Proceedings of ICAD2000
First International Conference on Axiomatic Design
Cambridge, MA – June 21-23, 2000
ICAD021

ENHANCING OBJECT-ORIENTED SOFTWARE DEVELOPMENT THROUGH AXIOMATIC DESIGN

Paul J. Clapis, Ph.D.
Metacom Inc.
PO Box 561
Southbury, CT 06488
pclapis@pantheon.yale.edu

Jason D. Hintersteiner
SVG Lithography Systems, Inc.
901 Ethan Allen Highway
Ridgefield, CT 06877
hintersj@svg.com

ABSTRACT

Several formal software design methodologies have evolved in the past few years to guide and document the development of object-oriented software. OMT (Object Modeling Technique) and UML (Universal Modeling Language) are two of the most widely used notations for graphically documenting the design of software objects. While these methodologies can convey significant software design detail, they do not provide a mechanism for documenting the functional requirements that drive the software development process. In addition, object-oriented design does not maintain traceability between functional requirements and the design of software objects.

In the past two decades, Axiomatic Design has emerged as a powerful tool for capturing the functional requirements and design parameters for a system. Due in part to its origins in mechanical engineering, Axiomatic Design has not been widely exploited in the software engineering field. In this paper we show how object-oriented software design can be enhanced by including Axiomatic Design in the formal design of object-oriented software systems. We will show the results of recent work which integrates the Axiomatic Design equations with OMT design documentation, enhancing the value of the software design process. A comparison will be made between the design parameters produced through Axiomatic Design and the classes and objects that result from the use of the OMT methodology.

Keywords: OMT, software design, object-oriented design, Axiomatic Design

1 INTRODUCTION

The explosive growth in computer processing capabilities during the last decade has led to an equally rapid expansion in the size and complexity of computer programs. To manage this complexity, computer scientists have developed a new paradigm known as object-oriented programming, which decomposes large programs into an interacting collection of modular software components. By encapsulating much of the software within private functions and providing simple public interfaces, objects seek to reduce the apparent software complexity while improving robustness and reducing maintenance costs. Not surprisingly, the design rules for object-oriented software correspond closely to the rules for efficient Design for Manufacture (DFM): develop a modular design, use standard components, and design parts for reuse.

The success of object-oriented programming has led to a proliferation of object-oriented design methodologies and notations such as Schlaer-Mellor, OMT, UML, Booch, Use Cases, Fusion and Catalysis [7]. These methodologies increasingly emphasize the value of formal software design strategies, narrowing the disparity between use of formal design approaches in software and more traditional engineering disciplines such as mechanics, electronics and optics.

Despite this maturation of the software design process, an important aspect has been largely neglected. Sparse attention has been paid to formalizing an approach to optimizing the design of software objects and minimizing coupling between them. Current textbooks focus on graphical design notations, coding standards and implementation issues but provide only anecdotal or heuristic guidance in partitioning software functionality or mapping functional requirements to object designs. In fact, all but one methodology (Use Cases, developed by Ivar Jacobsen) ignores the issue of software functional requirements altogether. Perhaps more importantly, little emphasis has been placed on designing software objects from a systems engineering perspective; that is, achieving an overall design that optimizes the performance of the system rather than the performance of individual objects.

Since its inception in the late 1980's, Axiomatic Design has provided a powerful set of guidelines to aid in developing optimal designs of physical hardware. Originally applied to the design of mechanical assemblies, awareness of Axiomatic Design has expanded to include other disciplines such as systems engineering and manufacturing [1]. In the past decade, significant interest has emerged in applying the principles of Axiomatic Design to software as well [2-6].

Axiomatic Design complements modern object-oriented software methodologies in several ways:

- It provides traceability between a system's functional requirements and the consequent design of each object's functionality
- It establishes metrics for optimizing the public interfaces between objects from the perspective of the overall system
- It encourages a recursive design process in which objects are hierarchically decomposed into smaller objects, with each level of the hierarchy mapping to a subsystem

Subsequent sections of this paper will describe these contributions of Axiomatic Design to the formal object-oriented design of large software systems.

2 BACKGROUND

Silicon Valley Group's Lithography division (SVGL) designs and manufactures state of the art lithographic instruments which are used in producing memory chips, integrated circuits and microprocessors. SVGL's instruments span many engineering disciplines including optics, mechanics, electronics and chemistry. The complexity of such hardware creates a correspondingly complex matrix of software requirements to control and manage the operation of the instrument.

In 1996, SVGL embarked on an effort to apply object-oriented software principles to the design and coding of a tool control system. This was part of a DARPA-sponsored initiative to develop an advanced software control system that could be extended to a variety of lithography tool platforms. The top-level requirements of this system included process recipe management, control of processing operations, interfaces to external users and factory host computers, and hardware diagnostics such as self-scheduled preventive maintenance and health checks. To manage this complexity, SVGL turned to object-oriented design to define the architecture of the software control system.

3 TRADITIONAL OBJECT-ORIENTED DESIGN APPROACH

The traditional sequence in object-oriented design is to design software objects iteratively from the top down, starting with the major subsystems and successively designing smaller and more specialized objects that represent increasingly fine levels of detail. The general sequence of steps is:

- Identify domain objects at this level
- Assign functionality to each object
- Define the public interface to each object
- Document this level of the design using a graphical notation such as OMT
- Code and test each object individually
- Integrate the objects and test them as a subsystem

OMT graphical notations provide a rich representation of each object's functionality and interfaces using diagrams known as object models. Other types of OMT models contribute additional information: dynamic models document the sequence of interactions between objects, their time-dependent behavior and control flow, while functional models capture the design of the objects' algorithms, data flows, behavioral states and state transitions. The popularity of object-oriented design stems in part from the effectiveness of these graphical models in communicating the software design to managers, engineers, and other stakeholders who are unfamiliar with the often-complex syntax of modern programming languages. By design, these object-oriented design models are language-independent, helping to separate the software design process from the ultimate implementation of the design in code. This approach is identical to the graphical nature of Axiomatic Design, which seeks to clarify and document the functional design of hardware components.

An important aspect of a software object's design is that it represents a conceptually complete package of functionality. Unlike traditional procedural computer programs, which provide few mechanisms for grouping functionality, object-oriented programs organize logically related functions into objects. Good object design dictates that strongly coupled functions should be incorporated together into an object, and each object should have minimal functional coupling to other objects. Because objects can interact only through explicitly defined functional interfaces, the strongly coupled functions in one object can be essentially hidden from other objects in the system. Axiomatic Design follows a similar convention, grouping coupled functionality (i.e. dependent FRs, or Functional Requirements) at the same level in the design hierarchy so that higher levels of the hierarchy can remain uncoupled [11]. As with object-oriented designs, this reduces the apparent complexity of the design. In essence, software objects combine dependent DPs into independent packages that satisfy functional hierarchies. From an Axiomatic Design perspective, a software object is essentially a package of dependent DPs.

Another strong similarity between Axiomatic Design and object-oriented design is their use of a hierarchy to represent increasingly detailed levels of the design. Both approaches use hierarchical representations to recursively decompose the system into increasingly fine-grained subsystems. Figure 1 shows a typical decomposition hierarchy of objects using OMT object model notation. The diamond symbol represents decomposition; that is, each object directly connected to the diamond is composed of the objects linked to the diamond by lines. OMT refers to this relationship as ownership or aggregation. Figure 1 indicates that lithography tool software is composed of machine mechanism objects, which in turn are composed of other machine mechanism objects.

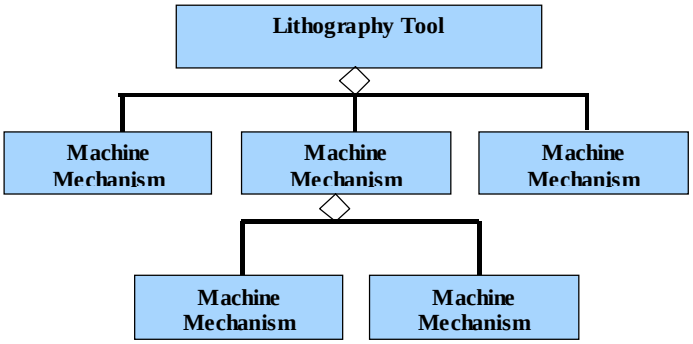

Figure 1: OMT Representation of hierarchical objects.

This similar use of a hierarchy does not end there. In Axiomatic Design, a system's FRs and DPs are represented as decomposition hierarchies [12]. The Axiomatic Design process encourages us to alternate between the functional and physical domains as we move down the hierarchy. This corresponds to the object-oriented design convention of designing software objects hierarchically, from the outside in; thus, the aggregation of objects represents a hierarchical decomposition of design parameters.

4 ENHANCING OBJECT ORIENTED DESIGN

Despite these similarities, object-oriented design lacks a fundamental feature: a formal process for optimizing the quality of the design. Object methodologies such as OMT do not directly address functional independence and information minimization, which are the fundamental axioms of Axiomatic Design. Nevertheless, these axioms can be adapted to the software design process in a straightforward way. The goal of the first axiom is to decouple the DPs of a design in order to maximize modularity; in object-oriented design this equates to minimizing coupling between classes. The second axiom encourages us to minimize the information content of our design. What does this mean in the context of a software design? The answer is clear when we consider the original intent of this axiom. In the physical world, the largest post-design cost of hardware is the expense of the manufacturing process. Minimizing the information content of a mechanical design results in reduced manufacturing cost. In the software world, however, the greatest post-design cost is maintenance – that is, modifying the software to meet new or changing requirements or to correct programming flaws. Object-oriented design has demonstrated that reduced maintenance is achieved by minimizing the *apparent* complexity of the software design – that is, by minimizing the public data and public functions in each class and using private data and private functions wherever possible. This principle is known as *information hiding*. In the context of software design, then, the second axiom is to minimize the public interface (i.e. public functions and public data) of each object.

We can now propose a revised sequence of steps to the object-oriented design process:

- Use Axiomatic Design to identify an optimally independent set of software FRs representing a subsystem
- For each top-level FR in this subsystem, design an object which satisfies it
- Design private data and methods which implement the FR
- Design public methods that present a minimal interface to other objects
- Document this level of the design using the Axiomatic Design equation, mapping FRs to software objects
- Code and test each object individually
- Integrate the objects and test them as a subsystem
- Iterate / decompose the FR hierarchy as necessary

This sequence enhances the steps described previously in section 3 in several key ways. By defining FRs that are optimally independent, we can then design software classes that each satisfy one and only one top-level FR. This guarantees that our software objects are modular and have minimal coupling to other objects. By documenting the corresponding Axiomatic Design equation for this level of the design hierarchy, we produce a requirements traceability matrix that aids maintenance by identifying which objects satisfy each requirement. The design equation also assures that all FRs are addressed by the design.

Fundamentally, this sequence is identical to the standard Axiomatic Design "zigzagging" process of hierarchically identifying hardware FRs and designing corresponding DPs. In both the hardware and software domains, we move back and forth between the FR hierarchy and DP hierarchy as we iterate to lower and lower levels. However, it is important to note that the resultant hierarchy of software FRs does not always correspond directly to the hierarchy of hardware FRs. Nevertheless, it is an established convention to design software objects that correspond to objects in the hardware domain, in part because this improves the ability of other engineers to understand the software architecture. Recent work has been done to apply Axiomatic Design from a systems engineering perspective, combining the FR definition process for both hardware and software into a hierarchy of subsystems[13-14].

5 AN EXAMPLE DECOMPOSITION

To demonstrate the process discussed above, we will present a decomposition of one small part of a lithography tool known as a wafer prealigner. Here is an excerpt of the FR and corresponding DP at this level of the hierarchy [15]:

FR.2.1: Determine and correct the wafer rotational position and centering offsets prior to lithographic processing

DP.2.1: Design a Wafer Prealigner device which performs a coarse alignment of the wafer to put it into the proper rotational position for the wafer stage. In addition to a wafer rotator, the prealigner consists of an optical sensor which is used to detect the location of the notch in the wafer, and serves as a source of

information on the wafer's offset from center, based on its spin characteristics.

At this level, the hardware solution is to design a wafer prealigner assembly. Following our object-oriented design sequence, we would design a corresponding Wafer Prealigner software class whose sole responsibility is to encapsulate the software necessary to control the prealigner hardware assembly and determine the notch location. Note that the Axiomatic Design justification for this decision is identical for both hardware and software:

Hardware perspective: The Wafer Prealigner hardware assembly satisfies one and only one FR, to orient the wafer's rotational position prior to lithographic processing. Subsequent changes to the design of this hardware should have minimal impact on the design of other hardware components.

Software perspective: The Wafer Prealigner software object satisfies one and only one FR, to control the wafer prealigner hardware and determine the location of the wafer notch. Subsequent changes to the design of this software object should have minimal impact on the design of other software components.

Moving down a level, we can decompose FR2.1 into five software subrequirements:

FR.2.1.1: Rotate the wafer in order to find the notch and to subsequently orient the wafer

FR.2.1.2: Maintain the lateral position of the wafer while rotating so that no loss of x-y position information occurs

FR.2.1.3: Detect the position of the wafer edge as the wafer is rotated

FR.2.1.4: Calculate the notch location from the detected edge information

FR.2.1.5: Calibrate the wafer prealigner hardware

Following our axiomatic software design strategy, these FRs lead us to define five corresponding software classes:

DP.2.1.1: Wafer Prealigner Rotator – controls the rotation of the wafer turntable

DP.2.1.2: Pneumatic Wafer Holder – controls a pneumatic (vacuum) valve that clamps the wafer to the turntable, preventing lateral sliding of the wafer during rotation. Also contains a vacuum sensor which confirms that the wafer is adequately retained

DP.2.1.3: Wafer Edge Detector – acquires data from the optical sensor, which is a linear array of optical detectors. The raw sensor data is processed by the detector object and transformed into form that represents the position of the wafer edge as a function of rotational position

DP.2.1.4: Wafer Geometry Analyzer – algorithmically processes the wafer edge data from the detector object and determines the angular location of the wafer notch

DP.2.1.5: Wafer Prealigner Calibrator – uses the other four objects to rotate and acquire data from a special calibration disk, then determines physical parameters such as spacing of the optical sensor so that the analyzer can subsequently report the notch location in calibrated units

Table 1 shows these FRs and DPs in condensed form. We can see that the software DPs mimic and are closely coupled to their corresponding hardware DPs. For example, if the hardware engineer chooses to satisfy FR.2.1.2 by designing a pneumatic clamp that holds the wafer by applying a vacuum to its back surface, the corresponding software object is then responsible for controlling the vacuum valve and sensing the vacuum pressure. Alternatively, the hardware engineer might choose to use a mechanical edge-gripping device, which would require a totally different software object. Historically, we have found that hardware design often precedes the design of software, resulting in a domain-sequential design process. While this may be appropriate in some circumstances, we believe that a superior approach is to perform Axiomatic Design from a systems perspective, addressing the FRs of both hardware and software at each level. This holistic process can produce a more optimal design, since co-dependencies and tradeoffs between hardware and software can be recognized earlier in the design process [14].

Table 1: FRs and DPs for the wafer prealigner software.

	Functional Requirements (FRs)	Design Parameters (DPs)
2.1.1	Control prealigner rotaton	Wafer prealigner rotator
2.1.2	Maintain lateral position	Pneumatic wafer holder
2.1.3	Detect wafer edge positions	Wafer edge detector
2.1.4	Calculate notch location	Wafer geometry analyzer
2.1.5	Calibrate prealigner hardware	Wafer prealigner calibrator

The following equation displays the corresponding design matrix for the wafer prealigner software. The off-diagonal elements in the first four rows and columns of the coupling matrix are zero, indicating that no coupling exists between these FRs and DPs. The last row of the matrix is non-zero in all positions, reflecting the fact that calibration of the prealigner hardware requires use of all five software objects. Although a calibration procedure normally precedes other operations, we have positioned calibration in the last row to show that the design equation represents a decoupled design.

$$\begin{Bmatrix} FR.2.1.1 \\ FR.2.1.2 \\ FR.2.1.3 \\ FR.2.1.4 \\ FR.2.1.5 \end{Bmatrix} = \begin{bmatrix} X & O & O & O & O \\ O & X & O & O & O \\ O & O & X & O & O \\ O & O & O & X & O \\ X & X & X & X & X \end{bmatrix} \begin{bmatrix} DP.2.1.1 \\ DP.2.1.2 \\ DP.2.1.3 \\ DP.2.1.4 \\ DP.2.1.5 \end{bmatrix}$$

Just as the five lower-level FRs are an Axiomatic Design decomposition of FR.2.1, the five lower-level software classes (rotator, holder, detector, analyzer and calibrator) represent a decomposition of the Wafer Prealigner class, i.e. they each represent a specific, modular portion of the wafer prealigner object's functional responsibility. Since they are contained within and accessed only by the wafer prealigner object, they can be hidden from the rest of the lithography tool control software, simplifying the apparent complexity of the overall software. In this way, a good object-oriented software design improves maintainability. Since each object represents an encapsulated body of functionality, the software objects can be reused on other projects, reducing future software development costs.

The design of these classes as well as their interfaces and coupling between classes has been documented using OMT notation. A fully functional prototype of the design has been coded in the Smalltalk programming language and has been demonstrated to the project sponsors. An Axiomatic Design decomposition of the lithography tool control system was generated as part of this project and is now contributing to the design of subsequent projects.

6 SUMMARY AND CONCLUSIONS

Object-oriented software design provides a powerful improvement over traditional software development methodologies by organizing and encapsulating data and functionality into modular, reusable packages. Axiomatic Design complements object-oriented design by providing a straightforward, repeatable process for defining the functionality of each software object. The result is an object design process that is more provably correct and less dependent on the creative judgment of the developer.

We have shown that the fundamental axioms of Axiomatic Design can be interpreted within the context of object-oriented software design, leading to a recursive design process in which objects are hierarchically decomposed into smaller objects, with each level of the hierarchy mapping to a set of independent functional requirements. Documenting an object design in the form of an Axiomatic Design equation quantifies the quality of the design and documents traceability to the original functional requirements.

While a good object-oriented design consists of software classes that are minimally coupled to each other, the software classes representing hardware are strongly tied to the hardware design. Our experience with applying the Axiomatic Design process to object-oriented software suggests that the best software designs are produced through concurrent engineering, where the design of hardware DPs and software objects are pursued simultaneously. Such a systems approach to design results in an architecture where software functionality is integrated into every level of the design hierarchy.

7 ACKNOWLEDGMENTS

The authors would like to acknowledge the many people at SVG Lithography Systems, Inc., including Ed Duwel and the engineers involved with the MAL program, that provided the support and input necessary to make this effort possible.

In addition, the authors would like to acknowledge Professor Nam P. Suh at the Massachusetts Institute of Technology for his comments and insight.

8 ABOUT THE AUTHORS

Paul J. Clapis received a Ph.D. in physics from the University of Connecticut, and holds several patents in software applications for the semiconductor industry. He is president of Metacom Inc., a software development and consulting firm based in Connecticut which provides object-oriented training, mentoring and software development in C++, Smalltalk and Java.

Jason D. Hintersteiner received a Bachelor of Science and Master of Science in Mechanical Engineering at the Massachusetts Institute of Technology. His experience includes research projects in robotics, digital control, error modeling and compensation, computer networking, as well as two years of postgraduate work with Professor Nam P. Suh at MIT on the application of Axiomatic Design to large-scale systems. He is currently a Senior Staff Engineer at SVG Lithography Systems, Inc., and is chiefly responsible for providing systems engineering support and coordinating the implementation of Axiomatic Design throughout the engineering organization.

9 REFERENCES

[1] Hintersteiner, J. D. "Addressing Changing Customer Needs By Adapting Design Requirements", *Proceedings of the 4th International Conference on Engineering Design and Automation*, Miami, FL. August 2000.

[2] Do, S.H. and Suh, N. P. "Design of Object-Oriented Software Systems using the Axiomatic Design Framework", publication pending

[3] Kim, S. J., Suh, N. P., and Kim, S. K. "Design of Software Systems Based on Axiomatic Design", *Annals of the CIRP*, Vol. 40, No. 1, pp. 165-170, 1991

[4] Park, G. J. "Axiomatic Design vs. Software Engineering", *NSF Sponsored Axiomatic Design Workshop for Professors*, MIT, Cambridge, MA, USA June 1998

[5] Do, S. H. and Park, G. J. "Application of Design Axioms for Glass-Bulb Design and Software Development for Design Automation", *Third CIRP Workshop on Design and Implementation of Intelligent Manufacturing*, pp. 119-126, June 19-22, Tokyo, Japan, 1996

[6] Do, S. H., Tate, D., Harutunian, V., and Suh, N. P. "Axiomatic Design Software", *NSF Sponsored Axiomatic Design Workshop for Professors*, MIT, Cambridge, MA, USA June 1998 (unpublished copyrighted software)

[7] Pressman, R. S. "Software Engineering, A Practitioner's Approach", 4th edition, McGraw Hill, 1997

[8] Cox, B. J. "Object-Oriented Programming", Addison-Wesley, 1986J.

[9] Booch, G. "Object-Oriented Analysis and Design With Applications", 2nd edition, The Benjamin / Cummings Publishing Company Inc., 1994

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

[11] Suh, N. P. "Principles of Design", Oxford University Press, 1990, p. 28

[12] Ibid, p. 36

[13] Hintersteiner, J. D. "A Fractal Representation for Systems", *Proceedings of the 1999 International CIRP Design Seminar*, Enschede, the Netherlands. March 24-26, 1999.

[14] Hintersteiner, J. D. and Nain, A. "Integrating Software into Systems: An Axiomatic Design Approach", *Proceedings of the 3rd International Conference on Engineering Design and Automation*, Vancouver, B. C. Canada. August 1-4, 1999.

[15] DelPuerto, S. and Garcia, J. "Axiomatic Redesign of Guide Flexures: Improving product reliability and reducing manufacturing cost", *Proceedings of the First International Conference on Axiomatic Design*, Cambridge, MA June 21-23, 2000.

