A DECISION-BASED METHODOLOGY
FOR
OBJECT-ORIENTED DESIGN

Patrick D. Barnes and Thomas C. Hartrum
Department of Electrical and Computer Engineering
School of Engineering
Air Force Institute of Technology
Wright-Patterson AFB, Dayton, Ohio, 45433

Abstract
The task of object-oriented development raises a new set of design problems. Addressing the decisions which must be made in applying object-oriented principles to design is the focus of this paper. A structural object model is presented and the concepts of decision support systems (DSS) are applied to the formulation of a decision-based methodology for object-oriented design. An overview of the development of a decision aid for evolution of the methodology is also given.

1 Introduction
Escalation of software development and maintenance costs as well as demand for software solutions to increasingly complex problems have mandated new techniques for engineering reliable, maintainable computer software. One approach to improving software quality is the use of the object-oriented paradigm for design.

But there is more to design than the paradigm we choose for structuring, conceptualizing, or representing a system. The design process can be seen as combining...

...intuition and judgement based on experience in building similar entities, a set of principles and/or heuristics that guide the way in which the model evolves, a set of criteria that enables quality to be judged, and a process of iteration that ultimately leads to a final design representation.

This description indicates that a software design environment must support judgment and choice, embody design principles and/or heuristics, guide an iterative development process, and enable qualitative evaluation of the finished product. While several methodologies have been proposed for an object-oriented approach to design ([9], [7], [5], [6], [15], and [2]), they seem to focus primarily on the representation of the design rather than the process.

This paper presents an approach to developing an object-oriented design methodology based on the concepts of decision support systems. The OOD process is not redefined; rather it is stated in terms of the decisions a designer must make while accomplishing OOD tasks. First, a general object-oriented model for design is presented. The decisions involved in OOD are then stated and a methodology is elaborated based on those decisions. Finally, an overview of the first stage development of a decision aid is discussed.

2 An Object Model for Design
2.1 Defining the Model
Two models of the object-oriented paradigm were analyzed for application to design, a theoretical model [4] based on objects, behaviors, and attributes, and the Smalltalk language-based model [8] which adds class, inheritance, messages, and methods. The theoretical model proved to be too ambiguous to rigorously depict relationships between objects, and the OOP model too restrictive to implementation constructs. A new model was derived from the more abstract theoretical model by adding refinements derived from the Smalltalk experience to solve design related problems. The resulting object model is formally defined as follows:

object A unique entity defined by attributes which serve to identify the object, and relations which associate it with other objects, relations, and operations.

operation The description of how an object performs some behavior. As with objects, attributes serve to identify the operation and relations associate it with other objects and operations.

attribute Serves to identify an object or operation. Required attributes for objects are name, behavior, and domain. Required attributes for operations are name and algorithm.

1This work sponsored by the Strategic Defense Initiative Organization

U.S. Government work not protected by U.S. copyright.
relation A complex attribute representing an association of an object or operation with other system objects and operations. Relations on objects include its class as well as sets of operations, component objects, actor objects, and server objects. Relations on operations include its object as well as sets of modified objects, argument objects actor operations and server operations.

class A complete design of an object which may be used as a template from which another object derives its characteristic structure and function.

name A string serving to identify an object or operation which must be unique within a context.

behavior A text description of an object's function when provided with certain stimuli.

domain A text description of the set of states to which an object may change.

actors A relation which denotes which objects or operations require services of some other object and operation pair.

servers A relation which denotes which objects or operations provide services to some other object and operation pair.

components A relation which denotes the parent/child relationships between objects.

arguments A relation which denotes which objects are required as arguments in the interface of an operation. This relation has the attribute mode which may be either input or output.

modifies A relation which denotes which objects are modified by the execution of an operation.

This model retains the function of the theoretical model, and adds the practical aspects of the programming model. The implementation of an object is not specified, nor is the syntax of the communication between objects limited to a specific method. Yet provisions are made for describing the interface between objects and operations of other objects, as well as for representing the fully recursive nature of real world objects.

2.2 Representing The Model.
Software developers have produced a plethora of graphical methods for representing software systems. A number of techniques have been proposed to represent an object-oriented design. The main purpose of graphics is to communicate the design more clearly than does the text. While the use of graphics is strongly advocated, a methodology that allows the graphic techniques to drive the design can be counter productive. Many of the methodologies investigated were developed specifically for designing Ada programs. The graphical representation presented in this paper takes a more general approach.

A multi-view approach is suggested consisting of three parts: a block diagram, an interface (detail) diagram, and a control flow or state diagram. Figure 1 shows an example of these three views. The block diagram used is similar to the high level object diagram of [15]. It depicts the objects in the system and the dependency relationships between them. In the case of an actor/server relationship, messages or operation calls flow across the directed arrows. The detail diagram is a modification of the modular design chart [17]. Objects begin with a capital letter, and operations begin with lower case. A petri-net graph similar to the one found in APEX [2] is used to depict a state diagram or object interaction in the case of concurrent communicating objects.

3 Overview of the Methodology
This section describes the methods or steps to deriving a design using the object model. The methodology is based on providing rules or postulates (design heuristics) to support object-oriented design decision making.

The steps in the methodology were developed by identifying the decisions involved in OOD. Thus the OOD process is presented in terms of decisions rather than the usual set of products associated with the design specification.

The OOD process is pictured in the concept map in Figure 2. The decision steps highlighted in the figure are defined as follows:

1. Analyze the problem and requirements specification to decide on an initial scope and a strategy for its solution.

2. Identify the abstract objects, operations, and their attributes from the solution strategy and requirements specification; then decide which are central to the solution strategy.

3. Encapsulate the objects, operations, and attributes into modules and determine the relationships, or interfaces, between those modules. In other words, decide which operations naturally go with which objects.

4. Decompose complex modules by repeating the process with objects or operations as separate problems, or begin detail design. Detail design requires deciding whether to construct modules from known components such as other objects, library modules, predefined functions or data types; or to produce an algorithmic description using pseudocode or flow diagrams.
OOD is unique in respect to what needs to be identified in analyzing the problem, how data structures and algorithms are encapsulated into system modules, and in how system modules are constructed from known, more general data types or classes. However, it should be clear that the main thrust of the decisions discussed here are basic to software design—regardless of the paradigm involved.

The following sections provide a more detailed description of the decision steps in the methodology.

3.1 Analyze the Problem to Determine a Solution Strategy
The first decision the designer must make is in limiting the scope of the problem to be solved. The problem statement should be determined from the problem space and stated in user-oriented terminology. It is important for the designer to interact with the user whenever possible in accomplishing this step. Using the concept map [12] to elicit such problem-oriented information encourages this interaction and may communicate more effectively and ensure mutual understanding.

Concept maps should be developed from both the users and the requirements specification, then compared and refined to provide a better understanding and statement of the problem.

Summary of the Analysis Step.
1. Interview one or more users and develop concept
2. Develop additional concept maps from the system's functional requirements.

3. Synthesize from the concept maps a single sentence statement of the problem.

4. Develop a single concept map which depicts a strategy for solving the problem.

### 3.2 Identify Objects, Attributes, and Operations

Dave Bullman [10] states that finding the right objects is hard, and that associating operations with the right objects is even harder. A number of heuristics have been suggested for the identification of objects. This step consists of the application of such heuristics to identify and define the objects, attributes, and operations which apply within the scope and level of abstraction we are dealing with. Some valuable heuristics include the following:

- **Object Selection Criteria** lists general software engineering heuristics such as information hiding, abstraction and inheritance for determining good objects [19].

- **Grammatical Analysis** makes selections from nouns and verbs [1].

- **Abstraction Analysis** makes selections based on data flow diagrams [15].

- **Class Abstraction** makes selections based on classes of physical objects [11].

- **Concept Analysis** makes selections based on concept map entities and has the following steps:
  1. Generate a first cut list of objects from the entities on the concept map.
  2. Identify from the list of objects which are long-lived and which are transient. Transient objects tend to be operation arguments or local variables. Long-lived objects tend to represent...
abstract state machines.

3. Identify which objects are subordinate, natural components of, or clearly attributes of other objects.

4. Identify the action words in the relationships between entities as candidate operations. Describe the behavior of these actions as to what objects are modified, what information is required, which objects invoke which operations, and what other operations might they naturally require of other objects.

The primary objective of this step is identification with some basic definition. As identifiers of objects and operations, attributes should be associated with appropriate entities after they are identified. Listing object and operation attributes serves to define those entities in greater detail. The requirements document will often need to be consulted to fully describe program entities.

Summary of the Identification Step.
1. Apply identification heuristics to identify the set of objects in the system.
2. Analyze each object and describe its attributes and structure in the solution strategy.
3. Apply identification heuristics to identify the set of operations performed.
4. Analyze each operation to determine and define its stimulus/response attributes.

3.3 Encapsulate Objects, Attributes, and Operations into Modules

Associating operations with objects is not as straightforward as it may seem. Objects seldom behave independently of other objects. Consequently, observed behaviors may represent a complex interrelationship among objects. Thus guidelines, rules, or heuristics are needed to guide the encapsulation of objects and operations in such a way as to produce good modules.

In choosing which objects and operations to encapsulate into modules, the interrelationships between modules are clarified. Those relationships or interfaces are specified by first determining the dependency between modules. The specific operations of an object required by each operation of each external object need to be diagrammed. This includes identifying the attributes or arguments an operation requires, and which attributes or internal objects are affected through such an operation.

Heuristics for encapsulation include the following:

Object Classification requires identifying an object's operation as one of several general types such as actors or agents [4] [9].

Application Classification requires identifying an object or operation as one of a set of predefined types specified as a set common to the program application area [2].

Structural Classification requires identifying an object's structure as one of four general types (e.g. an abstract state machine) [3].

Summary of the Encapsulation Step.
1. Apply encapsulation heuristics to the objects and operations to determine a set of system modules.
2. Determine the interrelationships between modules and diagram the dependencies.
3. Analyze each module dependency to determine and diagram the detailed interfaces between each dependent module's operations and the executors of those operations.
4. Refine the descriptions of the operations of each object in view of the various conditions under which it might be required of some other object and develop a state transition diagram if appropriate.

3.4 Decompose the Modules or Begin Detail Design

Decomposition deals with whether each module should be further decomposed, constructed from known components, or algorithmically defined. Inheritance may be applied since, a full description of each object at a given level of detail is available. Inheritance is applied based on the object or module classifications made in the previous step. Such classifications are helpful in identifying objects as instances of classes in the system, or as matching preexisting templates in a class library. The decision to use inheritance is always a tradeoff between the cost of new development and the cost of modifications to existing templates. Should inheritance fail to provide a solution the module must be decomposed, or, at its lowest level, described as data structures and algorithms.

All or part of a module may be decomposed. A module containing sets of objects and a set of operations, may have elements of those sets at their lowest level, and other elements of sufficient complexity to warrant decomposition. Decomposition may take a variety of forms. Conventional functional decomposition may be adequate. If the operation exhibits concurrency, a process-oriented approach would be appropriate, with each sub-operation represent-
ing a single concurrent operation. The existence of other independent objects might indicate an object-oriented approach. The problem should lead to an appropriate design technique, rather than forcing the problem into an unnatural methodology.

**Summary of the Decomposition Step**

1. Analyze the modules in the system for signs of common classes. If such a class hierarchy is apparent, indicate objects as instances of the class and further design the class.
2. Analyze the classification of modules in regard to existing generic structures or functions. Determine cost effectiveness of redesign versus reuse.
3. Analyze the complexity and determine which module components must be further decomposed.
4. For each component which must be decomposed, determine the appropriate design method and proceed with the design. Those components which require an object-oriented design, should be treated as new problems and designed using this methodology in an iterative fashion.
5. For each operation which need not be decomposed, describe its operation algorithmically.
6. For each object or attribute which need not be decomposed, describe its data structure.

4 **Developing a Decision Aid for OOD**

This section provides an overview of the top level design of a decision aid to implement the OOD methodology, and a brief discussion of a Smalltalk implementation.

In the field of Decision Support Systems (DSS), requirements determination requires four steps: understanding the problem, selecting a kernel system to implement, developing a representation or model of the system in the form of storyboards, and describing the database and modelbase requirements to support the system. The storyboards and associated feature chart then serve as a top level design of the dialogue, database, and modelbase components of the decision aid.

4.1 **Understanding the Problem**

The problem is to provide a methodology for object-oriented design which addresses the decisions a designer must make. A solution was determined from concept maps of the OOD process and the resulting model and methodology were proposed.

4.2 **Selecting the Kernel**

The concept map of Figure 2 was used to show the OOD decision processes. The feature chart [14] shown in Figure 3 depicts the support and interaction required by the four steps in the methodology. Storyboards were developed from the feature chart and represent key decisions and the support each decision requires.

The storyboards are linked together through the main menu which is be available at each storyboard to allow switching to any other storyboard. The main menu also provides a means of exiting the system and allows access to context sensitive help and the hook book, an on-line note pad to users to note possible modifications to the tool. Several functions overlap. For instance, the object and operation definitions created in the Identification storyboard are used again in both subsequent storyboards.

4.3 **Representing the Kernel**

Figure 4 shows an example storyboard developed in the design of the DSS kernel. In general, each storyboard contains at least three sub-windows or panes: a features pane, an objects pane, and a text pane. Selecting an element in the features pane causes a list of files or objects to appear in the objects pane. Selecting an element in the objects pane causes initialization of the text pane, or brings up a sub-window—either one of which will be used to carry out some sub-step in the methodology.

4.4 **Supporting the Kernel**

4.4.1 The Database Requirements.

The database involves the storage, representation, and manipulation of design objects as well as on-line access to a requirements specification. The functions described in the storyboards require the ability to display graphics, text, and data dictionary information.

4.4.2 The Modelbase Requirements.

For the purposes of this paper, the object model of Section 2 and the heuristics and methodologies listed in Section 3 comprise the “relations and associations” which govern the design decisions in the OOD process. The system must be able to manage this information and present it to the user in a meaningful and timely manner.

4.5 **A Prototype Decision Aid**

Case studies of DSS usage show that “Key factors explaining successful development are a flexible design and architecture that permit fast modification and a phased approach to implementation” [15].

An evolutionary design approach was applied to developing a prototype which would allow user response and
Figure 3: Feature Chart for the OOD Decision Aid

Figure 4: Storyboard: Analyze the Problem

540
feedback to determine the potential of these concepts.

It began with implementing the storyboards using the Smalltalk/V Object-Oriented Programming System. A single standard windowing style was used and as much functionality as possible was implemented. Implementation of the object model in Smalltalk consisted of declaring several new classes and selecting the data structures to represent the model. A simple relational approach was taken, directly implementing the relations derived from the model description.

The modelbase was implemented as a context sensitive set of text help files representing design heuristics and methodology instructions. The executive control module maintains lists of help files which can be edited, removed, or added to by the users. Each storyboard contains its own list of heuristic files developed from the examples discussed previously.

To aid in evolutionary development the hook book was fully implemented as a separate object with its own browser for entering, adding, and removing entries.

5 Conclusions

This paper has introduced an adaptive approach to developing software support tools and environments. The specific target was an object-oriented design methodology. The central hypothesis of this effort is that design is essentially a decision process and if good systems are to be produced, good decisions must be made. The software engineering community must take as hard a look at improving the engineers' decision making capabilities as it does in representing those decisions with flashy graphics and powerful databases.

References


