentially into the system in real time and return responses accordingly. The black box function, say f, will map historical sequences of such stimuli, in this case S^*, to responses, R, shown in the form

$$f\colon S^* \rightarrow R$$

The description of function f may be very complex for an airlines reservation system, but it is still only a function. This description of the black box assumes no data storage between stimuli, even though such storage may be known to exist, or be planned for development.

In a simple illustration, consider a stack object of integers, defined by a set of commands, say RESET, PUSH, POP, EMPTY?, and TOP?, whose functions are easily inferred from the names. A stimulus is a command plus data, if required. For example, a possible sequence of stimuli might be:

RESET, EMPTY?, PUSH 17, PUSH 31, POP, TOP?,
PUSH 11, . . .

The responses for the stimulus histories returning data are:

RESET, EMPTY? → yes

RESET, EMPTY?, PUSH 17, PUSH 31, POP,
TOP? → 17

Although a data stack can be readily imagined, the responses can be determined by examining only the stimulus histories, as above.

The state box of a system or subsystem expands the black box by identifying data at this system level to be stored between stimuli so that only a current stimulus is required, but not previous history. Let T be a set of possible data states at the top level, and let t be the initial state of the system or subsystem. As noted above, the state box contains an internal data abstraction that is defined by another black box, say g. In this case, the internal black box has a compound stimulus consisting of the external stimulus and the internal state, and a compound response consisting of the external response and the new internal state. That is, g has the form

$$g\colon (S \times T)^* \rightarrow (R \times T)$$

Then, each pair ⟨t,g⟩ of an initial state and an internal black box function will uniquely define the behavior of the system. Note that the internal data abstraction will be capable of maintaining more deeply stored data, with the internal black box using its compound stimulus histories.

To continue the stack illustration, consider the state to be a list of integers, with the initial state being the empty list. Then the commands RESET, PUSH, POP, EMPTY?, and TOP? are functions from the stimuli and state resulting in a response and new state. For example, the sequence of stimuli above will produce states as well as responses as follows:

(RESET, ⟨ ⟩) → (null, ⟨ ⟩)

(EMPTY?, ⟨ ⟩) → (yes, ⟨ ⟩)

(PUSH 17, ⟨ ⟩) → (null, ⟨17⟩)

(PUSH 31, ⟨17⟩) → (null, ⟨31, 17⟩)

(POP, ⟨31, 17⟩) → (null, ⟨17⟩)

(TOP?, ⟨17⟩) → (17, ⟨17⟩)

(PUSH 11, ⟨17⟩) → (null, ⟨11, 17⟩)

Furthermore, all intermediate states of this state box can be eliminated by mathematical substitution to derive a black box function, say k, in which the initial state will serve as a parameter. Thus, whenever a specified black box, say f, has been designed into a state box, say ⟨t,g⟩, the correctness of ⟨t,g⟩ can be verified by comparing its behavior, say k, to the intended behavior f.

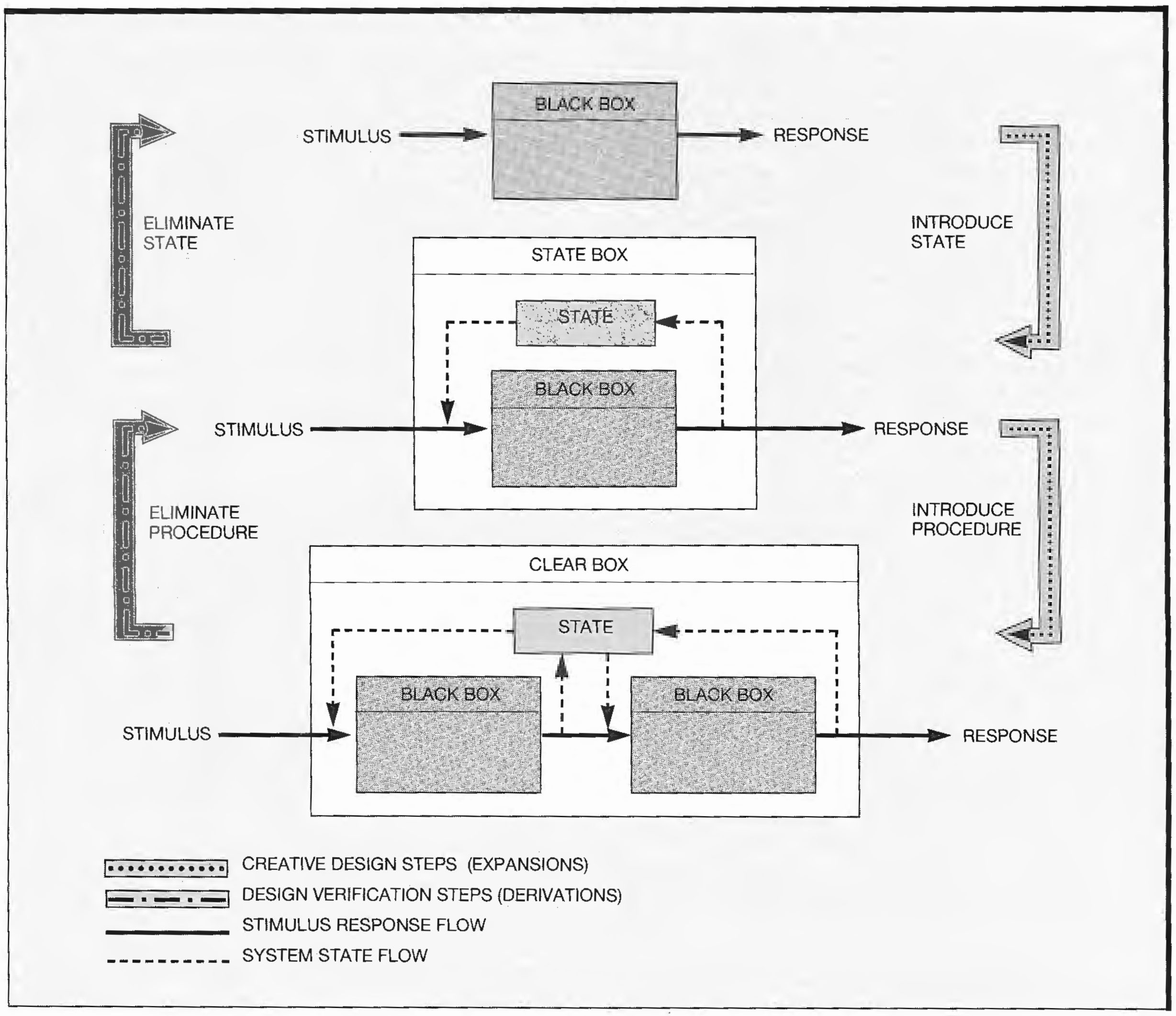
Continuing, a state box can be expanded into a clear box by replacing the internal data abstraction with a procedural structure of new data abstractions in either sequential or concurrent logic. Sequential structures may involve simple sequence, alternation, or iteration whose semantics are well known from sequential programming. Since sequential programs are rules for mathematical functions, from initial states to final states of computation, a clear box in sequential structures defines the functional behavior in terms of the next level black boxes. Concurrent structures require more analysis and discipline in use because of their potential complexities.

Such a procedural structure of data abstractions can also be eliminated to produce the effect of a single internal data abstraction and the state, in much the same way as the state was eliminated to derive a black box. Sequence and alternation structures are eliminated by function composition and disjoint union directly. Iteration structures can be reformulated as noniterative decision structures or recursive structures.[20] Again, concurrent structures require more specific treatment. In this way, clear box designs can be verified against state box specifications, as well.

Figure 1 shows the relationships among the three views of a single data abstraction. The creative design steps, along the right side of the figure, are called expansions. The design verification steps, along the left side of the figure, are called derivations. A given black box can be expanded into many correct state box designs. Conversely, a state box will define a unique black box by derivation. Also, a given state box can be expanded into many correct clear box designs, and conversely, a clear box will define a unique state box by derivation.

In order to gain intellectual control over the development of a complex system, it is necessary to be able to decompose the system into smaller, more manageable parts. A box structure usage hierarchy represents the use of black box abstractions in a higher-level clear box abstraction. A usage hierarchy of abstractions provides referential transparency among all black boxes within a

Figure 1 Box structure expansion and derivation



clear box.[21] Thus, each black box in a clear box can be designed independently of the others.

The effective use of box structures for the development of information systems is guided by the use of four basic box structure principles: referential transparency, transaction closure, state migration, and common services. We briefly define each of these principles.

Referential transparency—Referential transparency occurs when a black box abstraction is completely defined within the clear box at the next higher level in the usage hierarchy. The black box is then logically independent of the rest of the system, and can be designed to satisfy a well-defined behavior specification. The principle of referential transparency provides a crisp discipline for management delegation and assignment of responsibility.

Transaction closure—The principle of transaction closure defines a systematic, iterative specification process to ensure that a sound and complete set of transactions is identified to achieve the required system behavior. The closure process can be performed at each box structure view of an object abstraction. At the black box, checks are performed to ensure that the system stimuli are necessary and sufficient to generate the required system responses. At the state box, the defined transactions must be necessary and sufficient for the acquisition and preservation of all state data, and the state data must be necessary and sufficient for the completion of all transactions. At the clear box, the procedural design and the internal black boxes must include all transactions.

State migration—State data should be identified and stored in the system part (i.e., data abstraction) at the lowest level in the box structure hierarchy that includes all references to those data. At any time in the systems development process, state data can be migrated upward or downward in the hierarchy in order to achieve some system objective, such as minimizing data scope.[22] State migration must be performed carefully in order to maintain the consistency and mathematical correctness of data abstractions throughout the hierarchy.

Common services—A common service is a data abstraction that is described in a separate box structure hierarchy, and used in other box-structured systems. System parts with multiple uses should be defined as common services for reusability. Also, predefined common services, such as database management systems and input/output interfaces, should be used to advantage throughout the box-structured system. The advantages of reusable common services for systems development are obvious. Box structures directly support the identification and reuse of common services within and among systems.

More complete descriptions of box structure theory and principles can be found in References 16–18.

Box structures as objects

Similar to box structures, the object concept can be seen as an extension of abstract data types in programming languages.[23,24] A precise mathematical definition of an object has not been widely accepted or used. An *object* can be informally defined as a unique unit of information and descriptions of its manipulations. More concisely, Booch defines an object as having "state, behavior, and identity."[8] A collection of objects that have a common behavior and structure is termed an *object class*. Booch establishes four major and three minor elements of any object-oriented model.[8] The four major (i.e., essential) elements are abstraction, encapsulation, modularity, and hierarchy. The three minor (i.e., useful, but not essential) elements are typing, concurrency, and persistence.

An object-oriented systems development process must support all major object elements and should support the minor object elements as appropriate for its application environment. In this section, we demonstrate that the box structure theory incorporates the essential elements of the object model. We also discuss the box structure approaches for supporting other useful elements of the object concept.

Essential object-oriented elements. We now briefly show that box structures provide the concepts of abstraction, encapsulation, modularity, and hierarchy.

Abstraction. An object is an abstract representation of an entity in the problem domain. Much creative skill and experience are needed to identify and design a good set of system objects and classes. Box structures provide an excellent set of abstraction capabilities for system description. During analysis, a potential object can be defined and studied in any of the three box structure views. In particular, the black box view gives the external, design-free system behavior that provides the essence of a system abstraction.[25] The state box views the object as a data abstraction with the state visible. Within the clear box view complex object abstractions can be rigorously decomposed into simpler objects and simple objects can be grouped into larger objects.

During top-down design, the box structure usage hierarchy provides a framework in which to capture multiple levels of system abstraction in a controlled manner. All system design units, from the top-level complete system to the smallest subsystem components, and even down to simple variables, are viewed and described as box structure objects. Throughout the hierarchy, the ability to manage abstraction applies to all system components; stimulus (i.e., input), responses (i.e., outputs), state (i.e., internal data), and procedures.

The ability to handle abstractions is also important for the reverse engineering of existing systems. Bottom-up system analysis abstracts functionality (i.e., black box behavior) from system implementation details of procedure and state. The application of box structure theory to system reverse engineering is presented in Reference 26.

Encapsulation. Encapsulation, also known as information hiding, is supported by the state box and clear box views of a box structure object. The state of an object and the procedural operations on that state are hidden within the box structure as design constructs. The essential behavioral abstraction, or interface, of the object is described by the black box view.

An extension to object encapsulation can be found in the box structure principle of state migration. As box structure objects are decomposed and composed in a usage hierarchy, opportunities for state migration may exist. Beneficial state migrations provide insights into new class inheritance structures. Upward migration of state can identify new superclass structures and downward migration of state can identify new subclass structures.

Modularity. Modularity in systems development involves dividing the complete system into manageable units of analysis, design, and implementation. Each system module must be internally cohesive and loosely connected to the other modules of the system.[8]

Modularity is one of the major strengths of the box structure development process. The principle of referential transparency throughout the box structure usage hierarchy provides module independence for all box structures in the system design. Furthermore, referential transparency applies to both object decomposition and object composition in the systems development process, as discussed in the next section on hierarchy.

Hierarchy. The concept of a system hierarchy is an essential component for systems development. Box-structured systems development with objects utilizes two distinct types of hierarchy: *usage hierarchies* to describe system behavior and *inheritance hierarchies* to describe object behavior via inheritance.

A usage hierarchy of box structures is constructed during system design by the application of both system decomposition and object composition. Top-down system decomposition enables an essential intellectual control in development. The system grows one level at a time. The mathematical structuring of systems in usage hierarchies of objects allows formal verification methods to be used. Also, the referential transparency of objects in a clear box provides an essential modularity and design independence to each object.

In addition, in this framework of a usage hierarchy, the advantages of object composition come into play. An object requirement, stated as a black box, can be matched with existing object classes stored for reuse in a repository. During the systems analysis phase the benefits and costs of object reuse and modification can be studied. Another opportunity for object composition arises during the design of the clear box. Knowledge of existing object classes or insight into desired object classes will influence the designer's invention of data abstractions as black boxes at the next level in the object hierarchy.

As an object is used in the system usage hierarchy, it carries with it a description of its inherent behavior as defined in an inheritance hierarchy. Inheritance is a fundamental aspect of object orientation. Inheritance is the means by which one object class, the subclass, inherits the information and operations of another object class, the superclass. The subclass can then be modified by adding or deleting information or operations of its own.

Inheritance is exhibited in the box structure development process by building new classes from existing classes during systems development. After an object has been instantiated in a system design, the designer has the freedom to modify the object design by altering the state design of the state box (e.g., via state migration) and the procedural design of the clear box. If the modified object is designated for reuse, then a choice can be made as to its representation in the reuse repository. The new object class, from black box to clear box, can be stored as a unit or the new subclass can be stored as a set of modifications with a pointer to the existing superclass. Thus, an inheritance hierarchy can be developed of object classes. The physical structure of the hierarchy is a representation issue based upon an optimization of the reuse repository. Thus, an object is defined and stored in the form of a generic common service.[27]

Important object-based features. Based upon the fact that the box structure theory supports the four essential elements of Booch's object model, we conclude that box structures support object-based systems development. In fact, box structures provide important extensions for systems development with objects. These extensions include the isolation and verifiability of all creative design steps in small units and systematic expansion of the design in a top-down hierarchy for intellectual design control. Thus, there is no need to develop transformational functions to tie objects together, as is required in some traditional object-oriented design methods.[28]

Next we discuss several important features of box-structured development methods, to include Booch's minor elements of typing, concurrency, and persistence, as well as reuse and object representations.

Typing. The typing of an object identifies the object as a member of a specific class with all inherent states and behaviors. Object typing ensures that differently typed objects may interact only in very restricted ways. The support in various OOP languages for typing ranges from weak enforcement to strong enforcement. The box structure syntax does not specifically enforce strong typing in design specifications.

Concurrency. We believe that the ability to analyze and design concurrent structures is essential for realistic systems development. The clear box structure provides the means to model the concurrent behavior of black box objects. We have defined analysis and design methods to optimize the use of concurrency in system specifications.[29] However, many difficult questions remain to be explored.

Persistence. Persistence through time and space is embedded in the organization-wide common services as discussed in the next section on reuse. The use or reuse of persistent common services, such as object-oriented database management systems, allows data and procedures to be shared across many system boundaries.[30,31]

Reuse. Reuse is a fundamental concept in object-oriented development.[32] The reuse of objects within and among systems has the potential to significantly raise the productivity of systems development and the certified quality of systems. Box structure methods support a high level of object reuse.

In a top-down manner, each object in the system hierarchy is stored in its three box structure views in a systems development repository. Certain of these objects, usually the smaller objects at lower levels in the hierarchy, can be selected for future reuse. Objects are stored in the form of their inheritance hierarchies. Special design requirements are imposed on the objects, such as interface standards, documentation standards, and certification requirements. These reusable objects are migrated to large organizational repositories as object classes for potential reuse across all development projects. By including all three box views of the object in the reuse repository, a verified design trail of the object from requirement to detailed design is available for evaluation and use during reuse decisions.

During design, reuse decisions are made for a given black box requirement. It may be possible to find a reusable object type in the reuse repository that meets the requirement. (Current research on repository structures and access methods for reuse is reported in Reference 33.) We recognize several forms of object instantiation for reuse during a systems development.

An *organization-wide object instantiation* would encapsulate information and operations used by many systems. Such objects would include database and file management systems, common user interfaces, and sensors that maintain the state of physical properties (e.g., temperature, pressure). A *system-wide object instantiation* would encapsulate information (e.g., data types and constants) and operations used in several different places in the system usage hierarchy, but not outside of the system. Examples would include commonly used data structures and their operations (e.g., files, stacks, queues) and monitors for critical sections of the system. A *one-time object instantiation* would allow reuse of information and operations without information sharing. This would be beneficial primarily for reusing existing program code. The first two forms of object instantiations are examples of box structure common services.

Table 1 The 16 box structure operations

1.	Requirements determination
2.	Black box definition
3.	Black box analysis
4.	Black box requirements review
5.	State box expansion
6.	State box analysis
7.	Black box derivation
8.	Clear box expansion
9.	Clear box analysis
10.	State box derivation
11.	Stepwise system decomposition
12.	System implementation
13.	System operations
14.	System analysis
15.	System box structure description
16.	Stepwise system abstraction

Object representation languages. The search for appropriate languages for object-oriented development has led to graphics-based aids such as Smalltalk icons[34] and Booch diagrams,[8] and syntactical forms such as Ada program description languages (PDLs).[35] Box structures have both a graphic notation and a syntactic notation, that being the box description language (BDL).[17] While graphics may be appropriate for small system designs and high-level presentations, we see no alternative for the use of a syntactically complete design language for large-scale object-oriented development of systems. The use of the Z notation has also been used to represent box-structured designs.[36]

An integrated box-structured environment for systems development with objects

Box structures and the box structure usage hierarchy provide the common, unifying concepts for achieving a truly integrated object-based development environment. No artificial bridges and transformation procedures are needed to exchange information among development activities. We have defined 16 fundamental box structure operations (see Table 1) and show these in a schematic structure of an integrated development environment (see Figure 2). These operations, used and reused in various patterns, contain all the required processing needed to perform all activities in object-based systems development. The box structure information is stored in well-defined box structure formats, box structure graphics, and the box description language in systems development repositories. In this section, we describe the 16 operations and discuss several important patterns of object-based development.

Each of the box structure operations shown in Table 1 is atomic, accepting stimuli from and producing responses to the system developer and the systems development repositories. At any point in systems development or systems evolution, an operation can be performed as needed as long as the stimuli for it are available. It is incumbent upon the system developer to put the operations to "good use" in the development process. Natural groupings of the operations are exploited in good-use patterns. The box structures that underlie all of the operations provide the essential formalism and integration required for rigorous systems development. We next briefly describe the objectives of each of the operations.

1. *Requirements determination* involves a series of investigation activities in which system requirements are specified. The information is gathered via techniques such as user interviews, questionnaires, documentation review, and analysis of existing applications. The gathered requirements information is represented in box structure formats.

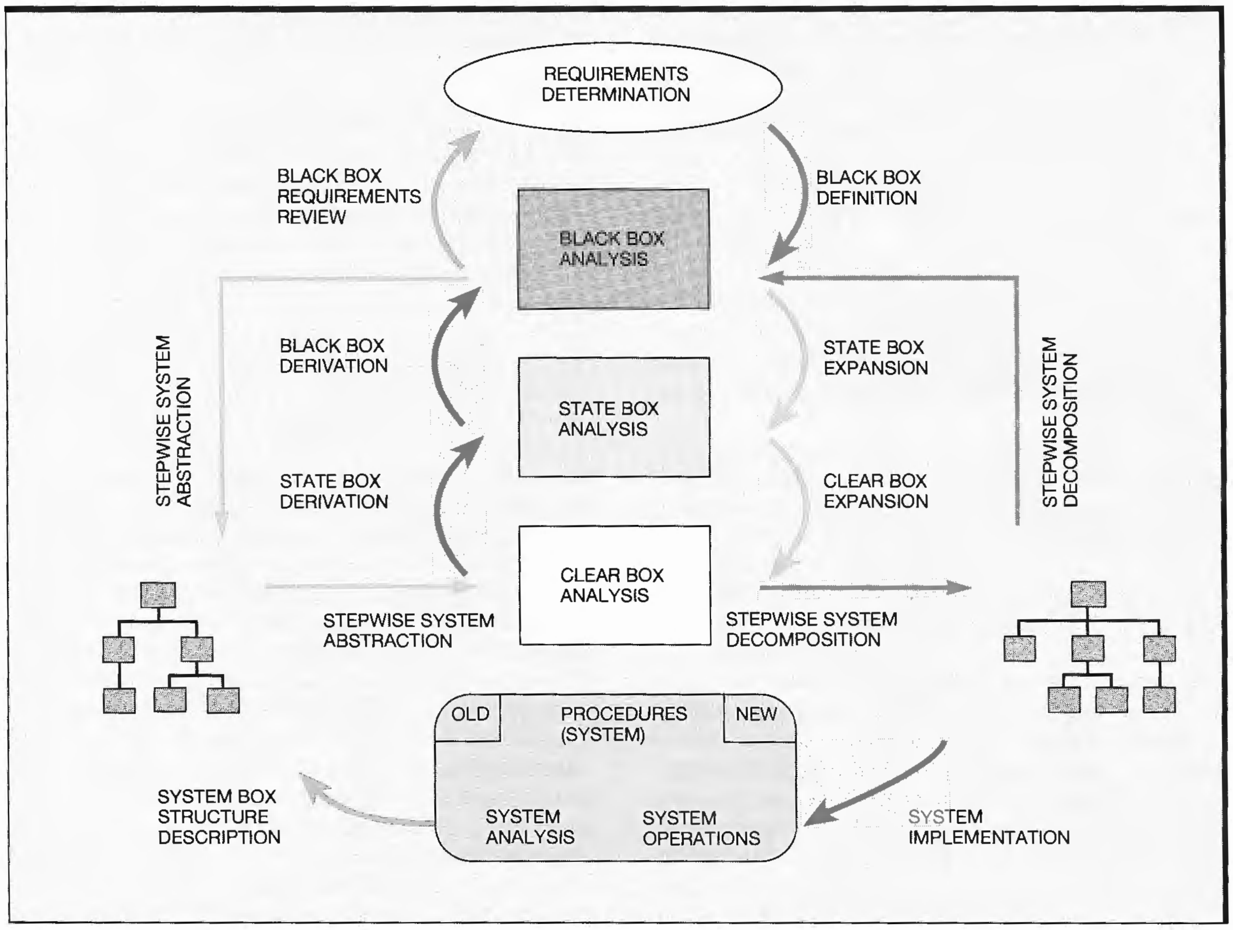
2. The black box of the system is completely defined (*black box definition*) based on the requirements for the system. The black box is described by its stimuli, responses, and the transactions that map stimulus histories into responses.

3. *Black box analysis* evaluates the quality and completeness of the black box specification. For example, transaction closure would ensure that all stimuli are necessary and sufficient in the system.

4. The defined black box is reviewed (*black box requirements review*) to determine whether it truly represents the desired system requirements. The review involves the customers, users, and managers of the system.

5. The state of the system is created (*state box expansion*) by encapsulating required stimulus history in a state box. Data design methods, such as entity-relationship models, are used to create a state design. An internal data abstraction is designed to map stimuli and state into responses and new state.

Figure 2 A schematic structure of an integrated development environment



6. *State box analysis* evaluates the quality and completeness of the state box design. The principles of transaction closure and state migration are applied. Data design metrics, such as level of data normalization, are used to evaluate the quality of the design decisions.

7. The *black box derivation* operation discovers the black box representation of a given state box. A state box can be verified as correct by deriving an equivalent black box and comparing it to the original black box requirement.

8. *Clear box expansion* is a creative step whose purpose is to design the procedural structure of the system. The uses of black box subsystems at the next level of design are identified. The intellectual control of stepwise system decomposition is contained in this operation.

9. *Clear box analysis* evaluates the quality and completeness of the clear box design. The principles of transaction closure, state migration, and common services are applied. Design metrics of structured programming can be used to study the clear box procedural design.

10. The *state box derivation* operation discovers the state box representation of a given clear box. A clear box can be verified as correct by deriving an equivalent state box and comparing it to the original state box.

11. *The stepwise system decomposition* operation continues the system design in a top-down manner by recursively applying the above operations to each black box at the next level of the box structure usage hierarchy. Common service box structures are identified and developed separately from the application system usage hierarchy.

12. *System implementation* accepts the design specification in the form of a box structure usage hierarchy and provides the capabilities and resources to implement it. Implementation may be an integration of hardware, software, and human behavior. Implementation objectives are to build and optimize the specified system and to prepare users and operators for its operation and maintenance.

13. Activities during *system operations* include maintenance, performance monitoring, integrity control, operations assurance, and system evolution. Box structures provide a rigorous and common means of understanding and controlling the system during operation.

14. For an existing system, *system analysis* is an investigation activity to support a better understanding of system behavior. Operational system metrics, such as performance, reliability, availability, etc., are computed and used to evaluate the quality and completeness of the system. Information is gathered from interviews and documentation reviews to better understand system behavior. This information is stored in a repository.

15. An existing system can be described in box structure representations to support further rigorous analysis and reverse engineering. Our goal is to enhance system understanding by describing the system (*system box structure description*) as a usage hierarchy of referentially transparent clear boxes. Methods for transforming natural procedures into clear box formats are presented in Reference 17.

16. *The stepwise system abstraction* operation builds an increasingly abstract description of an existing system in a recursive, bottom-up fashion. Detailed clear box descriptions of subsystems are derived to state box and black box representations. These subsystems are then represented as black boxes within procedural clear boxes at the next higher level of system description. This process continues until the complete system is described and understood at the top-level behavior. This operation is the basis of the reverse engineering of existing systems as presented in Reference 26.

The integrated systems development environment for box structures would include support for the 16 box structure operations and a common and controlled repository for storing box structure information. With the flexibility of being able to perform any of these operations at any time during systems development, the developer is no longer bound by a rigid systems development life-cycle paradigm. However, a discipline is still needed for the good use of the operations toward a well-defined systems goal.

The use of box structures can be adapted to any development situation in a flexible way by defining good-use patterns of operations. These patterns would be placed under strict management control and adapted dynamically to changing circumstances in the on-going systems development. Each box structure operation in the pattern has well-defined completion criteria, allowing immediate validation of the success or failure of any particular step in the development. In addition, since the creative invention operations (i.e., state box expansion and clear box expansion) are self-contained, it is easy to track and document the critical design decisions in the system.

To illustrate, consider the following examples of good-use patterns of box structure operations. For conciseness, we refer to the operations using their numbers as defined in Table 1.

Object description example. The description of an object would begin from the discovery of the object and a thorough requirements determination (operation 1). The object would be designed as part of an existing inheritance hierarchy (i.e., common service) or would initiate a new inheritance hierarchy. In either case, the design of the object would proceed through defining the black box, state box, and clear box views (operations 2–10). Subclasses of the object are defined using recursive application of these operations in the inheritance hierarchy (operation 11).

New system development example. The development of a system from the beginning would start from extensive requirements determination (operation 1) and proceed recursively through the top-down construction (operations 2–11) of the box structure usage hierarchy of the system design. Finally, the system would be implemented (operation 12) and brought into operation (operation 13). While this pattern of operations is optimistically possible, it is rare in practice. New system development will require many iterations of requirements determination, box structure analysis (to include reuse analysis), box structure design, and system implementation. The flexibility to dynamically select and perform the operation needed next is of great benefit.

Reverse engineering of systems example. Reverse engineering is defined as "the process of analyzing a subject system to identify the system's components and interrelationships and to create representations of the system in another form or at a higher level of abstraction."[37] A pattern of operations to support reverse engineering would be defined by the application of system analysis and system box structure description (operations 14 and 15). Then stepwise system abstraction would be performed as a recursive pattern of analyses and derivations (analysis operations 3, 6, and 9, derivation operations 7 and 10).

Prototyping example. A prototype is a limited version of a system built to provide requirements and operations information. Prototypes can range in scope from a simple study to see if software packages can exchange data correctly to a large-scale prototype of the complete system. Once the decision is made to prototype a portion of a system, the prototype development takes on an independent existence of its own. The pattern of box structure operations would be similar to the pattern for developing a new system. However, not all branches of the box structure usage hierarchy would be completed. Only the portions of the system to be studied would be designed and implemented. By developing the design with the usage hierarchy, referential transparency of all system parts in the prototype is maintained. This supports the ability to make use of these prototype subsystems in the design and implementation of the desired final system.

The box-structured systems development process

In this section, we apply the integrated systems development environment discussed in the previous section to build a good-use pattern of box structure operations for the object-based development of a system. We propose an object-based systems development process that consists of five phases. The order of performance of the phases during a system development is based on the spiral paradigm in which the next phase of development is determined by the results of the previous phases.[18] This requires definite result milestones and strict management control of the development process. The development phases follow.

Problem definition—A clear problem statement must be generated to provide a basis for systems development. Extensive domain analysis is essential for complete problem understanding.

Requirements definition—Requirements are elicited from the system domain experts and system users. The requirements are represented in formats that facilitate review and feedback.

Systems analysis—The system requirements are analyzed and information is gathered to support subsequent design decisions. The discovery of relevant, reusable objects is an important part of systems analysis.

Systems design and verification—Definitive design decisions are made and the system design is grown via top-down functional decomposition in a usage hierarchy. Each creative design step is verified to be a correct expansion of the existing design.

Systems implementation—The system design is transformed into an operational system. The final system will be a combination of hardware, software, firmware, and human behavior components. The boundaries and interfaces among these components must be specified in the final system design.

Our emphasis in this section is to detail the processing found in the middle three phases and to demonstrate the inherent object basis of the box structure development process. The phases of requirements definition, systems analysis, and systems design and verification will be performed as a tightly-integrated, iterative process. The ability to achieve this tight integration comes about because of the unifying box structure concepts and representations. (We use the term "box structure" to refer to a component in the system hierarchy; however, the term "object" could be used with equivalent meaning.)

Requirements definition. The input into the requirements definition phase is a complete problem statement, typically presented as a structured English document. Investigation tasks are performed in order to precisely determine the requirements of a system that solves the presented problem. Note that the requirements definition phase is performed for each box structure in the usage hierarchy.

Requirements for any level of system object can be represented in a box structure format. The ultimate goal would be to state all requirements in a state-free, procedure-free black box. Defining requirements solely as a black box places no constraints on the eventual design. The first four box structure operations (requirements determination, black box definition, black box analysis, and black box requirements review) are performed iteratively during this phase.

The transactions in a black box are defined as mathematical functions for deterministic behavior or mathematical relations for nondeterministic behavior. For high-level, complex box structures it may be necessary to provide the function or relation in the natural language of the problem domain, often a mixture of formal and informal language. Whatever the notation, the black box description is a set of mathematical functions, one per transaction.

Often system requirements do contain design constraints on such things as the availability and use of data or the need to conform to a defined procedure. Such requirements cannot be recorded in a black box; thus, a clear statement of state box and clear box design constraints must be provided. In addition, certain "nonfunctional" requirements, such as performance and documentation standards, can be stated in structured English forms. It is important during requirement reviews that the system owners understand that any requirements beyond a black box are constraints upon the design freedom for the system. In this process, many nonessential "requirements" can be discovered and eliminated.

The results of the requirements definition phase are a precisely defined black box with accompanying state box, clear box, and nonfunctional design constraints. This box structure requirement is stored in a repository as the initial definition of the system object.

Systems analysis. Analysis tasks are performed to support the decisions that must be made during systems design. These tasks are performed as part of the creative state box expansion and clear box expansion structure operations. The box structure requirement is analyzed and information is gathered to support one or more feasibility, reuse, prototpye, or tradeoff types of activities.

Feasibility studies are performed to determine the feasibility and cost versus the benefit of potential designs. *Reuse opportunities* are explored in several ways. Repositories of system objects from the current project or existing systems will be investigated for requirements matching. The cost and benefit of reusing existing objects, along with any required modifications, would be determined. *Prototyping* is performed to evaluate design alternatives. The prototype development process will progress independently from other design activities with the five development phases performed in an iterative manner. Objects developed in the prototype may be candidates for reuse and modification in the final system. *Tradeoff studies* are used to determine the advantages and disadvantages of designing and implementing the current box structure as hardware, software, firmware, human behavior, or some combination thereof. Such decisions will impact reuse opportunities and interface designs. Finally, the *reuse potential* of the current box structure should be analyzed. If the decision is made to design the box structure as a reusable object, then reuse standards may dictate certain design decisions (e.g., interface standards).

The above types of analyses are essential to support high-quality system designs. The information, analysis, and conclusions of these studies are recorded with the evolving box structure in the system repository. Some analysis discoveries may cause changes in the system requirements, thus, iteration between the phases of requirements definition and systems analysis is to be expected and encouraged.

Systems design and verification. In this phase the box structure requirement and the analysis results are used to produce a complete design specification of the box structure. This phase encompasses operations 5 through 11 of the integrated box structure environment.

First the state box is designed from the black box requirement specification using the state box expansion, state box analysis, and black box derivation operations. The completed and verified state box is stored in the repository. The clear box then can be designed using the clear box expansion, clear box analysis, and state box derivation operations.

The design and verification of the clear box completes the detailed design of the current box structure. The complete specification of the box structure object, from the black box requirement, through the intermediate state box, to the final clear box design, is stored in the system repository. Finally, the stepwise system decomposition operation is used to build the complete system in a top-down manner.

The procedural clear box design, developed in the clear box expansion operation, ensures that each internal black box is referentially transparent from all other peer black boxes and common services in the clear box. Thus, each black box can be designed independently. For each black box requirement the development process of requirements definition, systems analysis, and systems design and verification begins. Note that much of the work performed (and dutifully recorded in the repository) for higher-level box structures in the hierarchy can be used in the analysis and design of lower-level box structures. The desired system is complete when no further black box requirements exist in the leaves of the box structure usage hierarchy. The detailed design of the complete system is then sent to the final phase of systems implementation.

The design of a Master File-Transaction File system

We demonstrate the application of object-based development with box structures to a simplified version of the classic example of a Master File-Transaction File system. The following problem statement is given:

A supply business maintains a master file of parts inventory with attributes of part identification (PARTID) and quantity on hand (QOH). Each day parts are received and shipped. For each transaction, a record is added to a transaction file with attributes of part identification (PARTID), action (ACTION), and quantity (QTY), where ACTION has the values of "in" or "out." The system transfers the transactions to the master file at the end of each day. A management control report is produced showing the disposition of each transaction record and its effect on the master file.

A classic example illustrates the application of object-based systems development.

We develop the top level of this system using a box structure box description language notation similar to typical program description languages (PDL), and it should be self-explanatory.

Requirements definition. We begin by listing all of the stimuli and responses of the desired INVENTORY system. They are:

Stimuli	Transaction file and master file
Responses	Updated master file and management report

The discovery of system requirements should point out omissions and needed extensions of the problem statement. For example, what are the correct actions to be taken when unusual or erroneous conditions arise? We deal with two such conditions in this example. If the transaction file is empty, the management report will note this and the system finishes. If the PARTID in the transaction file does not match any record in the master file, the transaction record will be written with an error message. All pertinent conditions and contingencies should be studied during the requirements definition phase.

The black box notation for the INVENTORY system requirement is:

```
Black Box Inventory

  stimulus
    Transaction_file : file of records
      record
        PARTID : integer,
        ACTION : type of ('in', 'out'),
        QTY : integer
      endrecord.
    Master_file : file of records
      record
        PARTID : integer,
        QOH : integer
      endrecord.

  response
    Master_file : file of records
      record
        PARTID : integer,
        QOH : integer
      endrecord.
    Report:
      record
        HEADER : report_header,
        BODY : report_body
      endrecord.

  behavior

  if    The transaction file is empty
  then  Write the management report
  else
    for Each record in the transaction file
    do
      Match the PARTID value into the
      master file
      if    A match exists
      then  Modify the QOH value by add-
            ing (ACTION = 'in') or sub-
            tracting (ACTION = 'out') the
            value of QTY;
            Write the transaction record
            and new master record in the
            management report
      else  Write the transaction record
            and an error statement in the
            management report
      if;
    od;
  fi;
end Black Box Inventory.
```

Note that the transaction statement in the black box is a mixture of keywords and structured English for exposition purposes. Equivalently, we could have presented a mathematical representation of conditional algebraic assignments for the transaction.

Systems analysis. We concentrate our analysis for the example in discovering reuse opportunities. We assume that a File_manager object type exists as a box structure design in the existing reuse library. The object type is designed to encapsulate a file of arbitrary design and size. Visible operations on the file would include typical file operations, such as the following:

OPEN	Establishes currency pointer at first record of file and checks access rights
ISEMPTY	Checks if file is empty, returns Boolean value
READ	Reads record at currency pointer, moves pointer to next record
ATEOF	Checks if currency pointer is at EOF, returns Boolean value
WRITE	Overwrites given record at currency pointer
FIND	Given a primary identifier value, finds the first record with that identifier; if no match is found, a STATUS value is returned
ADD	Given a record with a valid identifier, places the record in the file in correct order
DELETE	Given a record identifier, finds record and deletes it from file
CLOSE	Establishes file integrity and update commitments, releases any file locks

We assume that two object instantiations of File_manager encapsulate the master file and the transaction file. Since these files would also be used by other systems in the business, these objects would be organization-wide common services. We name the objects Master_file and Trans_file.

Systems design and verification. State box design of the INVENTORY system would discover the need to store the evolving management report as intermediate state. Thus, the state box design is given as follows:

```
State Box Inventory

  common services
    Master_file.
    Trans_file.

  stimulus

  response
    Report :
      record
        HEADER : report_ header,
        BODY : report_body
      endrecord.
  state
    Report :
      record
        HEADER : report_ header,
        BODY : report_body
      endrecord.

  behavior

  if   The transaction file is empty
  then Write Report
  else
    for Each record in Trans_file
    do
      Match the PARTID value in Master_file
      if   A match exists
      then Modify the QOH value in Master_file
           by adding (ACTION = 'in')
           or subtracting (ACTION = 'out')
           the value of QTY from Trans_file;
           Write Trans_file record and new
           Master_file record in Report
      else Write Trans_file record and an
           error statement in Report
      if;
    od;
  fi;
end State Box Inventory.
```

The state box can be verified as a correct design of the black box requirement in a straightforward manner. Although we do not present all of the details here, the critical tasks would be to verify the correct uses of Master_file and Trans_file objects and the Report state in the state box transaction.

During the clear box design, an important design decision presents itself. Should Report remain as global state in the system or should it be encapsulated into a data abstraction with visible operations? We choose to develop a system-wide common service object called Mgmt_report with Report as encapsulated data and four visible operations:

NEW	Initializes Report with defined header information, such as date, time, titles, and column headings
ADD	Adds correctly processed Trans_file record and new Master_file record to body of Report
ERROR1	Adds a Trans_file record and error statement to body of Report when no match is found in Master_file
PRINT	Prints the current state value of the Report

The Mgmt_report object will be completely developed and verified, from black box requirement to clear box design, and used in the INVENTORY system as a common service object. The clear box design of INVENTORY could be presented as:

```
Clear Box Inventory

  common services
    Master_file.   (* organization-wide
                      common service *)
    Trans_file.    (* organization-wide
                      common service *)
    Mgmt_report.   (* system-wide
                      common service *)

  stimulus
  response
  state

  behavior
  data (* temporary data *)
    TEST1 : Boolean,
  proc
    use Mgmt_report(in: NEW);
    use Master_file(in: OPEN);
    use Trans_file(in: OPEN);
    use Trans_file(in: ISEMPTY, out: TEST1);
    if NOT TEST1 then use Update_master fi;
    use Mgmt_report(in: PRINT);
    use Master_file(in: CLOSE);
    use Trans_file(in: CLOSE)
  corp
end Clear Box Inventory.
```

Again, the verification of the clear box can be done and will not be presented here.

The only new object at the second level of the system hierarchy is the Update_master black box. We would iterate the development process for this object, defining the black box, performing systems analysis, and, finally, designing the state box and clear box. For purposes of space, we show the final clear box design.

The design work is complete when there are no undefined black boxes and the system is completely specified.

```
Clear Box Update_master

  common services
    Master_file.    (* organization-wide
                       common service *)
    Trans_file.     (* organization-wide
                       common service *)
    Mgmt_report.    (* system-wide
                       common service *)

  stimulus
  response
  state

  behavior
  data (* temporary data *)
    TEST2 : Boolean,
    T_REC :
      record
        PARTID : integer,
        ACTION : type of ('in', 'out'),
        QTY : integer
      endrecord.
    M_REC :
      record
        PARTID : integer,
        QOH : integer
      endrecord.

  proc
    use Trans_file(in: ATEOF, out: TEST2);
    while NOT TEST2
    do
      use Trans_file(in: READ, out: T_REC);
      use Master_file(in: FIND,
      T_REC.PARTID out: M_REC, STATUS);
      if STATUS = NOT_FOUND
      then use Mgmt_report(in:
            ERROR1, T_REC)
      else
        if T_REC.ACTION = 'in'
        then M_REC.QOH ← M_REC.QOH
                            + T_REC.QTY
        else M_REC.QOH  ← M_REC.QOH
                            − T_REC.QTY
        fi;
        use Master_file(in: WRITE,
        M_REC);
        use Mgmt_report(in: ADD,
        M_REC, T_REC)
      fi;
      use Trans_file(in: ATEOF, out: TEST2);
    od
  corp
end Clear Box Update_Master.
```

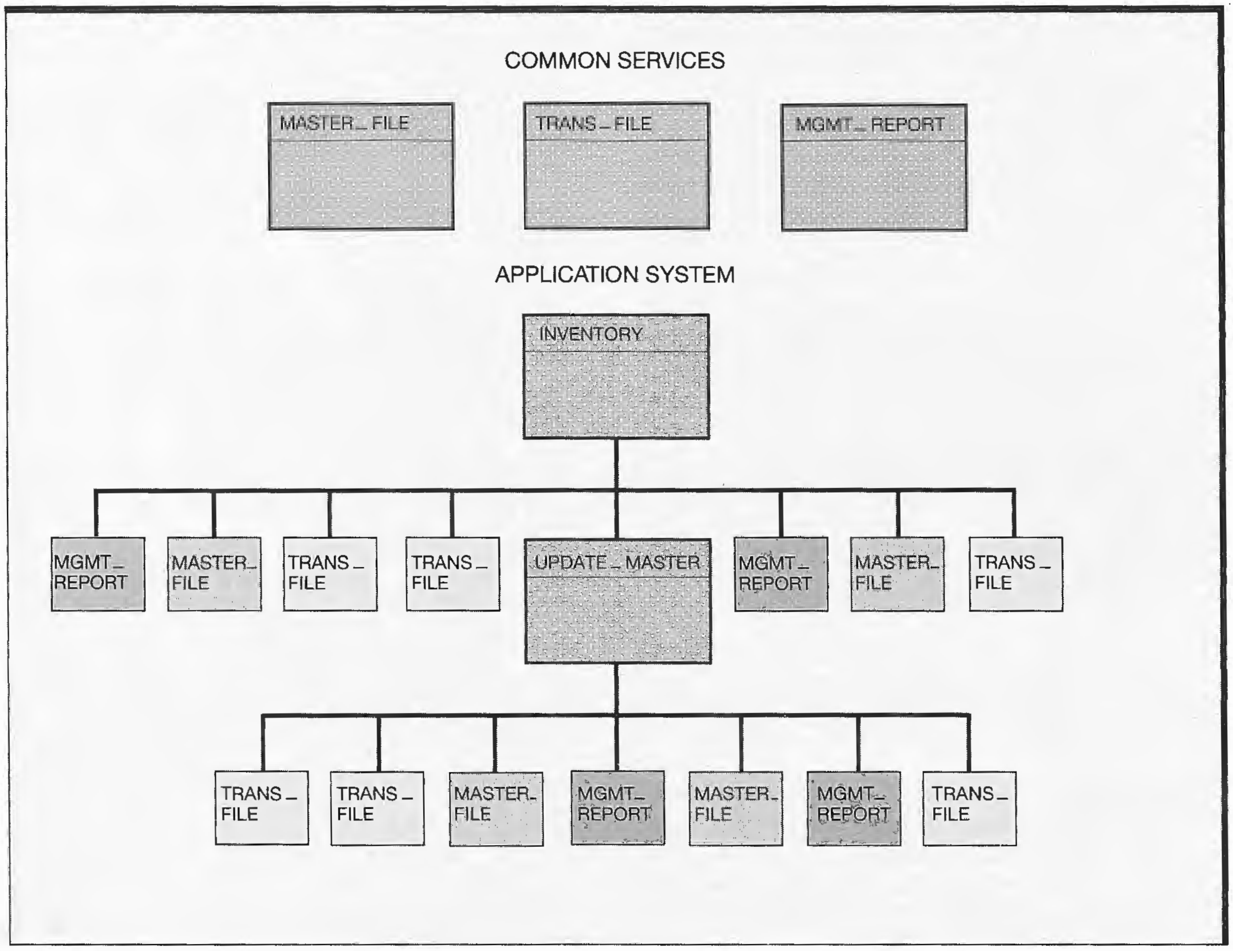
Since there are no undefined black boxes in Update_master, no further design work is needed and the INVENTORY system is completely specified as a hierarchy of object uses. Figure 3 shows the box structure usage hierarchy for this resulting system.

Observations for this example. In the INVENTORY system development, we have identified and created five objects: Inventory, Update_master, Master_file, Trans_file, and Mgmt_report.

Master_file and Trans_file are instantiations of a file management object type to encapsulate the master inventory file and the daily transaction file, respectively. The objects are organization-wide common services to all application systems that require access to these files. For example, an on-line application system will place transaction records into Trans_file during the daily inventory processing.

The Mgmt_report object can be an instantiation of an object-type that standardizes report formats and operations in the organization or it can be developed from scratch for this application. If it is newly developed, then the object becomes a

Figure 3 Inventory system box structure usage hierarchy



system-wide object for use throughout the INVENTORY system. If the encapsulated management report is to be used further in other system applications, then the Mgmt_report object can be designed to become an organization-wide object.

Inventory and Update_master are objects unique to the INVENTORY application. While the final designs of Inventory and Update_master encapsulate no persistent data (all persistent data are in the common services), the analysis and design of these objects provide the insights and the creative opportunities to perform the necessary object decomposition and composition for this system. This example also demonstrates the ability to design objects within objects since Update_master is wholly contained within the Inventory object.

Summary

Our goals in this paper have been to discuss and demonstrate the use of box structures in a rigorous and systematic object-based systems development process. Box structures provide a bridge between structured development methods and object-oriented development methods. The following observations support and summarize our discussion.

- Box structures provide for the definition of data abstractions and objects in three mathematical views.
- The box structure usage hierarchy allows intellectual control over the development process.

Each box structure in the system usage hierarchy is an object.

- All design inventions are separated into clearly identified small steps. Design verification is performed after each inventive step of design and provides a systematic basis for inspection.
- An object is stored in the system repository in all box structure views, from black box requirement to clear box detailed design. The object is described in an inheritance hierarchy as a common service.
- Box structures support an integrated development process, in that there is no need to transform the representation or content of development information from one phase to another.
- The systems development process is completely flexible between development phases. The next phase to be performed is based upon feedback from previous work results. The development of a system box structure usage hierarchy provides a discipline of sound and complete design.

Future research will expand upon the critical issues in this development process. We are currently performing research in three areas:

- *Requirements definition*—The process of eliciting requirements and representing system requirements in box structures needs important new research.[38] While the goal of requirements definition is to place all requirements in abstract black boxes, there are often essential requirements on data, procedure, and nonfunctional requirements, such as system performance.
- *Concurrent and real-time systems*—Current box structure theory supports the design and verification of sequential systems. Our recent research has provided extensions of box structures to the design and verification of concurrent systems.[29] Much more research is needed, however, to handle all of the complexities of real-time systems development.
- *Integrated CASE*—An eventual goal of this research is to design and build a comprehensive CASE system that provides integrated support of object-oriented development from requirements definition through system implementation. Our current research focuses on the representations of box structure information in common system development repositories.[39]

Cited references and note

1. E. Yourdon, *Modern Structured Analysis*, Yourdon Press, Prentice-Hall, Inc., Englewood Cliffs, NJ (1989).
2. J. Cameron, *JSP & JSD: The Jackson Approach to Software Development*, IEEE Computer Society Press, Washington, D.C. (1989).
3. J. Martin, *Information Engineering: Book 1—Introduction; Book 2—Planning and Analysis; Book 3—Design and Construction*, Prentice-Hall, Inc., Englewood Cliffs, NJ (1989).
4. S. Bailin, "An Object-Oriented Requirements Specification Method," *Communications of the ACM* **32**, No. 5, 608–623 (May 1989).
5. P. Ward, "How to Integrate Object Orientation with Structured Analysis and Design," *Software* **6**, No. 2, 74–82 (March 1989).
6. P. Coad and E. Yourdon, *Object-Oriented Analysis*, Second Edition, Prentice-Hall, Inc., Englewood Cliffs, NJ (1991).
7. S. Shlaer and S. Mellor, *Object-Oriented Systems Analysis*, Prentice-Hall, Englewood Cliffs, NJ (1988).
8. G. Booch, *Object-Oriented Design with Applications*, Benjamin/Cummings Publishing Co., Redwood City, CA (1991).
9. E. Seidewitz and M. Stark, "Towards a General Object-Oriented Software Development Methodology," *Proceedings of the 1st International Conference on Ada Programming Language Applications for the NASA Space Station*, D.4.6.1-D.4.6.14 (1986).
10. B. Meyer, *Object-Oriented Software Construction*, Second Edition, Prentice-Hall, Englewood Cliffs, NJ (1991).
11. P. Coad and E. Yourdon, *Object-Oriented Design*, Prentice-Hall, Inc., Englewood Cliffs, NJ (1991).
12. L. Cardelli and P. Wegner, "On Understanding Types, Data Abstraction, and Polymorphism," *ACM Computing Surveys* **17**, No. 4, 471–522 (December 1985).
13. B. Henderson-Sellers and J. Edwards, "The Object-Oriented Systems Life Cycle," *Communications of the ACM* **33**, No. 9, 142–159 (September 1990).
14. J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen, *Object-Oriented Modeling and Design*, Prentice-Hall, Inc., Englewood Cliffs, NJ (1991).
15. R. Wirfs-Brock, B. Wilkerson, and L. Wiener, *Designing Object-Oriented Software*, Prentice-Hall, Inc., Englewood Cliffs, NJ (1990).
16. H. Mills, "Stepwise Refinement and Verification in Box-Structured Systems," *Computer* **21**, No. 6, 23–36 (June 1988).
17. H. Mills, R. Linger, and A. Hevner, *Principles of Information Systems Analysis and Design*, Academic Press, Inc., Orlando, FL (1986).
18. H. Mills, R. Linger, and A. Hevner, "Box Structured Information Systems Development," *IBM Systems Journal* **26**, No. 4, 395–413 (1987).
19. R. Cobb and H. Mills, "Engineering Software under Statistical Quality Control," *Software* **7**, No. 6, 44–54 (November 1990).
20. R. Linger, H. Mills, and B. Witt, *Structured Programming: Theory and Practice*, Addison-Wesley Publishing Co., Reading, MA (1979).
21. D. Parnas, "On a 'Buzzword' Hierarchical Structure," *Proceedings of the IFIP Congress 1974*, North-Holland Publishing Co., Amsterdam (1974).
22. A. Hevner and R. Linger, "A Method for Data Re-Engineering in Structured Programs, *Proceedings of the*

22nd Annual Hawaii International Conference on System Sciences, Volume 1—Software Track, IEEE Computer Society Press, Los Alamitos, CA (January 1989), pp. 1025–1034.

23. B. Liskov and S. Zilles, "An Introduction to Formal Specification of Data Abstractions," *Current Trends in Programming Methodology: Software Specification and Design*, Vol. 1, Prentice-Hall, Inc., Englewood Cliffs, NJ (1977).
24. S. Danforth and C. Tomlinson, "Type Theories and Object-Oriented Programming," *ACM Computing Surveys* **20**, No. 1, 29–72 (March 1988).
25. H. Abelson and G. Sussman, *Structure and Interpretation of Computer Programs*, The MIT Press, Cambridge, MA (1985).
26. P. Hausler, R. Linger, M. Pleszkoch, and A. Hevner, "Using Function Abstraction to Understand Program Behavior," *Software* **7**, No. 1, 55–63 (January 1990).
27. This description of an object inheritance hierarchy is similar to the use of the Ada generic structure. Thus, we use the term "object-based" to describe the box-structured methods of systems development with objects.
28. P. Jalote, "Functional Refinement and Nested Objects for Object-Oriented Design," *IEEE Transactions on Software Engineering* **15**, No. 3, 264–270 (March 1989).
29. S. Becker and A. Hevner, "Concurrent System Design with Box Structures," *Proceedings of the 13th Annual International Computer Software and Applications Conference (COMPSAC)*, IEEE Computer Society Press, Washington, D.C. (September 1989), pp. 32–40.
30. *Object-Oriented Concepts, Databases, and Applications*, W. Kim and F. Lochovsky, Editors, Addison-Wesley Publishing Co., Reading, MA (1989).
31. J. Hughes, *Object-Oriented Databases*, Prentice-Hall International Series in Computer Science, Hartfordshire, England (1991).
32. B. Meyer, "Reusability: The Case for Object-Oriented Design," *Software* (March 1987).
33. T. Biggerstaff and A. Perlis, *Software Reusability: Vol. 1—Concepts and Models, Vol. 2—Applications and Experience*, Addison-Wesley Publishing Co., Reading, MA (1989).
34. A. Goldberg and D. Robson, *Smalltalk-80: The Language and Its Implementation*, Addison-Wesley Publishing Co., Reading, MA (1983).
35. *Ada for Specification: Possibilities and Limitations*, S. Goldsack, Editor, Cambridge University Press, Cambridge, England (1985).
36. D. Fetzer and J. Poore, "Using Box Structures with the Z Notation," *Proceedings of the 25th Annual Hawaii International Conference on System Sciences, Vol. II—Software Technology Track*, IEEE Computer Society Press, Los Alamitos, CA (January 1992).
37. E. Chikofsky and J. Cross, "Reverse Engineering and Design Recovery: A Taxonomy," *Software* **7**, No. 1, 13–17 (January 1990).
38. A. Hevner, "Box Structured Requirements Determination Methods," *Proceedings of the First Workshop on Information Technologies & Systems (WITS-91)*, MIT Sloan School of Management, Cambridge, MA (December 1991).
39. A. Hevner, S. Becker, and L. Pedowitz, "Integrated CASE for Cleanroom Development," *Software* **9**, No. 2, 69–76 (March 1992).

Accepted for publication August 17, 1992.

Alan R. Hevner *College of Business and Management, Management and Public Affairs Building, University of Maryland, College Park, Maryland 20742.* Dr. Hevner is an associate professor and chairman of the Information Systems Department at the University of Maryland. He is a faculty member of the Institute of Systems Research at Maryland. He has published over 50 refereed papers in the research areas of distributed database systems, information systems development, and systems engineering. Dr. Hevner is a member of ACM, the IEEE Computer Society, and the Operations Research Society of America (ORSA).

Harlan D. Mills *Computer Science Department, Florida Institute of Technology, Melbourne, Florida 32901.* Dr. Mills is a professor of computer science at the Florida Institute of Technology and President of Software Engineering Technology. He has written or coauthored six books and over 50 refereed technical journal articles on topics related to software engineering. Dr. Mills received a Ph.D. in mathematics from Iowa State University. He is an honorary Fellow of Wesleyan University and a Fellow of IBM, the American Computer Programming Association, and the IEEE. He also holds the Warner Prize for contributions to computer science.

Reprint Order No. G321-5511.