HEWLETT PACKARD

Criteria for Comparing Object-Oriented Development Methods

Patrick Arnold, Stephanie Bodoff, Derek Coleman
Helena Gilcrist, Fiona Hayes
Information Management Laboratory
HP Laboratories Bristol
HPL-91-51
June, 1991

object-oriented methods, comparison

Recently there has been a profusion of object-oriented analysis and design methods coming from a variety of backgrounds. The aim of this paper is to aid the objective comparison of methods. A set of criteria for making explicit the differences and similarities between methods is proposed. The criteria are presented as a set of questions together with some preceding commentary. There are four main categories of criteria: Concepts, Models, Process and Pragmatics. Concepts deals with what makes a development method object-oriented. The criteria for models consider what abstract system descriptions a method requires and the notations it proposes to capture those descriptions. The process criteria focus on mechanisms for developing the models. The pragmatic criteria concentrate on non-technical features.

This paper should be of interest to potential users of object-oriented methods as well as to the developers and advocates of these methods.

1 Introduction

Experience of the industrial use of object-oriented technology indicates that a disciplined software process is the essential factor determining success [4]. Key components of a software process are systematic analysis and design techniques. The first efforts at employing such techniques for object-oriented software development attempted to use traditional methods such as SA/SD ([5] [12]). However it rapidly became apparent that object-oriented methods were necessary because methods based on the functional decomposition of a system clash with the object-oriented approach.

Recently there has been a profusion of object-oriented analysis and design methods coming from a variety of backgrounds. Some, such as HOOD [6] and Buhr's [2] are targetted at the ADA community. Booch's [1] method is similar but has been extended to make it more truly object-oriented. Entity-Relationship Modelling [3] is the basis of methods like Rumbaugh's et al. [8], whereas the responsibility-driven method of Wirfs-Brock et al. [10], stems from an operational view of object interaction.

The emergence of these methods naturally raises the question as to which one is the best. It is not the goal of this paper to tackle this problem. We defer tackling this task since it is only with the benefit of extensive and metricated practical experience that such a question can be answered.

The aim of this paper is to aid the objective comparison of methods by proposing a set of criteria. We hope the criteria will aid in answering questions like "What is the difference between method X and method Y?" , "Are method X and method Y complementary?", " Could they be used together?". The criteria are tentative. We arrived at them from our accumulated software engineering experience and as a result of studying a number of the new methods. We were also influenced by other method surveys ([11], [7]). We have devoted enough time and energy on the criteria to know that they can only be considered as a first shot at the task. However we believe that they are worth consideration and are a step in the right direction.

The criteria are presented as a set of questions together with some preceding commentary. The aim of each commentary is to clarify the associated question(s) and to indicate some technical terms that may be useful in framing the answer. We do not provide definitions, except in the case of terms that may be ambiguous or unfamiliar to members of the object-oriented community.

The document is structured according to the four main categories of criteria: Concepts, Models, Process and Pragmatics. The concepts section addresses the question of what makes a development method object-oriented. The core of all software development methods, namely the models advocated and the process for developing those models, are examined in subsequent sections. Finally, the pragmatics section considers non-technical features of methods such as availability of resources. We also offer some directions for future research and conclusions.

Note In this document, evaluation criteria are indicated by use of italics.

2 Concepts

In order to be considered object-oriented a method should support the expression of those concepts which assume the most prominent role in object-oriented software systems. These concepts are mostly derived from object-oriented programming languages. As with object-oriented languages [9], there is no universally agreed upon set of features (apart from objects) that a method should support. This section introduces criteria for describing the semantics of the object model and thus provides a framework for evaluating the extent to which a method is object-oriented.

2.1 Objects and Classes

The fundamental concept that must be supported by an object-oriented method is the *object.* An object encapsulates its internal state (or attributes) and provides an interface (a set of operations) for manipulating the state. The way information hiding interacts with the type system to provide objects differentiates Ada based methods from those coming from the object-oriented world.

In *class-based* methods the information hiding module, called a class, has an associated type. A *class* is a template which describes the attributes and interface of a set of objects. Object instances are produced by defining class variables.

Package-based methods separate the type system from modules by providing untyped modules for encapsulating data. If a module exports a private data type then each variable of the type produces an object instance; otherwise the module corresponds to a single object.

Both package-based and class-based methods can employ *generic* modules. A generic module or class is a parameterised template that can be instantiated to give a simple module or class. According to the method, the parameters can be types, classes or operations.

A *metaclass* is a template whose instance is a *class object.* A class object has attributes that contain information common to an entire set of objects of one class and an operation to create new instances of that class.

> *Is the method class-based or package-based?*
> *Does it support generic modules and/or metaclasses?*

2.2 Inheritance

Inheritance is a relationship between classes in which the features of one class, called a *subclass*, are defined in terms of one or more *superclasses.* This facility permits the incremental development of designs and implementations. In *single inheritance* each subclass is allowed just one immediate superclass, whereas *multiple inheritance* permits more than one immediate superclass.

A design method can support *subtype inheritance* in which the subclass behaves like its superclass for all the operations of the superclass. It may also support *unrestricted* inheritance which permits the subclass to change the signature or behaviour of operations.

Some methods allow the definition of *abstract classes* which cannot have instances and exist solely to partially define the properties of its subclasses.

What type of inheritance does the method support?

2.3 Visibility

An object uses another object to perform a service by invoking an operation in the used object's interface. This is called the *client/server* relationship between objects and is fundamental because work in object-oriented systems is accomplished by collections of interacting objects. In order for a client to be able to use a server, the server has to be visible to the client. Methods differ in the degree to which object visibilities can be expressed.

The *aggregation* relationship holds when one object is a component of another object. Components have the same lifetime as the whole and are visible to the whole.

Objects can use objects other than their components. A method may make no restrictions and assume *static global* visibility. Scoping allows visibility to be *restricted statically.* Visibility may be *dynamic*, for example a server object can become visible to a client by parameter passing.

What visibility relationships does the method support?

2.4 Lifetimes

A method may be restricted to dealing with *static systems* of objects in which all objects have the same lifetime as the system. If this is not the case then the method must contain some facility for dynamically *creating* objects, for example by instantiating a class. It is also desirable to be able to specify object *destruction.* These two operations allow fully general systems to be modelled.

Does the method support object creation and destruction?

Not all objects are transient. There are a number of reasons why mechanisms to maintain objects that live indefinitely are necessary. Some objects may simply outlive one (or all) the executions of a program. In long lived object systems, some objects may have to be written to storage for reasons of resource management. To provide for these circumstances, a method may contain a facility for indicating object *persistence.*

Does the method support object persistence?

2.5 Concurrency

Because the real world is concurrent, concurrent objects are often used in the analysis stage to model it. Objects mesh nicely with concurrency since their logical autonomy makes them a natural unit for concurrent execution. However, concurrent sharing is more complex than sequential sharing, requiring mutual exclusion and temporal atomicity. The interfaces, internal structure and communication protocols of concurrent objects are more complex.

Normally objects are *passive*, because they are inactive until an operation is invoked by a client. In contrast *active* objects have their own thread of control and may be executing when the client attempts to send a message (i.e. invoke an operation). An active object is *internally concurrent* if it has more than one thread of control. Methods should support ways (e.g. monitors) for guaranteeing mutually exclusive access to shared data in concurrent systems.

What models of concurrency does the method support?

2.6 Communication

Objects constitute a loosely coupled model of computation, in which communication provides both information flow and synchronisation. The usual model of communication is that only two objects are involved in any one communication with the sender having to know the receiver's identity but not vice-versa. The information flow however, may be uni- or bi-directional.

Synchronous communication requires the sender to suspend execution until the receiver accepts the message, whereas *asynchronous* communication allows the sender to continue. Further qualifications of synchronous communication are *balking* (abort if receiver not ready) or *timeout* (abort if receiver not ready after some specified period). Communication is *reliable* if the sent message is guaranteed to remain available until the receiver is ready to accept it.

At the analysis stage methods often use an *event model* in which the communication is instantaneous and atomic. For design, development methods often use more complex communication primitives like those provided by implementations, e.g. the *procedure call* for sequential systems and the *rendezvous* and *remote procedure call* for concurrent or distributed object systems.

Mutual messaging between objects is important because of its use in model-view-controller type designs. This category of object communication includes *recursion*, where an object sends a message to itself and *callbacks*, where a server sends a message to a client during the evaluation of a message from a client.

What models of communication does the method support?

3 Models

A development method proceeds by developing abstract descriptions, or *models*, of the system under analysis or design. Each model is expressed in some *notation*. In assessing a method it is necessary to consider the models it constructs and the notations that it uses. The prime requirement is that the set of models should form a *complete* and *consistent* description.

3.1 Kinds of Models

Three kinds of models can be produced. A *physical* model is concrete and concerned with the actual structure of the software system and typically deals with such things as code modules and processors. *Logical* models capture the key abstractions of the system. Logical models can be separated into *static* models which emphasise the structure of a system and *dynamic* models which deal with temporal and functional behaviour. Another distinguishing feature is whether a model pertains to the *system* or an individual *component*. A component can be atomic, i.e an individual class or package, or a *subsystem*. In cases where more than one model captures the same information there should be rules for checking *consistency* between the models.

> *What models does the method prescribe and what notation is used for each?*
> *Are there any aspects of a system that are omitted or any that are covered by more than one model?*

3.2 Notation

This section is concerned with the properties of notations used to capture models. We consider their expressive power, whether their syntax and semantics are well-defined and how well they scale-up.

3.2.1 Expressivity

The main issue for a notation is *fitness for purpose*. Notations can be pitched at different levels of abstraction: they can use *abstract* or *concrete* data types and can be *declarative* or *operational*. If a notation cannot directly represent the essential concepts of the model, then the user has to encode this representation explicitly in the terms of the notation. This leads to more complex and less easily understood descriptions. These kinds of problem also afflict notations that are too *verbose*.

> *Are the method's notations appropriately expressive?*

3.2.2 Syntax and Semantics

Not only should the notation be sufficiently expressive but it should also be well-defined. The *syntax* of a notation is a set of rules which describe the primitive

components of a notation and the legal combinations of those symbols. Notations can be *textual* or *diagrammatic*. There are well-known techniques, such as BNF, for formally defining textual syntax. Techniques for defining the syntax of diagrams are less well-established, however there should be a clear definition of the icons and their legal combinations. A defined syntax is a requirement for effective use and also for automated tool support.

> *Is there a syntax definition or does the syntax have to be deduced from examples?*

The *semantics* of a notation is a set of rules which gives the meanings of the syntactic primitives and their combinations. In general, semantic definitions are more complex than syntactic definitions. A well-defined semantics eliminates ambiguity and is a pre-requisite for advanced tool support such as code generation or simulation. More importantly, a semantics is necessary for allowing analysis and design models to be examined and evaluated during development. There should be rules for *reasoning* about models and for *transforming* one model into another.

> *Is there a semantic definition or does the semantics have to be deduced from examples?*
> *Does the semantics have a formal foundation?*
> *Is there a logic for reasoning about or transforming models?*

3.2.3 Scalability

Scalability is concerned with whether a notation can be used effectively on large systems. Notations need a mechanism for *partitioning* descriptions into smaller and more manageable modules and *composing* the whole from those modules. It should also provide some means of *controlling the visibility of names* across modules, in much the same way as programming languages provide mechanisms for controlling the scope of names.

> *Does the notation provide a partitioning mechanism?*
> *Are there rules for composing the meaning of a system from the meaning of its modules?*
> *Is there an explicit mechanism for defining the scope of names?*

4 Process

We use the term *process* to characterise the steps that make up a method. A process has two main roles: to drive the development to an appropriate implementation and to assist progress tracking through the definition of milestones and deliverables.

First we look at the context of the software development in which a method is useful and what part of the lifecycle it covers. We then discuss the properties of a process including pragmatic issues such as flexibility and heuristics.

4.1 Development Context

Software development occurs in many different contexts. Most development methods are aimed at *greenfield* developments where there is no previous history of software development and the only environment is that provided by for example an operating system.

Adding functionality and *reengineering* requires a provision for the capture of functionality and the extraction of suitable abstractions of an existing system before the design can be modified to include the new functionality.

> *Does the process provide support for adding functionality to existing systems and reengineering?*

A further kind of development context is that of design with reuse. A process which supports reuse requires a look ahead approach such that the common, useful and hence reusable components can be identified. Once candidates for reuse are identified one can search a library to see if reusable components already exist.

> *Does the process address the issue of design WITH reuse?*

Reusable components and designs have to be developed, they are not just a by product of using objects. A process needs to explicitly provide activities which are intended to identify reuseability and support the development of reusable components and designs. Typically the development of a reusable component will be a design exercise in its own right, as a reusable component must not only satisfy the immediate needs of the current development but must take a broader view of the requirements for reuse.

> *Does the process address the issue of design FOR reuse?*

4.2 Coverage of Lifecycle

In this section we identify some of the activities which constitute a software development process. Many methods cover different parts of the lifecycle, not just analysis or design. Therefore it is more useful to describe a method in terms of the development activities it supports. These activities can be combined in various ways to make up a particular process model, for example the spiral model.

The term design is applied very loosely by authors of methods, so that many methods which claim to be design methods also include aspects of analysis and implementation. For our purposes we will use the following definitions:

Analysis The purpose of analysis is to construct the logical model of the system and its environment. At this stage there is an emphasis on describing properties rather than the mechanisms which implement them.

Design In the design phase the system to be built is differentiated from its environment. The logical models produced during analysis are successively refined and made more concrete, and a physical model is produced. The emphasis is on the realisation of the properties as a software structure.

Implementation Implementation encodes the physical and logical models in a particular programming language. At this point in the process all of the structure and behaviour of a system will have been defined, and the emphasis is on providing an encoding of the design using the primitives of a particular language.

Which of these activities does the process support?

4.3 Process Properties

A process should be repeatable and flexible so that it can be reused and adapted to meet local requirements. Each process step must be defined in terms of its inputs and outputs, or in some other way. Since one step may produce an input for another step, there are usually constraints on the order in which the steps can be tackled. However a process definition should not force unnecessary sequentialisation. Wherever possible it should allow steps to be overlapped in time, in order to exploit potential parallelism in the development. Similarly deliverables should not be tied to particular notations because this makes it difficult to substitute alternative approaches for particular activities.

Are the process steps well-defined?
Is the process flexible?

The adaptability of a process can be improved by the inclusion of guidelines, or *heuristics*. These provide a means of identifying common situations and tackling them in a previously used way. They should help to identify when it is appropriate to perform a particular activity and how to begin the activity.

Are there heuristics?

The reasoning behind the decisions embodied in an implementation is invaluable during software maintenance. It is important therefore to be able to *trace* the connection between requirements and implementation. Traceability is aided if the deliverable from one step is explicitly refined or developed during some subsequent step. Naming conventions can also help by indicating relationships between models.

Is it possible to locate the origin of design decisions made during the development?

A process should provide mechanisms to *verify* that an implementation meets its requirements. Verification involves demonstrating that at each phase the models are consistent with each other and with those from the previous phase. This requires steps which show consistency by inspections, testing or proof.

Does the process provide for verification?

A process should also include steps for *validating* whether a development meets the customer's needs. This can be done through the construction of executable models, e.g. through simulation or the use of prototypes, or by using notations which allow the properties of models to be deduced.

Does the process provide for validation?

5 Pragmatics

There are many pragmatic concerns that influence a methods uptake in the software engineering community. These concerns can be divided into two categories: those having to do with the human-method interaction and those pertaining to the utility of a method in a particular application domain. Within these two categories further distinctions can be made between those properties that are intrinsic to the method itself and others that are external and possibly even transient in nature.

5.1 Resources

A concern when considering which method to adopt are the variety of resources available to support its introduction and use. A course is often an appropriate vehicle for the first introduction. A textbook may be sufficient for more experienced developers and can serve as a reference document. Other discriminants for a method include whether it is supported by more than one consultancy firm or CASE vendor. Similarly the existence of user groups, workshops and conference tutorials tend to suggest that a method is in widespread usage.

What resources are available to support the method?

No matter how straightforward a method is, almost all projects beyond a certain size will require some form of tool support to assist the development of analysis and design models. CASE tools can be distinguished by whether they provide syntactic or type checking. Semantic processing is also desirable; simulation, code generation and proof tools fall into this category. In general the existence of CASE tools encourages the development of defacto standards for the method.

Are there CASE tools available to support the method?

5.2 Accessibility

Users are also concerned with how difficult it is to learn to use the method and once learned, how usable it is. The background required of the user must be taken into account. A distinguishing characteristic of methods is the level of mathematical sophistication required to use its notations.

What background is necessary for someone learning the method?

5.3 Applicability

A method that is targeted at a particular implementation language is likely to have limited applicability, since it may not fit well with languages that have a different underlying semantic model.

Is the method targeted at a specific language?

Methods may be restricted to certain application domains. Rumbaugh et al [8] give the following list as a starting point for what areas a method might reasonably be expected to address:

Batch - A data transformation executed once on an entire input set.

Continuous transformation - A data transformation performed continuously as inputs change.

Interactive interface - A system dominated by external interactions.

Dynamic simulation - A system that simulates evolving real world objects.

Real-time system - A system dominated by strict timing constraints.

Transaction manager - A system concerned with storing and updating data, often including concurrent access from different physical locations.

Distributed system - A system subject to communication latency.

For what application areas is the method suitable?

6 Future Work

In this paper we have presented a set of criteria for comparing methods. We intend to validate them by using them to compare all the major object-oriented analysis and design methods. This study will be carried out as preparation for developing a course on object-oriented development to be used by Hewlett-Packard engineers. We expect that the systematic nature of our study will allow us to synthesise a method which offers significant advantages over extant methods.

We would like to encourage others to use and improve our criteria by applying them to diverse methods. We are keen to receive feedback on any such efforts. We believe that this paper is a contribution to the systematic study of object-oriented methods. In the long term it is only by the increased understanding of object-oriented methods that the user community will get the methods that meet their requirements.

7 References

[1] G. Booch. *Object-Oriented Design with Applications.* Benjamin Cummings, Redwood City, CA (USA), 1991.

[2] R.J.A. Buhr. *Practical Visual Techniques in System Design: with Applications to Ada.* Prentice Hall, Englewood Cliffs, NJ (USA), 1991.

[3] P. Chen. The entity-relationship model - toward a unified view of data. *ACM Transactions on Database Systems*, pages 9–36, March 1976.

[4] D. Coleman and F. Hayes. Lessons from Hewlett-Packard's experience of using object-oriented technology. In *TOOLS 4*, pages 327–333, Paris, 1991.

[5] T. DeMarco. *Structured Analysis and System Specification.* Yourdon Press, New York, NY (USA), 1979.

[6] HOOD Technical Group. HOOD reference manual, October 1990.

[7] P. R.H. Place, W.G. Wood, and Mike Tudball. Survey of formal specification techniques for reactive systems. Technical Report CMU/SEI-90-TR-5, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA (USA), May 1990.

[8] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design.* Prentice Hall, Englewood Cliffs, NJ (USA), 1991.

[9] P. Wegner. Concepts and paradigms of object-oriented programming. *OOPS Messenger*, 1(1):7–87, August 1990.

[10] R. Wirfs-Brock, B. Wilkerson, and L. Wiener. *Designing Object-Oriented Software.* Prentice Hall, Englewood Cliffs, NJ (USA), 1990.

[11] D. P. Wood and W.G. Wood. Comparative evaluations of four specification methods for real-time systems. Technical Report CMU/SEI-89-TR-36, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA (USA), December 1989.

[12] E.N. Yourdon and L.L. Constantine. *Structured Design.* Prentice Hall, Englewood Cliffs, NJ (USA), 1979.