On Bridging the Analysis-Design and Structure-Behavior Grand Canyons with Object Paradigms

SINCE THE LATE 1970s, software developers have faced two "Grand Canyons."[1] The first chasm is the process-data or behavior-structure, represented by data flow diagrams (DFDs)[2] on the one hand and entity-relationship diagrams (ERDs)[3] on the other. In 1978, DeMarco noted that DFDs were not strong enough for data held over time. His solution was introducing an ERD as a second type of diagram that emphasizes data rather than processes. The second, more general chasm has been between analysis and design. For many years, professionals have been stymied by the underlying representation shift that accompanied the transition from analysis to design. This major shift prevented designers from systematically adding design-dependent details to the analysis results. Relating to the issue of different representations for different parts of information system development, Martin and Odell[4] point out that there really are three views to the world: the analyst view, the designer view, and the programmer view. Object-oriented (O-O) methods that have evolved in recent years have attempted at improving this situation. While they contribute towards narrowing the gap between analysis and design, this gap still exists. Many O-O approaches address the problem of bridging the gap between analysis and design. For example, the Object Modeling Technique (OMT)[5] proposes a smooth transition between consecutive phases of system development. While the analysis-design gap has been narrowed through the use of O-O techniques, no serious attempt has been made to close the process-data gap. Moreover, this structure-behavior gap seems to widen as attempts to bridge the analysis-design gap are starting to bear fruit.

In the first part of this article we survey and compare several O-O methodologies from the point of view of how they treat the structural and behavioral aspects of the system under construction, and how they propose to carry out the analysis results into the design phase.

In the second part we elaborate on how the object-process paradigm[6] integrates structure and behavior by providing for the possibility of defining high-level processes as entities that are not necessarily encapsulated within objects.

Dov Dori
Technion, Israel Institute of Technology

Moshe Goodman
Technion, Israel Institute of Technology

DELINEATING THE BORDER BETWEEN ANALYSIS AND DESIGN

The border between the analysis stage and the design stage is often fuzzy. Coad and Yourdon[1] propose a general rule from the analyst and designer points of view. While the systems analyst is concerned explicitly with the user's world, the problem domain, and the system responsibilities, the designer is concerned with translating and adapting the analysis results into a particular hardware/software implementation. The design should consist solely of expanding the analysis results to account for the complexities introduced in selecting a particular implementation.

According to Rumbaugh et al.,[7] the distinction between analysis and design may at times seem arbitrary and confusing. They define simple rules for differentiating between the two stages. The analysis model should include information that is meaningful from a real-world perspective and should present the external view of the system. In contrast, the design model is driven by relevance to the computer implementation. In other words, *analysis* is understanding a problem, *design* is devising a strategy to solve the problem, and *implementation* is building the solution in a particular medium. With Rumbaugh's OMT method, there is no clear-cut border between the analysis and design phases. Instead, information is gradually added to the analysis model to form the design model.

Jacobson et al.[8] claim that there is no uniformly applicable answer to the question. On the one hand, we want to do as much work as possible in the analysis stage, where we can focus on essentials, but on the other, we do not want to be committed to concepts that may potentially be changed when adapted to the implementation environment. They conclude that the time for the transition between the stages should be decided separately for each application.

Martin and Odell[4] emphasize

the advantage of O-O methodologies over the traditional ones. While in the traditional development methods analysts, designers, and programmers spoke three different languages (analysts used ERDs; designers, DFDs; and programmers, COBOL, C, etc.), with O-O techniques developers at all stages, as well as the end users, use the same conceptual model. The transition from analysis to design becomes so natural that specifying when analysis ends and design begins is sometimes difficult.

Overall, the various O-O methods have indeed contributed to narrowing the analysis-design gap. However, the situation with the structure-behavior gap is much less favorable, as the survey in the next section shows.

LINKING LIFECYCLE PHASES IN CURRENT O-O APPROACHES

Support of the entire development cycle of an information system by a single methodology has been addressed in the literature, although such discussions are sparse. In this section we survey a number of O-O methodologies from two perspectives: (1) the way they handle the various aspects of the system during analysis and (2) the way they handle the transition from analysis to design.

Object Modeling Technique

OMT incorporates several concepts to obtain high granularity for expressing different modeling situations. OMT combines the following three models that relate to each other, each with its own graphical tool, written to its right:

- object model: *object model diagram*
- dynamic model: *state diagram* + *global event flow diagram*
- functional model: *data flow diagram* + constraints

These three models are used as part of the analysis procedure. Each one of the three OMT models contributes to system development. The object model contains most of the declarative structure, the dynamic model specifies the high-level control strategy for the system, and the functional model captures functionality of objects that must be incorporated into methods. An example of using OMT was shown in two recent columns by Rumbaugh.[9,10] The example concerns building a simulation for the evolution of "bugs." The simulation deals with a two-dimensional world, containing only bugs and bacteria. Bugs eat bacteria, split, and move in the world, while bacteria are like "manna from heaven" that appear randomly at different locations. In the analysis stage,[9] the objects comprising the system are identified and displayed in an object diagram. A state diagram is drawn for each object, and operations are shown in a DFD. To cope with the combinatorial explosion expected in a flat state diagram, Rumbaugh et al. use *statecharts.* Statecharts are a broad extension of the conventional state machines and state diagrams introduced by Harel.[11] Statecharts expand conventional state diagrams with three elements dealing with hierarchy, concurrency, and communication. In the design phase[10] the analysis model is expanded into the design model. The main focus at this stage is the expansion and realization of operations (from the dynamic and functional models) as methods of object classes, together with clarifications and modifications to the object diagram. These methods are presented as pseudocode. The object model is updated to make it amenable for efficient implementation without changing its fundamental meaning. This is achieved using transformations that include clarification of attributes, elimination of unnecessary information, and modification of associations for optimization purposes. In the implementation phase[10] the results of the design are mapped into a specific programming language. Being an O-O approach, the object model of OMT is particularly suitable for developing applications that are basically static in nature. The dynamic and functional models handle system aspects beyond structure. Comprehensive understanding of the system under consideration through OMT remains a complicated task, since each one of the three models has a different perspective and a separate graphic representation. This leaves the burden of integration among the various system aspects to the developer.

Object-Oriented Software Engineering

O-O Software Engineering (OOSE), presented by Jacobson et al.,[8] is a use case–driven approach. OOSE defines the following five models for the various states of software development:

1. **Requirements model:** The requirements model defines the limitations of the system and specifies its behavior. It consists of a use-case model, interface descriptions, and a problem domain model. The use-case model uses actors and use cases. These concepts are an aid in defining what exists outside the system (actors) and what should be performed by the system (use cases).
2. **Analysis model:** The analysis model aims at structuring the system independently of the actual implementation. At this stage, the objective is to capture information, behavior, and presentation of the system. The object types used in the analysis are entity objects, interface objects, and control objects, while in most other O-O analysis (OOA) methods, only one object type is used, typically entity objects. The introduction of control objects is motivated by the need to handle the dynamic model of a system, which cannot be performed satisfactorily using an approach that is exclusively O-O.
3. **Design model:** The main effort in developing the design model is to adapt the analysis model to an actual implementation environment. The design model is regarded as a formalization of the analysis model, where the latter is adapted to fit into the implementation environment. In the first phase of the design, each object in the analysis is mapped into a design block. Since the design model is an abstraction of the actual system, it should reflect how the implementation environment has affected construction. The goal here is to maintain the structure, as defined in the analysis model, while reflecting the semantics of the objects in the corresponding design blocks. Great care must be exercised in changing, adding, or deleting blocks during the design. In the second phase of the design the interaction between the objects is defined. This is done by the *interaction diagram,* and it is in this phase that use cases play a major role. The interaction diagram is expected to clarify the requirements from each object. These requirements specify the necessary interfaces among the blocks. The interfaces then serve as preliminary definitions of function prototypes, needed for the implementation stage. After the interface of each block is defined, the internal structure of the blocks can be outlined. A *state transition graph* is drawn to provide a simplified description for each block.

continued on page 31

continued from page 26

4. **Implementation model:** The implementation model consists of the annotated source code. The basis for the implementation is the design model. Once *state transition graphs* exist for every block, the implementation in a specific programming language describes how terms and properties used in design are translated into terms and properties in the implementation language. Examples of implementation rules in different languages are given.

5. **Test model:** The test model is developed for testing the system. Testing is done at a number of different levels of granularity. The first step is testing the lower levels, such as object modules and blocks. These are tested by the designers. Units at the subsystem level are then tested. The integration test does not come in a "big bang." Rather, it is introduced on varying levels when integrating parts of increasing magnitude.

Object-Oriented Analysis

The OOA method, presented by Coad and Yourdon,[12] consists of five layers: subject, class and object, structure, attribute, and service. These five layers correspond to the following five major OOA activities applied in this order:

- **finding classes and objects:** identifying objects and object classes that constitute the basis for the application
- **identifying structures:** determining generalization-specialization and whole-part relations that induce inheritance and reflect composition of objects, respectively
- **defining subjects:** aggregating objects into groups with related semantics, called subjects, that can be used as modules of the system under construction
- **defining attributes:** identifying the set of attributes of each object class
- **defining services:** identifying the set of services (methods) for each object class and the message connections among the classes

The first four activities model the structure of the system, while the last activity handles the dynamic aspect of the system. Each service is described by a *service chart*—an enhanced flow chart. Objects interact through message passing. The OOA approach prescribes a method of identifying the objects in the system. Hence, OOA is an effective tool for analyzing systems that are basically static in nature.

Object-oriented design (OOD)[1] comprises four major components: problem domain, human interaction, task management, and data management. Each component corresponds to a particular design activity, in which each one of the five analysis layers should be handled.

When designing the problem domain component, objects defined in the analysis stage should be transformed into design objects. The notation used in the two stages is identical. The dynamic parts of the system at this stage are also handled through message passing. However, when the system is dynamic in nature it features many services, in which case the web of messages passed among objects becomes intractable. The authors do not explicitly refer to the design of the dynamic aspects of the system that are not reflected by message passing. Due to lack of adequate tools, such aspects are also hard to represent in the analysis phase.

Designing of the human interaction component includes capturing how a human user will command the system and how the system will present information to the user. Analysts study people to get the context and content right during OOA. Designers need to continue to study people, this time designing the interaction specifics, using interaction technologies available for a particular system.

For certain applications, tasks simplify the overall design and code. Separate tasks separate behaviors that must go on concurrently. This concurrent behavior may be implemented on separate processors or may be simulated when a single processor is used in conjunction with a multitasking operating system. Task selection and definition are presented during the design of the task management component.

Large systems must usually include some method of saving and restoring data overtime—a database. The design of the database and its interface with the rest of the system are addressed during the data management component.

Object-Oriented Systems Analysis

Object-Oriented Systems Analysis (OSA), developed by Embley et al.,[13] is defined as an approach for capturing and organizing information pertinent to the design and implementation of a software system. OSA is a "model-driven approach," implying that there is no predetermined set of steps to carry out the analysis. Instead, the analysis comprises a collection of different models. The ability to work concurrently with these models is a basic concept in this model-driven approach. A model-driven approach is expected to be more suitable for system analysis than a method-driven approach because analysis cannot usually be prescribed as a given set of steps. OSA consists of three major parts, each using a different model. The object-relationship model (ORM) is used for modeling the objects in the system and the structural relations among them. The object-behavior model introduces the concept of states. Every object in a system can be in one of many states. These states and the conditions for switching among them are defined. Each object found in the ORM is "exploded" into a set of states, conditions (triggers) for changing these states, and the action(s) that should be performed as a result of each change, which may include interaction with different objects. The object-interaction model models the interaction among the objects. The three models relate to a variety of cases that may occur within a system: real-time requirements, multiple inheritance, time-constrained interactions, etc. Each one of the three models can also be represented at various levels of abstraction. Design and implementation are not addressed by OSA, but OSA lays a foundation for O-O systems design and O-O systems implementation.

Object-Oriented Analysis and Design

Martin and Odell[4] divide the analysis into two parts: object structure analysis (OSA) and object behavior analysis (OBA). These two activities should be performed together to form an integrated model of the system. For the OSA, Martin and Odell present the *object-relationship diagram*. As stated by the authors, this diagram is essentially the same as an ERD.[3] The following information is identified during OSA:

- What are the object types and how are they associated?
- How are the object types organized into supertypes and subtypes?
- What is the composition of complex objects?

While OSA is concerned with the static aspect of the system, OBA handles system dynamics. During OBA, the following information is obtained for each object:

- What states can the objects be in?
- What state transitions occur?

The resulting states and transitions among them are drawn in *state transition diagrams,* which address the following questions:

- What events occur?
- What operations take place?

Operations and events are drawn in an *event schema* referring to the following questions.

- What interactions occur among objects?
- What trigger rules are used to react to each event?
- How are the operations represented in methods?

In the state transition diagram, the states of each object, the transitions between these states, and the events that change the state of an object are defined. For each object, a *fence diagram (state transition diagram)* is drawn to model the object lifecycle. In the fence diagram, all object states are shown and the possible transitions between the states are modeled as arrows, pointing from one state to the next. Event schemata are drawn to show the events, the sequence in which they occur, and how they change the state of objects. Interaction among objects, operations (also known as services or methods), and external sources of events are defined. Hierarchical schemata can be used in the behavioral part of the analysis for simplifying the representation of complex operations, using different levels of abstraction.

Recognizing the difficulty involved in integrating these two separate analysis tools. Martin and Odell offer yet another diagram—the object-flow diagram (OFD)—which is used to represent the system at the strategic level. OFDs are similar to DFDs.[2] Both depict activities interfacing with other activities, but while in DFDs only data flows among activities, OFDs enable objects to be passed from one activity to the next. The OFD indicates the objects that are produced and the activities that produce and exchange them. To model the dependency between processes, the *process-dependency diagram* is drawn. The result of the analysis, according to Martin and Odell, is four sets of diagrams: the event schema, the process dependency diagram, the OFD, and the state-transition (fence) diagram. These four types of diagrams together constitute the *activity schema.* The interaction among the different diagrams is not quite clear. A way to connect the event schema with the object schema is suggested by drawing arrows from the event schema to the objects in the object schema that are affected by the event. The result, as illustrated on a small example, seems cumbersome.

OOD has two aspects: object structure design (OSD) and object behavior design (OBD). These two aspects are a continuation of the different aspects in analysis (OSA and OBA, respectively). However, since OOPLs encapsulate data structures and methods into classes, both OSD and OBD are intertwined and are addressed concurrently. In the design phase, the following information is identified:

- What classes will be implemented?
- What data structures will each class employ?
- What operations will each class offer and what will their methods be?
- How will class inheritance be implemented and how will it affect the data and operation specifications?

Martin and Odell present ways of mapping the elements of object and event schemata to OOPL constructs and methods, respectively. Design is conceived as a direct continuation of the analysis, and hence the gap between structure and behavior, introduced during analysis, is carried on to the design phase.

Object Lifecycles

Shlaer and Mellor[14] propose the object lifecycle approach for analyzing information systems. They propose to perform OOA in three steps, using the following three models.

Information models. The first step focuses on abstracting the world as a collection of objects, their attributes, and the relationships among the different objects that make up the system. The relationships are based on policies, rules, and physical laws that prevail in the real world. The resulting information model diagram is similar to the familiar ERD.[1]

State models. The second step is concerned with the behavior of objects and the relationships among them over time. Each object has a lifecycle—an orderly pattern of dynamic behavior. State models formalize these lifecycles. A *state transition diagram,* similar to a flow chart, is used to represent the different object states and the transitions among them. The diagram includes actions performed when an object is in a specific state and events that cause an object to change its state. Another diagram, the *object communication diagram,* is used for modeling the interaction among different objects.

Process models. All the processing required by the problem is contained in the actions of the state models. Each action is defined in terms of processes and object data stores, where a process is a fundamental unit of operation and an object data store corresponds to the data (attributes) of an object in the information model. Each action is displayed graphically in an *action data flow diagram* (ADFD), which is similar to a DFD, except that like Martin and Odell's DFD it also allows for the flow of objects. A separate ADFD is produced for each action in the state model.

Shlaer and Mellor offer a detailed and systematic method for transforming analysis results into OOD. They explicitly show how their analysis model can be directly mapped into a detailed design. OODLE is proposed as a language-independent notation for their OOD. The notation has been named OODLE, an acronym for O-O Design Language. The objective of OODLE is to represent the fundamental concepts of OOD in an intuitive manner. In OODLE notation, another set of diagrams has been developed to describe four significant aspects of design:

- *class diagrams* that depict the external view of each class
- *class structure charts* that show the internal structure of the code of the operations of each class

- *dependency diagrams* that depict the client/server and friend relationships that hold between classes
- *inheritance diagrams* that show the inheritance relationships among the classes

OOD can be directly derived from the models of OOA. The design of the system as a whole is expressed in terms of the design as a single program. Each program is made up of a main program, four architectural classes, and a number of application classes. The main program is responsible for intertask communication and the invocation of operations of the application classes to initiate threads of control. Three architectural classes—Finite State Model, Transition, and Active Instance—supply mechanisms required to initialize and traverse state machines. The fourth architectural class, Timer, provides a mechanism analogous to the Timer object of OOA. The application classes are analogous to, and derived from, the objects and state models of OOA. Each one is responsible for the same activities carried out by its OOA analog. For each of the classes—architectural and applications—a class structure chart (presented as part of OODLE) is used to show the structure of the code, as well as the flow of data and control within the class. During implementation, the modules of the archetype class structure charts are coded in the particular language of choice.

OBJECT-ORIENTED ANALYSIS AND DESIGN METHODOLOGIES: A COMPARISON

Table 1 summarizes the preceding survey by comparing the six O-O methodologies (and the object-process approach discussed below) from two perspectives: (1) What structure, behavior, and additional models does each methodology have within its analysis phase? (2) How is the transition from analysis to design supposed to proceed?

THE OBJECT-PROCESS ANALYSIS APPROACH

Object-process analysis (OPA)[6] combines ideas from OOA and DFDs to model both the structural and procedural aspects of a system in one coherent frame of reference. The structure-behavior model unification distinguishes the object-process paradigm from conventional O-O approaches including the ones surveyed above. In OPA, high-level processes have their own right of existence rather than being associated as services of objects. The detachment of processes from objects is a direct consequence of the observation that in many cases, nontrivial processes cannot be uniquely associated with a single object, or class of objects. The O-O approach heralds encapsulation—the packaging of procedures ("services" or "methods")—and enforces the attachment of each procedure to one particular object class. In complex systems, though, it is often the case that a process can occur only through the activation of more than one object. In such cases, the choice of which object to attach the service to is bound to be arbitrary, as

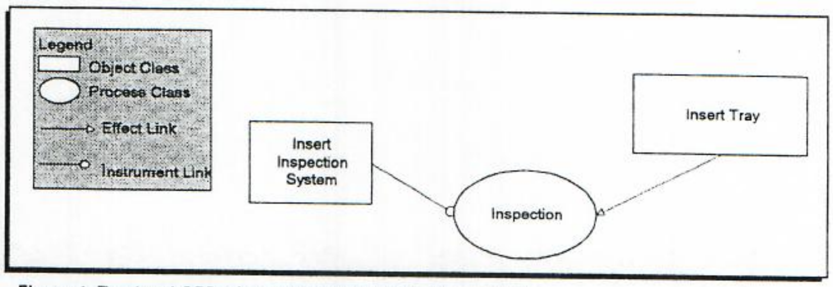

Figure 1. Top-level OPD of the Insert Defect Detection system.

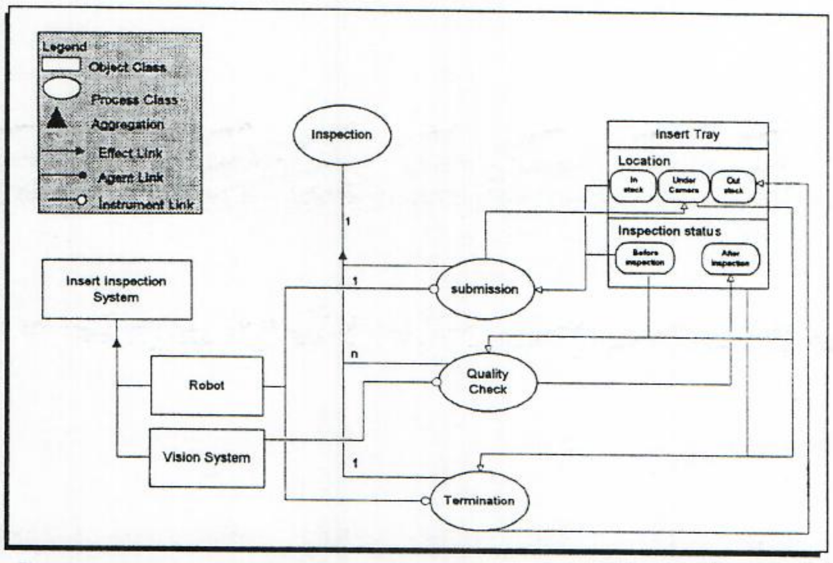

Figure 2. Unfolding of the process "Inspection" and the object "Insert Inspection System" of Figure 1.

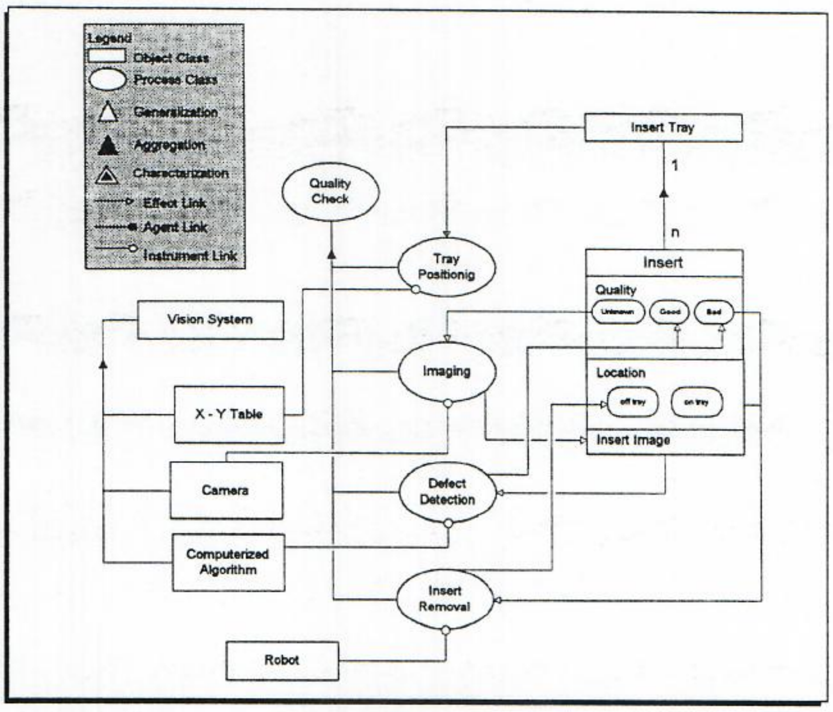

Figure 3. Unfolding of the process "Quality Check" and the object "Vision System" of Figure 2.

		Analysis Models				
Name	Main author	Structure model	Behavioral model	Additional model(s)	Transition to design	Comments
Object Modeling Technique (OMT)	Rumbaugh et al.[7] 1991	Object model	Dynamic model	Functional model	Expansion of the analysis model	
Object-Oriented Software Engineering (OOSE)	Jacobson et al.[8] 1992	Entity objects	Control objects	Interface objects	Adaptation and formalization of the analysis model	Includes requirement, test, and implementation models
Object-Oriented Analysis (OOA)	Coad & Yourdon[1,12] 1991	Objects with attributes	Services and message passing among objects	None	Design of the static aspect only	
Object-Oriented Systems Analysis (OSA)	Embley et al.[13] 1992	Object relationship model (ORM)	Object behavior model (OBM)	Object interaction model	Not addressed	A model-driven approach
Object-Oriented Analysis & Design	Martin & Odell[4] 1992	Object structure analysis (OSA)	Object behavior analysis (OBA)	Object flow diagram (OFD)	Supposed to be a direct mapping of analysis	
Object Lifecycles	Shlaer & Mellor[14] 1992	Information models	Process & state models	None	A direct mapping of analysis using OODLE	
Object Process (OP)	Dori[6] 1995	Object-process		None	Under development	Unified structure-behavior analysis model

Table 1. Summary of analysis models and transition to design within the various O-O methodologies.

it is impossible to pinpoint one particular object that is solely "responsible" for the process. The mechanism of message passing among objects is the way O-O methodologies provide for interaction among objects. This mechanism frequently yields unintuitive awkward modeling. In OPA, processes are not confined to be services or methods of any particular object. Moreover, the effect of an OPA process is to change the state of at least one object. This makes object orientation's mechanism of message passing among objects unnecessary. Avoiding message passing prevents the difficulty this mechanism potentially introduces. This, in turn, enhances the expressive power of OPA.

An important feature of objects and processes in OPA is that they are recursively scalable. Scalability provides for complexity management of systems through controlling the visibility and level of detail of things in the system. Scaling is a process of changing the level of detail of a thing (object or process). Scaling up is a process of increasing the level of detail of a thing. Scaling down is a process of decreasing the level of detail of a thing.

To demonstrate some of OPA's features we provide an example of a system analyzed by OPA. In an industrial process, "inserts" for metal-cutting tools are manufactured. The inserts are then sent to an automated quality check, one tray of inserts at a time, and defected inserts are removed from the tray. A robot takes one tray at a time and positions it on a special "x-y table" under the camera. The table moves so as to allow the camera to focus on one insert at a time. The image taken by the camera is passed to a defect-detection algorithm, and defected inserts are removed from the tray by the robot.

Figure 1 is a top-level object-process diagram (OPD) showing two objects and one process. The inspection process affects (changes the state of) the object Insert Tray through the instrument Insert Inspection System.

In the OPD of Figure 2, the Inspection process and the Insert Inspection System object are unfolded to expose their constituents. Inspection comprises submission, Quality Check, and Termination. The inspection system is made up of a robot, which is the instrument for the submission and termination processes, and a vision system, which is the instrument for the quality check process. The object Insert Tray is expanded to show that it is characterized by the attributes location (with possible values "in stack", "under camera" and "out stack") and inspection status ("before inspection" and "after inspection"). Submission affects a tray in "in stack" location and "before inspection". This process changes the location (and hence the state, which is the vector of attribute values) to "under camera". Likewise, the quality check process changes the inspection status to "after inspection", while termination changes the location to "out stack."

An additional level of up-scaling is introduced in the OPD of Figure 3, where Vision System and Quality Check are further unfolded, and Insert Tray is further expanded. At this level of detail we can trace what happens to a particular insert if it is found to be defective. Two major features are demonstrated in this example:

1. OPA integrates system structure (objects and their aggregation and generalization) and behavior (processes and how they affect object states) within one diagramming methodology.
2. OPA is capable of recursive scaling to show system details at any granularity level.

CONCLUSION

In all the O-O methodologies surveyed above, the analysis is dispersed among many models, depicted by a variety of diagram types. The multiplicity of diagram types is dragged into the design and implementation phases, yielding an application that is more complex, hard to understand and difficult to maintain. This has been the main motivation for developing the object-process (OP) paradigm as an extension of the object paradigm. OP departs from accepted O-O methodologies in the recognition of processes as parts of the analysis that are integrated within one graphical tool—the object-process diagram (OPD). We suggest that the analysis results obtained from OPA can potentially evolve into the design phase, while maintaining the advantages of unification and visibility control gained by using the OPA approach. Using a single tool for the different aspects of the system and throughout the various stages of the system development should make a significant contribution to information systems engineering by delivering systems that are more understandable than those obtained by the current multiple-models/multiple-diagrams O-O methodologies.

We intend to specifically define the activities involved in the transition from analysis to design and from design to implementation using the object-process methodology. A CASE tool for building software systems using the OPA paradigm is planned. The C++ programming language may require some extensions to support aspects of OPA that cannot be implemented directly using standard C++.

Acknowledgment

This research was supported by Technion V.P.P. Fund–B. and I. Green Research fund.

References

1. Coad, R. and E. Yourdon. Object-Oriented Design, Prentice Hall, Englewood Cliffs, NJ, 1991.
2. DeMarco, T. Structured Analysis and System Specification, Yourdon Press, New York, NY, 1978.
3. Chen, P.P. The entity relationship model—toward a unifying view of data, ACM Transactions on Data Base Systems 1:9–36, 1976.
4. Martin, J. and J. Odell. Object-Oriented Analysis & Design, Prentice Hall, Englewood Cliffs, NJ, 1992.
5. Rumbaugh, J. The life of an object model, Journal of Object-Oriented Programming 7(1):24–32, 1994.
6. Dori, D. Object-process analysis: Maintaining the balance between system structure and behavior, Journal of Logic and Computation 5(2):227–249, 1995.
7. Rumbaugh, J., M. Blaha, W. Premerlani, F. Eddy, and W. Lorenson. Object-Oriented Modeling and Design, Prentice Hall, Englewood Cliffs, NJ, 1991.
8. Jacobson, I., M. Christersson, P. Jonsson, and G.G. Overgaard. Object-Oriented Software Engineering, Addison-Wesley, Reading, MA, 1992.
9. Rumbaugh, J. The evolution of bugs and systems, Journal of Object-Oriented Programming 4(7):48–52, 1991.
10. Rumbaugh, J. Designing bugs and dueling methodologies, Journal of Object-Oriented Programming 4(8):50–56, 1992.
11. Harel, D. Statecharts: A visual formalism for complex systems, Science of Computer Programming 8:231–274, 1987.
12. Coad, R. and E. Yourdon. Object-Oriented Analysis, Prentice Hall, Englewood Cliffs, NJ, 1991.
13. Embley, D., B. Kurtz, and S. Woodfield. Object-Oriented Systems Analysis, Prentice Hall, Englewood Cliffs, NJ, 1992.
14. Shlaer, S. and S. Mellor. Object Life Cycles—Modeling the World in States, Prentice Hall, Englewood Cliffs, NJ, 1992.

Dr. Dov Dori is Head of the Information Systems Area of the Faculty of Industrial Engineering and Management, Technion, Israel Institute of Technology. Moshe Goodman is an Information Systems Engineer and MS candidate of Information Management Engineering at Technion. They may be contacted at dori@ie.technion.ac.il.