Abstract
Object Interaction Diagrams (OIDs) model dynamic behavior over a period of time by showing how system components interact to complete core tasks defined in use case modeling. While seemingly intuitive, diagram elements are not consistently defined, and methods for constructing an OID have not been described in the literature. Also there is a lack of understanding about how OIDs relate to each other and to other system diagrams.

The goal of the paper is to resolve these issues by systematically examining the structure and role of the OID in object-oriented (OO) systems design. First, we take a look at how prominent developers use OIDs in their designs. Next the structure of an OID is examined in detail, and recommendations for clarification are presented. Then a heuristic method for OID construction is described. An analysis of a set of library OIDs corroborates the heuristic by reproducing the profile of entity, control, and boundary object stereotypes described in the literature. The heuristic method represents a first effort in the field of OO design to describe how an OID should be constructed.

We also show that communicating OIDs are not systematically connected to each other. Relationships can be fully understood if OIDs are organized hierarchically with strategically positioned connector buttons that enable the viewer to both “drill down” to subordinate OIDs and to “roll up” to calling OIDs for a complete, detailed trace through a system. We conclude the paper by bringing together all of our recommendations for syntax changes into a single enhanced OID that will hopefully improve the quality of software design.

1. Background
Object Interaction Diagrams graphically show how system components communicate with each other to fulfill user requests. They are implicitly dynamic in nature, and can be visualized as a sequential execution trace of events through a system. Their inventor, Ivar Jacobson, originally developed them as extensions of the use case from his work with large telecommunications systems [4].

Other developers have incorporated OIDs into their models. Rumbaugh, for example, refers to them as event trace diagrams that describe a “sequence of events and the objects exchanging events … shown in an augmented scenario”. [7]. Rumbaugh does not explicitly state that the OID comes from the use case in his model, but Yourdon does [9], and in the Unified Modeling Language (UML) an OID or sequence diagram is simply defined as an instantiation of a use case [3].

2. Components of an Object Interaction Diagram
The template for an OID pictured in Figure 1 is read from the upper left-hand corner going downwards to the right. The last item reviewed is the text associated with the lowest arrow on the page. Based on time, the vertical axis has an ordinal, interval, or a rational scale. Time is the independent variable in an OID. The horizontal axis is strictly nominal containing the names of the object classes that are affected by the trace through the system.

![Figure 1. Object Interaction ~ Fence Format [9].](image)

2.1 Block Label
According to Yourdon object classes are displayed horizontally across the top of an Object Interaction Diagram [9]. Booch differs from Yourdon by saying that objects or instantiated classes are named along the horizontal axis of an OID [1]. In fact, Booch separates his diagrams by placing them into a static class or dynamic object category [2]. Jacobson tends to side with Booch but acknowledges Yourdon’s point of view when he says “a column most often represents an instance but might also represent a class” [5].

To fully understand a model, all components must be precisely defined. If we look at the OID as an abstraction of a system, then class might be the appropriate definition for Jacobson’s block. However, if we are looking at the model to dynamically trace a set of events through a system, then we are modeling instantiation, and one would think that block labels should reference objects. However, the traces, themselves also have different levels of abstraction meaning that multiple scenarios can be incorporated into a single trace.

In Object-Oriented Modeling and Design, Rumbaugh sheds light on the level of abstraction needed for a trace when he says “every event is a unique occurrence, but we group them into event classes and give each event class a name to
indicate common structure and behavior”[7]. Furthermore, Rumbaugh asserts that the number of classes in a system is finite whereas the number of scenarios and objects can grow to infinity [6]. Given the existence of a finite number of OIDs in a system, one would consider the matter closed in favor of the class definition. However, a reversal occurs in UML Toolkit where sequence diagrams are used to illustrate how objects interact with each other” [3] (italics added).

We propose to end the debate by extending Yourdon’s terminology to reference a new hybrid term object class. We need a notation that enables us to stand outside the class/object dichotomy, because we are modeling instantiation itself. Given our new construct, a single block label can reference both a class and an object as shown in Figure 2:

![ObjectClass #2 is a class prior to construction.](image)

Yourdon has also provided additional notation to distinguish between entity, control and interface (boundary) object classes in his OIDs [9].

- **Entity** object classes represent real-life objects or concepts that are internal to the system. External actors usually have no direct contact with them.
- **Interface** object classes handle the communication with external actors, and they encapsulate environmental-dependent behavior. (UML has substituted the word boundary for interface).
- **Control** object classes describe behavior that doesn’t belong to either entity or boundary objects such as the coordination of series of events among multiple objects.

UML Toolkit refers to these categories as stereotypes, and asserts that entity types can be persistent whereas controls are not [3].

### 2.2 Operations

Jacobson states that the text in the left-hand margin of the OID determines the length of the rectangle attached to the block bar [4]. It would appear from his description that the operation represented by the rectangle is a set of logically connected events that accomplish a sub-task in the use case.

The rectangular bars that appear in OIDs are also intended to show when an object class is active [10]. However, parallel bars can convey concurrency, because the vertical axis represents the passage of time.

When arrows are added to an OID, one would hope to get a better picture about which block is actually in control. However, arrows in OIDs are used to define a client/server relationship between blocks with the arrowhead positioned over the server [1]. Yourdon even deletes return arrows from his diagrams, because the viewer is to assume that servers automatically respond to client requests [9].

These problems are solved in UML Toolkit. First the term activated is substituted for active, and it refers to an object class that is “either executing its own code or is waiting for the return of another object to which it has sent a message”. UML then reinserts the return arrow into the OID, but gives it an identifying format so that it won’t be confused with a client request. Lastly, a third type of asynchronous arrow is used for denoting concurrency [3].

![Operations depicted as parallel bars better reflect the structure of an OID as an ordered set of subtasks where client/server relations can be tracked. However, parallel operations can be mistaken for concurrent processes.](image)

### 2.3 Descriptions

Descriptions appear three places in the OID. First, object classes are properly labeled using nouns such as borrower, or caller. The highest degree of abstraction should be captured in the block label to reduce the number of OIDs required for a system.

Secondly, messages with optional parameters are displayed above the trace arrows. Messages define server rather than client roles in a system. Since messages are defined as operations in the Class Diagram, OID operations are not the same as Class Diagram operations.

Finally, an abbreviated version of the use case description is placed to the left of the system border and at the same height as the related trace. The textual description can be written in structured English or in pseudo-code to accommodate iterative or conditional processing. Yourdon doesn’t use pseudo code, because he maintains that each trace requires its own OID [9].
3. Using a Heuristic for Developing an OID

Before presenting the eleven-step heuristic for constructing the *Check-In Materials* OID, the pre-existing use case diagram and transaction sequence should be reviewed:

![Figure 4. Use Case Diagram for Check-In Materials.](image)

Given the use case diagram and a screen-annotated use case description, the developer produces the *Check-In Materials* OID partially shown in Figure 3 by following the eleven-step heuristic described below:

1) Identify the actor and initiating external event from the use case description.
2) Begin with the major boundary object.
3) Identify the major display screens needed for implementing the use case.
4) Create one boundary object for each of the major screens.
5) Identify all object classes by reviewing the use case description. If any class identified from the use case description does not exist in the Object Structure Diagram, add it.
6) Use those object classes just defined as block labels in the OID.
7) Classify events from the use case description as:
   - Instance creation or destruction
   - Association forming
   - Attribute modification
   - Calculation
   - Interface with external objects or systems
   - Report generation
8) Order the sequence of messages among the object classes for implementing the use case.
9) Introduce a control stereotype to encapsulate a set of highly coupled, related messages.
10) Name each message and supply it with optional parameters.
11) Review the OID to see if subtasks map better to a stair or fork configuration.

The heuristic presented above is user-centric in that a special effort is made to identify the external initiating event from the use case diagram, and screen prototypes are translated into boundary object classes. The heuristic also promotes continuity and complete coverage by translating messages from the OID into operations in the Class Diagram and by directing the developer to make sure the OID and Class Diagram contain the same object classes. Control and boundary object classes are also transferred to the Class Diagram along with the dotted line paths in the Class Diagram that connect the various types of classes.

The heuristic is not designed to handle concurrency, branching or looping, but rather builds OIDs one at a time from the use case descriptions. By limiting the initiating event to external stimuli, concurrency cannot be managed, because object classes in a concurrent real-time system are frequently self-starters [3].

4. Collecting Information about Client/Server Roles

We collected information about clients and servers by polling all labeled messages in the OIDs from the library system mentioned earlier. The results summarized in Figure 5 underscore the fact that controls tend to be clients, entities are servers and boundary stereotypes play both roles.

We recommend that developers double-check control servers and entity clients for errors, since they are the exceptions to the trends established in Figure 5. The exceptions may be perfectly legitimate, however, since one entity may seek information from another entity as a type of join, or a control may act as a server when it is created.

![Summary Statistics Client/Server Roles](image)

5. Categories of OIDs

Jacobson categorizes OIDs by structure [4,]. The fork or centralized OID contains one primary client operation that controls the flow of signals to multiple server operations. (Arrows always point to servers). In contrast,
the stair or decentralized OID uses delegation as the primary means for structuring communications among multiple objects. More specifically, Jacobson describes a decentralized OID as one where “each object only knows a few of the other objects and knows which objects can help with a specific behavior. Here we have no ‘central’ object” [4]. Figure 6 below is a reproduction of Figure 8.15 that appears in Object-Oriented Software Engineering: A Use Case Driven Approach.

![Figure 7. Fork and Stair configurations are displayed. Control stereotypes always present as forks whereas communicating boundary stereotypes positioned next to each other look like the stair diagram in the Figure. Entities can be servers in either a fork or a stair diagram.](image)

Our diagram for Check-In Materials in Figure 3 highlights the stair format. Early on, the main menu delegates authority to the check-in display which goes through the Check In control object to transfer control to the Emp/Dept Profile Display. This delegation makes sense given the hierarchical arrangement of the window displays.

6. Relationships Among OIDs

Relationships among OIDs are determined at the use case level in systems design. Rumbaugh suggests that the first step in defining a use case should be to “group together all transactions that are ‘similar’ in nature, which a user would think of as being variations on a theme” [6]. To follow through on Rumbaugh’s metaphor, the theme becomes the standard path and the variations become alternative paths in use case descriptions. When the descriptions are translated into diagrams, only one theme is allotted per diagram. In addition, each OID in a system must represent a complete flow of events in a system [5].

With these stipulations in mind, use cases can be made complete by combining fragmentary subsequences with uses, and alternative paths can be accommodated with an extends combination of use cases.

Like a super-class, the base sequence in an OID is unaware of the existence of an extends sequence. Rumbaugh goes on to add that in the uses case, the subsequence is mandatory whereas it is optional for an extends case [6]. What he doesn’t say is that the uses case can either be fragmentary or complete, and that the same use case can play different roles in separate diagrams. With such flexibility the two-tier structure of primary and secondary use cases can quickly become very complicated.

While a probe position is inserted into an extends OID showing the conditions under which it is invoked, nothing appears inside the base or super-class OID itself. The absence of an actual invocation demonstrates that the base OID is unaware of any extension’s existence. Unfortunately, the developer is hard pressed to find a connection as well. Conversely, a uses OID is clearly marked with a dotted line when it is invoked [9]. But it too is limited, because once inside a uses OID the developer has no way of tracing the invocation.

We propose to overcome these limitations by inserting flowchart connector symbols into our OIDs that act like hotspots. When the viewer presses one of the solid disks in the diagram, he either “drills down” to a subsequent OID or “rolls up” to a prior one. Since uses and extends also reference participation constraints fundamental to database theory, a mandatory uses subsequence is marked by a solid line, and the optional extends sequences gets a dashed line.

7. Conclusion

In Figure 11 we bring together our recommendations for syntax changes into an enhanced OID. Hopefully complex software projects can be accurately developed in a shorter period of time with a more clearly defined syntax, detailed heuristic, and enhanced connector notation described in this paper.

![Figure 11. Enhanced OID with syntax changes. A shaded prior OID shows the OIDs that can invoke the current OID. Object classes previously instantiated are emboldened. Concurrent operations are also emboldened. Both time and connections to all types of OIDs are made explicit in this diagram. Flow chart connector symbols are used for drilling down to subsequent OIDs or rolling-up to prior OIDs.](image)

8. References

Due to the page limitation, references will only be supplied upon request.