

Object-Oriented Techniques Based on Specifications

Valdis Berzins

Computer Science Department
Naval Postgraduate School
Monterey, CA 93943

ABSTRACT

Object-oriented techniques form a promising approach for realizing an integrated computer-aided software development environment capable of detecting and correcting errors early in the development process. We discuss the connection between formal specifications, object-oriented data models, reusable components, and engineering databases.

Introduction

Object-oriented techniques are characterized by the following:

- (1) Abstract objects whose interactions are limited to explicit interfaces.
- (2) A subclass hierarchy with an inheritance mechanism.

Object-oriented techniques are important in software development because they reflect the shift from a single processor environment to a distributed, multi-processor environment. Objects are potentially active agents that can perform actions independently and concurrently. An object-oriented viewpoint exposes the parallelism inherent in a problem, and allows simpler descriptions of many problems by allowing logically separate activities to be defined independently of each other, rather than forcing their combination into a single sequential process.

Most of the effort in the early stages of software development is spent on building models. A model of the environment for the proposed system is usually developed in the requirements analysis phase. A model of the external interfaces of the system is constructed in the functional specification phase [1], while models of the internal interfaces are constructed in the architectural design phase. These models are the basis for the software tools in advanced computer-aided software development environments. Such environments depend on engineering database support for effective tool integration and coordination of the concurrent activities of a design team [10]. Object-oriented methods are important for developing models of software systems and the automated tools supporting the construction and analysis of those models.

Object-Oriented Domain Models

The domain model developed during requirements analysis can be conveniently expressed in an object-oriented manner. The domain model supplies the concepts needed for describing the world in which the proposed system will operate. These concepts consist of the types of objects in that world, the attributes of those objects, the relations between those objects, and the laws governing those objects and relations. The domain model is important because it forms the basis for all agreements between the customer and the developer.

A type is a set of objects which are all subject to the same attributes, relationships, and laws. An attribute is a single-valued mapping from one or more types to another type. A relationship is a mathematical relation, corresponding to a predicate which is true for exactly those n -tuples contained in the relation. The number of tuples in a relation need not be finite. Laws are relationships that must be true for all possible n -tuples of objects from the given types.

The domain model presents a simplified view of the world containing only a fixed set of types, attributes, and relationships. This includes both the aspects of the world that will have to be represented in the proposed software, and the aspects that impose external constraints on the system and motivate the customer's requirements. The domain model includes the known laws governing the behavior of the environment, which can be either

descriptive or prescriptive. Laws can be inherited from general purpose types and relationships in the model library by means of a subclass mechanism.

Object-Oriented Specifications

One of the primary difficulties in the development of large software systems is conceptual complexity. Conceptual complexity can be reduced by constructing a set of independent abstractions to describe a complex concept or system [7, 8]. Abstractions are concepts that can be treated as black boxes [2]. A concept qualifies as an abstraction if it can be understood, specified, and analyzed independently of the mechanism used in its implementation. Formal specifications are essential for the effective use of abstractions, since the specifications must be precise to allow use of the abstraction without the need to examine its implementation. Formally defined notations for specifying black boxes are essential for achieving a high degree of automation [6]. Object-oriented approaches can make the formal methods easier to use and automate.

Objects in Formal Specifications

A natural and convenient approach to formal specifications is based on an object-oriented event model of computation [4, 5]. In the event model, computations are described in terms of modules, events, and messages. A module is a black box that interacts with other modules only by sending and receiving messages. An event occurs when a message is received by a module at a particular instant of time. A message is a data packet that is sent from one module to another. Modules and messages are the most important kinds of objects in the event model.

Modules can be used to model external systems such as users and peripheral hardware devices, as well as software components. Modules are active black boxes, which have no visible internal structure. The behavior of a module is specified by describing its interface. The interface of a module consists of the set of stimuli it recognizes and the associated responses. A stimulus is an event, and the response is the set of events directly triggered by the stimulus.

Messages can be used to model user commands and system responses. Messages represent abstract interactions that can be realized in a wide variety of ways, including procedure call, return from a procedure, Ada rendezvous, coroutine invocation, external I/O, assignments to non-local variables, hardware interrupts, and exceptions.

Another kind of object useful in formal specification is the *concept*. A concept is a predicate, function, constant, or type that is needed to describe the behavior of a module. Concepts are one level removed from the objects in the model, because they appear only in explanations of the required behavior, rather than acting as direct participants in interactions with the proposed system or module. Modules and messages are objects contained in the model of the system, which concepts are meta-objects for that model. Concepts are important for organizing complex descriptions into independent pieces small enough to be readily understood. Concepts are part of the specification language rather than the underlying event model.

Object Hierarchies

Hierarchical structures are important for making complex systems manageable. A popular approach to functional specification uses data flow diagrams to decompose a system into sub-processes, until a level is reached where the processes are easy to describe in terms of a fixed set of primitives. The behavior of a process is described directly only for processes that are not decomposed further. A more recent approach uses abstraction rather than decomposition to arrive at simple descriptions of a proposed

system [2, 5]. The advantage of this approach is a reduction in the number of components in the specification that must be examined to understand any particular interaction between the proposed system and its environment, especially for large systems. Such a reduction depends on a powerful specification language. Predicates containing quantifiers and high level abstractions can describe the behavior of complex modules without introducing internal details via decomposition of the computation.

The abstraction approach does not treat large functional specifications as monolithic entities, however. Black-box specifications are divided into simpler pieces in several qualitatively different ways.

- (1) Very large systems often contain a number of nearly independent major subsystems, which have minimal interactions with each other. For example, a spacecraft may have a navigation subsystem, a communication subsystem, and a subsystem for controlling an exploration robot. Such major subsystems should be modeled as distinct central modules.
- (2) Each major subsystem typically has more than one interface. For example, the navigation subsystem may have interfaces with the pilot and with a number of different sensors. Each interface of a central module is specified as a separate view.
- (3) The description of each interface of a module is partitioned based on the messages it accepts, the normal and exceptional responses to each message.
- (4) The concepts needed to describe the behavior of each message are described by a structured set of definitions. In complex systems the definitions of these concepts can have a hierarchical structure, in which the more abstract concepts are defined in terms of more primitive ones at several levels of detail. The concept hierarchy replaces the data dictionaries used in earlier approaches, and is more general because it includes predicates and functions in addition to data types and constants.
- (5) The individual events of a system can be organized into atomic transactions to describe the degree of interleaving allowed between concurrent interactions. The transactions in a complex protocol can also be defined hierarchically.

Object-oriented models usually include subclass hierarchies with inheritance. Inheritance is an important feature for specification languages [3], which has several important uses in specifying large software systems.

- (1) Inheritance can be used to standardize command interfaces across the subsystems of a large system. Consistency in the interpretations of similar commands in different subsystems is an important factor in making large systems easier to use. Specifying subsystem interfaces as subclasses of an object type defining the standard system-wide interface conventions provides a way to specify and mechanically check conformance to such conventions in large systems.
- (2) Inheritance provides a mechanism for specializing reusable fragments of specifications. A library of reusable specification components should contain partially specified general purpose building blocks that can be tailored to the needs of each application.
- (3) Multiple inheritance can be used to support view integration for systems with multiple interfaces. If each interface of the system as a separate module legibility is improved and concurrent development of different interfaces by different people is enhanced. The entire system is specified by a module that simultaneously inherits the details of the individual interface specifications.
- (4) Inheritance can be used to record the steps in a stepwise refinement, where refinements correspond to subclass elaborations. This is especially useful for maintaining the distinction between the details in the user's views of the system and the details in the implementor's view that should not be visible to the users.

Object-Oriented Specification Languages

Specification and prototyping languages are important for computer-aided software engineering [11]. Specification languages are designed for the simple description of complex behavior, with automated tools for synthesis and error checking emphasized over the ability to execute the entire language. Prototyping languages are designed to be executable, with simplicity of expression emphasized over execution efficiency. Both kinds of languages can benefit from an object-oriented approach. Some examples of object-oriented specification languages are Spec [4,5] and MSG [1]. An

example of an object-oriented prototyping language is PSDL [12].

Object-Oriented Databases

Specialized engineering databases are essential for integrated, flexible CAD systems in general [10] and for computer-aided software engineering in particular [13]. Such databases are used for maintaining the versions of the software being developed as well as the library of reusable components and the knowledge bases used by expert systems embedded in the CAD tools. The required capabilities can best be provided by object-oriented databases [9], which were developed primarily to support engineering applications.

The essential problem in the organization of object-oriented databases for managing reusable components is to allow the representation and retrieval of an unbounded number of components with finite memory and processor speed. An unbounded number of components must be considered because software designs can contain arbitrary user-defined abstract data types, and the reusable components in the component database must be applicable to all of the types in this infinite set to be useful. It is also necessary to allow the retrieval of assemblies composed of finite numbers of available reusable components, because it is unlikely that reusable components can be provided to serve all possible functions. It is desirable for the designer to be able to think top-down while the bottom-up search for available components is aided by the database management system, because the set of reusable components can get very large, making it impractical to expect each designer to be familiar with all of the available reusable components.

Conclusions

Object-oriented approaches are a promising means to achieve a high level of automation in the software development process. An integrated approach is needed to realize the potentials of this approach. The design and analysis methods, the notations and tools that support them, and the engineering databases coordinating the process must all be tied together in a consistent object-oriented style. This involves object-oriented data models at several different levels, which must be tied together by transformations and constraints. This is a promising area for research, with many interesting and tractable problems whose solutions promise great practical benefits.

1. V. Berzins and M. Gray, "Analysis and Design in MSG.84: Formalizing Functional Specifications", *IEEE Trans. on Software Eng. SE-11*, 8 (Aug. 1985).
2. V. Berzins, M. Gray and D. Naumann, "Abstraction-Based Software Development", *Comm. of the ACM* 29, 5 (May 1986), 402-415.
3. V. Berzins and Luqi, *The Semantics of Inheritance in Spec*, Computer Science, Naval Postgraduate School, 1987. NPS 52-87-032.
4. V. Berzins, "The Design of Software Interfaces in Spec", in *to appear in Proceedings of the International Conference on Computer Languages*, Miami, Oct. 1988.
5. V. Berzins and Luqi, *Software Engineering with Abstractions: An Integrated Approach to Software Development using Ada*, Addison-Wesley, 1988.
6. V. Berzins and Luqi, "Languages for Specification, Design and Prototyping", in *Handbook of Computer-Aided Software Engineering*, Van Nostrand Reinhold, 1988.
7. G. Booch, *Software Engineering with Ada*, Benjamin/Cummings, Menlo Park, 1983.
8. J. B. Dennis, "Modularity", in *Software Engineering: An Advanced Course*, vol. 30, F. L. Bauer (editor), Springer Verlag, 1975.
9. M. Ketabchi, V. Berzins and S. March, "ODM: An Object Oriented Data Model for Design Databases", in *Proc. ACM Computer Science Conference*, Feb. 1986.
10. M. Ketabchi and V. Berzins, "Modeling and Managing CAD Databases", *IEEE Computer* 20, 2 (Feb. 1987), 93-102.
11. Luqi, "Specification Languages in Computer Aided Software Engineering", *IEEE Software Design and Network Conference*, Santa Clara, CA, April 1988.
12. Luqi, V. Berzins and R. Yeh, "A Prototyping Language for Real-Time Software", *to appear in IEEE Trans. on Software Eng.*, 1988.
13. S. Simmel and V. Berzins, "A Software Management System", in *Proc. First International Workshop on Computer-Aided Software Engineering*, Cambridge, MA, May 1987, 767-783.