Perspectives on Object-Oriented Design

Peter Wegner

Department of Computer Science
Brown University
Providence, Rhode Island 02912

Technical Report No. CS-91-01
January 1991

Perspectives on Object-Oriented Design *
Review of Concepts and Literature

Peter Wegner, January 1991 (draft)

* This review of concepts and literature in object-oriented design focuses on two new texts. This document serves a number of purposes. It may be viewed as an extended syllabus for an object-oriented design course being taught at Brown in the spring semester of 1991. It will be distributed at a January 1991 workshop on curriculum development as a potential prototype for in depth curriculum description of subfields of computer science. A talk based on this material is planned for the medical informatics workshop in Palm Springs in early February. It is being submitted for publication and suggestions for its improvement are welcomed.

Perspectives on Object-Oriented Design
Peter Wegner, January 7 1991 (draft)

1. Introduction

Software technology is making a transition from the view that programs are action sequences to the view that programs are collections of interacting software components. Object-oriented programming is a form of *component-based software technology* whose software components are objects and classes. Focusing on design promotes a language-independent view of underlying principles that deepens our understanding of the object-oriented paradigm.

In reviewing concepts and literature of object-oriented design we focus on two new texts and consider their impact on broader issues of software engineering.

(1) *Object-Oriented Design with Applications* (OODA), by Grady Booch, uses objects and classes as a structuring mechanism, develops concepts and notations for design, and devotes half its 500 pages to five extensive case studies in the areas of process control, scientific tools, information management, heuristic search, and command and control.

(2) *Object-Oriented Modeling and Design* (OOMD), by five authors from General Electric Research Labs [RBPEL], focuses on the modeling process rather than on case studies. Its three-pronged object modeling technique (OMT), which includes object, dynamic, and functional models that capture complementary views of software, offers a visual notation and methodology for object-oriented design.

To motivate our discussion, we examine the nature of design as a human activity and compare OODA and OOMD to other software engineering texts.

2. The Nature of Design

2.1. Design as an Artificial Science

What is the nature of design? *Design* by architects, tailors, or shipbuilders is the process of conceptualizing the form and structure of artifacts prior to their construction. Design is a normative activity that explores how things ought to be, while science describes and analyzes how things are. It focuses on creating and constructing an artificial world, while science focuses on discovering and describing the natural world (see Figure 1).

Simon views design as one of the *artificial sciences* [Si], perhaps the queen of the artificial sciences, and contrasts its techniques of synthesis of artificial reality with the analytic description of actual (natural) reality. He suggests that design is more closely related to the professions of medicine, law, and engineering than to the sciences of physics, chemistry, and biology.

A design in any domain has goals, structure, behavior, and constraints:

(1) A design should satisfy explicit or implicit *goals* specified by its requirements

(2) Design *structure* should mirror artifact structure (phylogeny should reflect ontogeny)

(3) A design should determine *behavior* that realizes the required goals

(4) A design should satisfy *constraints* of the design medium and application domain

For example, a car design has transportation goals, a structure that includes doors, wheels, an engine, and an electrical system, behavior that supports self-propulsion, steering, and refueling, and constraints determined by construction materials and laws of motion.

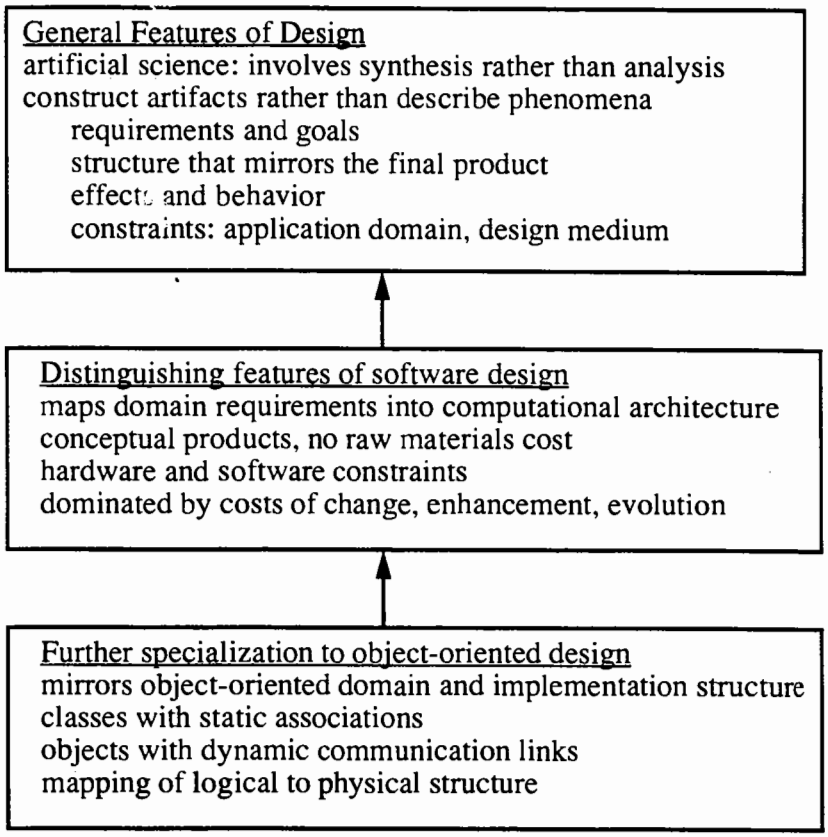

Figure 1: Design → Software Design → Object-Oriented Design

A product life cycle has a create stage, a use and maintain stage, and an enhance or replace stage. Create includes design followed by construct or implement.

life-cycle = create → use and maintain → enhance or replace
create = design → construct or implement

Design occurs early in the life cycle and determines the process of construction, the quality of the product, and the quality of the service provided by the product to its users. For example, an architectural design determines the process of construction, the quality of what is constructed, and the quality of life experienced by its users.

A design must serve a variety of clients, including the sponsors, implementers, users, maintainers, and enhancers. The needs of different clients must be balanced against each other. In particular, every design determines tradeoffs among *cost* to sponsors, *efficiency* of implementation, *simplicity* of use, *robustness* of maintenance, and *flexibility* of enhancement. These factors play a role in any domain, but assume a special form in software design.

2.2. Specialization to Software Design

The economics of software differs from that of physical products in a number of ways. Software construction (called implementation) has no raw materials cost and making copies has negligible incremental cost. Software components do not wear out, and software maintenance consists of fixing errors, improving performance, and changing functionality. Software creation (design and implementation) costs are generally dominated by costs of maintenance and enhancement, in part because software is more versatile in its range of uses and more adaptable to change than physical products. Software qualities needed to handle maintenance and enhancement (robustness and flexibility) are therefore relatively more important in controlling life-cycle costs.

Clean design that provides for disciplined evolution can greatly reduce subsequent life-cycle costs: more effort in the early stages of the life cycle can greatly reduce costs in later stages.

The primary role of software is to model (simulate) application domains by program execution. Designs provide a bridge between the domain being modeled and the implementation that performs the modeling. They should reflect the structure of both the domain and the implementation, since preservation of domain structure in the design and implementation facilitates traceability. Requirements changes in the domain can more easily be mapped into computational changes and conversely implementation changes can be traced back through the design into the domain.

The *waterfall* life-cycle model views software development as a linear process that progresses inexorably from requirements through design to implementation, testing, and enhancement. Recently developed spiral and iterative life-cycle models allow the products of design as well as other phases of the life cycle to be continually modified and reevaluated, and provide better support for enhancement and evolution of software products.

Component-based software technology meshes nicely with spiral and iterative life-cycle models, supporting early decomposition of a system into autonomous components that can be independently constructed, maintained, and enhanced. It facilitates a *divide and conquer* strategy for managing complexity: divide a system into components and conquer each component separately. It combines modularity with abstraction, making use of modularity to divide the system into components and of abstraction to conquer each component.

How does software design contribute to the management of complexity? The key is choosing a modeling framework with primitive components that model the problem domain and composition rules that model interaction in the problem domain. The basic paradigm is that of formal systems (like Euclidean geometry or the predicate calculus) whose simple axioms and rules of inference model great mathematical complexity. Programming languages likewise model complexity by simple primitives and composition rules, but simplicity (say of Turing machines or the lambda calculus) is generally associated with low-level languages. Higher-level languages replace machine language instructions by *software components* as the primitive building blocks for programs. Procedure-oriented languages have functions and procedures as software components, while object-oriented languages have components that are objects and classes.

2.3. Further Specialization to Object-Oriented Design

Object-oriented designs restrict the design space to exclude procedure-oriented and other non-object-oriented software structures. Freedom to experiment with non-object-oriented designs is sacrificed to guide the designer towards an object-oriented solution. There is strong evidence that object-oriented models are generally natural and effective. However, object-oriented design is a heuristic modeling technique that has proved conceptually powerful and practically effective, but may in some cases be suboptimal.

How can we characterize the restrictions that make systems object-oriented? How can we justify the claim that object-oriented technology is effective? We address these questions first for component-based technology and then specialize to object-oriented systems.

The essence of component-based software technology is abstraction implemented by encapsulation. Each component is an abstraction that encapsulates implementation decisions. Components provide their clients with an abstract interface that determines their behavior independently of their implementation. Components may be distinguished by the kinds of information they encapsulate. In particular we may distinguish between functions and procedures which encapsulate actions and objects and classes which encapsulate data. An object-oriented system is a component-based system that supports objects and classes as its primitive components and inheritance as one (of several) composition mechanisms.

Classes specify the common behavior of a collection of objects in terms of a set of operations applicable to objects of the class. Inheritance can factor out behavior shared by several classes into superclasses and can express hierarchical relations among classes like generalization, containment, and evolution. Superclasses amy be both open (extendible) and closed (directly usable). Classes and inheritance impose structure on operations at two levels: classes determine operation clusters while inheritance further structures operation clusters associated with classes into hierarchies. This two-level structure among operations may be contrasted with the lack of structure (or flat structure) among procedures in procedure-oriented languages.

Objects are higher-level software components than procedures, since they are collections of procedures (operations) that share an encapsulated state. Objects are also higher-level modeling primitives, since they directly model objects of the domain while procedures merely model actions on objects of the domain. Modeling real-world entities by clusters of operations that share a state more directly captures the structure of application domains than procedure-oriented designs and determines a universal paradigm for modeling the behavior of entities by finite sets of behavioral attributes.

Notations are an important tool for design, just as language is an important tool for thought. Both OOMD and OODA use visual notations based on graphs whose nodes are objects or classes and whose edges are relations that may be labeled or otherwise distinguished to capture different kinds of static associations among classes and dynamic links among objects.

Visual notations provide intuitive specifications that are generally nonformal and sometimes incomplete. They may be backed up by text that provides a more complete specification of behavior. However, in many contexts a graphical representation, supplemented by brief textual annotation of nodes and edges, sufficiently captures relevant abstractions.

Once classes have been identified during domain analysis, design and implementation of classes can proceed independently and multiple designers can work in parallel on different parts of the design problem. Top-down and bottom-up design are special cases applicable to inherently hierarchical problems. However, incremental, iterative approaches, referred to in OODA as *round-trip gestalt design* are more compatible with iterative and spiral life-cycle models.

Round trip gestalt design supports the *easy-part-first* approach: "when faced with a problem you do not understand, do any part you do understand and then look at it again". The alternative *hard-part-first* approach, "do the hard parts of a design first and the easy parts will fall into place", is equally compatible with round-trip gestalt design. Object-oriented, round-trip gestalt design supports *spiral* software development within a single life-cycle phase, providing greater freedom to divide early and conquer incrementally than traditional design.

Although object-oriented design can in principle be used even when implementations are not object-oriented, it particularly supports object-oriented implementation, facilitating traceability of design components and controlled propagation of change. Object-oriented design encourages but does not require object-oriented architecture throughout the software life cycle.

3. Review of Software Engineering Texts

OODA and OOMD are a new kind of software engineering text that use visual design notations to describe more ambitious and realistic software systems than are found in traditional software engineering texts. We compare OODA and OOMD with other texts on object-oriented design and software engineering.

3.1. Object-Oriented Design with Applications – An Application-Driven Approach

Object-Oriented Design with Applications has three sections that examine object-oriented programming (150 pages), object-oriented design (70 pages), and applications (250 pages), as illustrated in Figure 2. Section 1 discusses the management of complexity, the nature of the

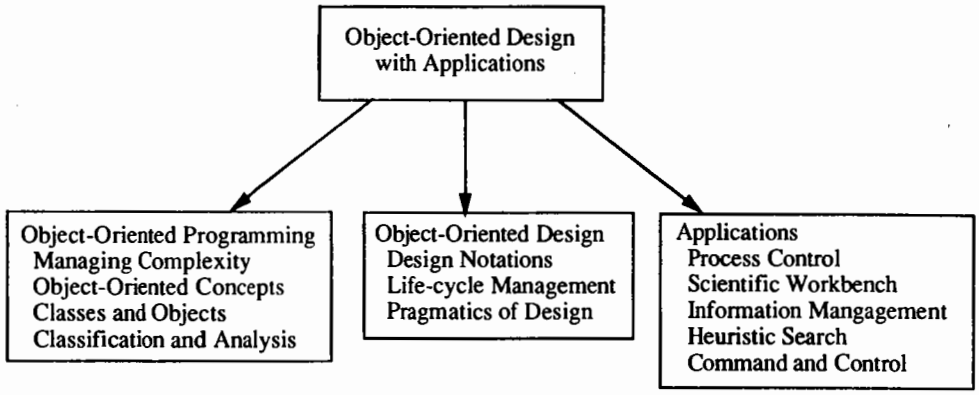

Figure 2: Object-Oriented Design with Applications (OODA)

object model, and the role of classification. Section 2 discusses notations, processes, and pragmatics of object-oriented design. Section 3, the longest and the most unique part of the book, consists of five detailed case studies in different problem domains: process control, scientific tools, information management systems, artificial intelligence, and command and control.

3.2. Object-Oriented Modeling and Design – A Model-Based Approach

Object-Oriented Modeling and Design is close to OODA in its goals and approach, but is less application driven (see Figure 3). Whereas OODA devotes half its pages to case studies, OOMD devotes only the last 50 of its 500 pages to case studies (compiling, computer animation, and an electrical distribution design system). Instead, it emphasizes modeling and introduces a three-pronged modeling technique (OMT) consisting of object modeling, dynamic modeling, and functional modeling. Its three-pronged graphical design notation (program design language) has the same flavor as the design notation of OODA, but is developed with greater precision and nicely illustrated by running examples. Its modeling approach complements the application-driven approach of OODA.

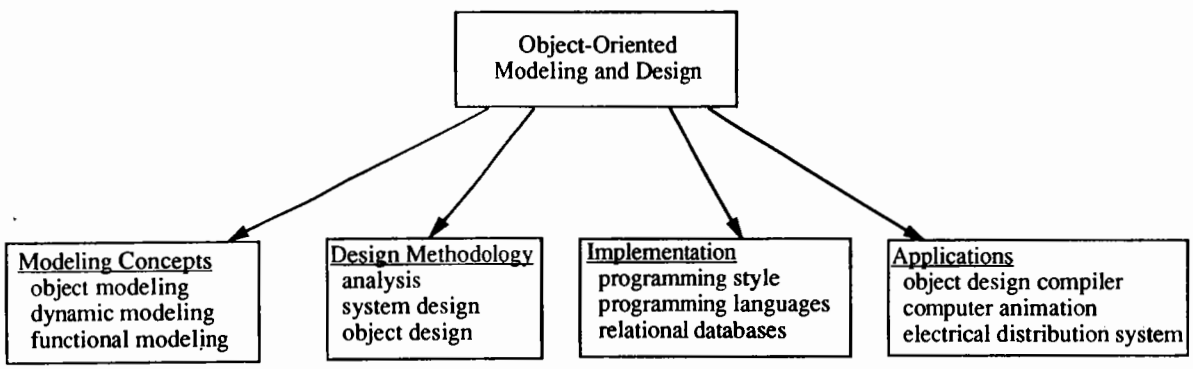

Figure 3: Object-Oriented Modeling and Design (OOMD)

3.3. Object-Oriented Software Construction – A Language-Based Approach

Object-Oriented Software Construction [Me], by Bertrand Meyer, explores software development in the context of a particular programming language, namely Eiffel (see Figure 4). It examines key software qualities such as correctness, robustness, extendibility, reusability, and compatibility. Its discussion of modularity in terms of composability, understandability, continuity, and protection is useful. Its *open-closed* principle that modules should be both open (extendible) and closed (usable) illustrate its nuggets of folk wisdom expressed as boxed aphorisms. (Another is: "Ask not first what a system does, ask what it does it to".) [Me] argues that top-down functional decomposition does not promote evolutionary reusability and advocates a bottom-up, data-driven design methodology starting from objects related to the application domain and progressively abstracting to classes, inheritance hierarchies, and generic classes.

Objects are specified by pre- and post-conditions that determine a contract between an object and its clients. The precondition states the responsibilities of the client, while the postcondition specifies the services provided when responsibilities specified by preconditions are met. The core of this text is concerned with the development of the programming language Eiffel, but software engineering issues are kept firmly in mind during language development. This text is therefore not only an introduction to a particular programming language but also a language-driven introduction to object-oriented design.

3.4. Abstraction and Specification in Program Development – A Component-Based Approach

Abstraction and Specification in Program Development [LG], by Barbara Liskov and John Guttag, focuses on problem decomposition into abstractions and on the informal and formal specification of abstractions (see Figure 5). Two primary abstraction techniques are considered: abstraction by parameterization (as in the lambda calculus) and abstraction by specification. These are used in defining three kinds of abstract software components: procedure, data, and control abstractions. The programming language CLU is introduced to support programming with abstractions. Data abstraction is viewed as the primary program decomposition mechanism, but the properties of procedure and control abstractions are also explored. Data abstractions are specified by constraints (axioms) on operations, while procedure and control abstractions are specified by pre- and post-conditions that determine their stimulus/response behavior. Data, procedure, and control abstractions are complementary views of software structure that respectively

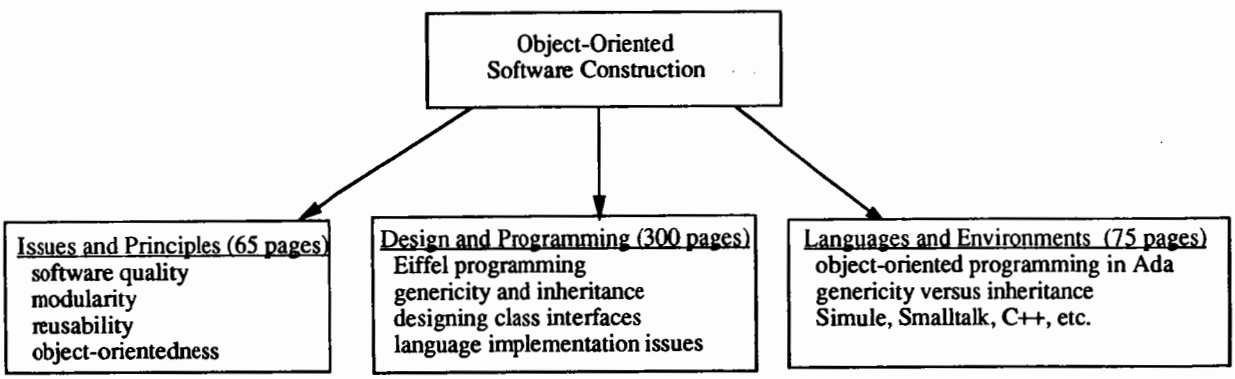

Figure 4: Object-Oriented Software Construction

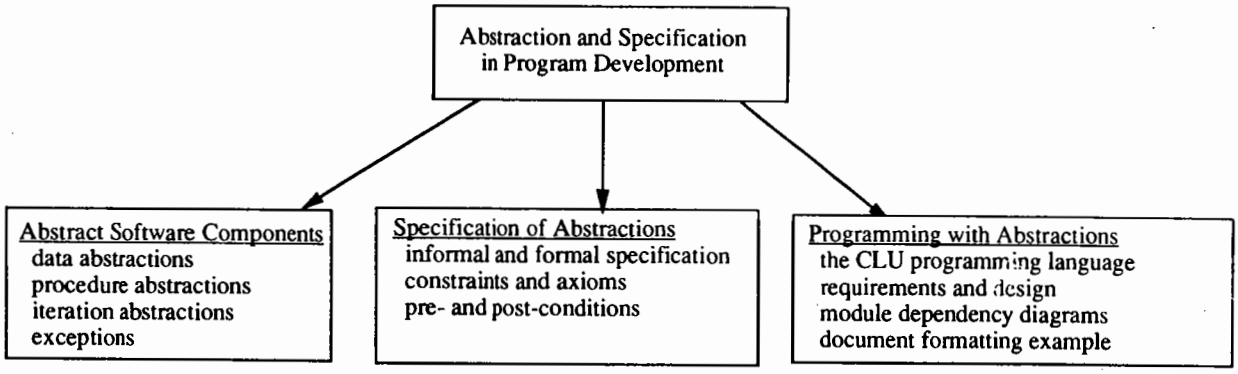

Figure 5: Abstraction and Specification in Program Development

model objects, actions, and iteration, and may be loosely related to the object, functional, and dynamic models of OOMD.

[LG] supports the teaching of specification techniques in a pragmatic framework, providing a basis for a more rigorous approach to design than that of other texts considered here. But the specification methodology is applied only to small problems and cannot easily be extended to large applications. The extension of formal specification techniques to large programs has proved elusive, but research aimed at this goal is continuing.

3.5. Traditional Software Engineering – A Life-Cycle Approach

Traditional software engineering texts such as Somerville [So] (see Figure 6) take a broad view of the subject, considering the complete range of software life-cycle activities and viewing object-oriented design as one among many design techniques. Somerville devotes only one of its five sections to design, and only one quarter of the design section to object-oriented design. Topics such as specification, testing, and design are disussed in a broader context that does not commit to a particular design framework. However, this greater generality is realized at a cost in

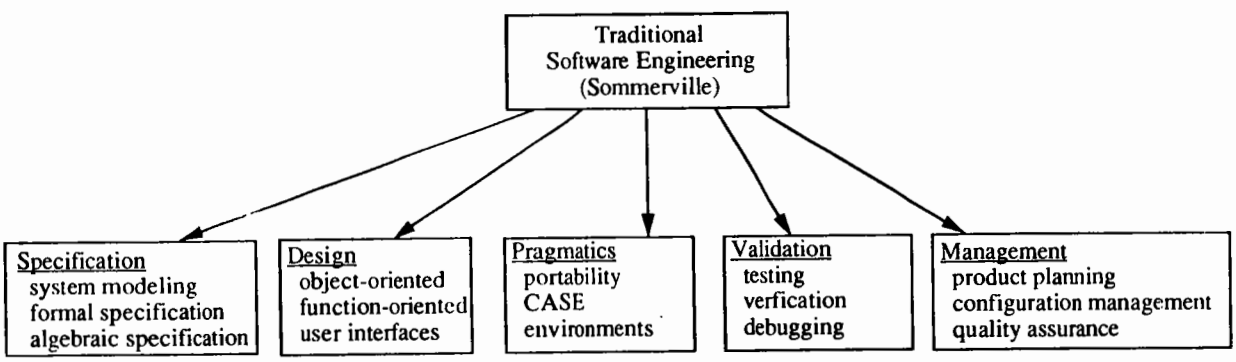

Figure 6: Traditional Software Engineering (Somerville)

relevance to practical problem solving. By committing to a particular methodology, object-oriented texts can go into greater depth in the design and implementation of specific problems.

3.6. Medical Informatics – An End-User Approach

Medical Informatics focuses on a specific application domain with a rich variety of applications. It is a thriving end-user application area that can benefit greatly from the systematic application of software engineering principles. The text *Medical Informatics* by Shortliffe and his colleagues at Stanford [SPWF], examines about a dozen medical application domains. Its application areas include all those discussed in OODA: process control (patient monitoring), scientific tools (radiology), information systems (patient records, laboratory rest records, drug information), heuristic search (rule-based diagnosis), and command and control (complete hospital systems). Moreover, these five categories appear to provide a *complete* classification framework in the sense that we do not need additional categories (see Figure 7). However, most substantial applications have aspects that fit several categories: radiology systems require not only scientific tools for image reconstruction but also information systems for patient records and process control for tracking sequences of images.

The descriptions of application areas provide insight into the expectations of end users, and are effectively in depth requirements for a comprehensive collection of medical applications. It would be an interesting exercise to develop object-oriented designs for the dozen medical domains described in [SPWF], and an even greater challenge to design and implement a comprehensive, open-ended, object-oriented hospital system that encompasses all domains. Such an effort would benefit greatly from an analysis of relations among hospital utilities such as billing systems, patient records, drug knowledge etc. Object-oriented techniques are likely to prove useful both for expressing hierarchical relations among behaviors through inheritance relations and for specifying more finely grained interface behavior of objects and classes in each application domain.

3.7. Software Engineering Economics – A Cost-Estimation Approach

Software Engineering Economics [Bo], by Barry Boehm, examines the economics of software development in terms of costs and benefits for both consumers and producers of software. It warns against viewing problems with a human aspect solely as programming problems (a man with a hammer should not view the world as a collection of nails). The factors that

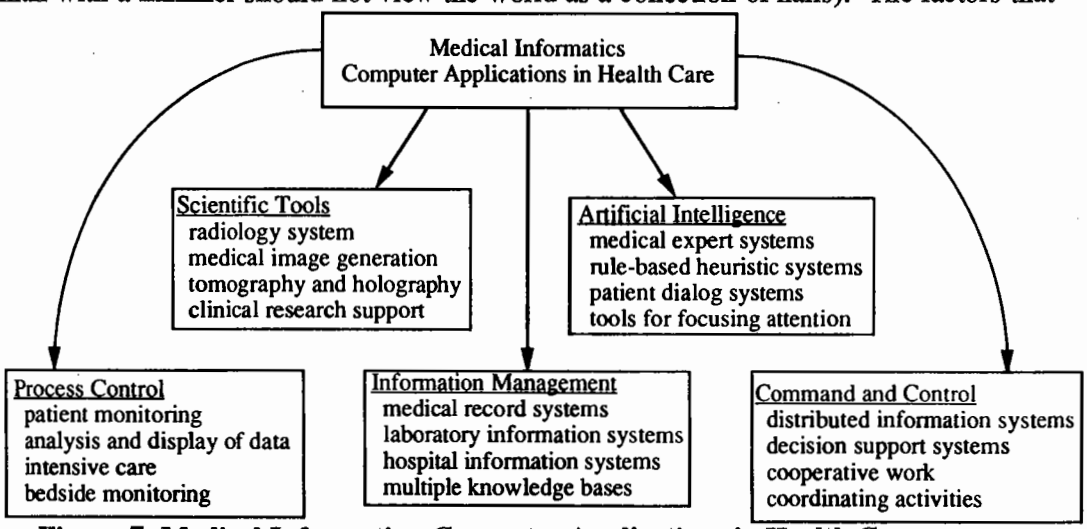

Figure 7: Medical Informatics: Computer Applications in Health Care

determine software costs in each life-cycle phase are carefully analyzed and a COnstructive COst MOdel (COCOMO) for estimating software cost is developed and described in detail for the waterfall life-cycle model.

The three principal parts of the book deal respectively with the development of COCOMO (135 pages), microeconomic techniques such as present value analysis, constraint optimization, risk analysis, and statistical decision theory (140 pages), and the detailed impact of cost estimation methodology on the software life cycle (400 pages) (see Figure 8). The seven basic cost-estimation steps of part 3 include methods for data and resource estimation and suggest the use of several independent cost estimation techniques (algorithmic, expert judgment, analogy, price-to-win). The examination of life-cycle activities from an economic point of view raises broader issues than those of simply constructing well-designed software systems. *Software Engineering Economics* therefore complements texts that deal with software design and development.

Each of these books discusses aspects of software engineering that cannot be found in any of the others. Is it possible to combine the strengths of all or some of these texts to build a stronger book or better framework for design? We present the strengths of OODA and OOMD, recognizing that the more ambitious task of addressing formal specification, end-user interests, and economic issues in the context of object-oriented design is beyond the scope of this review.

4. Concepts of Object-Oriented Programming

We review OODA's presentation of concepts of object-oriented programming, continue with design techniques taken from OOMD, and then return to OODA to discuss its extensive case studies. The introductory chapters of OODA nicely blend philosophy (for managing complexity and classification) with fundamental concepts, and establish a framework for the detailed examination of object-modeling techniques.

4.1. The Management of Complexity

What is software complexity, how is it related to the complexity of systems that it models, and why is object-oriented technology effective in managing software complexity?

The structure of complex systems in different domains is remarkably similar and may therefore be studied in a domain-independent manner. OODA, in chapter 1, describes the structural complexity of natural systems (living organisms and physical models of matter) and artificial

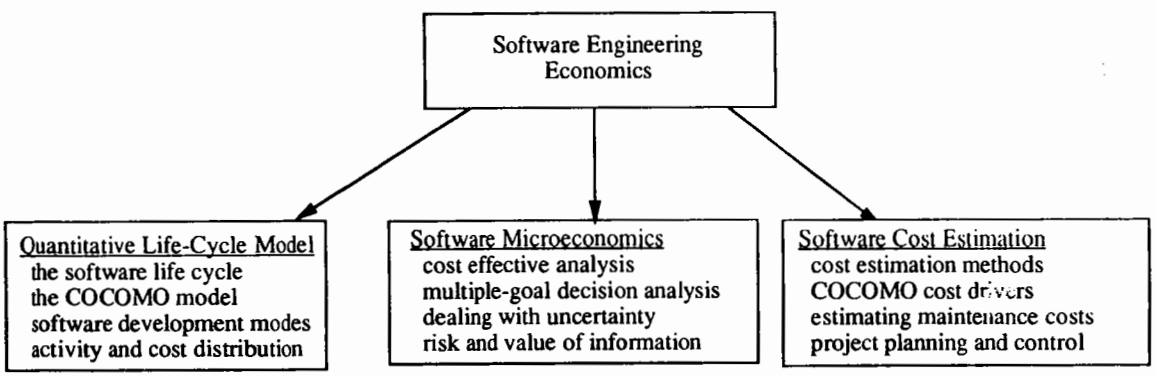

Figure 8: Software Engineering Economics

systems (personal computers and social institutions). Natural systems generally turn out to be well designed, while artificial systems are many orders of magnitude simpler if well designed than if poorly designed. However, software systems cannot be simpler than the behavior they model, and complexity is therefore an essential rather than an accidental property of software [Br]. Einstein's remark that nature should be described as simply as possible but no simpler has an artificial science analog that designs should be as simple as possible but no simpler.

Object-oriented technology is effective in managing software complexity because of its disciplined use of abstraction. An abstraction models an object, concept, or situation by its *relevant* attributes while ignoring *irrelevant* attributes. Abstractions are specified by finite sets of attributes, encapsulate their implementation, ignore attributes considered irrelevant, and permit adding further attributes through inheritance.

Abstractions are implemented by software components whose strong internal linkages and weak inter-component linkages permit independent specification and design. The problem domain is partitioned into collections of interacting but autonomous components that are both open (extendible) and closed (usable). Components may be related by "part of" and "kind of" relations ("has part" and "is a" hierarchies). Components may have static relations that determine potential behavior, and dynamic relations that realize actual behavior during computation.

Complex systems are built from primitive components by rules of composition. Application designers have great latitude in choosing the primitive components and linkages that express the structure of of computational models. Object-oriented design narrows the choice by requiring the primitives to be objects and classes, and provides guidance on how objects and classes may be chosen, but nevertheless leaves much flexibility in selecting specific primitives so that they simply and naturally model the application domain.

We strive to preserve simplicity by insisting on *simple* primitives and composition rules. However, computers extend our notion of what is simple by augmenting the limited human intellect with great manipulative power. Software components that are conceptually simple but internally complex may be included among the permissible primitives. More complex composition and computation rules (for example unification in logic programming) satisfy our notion of computational simplicity. Concurrent and distributed software components require new forms of abstraction primitives and composition rules to model synchronization and communication among autonomous components.

Small changes in primitives and composition mechanisms can greatly enrich the modeling technology without appreciably increasing the modeling complexity. In striving for a technology that increases our ability to model complexity we cannot abandon our requirement of simplicity of primitives and composition mechanisms, since this is fundamentally rooted in human limitations. However, richer primitives and composition mechanisms can increase our modeling power without violating underlying constraints of simplicity.

Algorithmic (procedure-oriented) decomposition results in hierarchies among procedure components very different from those among objects and classes of an object-oriented design. Algorithmic decomposition provides an alternative view of system structure with insights not readily derivable from an object-oriented decomposition. But object-oriented decomposition is higher-level, providing a basis for the encapsulation of data and procedures that hides data representation and provides clients with an abstract interface of accessible operations. Classes allow the common behavior of collections of objects to be modeled abstractly independently of the behavior of particular instances of the class. Objects and their classes have strong internal linkages and weaker inter-component linkages while providing a hook for hierarchies that capture generalization/specialization, incremental modification, and evolution. Object-oriented decomposition thus appears to satisfy general criteria for decomposition of complex systems better than algorithmic decomposition.

4.2. Fundamental Concepts

OODA traces the fundamental concepts of object-oriented programming in chapter 2 as a prelude to precisely specifying language mechanisms in chapter 3. In the late 1960s objects emerged simultaneously in many subfields of computer science, including computer systems, programming languages, programming methodology, and artificial intelligence. The concept of objects appeared in in capability-based operating systems, programming languages like Simula and Smalltalk, methodologies like information hiding and data abstraction, and frames and semantic nets in artificial intelligence. In the 1980s distributed networks of computers have provided an object-based hardware infrastructure and object-oriented databases are becoming an important technology for organizing information systems. Thus many different conceptual and technological trends appear to be reinforcing each other.

Object-oriented languages support modularity for decomposing a system into components, abstraction for specifying the behavior of components, and encapsulation as a technique for realizing abstraction through information hiding. Inheritance is a primary mechanism for managing hierarchies of abstractions. Typing is a mechanism for enforcing restrictions on the operands acceptable to an operator. Concurrency may be realized by *active objects* that may actively compute even when they do not receive messages, while persistent objects have a lifetime exceeding that of their creator and even that of the program in which they were created. A variety of object-oriented applications ranging from avionics and banking to robotics and VLSI is reviewed to illustrate that the conceptual universality claimed for object-oriented programming is being realized in practice.

4.3. Objects and Classes

The robustness of object-oriented technology is due in part to the strong correspondence between the informal and computational notion of objects. Informally an object can be ''a tangible thing'', ''an abstract entity'', or ''a goal towards which an action is directed''. Computational objects, defined in terms of their state, behavior, and identity, can uniformly model tangible, conceptual, and goal-oriented objects in terms of finite sets of attributes and operations. The state of a computational object is represented by instance variables and its behavior is represented by operations or methods that may be activated by messages to the object. Objects may be assigned to variables by *shallow assignment* that allows an object's state to be shared by several aliases or by *deep assignment* that assigns a new unshared copy of the object.

Objects that receive but do not send messages are called *server objects* (for example, queues with APPEND and REMOVE operations), while objects that send messages to server objects are called *clients*. Relationships among objects include *using relationships* between servers and clients and *containing relationships* between an object and its components. These relationships may be represented by graphs whose nodes are objects and whose edges represent using or containing relations.

Classes define the properties of sets of objects with a common structure and behavior. They have a common *public* interface for all objects of the class and may have a separate *protected* interface for subclasses and a *private* interface for internal operations of the class. Classes are the key abstractions for describing application domains and may be related in a variety of ways to form the class structure of a design. Relations among classes may include *kind of relations* (a vehicle is a kind of car), *using relations* (the class ''library'' uses the class ''book'' in its definition), *generic instantiation relations* (the generic class ''queue'' can be instantiated as a queue of customers or a queue of integers), and *metaclass relations* (between a class considered as an object and its metaclass that specifies operations applicable to the class). Relations among classes may be specified by graphs having classes as their nodes and various kinds of edges corresponding to the different kinds of relations that may obtain among classes.

Classes are generally static in that their behavior and relation to other classes does not change during the lifetime of a program, while objects may be created, modified, and destroyed during program execution. The primary relation between classes and objects is instantiation. Design is concerned primarily with the description of application domains by relations among classes, but may also require relations among objects and between classes and objects to be expressed. Thus graphs that contain both classes and objects and many different kinds of edges (relations) may be required during design.

Although design is an art, guidelines may be useful in assessing the quality of object-oriented designs. Well-designed classes should have relatively weak interclass *coupling* and relatively strong intraclass *cohesion* (natural dependencies among attributes of a class). Class attributes should satisfy criteria of *sufficiency* (capture enough behavioral attributes of the class to be usable) but need not satisfy criteria of *completeness* (support all conceivable contexts of use). Only primitive attributes not definable in terms of other attributes need be specified, but sometimes it is useful to directly specify nonprimitive attributes whose specification in terms of other attributes would be inefficient.

4.4. Classification

The use of classification as a principle for organizing knowledge dates back at least to Aristotle, but was reintroduced by Linnaeus in the 18th century for classifying organisms by their genus and species, and by Darwin in the 19th century in his theory of evolution by natural selection. Chemistry has likewise progressed from rudimentary classification of matter into earth, air, fire, and water by the Greeks, to classification into over 100 elements, and subclassification by isotopes. Mathematical set theory is built on a foundation of classification, and classification (realized by classes and inheritance) is conceptually central in object-oriented programming. It is no accident that the term "class" is derived from "classification".

Although the classes used in describing mature domains may appear to be cut and dried, developing a classification is in fact a creative and uncertain process involving arbitrary decisions concerning boundaries between classes, the granularity of classes, and the framework for classification. In biology, boundaries between species are sometimes arbitrary. Less than 2 million species are currently classified and the estimate of the total number of species ranges from 5 million to 50 million. The number of ways of classifying finite collections of objects depends exponentially on the number of objects. The classification appropriate to a particular context depends on the goals. For example, in classifying the set of digits {0 1 2 3 4 5 6 7 8 9} we can use *even, multiple of 3*, and *small* as alternative criteria. We leave it to the reader to determine the classification principle that yields the three classes {(1 2 6), (0 4 5 9), (3 7 8)}.

Object-oriented classes are specified by finite sets of attributes (operations) and may be classified by inheritance hierarchies. We distinguish between *first-order classification* of objects into classes with common structure and behavior and *second-order classification* of classes into subclasses through inheritance [We]. Classes serve to classify objects while inheritance serves to classify classes. Classifying objects into classes or types is a feature of any typed language, but the seamless blending of first- and second-order classification is a special feature of object-oriented languages. By supporting inheritance, object-oriented languages combine state transition, communication, and classification paradigms in a way that allows classification to play a greater role than in traditional languages.

Three approaches to classification are identified in OODA: specification by finite sets of properties (the class of polygons), by a concept (the class of well-written programs), and by a prototype (the class of tables). The first is the most precise and is also the method for object-based classification, but is more applicable to mathematical than to real-world objects.

Prototypical real-world classes such as *table* have fuzzy boundaries, and are called *natural kinds* because *tablehood* cannot be precisely defined in terms of other properties (see [We], page 551, for further discussion). This is a fundamental limitation on the power of object-oriented modeling. *Table* is a natural kind because no necessary and sufficient set of properties precisely characterizes the set of all tables. However, most natural kinds can be approximated by classes with finite sets of properties: having four legs that support a flat surface on which objects may be placed captures the properties of most, though not all, tables.

Object-oriented systems use sets of properties specified by classes to classify sets of objects. Because adding properties reduces the size of the set possessing the required properties, property sets and object sets satisfying the properties are said to be contravariant.

5. Object-Oriented Modeling and Design

Having established a framework for object-oriented design through the eyes of OODA, we examine the object-oriented modeling framework of OOMD. OOMD develops an object modeling technique (OMT) in terms of three kinds of models:

(1) Object Model: describes the structure of a system in terms of objects, classes, and their relationships

(2) Dynamic Model: describes the control and execution structure of a system in terms of events and states

(3) Functional Model: describes the functional (transformational) structure of a system in terms of functions and values

The three models provide complementary views of program structure and behavior for the following software development stages:

(1) Analysis: starting from a problem statement, build a precise model in terms of objects and classes of the problem domain

(2) System Design: design the logical architecture of the program and match it to the physical architecture of the hardware

(3) Object Design: starting from the products of analysis and system design, design the objects, data structures, algorithms, and communication protocols of the implementation

(4) Implementation: implement the objects and classes specified by the design in a particular programming language

The object model constructed during analysis is refined through system and object design into an implementation. The dynamic and functional models serve to focus on abstractions not explicitly represented in the object model. They are auxiliary models useful during analysis and design and for checking that the implementation has the desired dynamic and functional properties. But the implementation is generally developed from the object model making use of dynamic or functional models to fill in details and check correctness.

5.1. Visual Design Notations

Graphs are a versatile visual notation for representing structures of interrelated components. Components are generally specified by nodes of the graph, with relations among components being specified by edges. However, it is sometimes useful to *reify* relations, representing relations among components by tangible nodes and the components themselves by less tangible edges. The versatility of graphs arises from the variety of interpretations that may be associated with their nodes and edges.

The object, dynamic, and functional models of OOMD are all represented by graphs and differ merely in the way that nodes and edges are interpreted. The object model has graphs whose nodes are classes and/or objects. The dynamic model has *state transition* graphs whose nodes are states and edges are transitions or events. The functional model has *dataflow* graphs whose nodes are functions and edges represent streams of data values.

The graphs of dynamic and functional models have dual (mirror image) representations of data and functions. In the dynamic model data is represented by nodes (states) while functions are represented by edges (events, state transitions). In the functional model the nodes represent functions while the edges represent data streams. The dynamic model views the world from the point of view of data that is being acted on by functions, while the functional model views the world from the point of view of functions that are the agents of data transformation. Both these view are useful but neither pays sufficient attention to the interaction of programs and data.

The object model represents both data and function information in its nodes, and its edges can model the interaction of programs and data. The greater richness of interpretations makes the model more complex than dynamic and functional models, which restrict the interpretations of nodes and edges of the object model. But the greater simplicity of these two specializations can provide valuable insights not directly derivable from the object model itself.

A more general discussion of visual graph-based formalisms can be found in [Ha], which introduces generalized graphs (higraphs) whose nodes can represent sets of entities and whose edges can represent relations among sets. Sets are represented by contours that may contain nested contours representing subsets. Disjoint subsets (*or relations*) are represented by nonoverlapping contours, while aggregates with components (*and relations*) are represented by a visual crossproduct notation. Edges may represent arbitrary relations between the source and target sets, including functions that map every element of the source set into a unique element of the target set, or more complex relations.

Higraphs enrich traditional graph notation by allowing visual nesting, relations between classes as well as instances, and composition through *and* and *or* relations. Composite components constructed by *and* (cross-product) relations may act autonomously and therefore concurrently. OOMD represents class diagrams by higraphs whose nodes are classes and whose edges represents represent relations among classes. In the context of class diagrams, *or* relations represent *is-a* inheritance while *and* relations represent *has-part* aggregation.

Although our focus is on using visual notation for object-oriented systems, the dual problem of using object-oriented systems for visual notation is also interesting. Smalltalk has from its beginnings emphasized visual interfaces in its management of objects (the class browser) and its editing and debugging environments. Its *model-view-controller* (MVC) paradigm [LP] recognizes the distinction between *models* that specify objects or systems, *views* by which relevant attributes of models are displayed, and *controllers* that control the modification of models and views. Object-oriented systems facilitate the support of visual notations in a broader context than design, and may suggest uses of visual notation in design beyond object, dynamic, and functional models. General-purpose, object-oriented visual programming environments are beyond the scope of this review but certainly relevant to a comprehensive discussion of object-oriented software development environments.

5.2. The Object-Modeling Technique (OMT)

We review object, dynamic, and functional models and describe how they are used during design, analysis, and implementation.

5.2.1. The Object Model

The object model captures relations among object and classes by two kinds of *object diagrams*: *class diagrams* and *instance diagrams*. A class diagram specifies a collection of class templates that serves as a pattern (schema) for describing relations among objects generated as instances of the class. An instance diagram describes how a particular set of objects relate to each other, and may be used to document test cases or to help clarify particular features of a complex class diagram.

OMT distinguishes between *associations* that represent relations between classes in a class diagram and *links* that represent relations between objects in an instance diagram. Associations and links may have labels indicating the nature of the relation. For example, the classes *Person* and *Company* may be related by the relation *works for* or alternatively by the relation *buys from*. Cities and countries may be related by the relation *belongs to* or *is capital of*.

Associations describe the behavior of groups of links with a common structure and semantics, just as classes describe the behavior of groups of objects. Associations may be modeled by classes with finite sets of attributes, such as their *arity* (the number of entities they relate) and their multiplicity. Most associations may be decomposed into binary associations, but sometimes nondecomposable ternary associations are needed (programmer A uses language B on project C). The multiplicity of an association may be one-one, one-many, many-one, or many-many.

Entities linked by an association play roles that may be named by role names. Sometimes role and class names can be identified, but in some cases, for example when a binary association is between objects of the same class, a role name distinct from the class name is necessary. For example the association *works for* between two employees requires the role *employee-as-manager* to be distinguished from the role *employee-as-worker*.

The relations of aggregation and generalization/specialization are important kinds of associations represented in class diagrams. The *is-part-of* relation between components and the whole is the primary aggregation relation. One of the properties of aggregation is transitivity: if A is-part-of B and B is-part-of C, then A is-part-of C. Generalization/specialization, characterized by an *is-a* relation, is also transitive: if A is-a B and B is-a C then A is-a C.

Aggregation may be viewed as an *and relation* since all components must be present in the aggregate (a house has walls and doors and windows). Specialization is an *or relation* that specializes a general concept into precisely one of its alternative specializations (a car is a Toyota or a Chevy or a Porsche). In developing a design notation it is important to classify the kinds of associations being modeled and to determine properties that can be modeled as attributes of classes of an object-oriented design language.

5.2.2. The Dynamic Model

Dynamic modeling is realized by *state diagrams* with *states* whose state transitions are triggered by *events*. State diagrams have states as their nodes and state transitions representing events in the life of states as their edges. Events may have guards that specify the conditions under which the event can occur.

Causally connected events are sequential, while causally unrelated events are assumed to be concurrent for purposes of analysis but may be constrained to be sequential during design and implementation. Events with a common structure and behavior may be grouped into *event classes* (the event class *airplane flight departs* has an event instance *flight #123 departs from*

Chicago). Sequences of events that describe the execution of a process are called *scenarios*. The subsequence of events of a scenario that affects a given object is called the *event trace* of the object for the given scenario.

States and events may act on their environment by operations that are performed when the state or event obtains. States may be associated with ongoing *activities* such as display of a picture on a screen or ringing of a telephone. Events may be associated with instantaneous *actions* such as switching off the television or taking the phone off the hook. Actions may alternatively be associated with entry to or exit from a state (rather than with a transition between two states). For example, sending and receiving events for states that represent objects are associated with entry to and exit from the object.

State diagrams may be nested, for example by expanding a state associated with an activity into a more detailed state diagram. Events may be similarly expanded into state diagrams that realize an associated action. Grouping a set of states corresponding to a subtask into a superstate also results in a nested state diagram.

5.2.3. The Functional Model

Functional behavior is represented by graphs called *dataflow diagrams* whose nodes represent processes and whose edges represent data. Traditional dataflow diagrams that show the flow of data through processes are extended to include active objects (actors) that produce and consume values, and passive objects (data stores) that can store data for later use. The dataflow notation is further extended to include creation and subsequent use of objects. Nested dataflow diagrams permit processes represented by a single node to be expanded into dataflow diagrams. Control flows that show the sequencing of operations can be included in dataflow diagrams, but should be used sparingly since control is more appropriately modeled by dynamic models.

Operations of the functional model include queries, actions, and activities. Queries are read-only operations. Actions are operations that may change the state of the target object or have side effects on other objects. Activities are operations with duration in time that inherently have side effects.

The functional model shows *what* has to be done by a system, the dynamic model shows the sequence in which operations are performed, and the object model focuses on the doers (objects) on which the operations act. The three models are complementary abstractions of the same underlying software system, respectively viewing the system in terms of what its operations do, how its operations execute, and how its operations act on objects. OMT views the object model as the primary one for system development, but makes use of functional and dynamic models as valuable complementary abstractions.

5.3. From Analysis to Design and Implementation

5.3.1. Analysis

Object-oriented analysis starts from a problem description and develops an object model in terms of objects and classes of the domain. The first step of analysis is to understand the domain being modeled and to identify objects and classes of the domain. The next step is to identify associations among classes, to iteratively refine the class diagram, and to simplify the class diagram by factoring out common features of classes into superclasses. Once the class structure has been defined and is reasonably stable, dynamic and functional models for classes and subsystems may be developed. A dynamic model may be constructed with the aid of one or more scenarios that illustrate typical system interactions. Functional models specify important functions performed by the system by their inputs and outputs during analysis and refine input-output specifications into algorithms during design and implementation.

The automatic teller example introduced in chapter 8 of OOMD provides a nice running example that is nontrivial but not too complicated. It involves designing a computerized banking network including human cashiers and automatic teller machines (ATMs). Each bank has a local computer that holds local accounts and services its cashiers. ATMs are controlled by a central (consortium) computer that accesses accounts controlled by a local bank computer in performing automatic teller transactions.

Analysis yields the classes: Account, ATM, Bank, Cost-card, Cashier, Cashier-station, Central-computer, Customer, Transaction. The class diagram has associations like *Bank holds Account* and *ATM communicates with Central-computer*. Inheritance relations are introduced by factoring out common features of classes into superclasses. For example, the common features of *cashier transaction* and *ATM transaction* can be factored out into a superclass *transaction*.

A typical ATM scenario involves a user inserting a card, typing a password, selecting a transaction type, performing a transaction (say withdrawing cash), and receiving the cash and a receipt. The events at the interface must be supplemented by internal events such as verifying passwords, transmitting a request to the local computer, and withdrawing cash from the account.

The event sequences of scenarios are refined by identifying the objects involved in each event and constructing event traces for each object that indicate the event sequence in which that object participates. The objects involved in this scenario are the customer, an ATM, the central computer, and a local bank computer. The user transaction scenario determines an event trace for each of these four objects. A set of partial states and state transitions can be constructed from these event traces that can be extended into a state transition diagram that captures all possible event traces for each of the object classes. The state diagrams and the process of constructing them are nicely illustrated in the text.

The functional model specifies that the interaction of a user with the ATM involves entering a cash card, password, transaction type, and cash amount into the ATM and receiving cash and a receipt. This high-level functional specification can be broken down by specifying the system functions needed to realize the ATM behavior. The functions in this example are relatively simple, since the complexity arises from checking passwords, communicating with local banks, and checking account balances rather than from extensive transformation of data.

5.3.2. Design

Design includes *system design* to determine the system architecture followed by *object design* to specify the full definitions of the classes and associations identified during analysis. The object model, dynamic model, and functional models developed during analysis provide a basis for system design and then object design. System design decomposes the classes, objects, and associations into closely linked subsystems, each of which provides a specific set of system services. Systems may be decomposed horizontally into layers (hardware, operating system, programming language, application package), or vertically into slices (window graphics, dialog control, text editor). The relation between two subsystems can be client-server (client knows the server's interface) or peer-to-peer (each knows the other's interface).

The automatic teller system can be broken down into three subsystems: the ATMs, the central computer, and the bank computers. This hardware decomposition in turn determines a software decomposition by services provided, resulting in a correspondence between logical and physical system structure.

Object design carries most of the classes from analysis directly into classes of the design. But occasionally an analysis object is distributed among several design objects for efficiency and the design may contain auxiliary classes for purposes of implementation that have no counterpart is the implementation-independent analysis model.

The object, dynamic, and functional models developed during analysis may be combined into a single model during design. The object model is the controlling model in this process, with the dynamic model being used to develop control structures and the functional model being used to guide the construction of algorithms that implement methods. The dynamic and functional models serve to constrain the process of detailed design of classes and associations, and may be used to check the correctness of the detailed design.

5.3.3. Implementation

Implementation of a good design should be easy because all the difficult decisions should have been made during design. Implementation of an object-oriented design in an object-oriented programming language should be particularly easy since the design can be directly mapped into the language. However, programming is as much an art as design, and poor programming can ruin the efficiency, readability, reusability, and maintainability of the final product just as much as poor design. Thus a review of good programming style is included in OOMD to facilitate preserving the quality of a good design in the implementation. The languages C++, Eiffel, and Smalltalk are reviewed by showing sample code for implementing a simple graphics editor. Implementing object-oriented designs in the languages C and Ada and in a relational database system is also discussed.

6. Case Studies

OODA charts new ground in the detail with which it develops its case studies. Each is specified by verbal requirements, and is developed in a different object-based language. A final section for each case study explores potential changes in functionality of the completed designs to demonstrate that object-oriented designs support flexible enhancement and evolution. The case studies are carefully chosen to illustrate the flavor of applications in the areas of process control, scientific tools, information management, artificial intelligence, and command and control.

These five categories have proved useful in classifying the medical informatics applications described in [SPWF], and may provide a basis for defining a small number of *generic applications domains* with characteristic support tools that collectively cover most applications. Each generic domain could be supported by a generic environment: for example a process control environment with support for process control applications. The reader may wish to generalize from each prototypical application to explore the structure of the generic application domain it represents.

6.1. Home Heating System: A Process-Control Application in Smalltalk

The first application is a home heating system for regulating the flow of heat from a furnace to individual rooms of a home. This is a process-control application in the sense that it controls a temporally evolving process with a potentially infinite time horizon. The system includes a thermostat in every room with a device to set the thermostat and an infrared sensor to determine if the room is occupied. Unoccupied rooms are maintained at a lower temperature than occupied rooms according to a *living pattern* for each room that causes the temperature to be raised thirty minutes before expected occupancy. The requirements include a furnace activation and deactivation procedure, a furnace control switch, and various furnace-monitoring devices.

The author approaches the design problem pragmatically, first pointing out incompleteness and overspecification in the requirements and then examining the vocabulary of the problem space as a prelude to defining the objects and classes of the design. Domain analysis identifies tangible things in the problem domain such as rooms, sensors, and interface panels as potential classes and objects. It documents key abstractions in terms of their imports, exports, and behavior. The reuse of classes in the class library is illustrated by adaptation of the Smalltalk class *BooleanView* to model the class *Toggle Switch*. The key abstractions are determined to be

the home with its rooms, the furnace with its automated controls, and an operator interface with manual controls. The process structure and the associated analog and digital processors are specified and some general background on process control applications is given. The implementations of the furnace, the rooms, and the operator interface are developed in detail in a long section that includes much Smalltalk code.

The final section points out that changing from a simulated to a real home heating system requires very little change at the level of class specification. Substituting real rooms and furnaces for simulated ones requires only substitution for simulated objects of real objects with the same specification. Changing requirements to include homes with two furnaces rather than one is likewise relatively simple.

The application is nontrivial, but its complexity is manageable and its domain is sufficiently familiar that analysis does not require the aid of domain experts. Its concrete class and object diagrams provide the reader with a firm grip on the application of object-oriented methodology. However, there are many open questions and many unexplored alternatives.

What can we learn about the general nature of requirements specifications for process control from this particular example? Process-control requirements describing an already existing process for which controls are to be developed must inevitably be detailed. The informal natural-language specification of this example specifies not only functional requirements for the home heating system, but also their implementation. This simplifies the design task, since most objects of the implementation are already present in the design. But it overspecifies the problem since the functional requirements can be realized by a variety of alternative designs.

By varying the problem specification we can learn much about the reusability of the knowledge of this design in the design and implementation of related applications. Small variations, such as replacement of a simulated by a real system, are considered by the author. Larger variations that generalize the problem domain from home heating to home management could also prove instructive. How should a home be modeled in terms of its rooms, their contents, and house utilities like furnaces in developing a general house management system? Object-oriented designs lend themselves easily to such generalizations of the problem domain, and provide a systematic framework for reusability of behavior in new (anticipated and unanticipated) situations.

This design may be used as the basis for a variety of design exercises in a software engineering course. The advantages of using a well-documented, already solved problem are similar to those of using an already written program as the basis of a programming assignment.

6.2. Geometrical Optics Kit: A Scientific Application in Object Pascal

The second application addresses a specialized application domain that is a subdiscipline of physics. But its requirements place heavy emphasis on user interface technology and on computation-intensive processing involving dynamic objects. The design therefore involves the construction or reuse of utilities central to many other object-oriented applications.

The geometrical optics construction kit must model ray tracing for an object through a sequence of lenses. It must be able to select and position lenses, support dialogue to specify the focal length, use the mouse to drag, cut copy, clear and paste lenses in the McApp environment, and track the rays for an image as such changes are being made. The requirements include both mathematical ray-tracing formulae from optics and object-oriented user interface requirements.

The author discusses views and windows of the McApp class library, taking the opportunity to discuss properties of class libraries in general, and the object structure of menus, mice, and McApp documents. The class and object structure of the optics construction kit are then developed, including a discussion of the optics model, user interface classes, and user commands for controlling the experiment. The design of the user interface is developed in terms of designs for windows, menus, and dialogues. The implementation of user interface classes is described in

some detail. The design and implementation of the optics model is built on top of the user interface model. The ray-tracing algorithm, which makes use of both the user interface model and the optics model, is left till last.

Changes in functionality in ray tracing, number of experiments, and focal point representation can be easily accommodated. Even a radical requirement change to handle mirrors as well as lenses does not destroy the fundamental structure of the implementation.

6.3. Problem Reporting System: An Information Management Application in C++

The problem reporting system must cope with error reports from multiple software products, multiple versions of each product, and multiple (distributed) computers. The reports are stored in a problem database on arrival. They may be assigned for analysis and correction to human handlers, be updated as information about the error is accumulated, and cause an error report to be generated. The database is distributed to accommodate clients at multiple locations, and must be designed to handle changing data formats and changing hardware configurations.

The design and implementation assumes that error reports are stored in a commercial relational database system and accessed in SQL. The design deals both with the representation of the error reports in the database and with the higher-level object-oriented representation at the user interface. Since the database is distributed, the mapping of logical structure onto physical processes is nontrivial. Building a C++ application structure on top of the relational structure requires 20-30,000 lines of C++ code. However, the commonality among key abstractions realized by the object-oriented structure considerably reduces both the amount of code needed and the conceptual complexity.

This application affords an opportunity to introduce relational and object-oriented database technology in the context of a concrete problem. Classes such as customer, submitter, auditor, employee, problem, report, and product are identified and organized into hierarchies. A netmail system is used for reporting and communication in a distributed network. A version control mechanism is introduced to keep track of versions of error reports. The notions of utilities applicable to many classes (query, mail, and logging utilities) and of application generators for specializing generic utilities such as screen and report utilities are introduced.

Changes needed to split error reports referring to multiple versions and to group multiple reports of the same error are considered. A graphical user interface can be added in a modular fashion by replacing the existing user interface, hardly disturbing other system components.

6.4. Cryptanalysis: An Artificial Intelligence Application in CLOS

The cryptanalysis application employs heuristics to decode letter-for-letter substitution ciphers such as THE $\rightarrow$ UIF. Although substitution ciphers are simple, their deciphering requires powerful heuristics to make the task algorithmically tractable. A remarkably rich set of heuristics may be used to speed up the deciphering task, including knowledge of one- and two-letter words, letter frequencies, double letters, prefixes and suffixes, and semantic patterns. Backtracking to undo the effect of erroneous guesses is generally required.

The overall structure of knowledge-based heuristic problem-solving systems with backtracking is discussed. A blackboard framework is introduced with guesses from potentially concurrent multiple knowledge sources being posted on a blackboard. In this example, the blackboard is a tree of partially decoded solutions, and the knowledge sources are heuristics that may be applied concurrently to different parts of the text. Knowledge sources generally specify a precondition that obtains when the knowledge is applicable and an action to be undertaken when the precondition is met.

The design of the blackboard requires specification of classes and objects that may appear on the blackboard, such as sentences, words, and letters, and of operations for adding and

removing them. A class hierarchy of knowledge sources is defined that distinguishes between knowledge about sentences, words, strings, and letters. The application of knowledge to blackboard objects is controlled by an inference engine.

Introspective functionality can be easily added to the system to keep track of how the solution is arrived at. Changes needed to decipher languages other than English, to handle transposition ciphers as well as substitution ciphers, and to learn from experience can easily reuse language-independent decoding heuristics.

6.5. Train Traffic Management: A Command and Control Application in Ada

Command and control applications involve a centralized command center that controls a collection of distributed activities. In this example the command center is a network control system that controls a dispersed set of train controllers and a set of trains. The two primary functions of the system are train routing and train system monitoring. The coordination of train management activities is complex and includes traffic planning, train location tracking, traffic monitoring, collision avoidance, failure prediction, and maintenance logging.

The information on which these activities are based is collected by sensors on the train and on the tracks and by a NAVSTAR global positioning system. Train sensors include monitors for oil pressure, oil temperature, fuel quantity, electric power, throttle setting, engine RPM, water temperature, and drawbar force. Track sensors include position transponders placed every kilometer on the track and wayside interfaces placed at junctions and other critical points. The subsystems include a locomotive analysis and reporting system, energy management system, on-board display system, data management unit, and train-location tracking system.

This application requires a distributed communication network with networking, database, user interface, and device control subsystems. The key high-level abstractions are trains, tracks, and plans. Communication requires messages of many different kinds, each specified by a class. The database must keep track of current locations and planned routes, giving rise to classes associated with trains and plans. The interface could specialize class definitions of an existing window system to the particular requirements of monitor and plan displays. The sensors require classes for each type of sensing device.

Adding a payroll system that makes use of crew assignments to calculate hours worked, or integrating an expert system that advises on routing for given traffic conditions, and changing the target hardware, can be realized without changing the structure of the system.

7. Conclusion

The complementary views of design of OODA and OOMD suggests their joint use in courses on software engineering or by practitioners. In using the two texts together, the reader should probably start by skimming OODA for its insightful discussion of complexity and classification and its analysis of applications. The models of OOMD should then be mastered and its exercises may be used in class assignments. The reader can then go back to a deeper study of the applications of OODA, possibly using the models of OOMD as a basis for analysis.

A more advanced course could be organized around the five generic application areas of OODA. Students could be organized into five teams with the goal of designing generic application environments for the areas of process control, scientific tools, information management, artificial intelligence, and command and control. This final-project goal could be approached through a sequence of intermediate assignments, first reformulating the requirements and design of already described OODA applications, then designing a second application in each category chosen by the instructor, and only then developing a design for the generic application domain. Students within each group could specialize to handle parts of each design problem, each group could evaluate the design of another group, and common behavior spanning generic application

boundaries could be factored into superclasses. Well-documented application areas such as medical informatics could be used as a source for further prototypical examples. The problem of applications that span several generic application areas (radiology which spans scientific tools, information management, and process control) could be addressed as a problem of coordination among independently designed, autonomous large systems.

Both books may be viewed as innovative first drafts susceptible to improvement. OODA's informal style is sometimes imprecise, its design notation is not as well developed as that of OODA, its appendix on object-oriented programming languages is tentative, and the programs that implement its case studies could be developed more clearly. OOMD is uneven in its level of detail, exploring analysis in greater detail than design. It switches from the automatic teller example for analysis to an incomplete discussion of the OMTool example for design, so that the reader cannot trace the excellent analysis models developed for ATMs to the design and implementation phases. OMTool would in fact be a good running example to illustrate the object-oriented specification of visual languages for object-oriented design. Its discussion of implementation and case studies is unsystematic, as though the authors of OOMD had run out of steam.

Perhaps the authors should pool their resources and develop a joint follow-on text that combines (by multiple inheritance) the application expertise of OODA with the precise design notation of OOMD. Such reusability of of concepts and structure in the development of a new multiparadigm product is a recommended software engineering practice. However, multiparadigm efforts rarely yield products that are a seamless joining of the best features of component paradigms, and great care would be needed to combine the best features of these two path-breaking books, either by collaboration of the authors or by a third party.

OODA and OOMD will affect the practice as well as the teaching of software engineering. They inaugurate a new, more mature phase of object-oriented technology by concretely defining object-oriented design. The rough spots in these first-generation texts need to be smoothed out, but partly because of these books the concepts of object-oriented design are now as precisely defined as was structured design in the previous generation. A firm basis has been established for transferring object-oriented technology from the drawing boards to the workplace. OODA and OOMD provide a definitive framework for object-oriented design firmly rooted in applications that will accelerate the transformation of object-oriented design into a mature technology. The existence of a well-defined object-oriented development technology will impact future software engineering texts and change both the teaching and practice of software engineering.

8. References

[Bo] Barry Boehm, *Software Engineering Economics*, Prentice Hall, 1981.

[Booch] Grady Booch, *Object-Oriented Design with Applications*, Benjamin Cummings, 1991.

[Br] Fred Brooks, "No Silver Bullet: Essence and Accidents of Software Engineering", *IEEE Computer*, April 1987.

[CY] Peter Coad and Edward Yourdon, *Object-Oriented Analysis*, Yourdon Press, Prentice Hall International, 1990.

[Ha] David Harel, "On Visual Formalisms", *Communications of the ACM*, May 1988.

[LG] B. Liskov and J. Guttag, *Abstraction and Specification on Program Development*, MIT Press and McGraw-Hill, 1986.

[LP] Wilf Lalonde and John Pugh, *Inside Smalltalk*, Volume II, Prentice Hall, 1991.

[Me] Bertrand Meyer, *Object-Oriented Software Construction*, Prentice Hall International, 1988.

[RBPEL] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen, *Object-Oriented Modeling and Design*, Prentice Hall, 1991.

[Si] Herbert Simon, *The Sciences of the Artificial*, Second Edition, MIT Press, 1982.

[So] Ian Somerville, *Software Engineering*, Third Edition, Addison Wesley, 1989.

[SPWF] E. Shortliffe, L. Perrault, G. Wiederhold, and L. Fagan, *Medical Informatics*, Addison Wesley, 1990.

[We] Peter Wegner, "The Object-Oriented Classification Paradigm", in *Research Directions in Object-Oriented Programming*, Eds. Bruce Shriver and Peter Wegner, MIT Press, 1987.