Proceedings of ICAD2000
First International Conference on Axiomatic Design
Cambridge, MA – June 21-23, 2000
ICAD027

OBJECT-ORIENTED SOFTWARE DESIGN WITH AXIOMATIC DESIGN

Sung-Hee Do
Dosh@axiod.com
Vice President, Technology
Axiomatic Design Software, Inc.
221 N. Beacon St.
Boston, MA 02135-1943, U.S.A.

Nam P. Suh
Npsuh@mit.edu
The Ralph E. & Eloise Cross Professor
Department of Mechanical Engineering
Massachusetts Institute of Technology
77 Massachusetts Ave. Rm 3-173
Cambridge, MA 02139, U.S.A

ABSTRACT

This paper presents a new software design methodology based on axiomatic design theory that incorporates object-oriented programming. This methodology overcomes the shortcomings of various software design strategies – extensive software development and debugging times and the need for extensive maintenance – since it is not heuristic and provides basic principles for good software systems. A simple software program is presented here as a case study following the methodology. This case study shows the systematic nature of axiomatic design that has been generalized and can be applied to all different designs. The axiomatic design framework for software overcomes many of the shortcomings of current software design techniques: high maintenance costs, limited reusability, the need for extensive debugging and testing, poor documentation, and limited extensibility of the software, in addition to high development cost of software. The methodology presented in this paper has helped software engineers to improve productivity and reliability.

Keywords: design, axioms, software, object-oriented

1 INTRODUCTION

Both the importance and high cost of software are well recognized. The high cost is associated with the long software development and debugging time, the need for maintenance, and uncertain reliability. It is a labor-intensive business that is in need of a systematic software engineering approach that ensures high productivity and reliability of software systems a priori. The goals of software engineering should be two: first to enhance algorithmic efficiency so as to reduces execution time and the other to enhance productivity so as to reduce the coding, extension, and maintenance effort. As computer hardware rapidly evolves and the need for large-scale software systems grows, productivity is increasingly more important in software engineering. The so-called "software crisis" is closely tied to productivity of software development [Pressman, 1997].

Several design methodologies for software systems have been proposed in the past. Two decades ago, structured methods, such as structured design and structured analysis, were the most popular idea [DeMarco, 1979]. As the requirement for productive software systems has increased, the object-oriented method has become the basic programming tool [Cox, 1986]. It emphasizes the need to design software right during the early stages of software development and the importance of modularity. However, even with object-oriented methods, there are many problems that intelligent software programmers face in developing and maintaining software over its life-cycle. Although there are several reasons for these difficulties, the main reason is that the current software design methodology has a difficulty to explain the logical criterions about good software design. Modularity alone does not ensure good software, since even a set of independent modules can couple software functions.

The concept of the AD framework has been successfully applied to software design [Kim, et al, 1991][Do and Park, 1996][Do, 1997]. The basic idea used for the design and development of software systems is exactly the same as that used for hardware systems and components, and thus the integration of software and hardware design becomes a straightforward exercise.

The methodology presented in this paper for software design and development uses both the AD framework and the object-oriented method. It consists of three steps. First, it designs the software system based on axiomatic design, i.e., decomposition of FRs and DPs, the design matrix, and the modules as defined by axiomatic design [Suh, 1990 and 2000]. Second, it represents the software design using a full design matrix table and a flow diagram, which provide a well-organized structure for software development. Third, direct building the software code based on a flow diagram using the object-oriented concept. This axiomatic approach enhances software productivity since it provides the roadmap for designers and developers of the software system and eliminates functional coupling.

2 OBJECTED-ORIENTED SOFTWARE DESIGN USING AXIOMATIC DESIGN[1]

Based on Axiomatic Design and object-oriented method, we have developed a generic approach to software design. The software system is called 'Axiomatic Design of Object-Oriented Software Systems (ADo-oSS)' that can be used by any software designers. ADo-oSS is a major new paradigm shift in the field of software engineering. It combines the power of axiomatic design

[1] This section is also presented in CIRP paper: Suh, N.P. and Do, S.H., "Axiomatic Design of Software Systems", *CIRP Annals*, Vol. 49, 2000.

with the popular software programming methodology called object-oriented programming technique (OOT) [Rumbaugh, et al, 1991][Booch, 1994]. The goal of ADo-oSS is to make the software development a subject of science rather than an art and thus reduce or eliminate the need for debugging and extensive changes.

ADo-oSS utilizes the systematic nature of axiomatic design, which can be generalized and applied to all different design tasks, and the infrastructure created for object-oriented programming. It overcomes many of the shortcomings of the current software design techniques which result in high maintenance cost, limited reusability, extensive need to debug and test, poor documentation, and limited extensionality of the software. ADo-oSS overcomes these shortcomings.

One of the final outputs of ADo-oSS is the system architecture, which is represented by the Flow Diagram. The flow diagram can be used in many different applications for a variety of different purposes such as:

a. Improvement of the proposed design through identification of coupled designs.
b. Diagnosis of the impending failure of a complex system.
c. Reduction of the service cost of maintaining machines and systems.
d. Engineering change orders.
e. Job assignment and management of design tasks.
f. Management of distributed and collaborative design tasks.
g. Reusability and extensionality of software.

In axiomatic design a 'module' is defined as the row of design matrix that yields the FR of the row when it is multiplied by the corresponding DP (i.e., data). The AD framework ensures that the modules are correctly defined and located in the right place in the right order. A 'V model for software' shown in Figure. 1 [modified from El-Haik, 1999] will be used here to explain the concept of axiomatic design of object-oriented software systems (ADo-oSS). The first step is to design the software following the top-down approach of axiomatic design, build the software hierarchy, and then generate the full design matrix (i.e., design matrix that shows the entire design hierarchy) to define modules. The final step is to build the object-oriented model with a bottom-up approach, following the AD flow diagram for the designed system.

Axiomatic design of software can be implemented using any software language. However, in the 1990's most software is written using an object-oriented programming language such as C++ or Java. Therefore, axiomatic design of software is implemented using object-oriented methodology.

To understand ADo-oSS, it is necessary to review the definitions of the words used in OOT and their equivalent words in axiomatic design. The fundamental construct for the object-oriented method is the *object*[2], which is equivalent to FRs. Object-oriented design decomposes a system into objects. Objects '*encapsulate*' both data (equivalent to DPs), and method (equivalent to relationship between FRi and DPi, i.e., module) in a single entity. Object retains certain information on how to perform certain operations, using the input provided by the data and the method imbedded in the object. (In terms of axiomatic design, this is equivalent to saying that an object is [FRi = Aij DPj].)

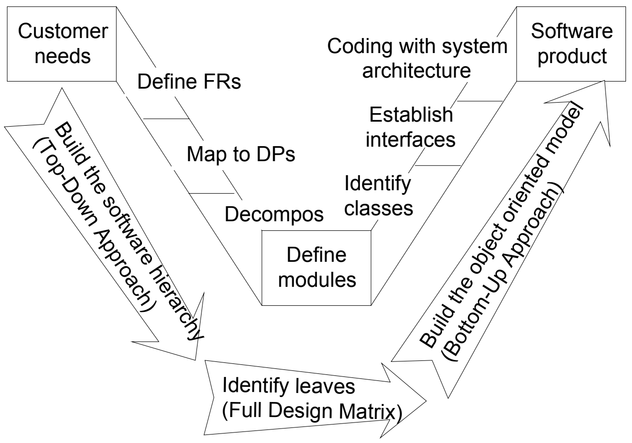

Customer needs
Define FRs
Map to DPs
Decompos
Define modules
Identify classes
Establish interfaces
Coding with system architecture
Software product
Build the software hierarchy (Top-Down Approach)
Identify leaves (Full Design Matrix)
Build the object oriented model (Bottom-Up Approach)

Figure 1: Axiomatic Design Process for Object-Oriented Software System (The V model)

Object-orient design generally uses four definitions to describe its operations: *identity*, *classification*, *polymorphism* and *relationship*. Identity means that data – equivalent to DPs -- are incorporated into specific objects. Objects are equivalent to a FR -- with a specified [FRi = Aij DPj] relationship-- of axiomatic design, where DPs are data or input and Aij is a method or a relationship. In axiomatic design, the design equation explicitly identifies the relationship between FRs and DPs. Classification means that objects with the same data structure (attributes) and behavior (operations or methods) are grouped into a class. The object is represented as an *instance* of specific *class* in programming languages. Therefore, all objects are instances of some *classes*. A class represents a template for several objects and describes how these objects are structured internally. Objects of the same class have the same definition both for their operations and for their information structure.

Sometimes an 'Object' is also called a tangible entity that exhibits some well-defined 'Behavior'. 'Behavior' is a special case of FR. The relationship between 'Objects' and 'Behavior' may be compared to the decomposition of FRs in the FR hierarchy of axiomatic design. 'Object' is the 'parent FR' relative to 'Behavior' which is the 'child FR'. That is, the highest FR among the two layers of decomposed FRs is 'Object' and the children FRs of the 'object FR' are 'Behavior'.

The distinction between '*Super Class*', '*Class*', '*Object*' and '*Behavior*' is necessary in OOT to deal with FRs at successive layers of a system design. In OOT, Class represents an abstraction of *Objects* and thus, is at the same level as an Object in the FR hierarchy. However, *Object* is one level higher than *Behavior* in the FR hierarchy. The use of these key words, while necessary in OOT, adds unnecessary complexity when the results of axiomatic design is to be combined with OOT. Therefore, we will modify the use of these key words in OOT.

In ADo-oSS, the definitions used in OOT are slightly modified. We will use one key word 'Object' to represent all levels of FRs, i.e., Class, Object, and Behavior. 'Objects with indices' will be used in place of these three key words. For example, *Class* or *Object* may be called *Object i*, which is equivalent to FRi, *Behavior*

[2] Italicized words in this section have specific definitions.

will be denoted as '*Object ij*' to represent the next level FRs, FRij. Conversely, the third level FRs will be denoted as *Object ijk*. Thus, Object i, Object ij, and Object ijk are equivalent to FRi, FRij, and FRijk, which are FRs at three successive levels of the FR hierarchy.

To summarize, the equivalence between the terminology of axiomatic design and those of OOT may be stated as:

- A FR can represent an Object.
- DP can be data or input for the Object, i.e., FR.
- The product of a module of the design matrix and DP can be a method, i.e., FR = A*DP.
- Different levels of FRs are represented as Objects with indices.

The Axiomatic Design of Object-Oriented Software System (ADo-oSS) shown in Figure 1 involves the following steps:

a. *Define FRs of the Software System*

The first step in designing a software system is to determine the customer attributes, in the customer domain, which the software system must satisfy. Then, the functional requirements (FRs) of the software in the functional domain and constraints (Cs) are established to satisfy the customer needs.

b. *Mapping between the Domains and the Independence of Software Functions*

The next step in axiomatic design is to map these FRs of the functional domain into the physical domain by identifying the design parameters (DPs). DPs are the 'how's' of the design that satisfy specific FRs. DPs must be chosen to be consistent with the constraints.

c. *Decomposition of {FRs}, {DPs}, and {PVs}*

The FRs, DPs, and PVs must be decomposed until the design can be implemented without further decomposition. These hierarchies of {FRs}, {DPs}, {PVs} and the corresponding matrices represent the system architecture. The decomposition of these vectors cannot be done by remaining in a single domain, but can only be done through zigzagging between domains.

d. *Definition of Modules – Full Design Matrix*

One of the most important features for the AD framework is the design matrix, which provides the relationships between the FRs and DPs. In the case of software, the design matrix provides two important bases in creating software. One important basis is that each element in the design matrix can be a method (or operation) in terms of the object-oriented method. The other basis is that each row in the design matrix represents a module to satisfy a specific FR when a given DP is provided. The off-diagonal terms in the design matrix are important since the sources of coupling are these off-diagonal terms.

It is important to construct the full design matrix based on the leaf-level FR-DP-Aij to check for consistency of decisions made during decomposition.

e. *Identify objects, attributes, and operations*

Since all the DPs in the design hierarchy are selected to satisfy FRs, it is relatively easy to identify the objects. The leaf is the lowest level Object in a given decomposition branch, but all leaf-level objects may not be at the same level if they belong to different decomposition branches. Once the Objects are defined, the attributes (or data) – DPs -- and operations (or methods) – products of module times DPs -- for the Object should be defined to construct the object model. This activity should use the full design matrix table.

The full design matrix with FRs and DPs can be translated into the OOT structure as shown in Figure 2.

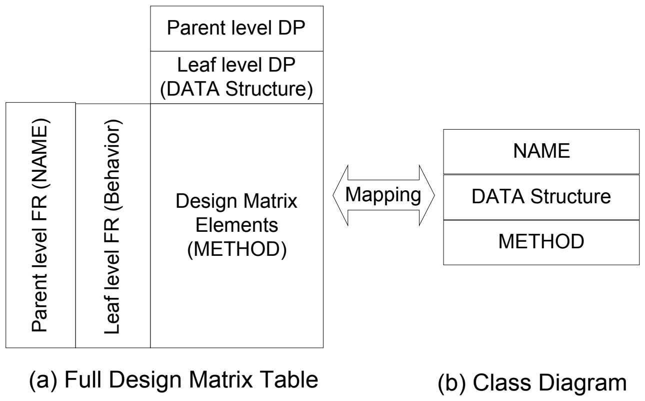


Figure 2: The correspondence between the full design matrix and the OOT diagram

f. *Establish interfaces by showing the relationships between objects and operations*

Most efforts are focused on this step in the object-oriented method since the relationship is the key feature. The axiomatic design methodology presented in this paper utilizes the off-diagonal element in the design matrix as well as the diagonal elements at all levels. A design matrix element represents a link or association relationship between different FR branches that have totally different behavior.

The sequence of software development begins at the lowest level, which is defined as the leaves. To achieve the highest-level FRs, which are the final outputs of the software, the development of the system must begin from the inner-most modules shown in the flow diagram that represent the lowest-level leaves. Then, move to the next higher level modules (i.e., next innermost box) following the sequence indicated by the system architecture; that is, go from the innermost boxes to the outer most boxes. In short, the software system can be developed in the following sequence:

a. Construct the core functions using all diagonal elements of the design matrix.
b. Make a module for each leaf FR, following the sequence given in the flow diagram that represents the system architecture.
c. Combine the modules to generate the software system, following the module junction diagram.

When this procedure is followed, the software developer can reduce the coding time since the logical process reduces the software construction into a routine operation.

3 EXAMPLE – SIMPLE DRAWING PROGRAM

In the preceding section, the basic concept for designing software based on Axiomatic Design of Object-Oriented Software Systems (ADo-oSS) was presented. In this section, a case study involving simple drawing software designed based on ADo-oSS will be presented.

a. *Define FRs of the Software System*

Let us assume the customer attributes as follows:

Table 1. Customer Needs

CA1	We need software to draw a line or a rectangle or a circle at a time
CA2	The software should work with mouse using push, drag, and release action

Then, the desired first level functional requirements of the software can be described in Table 2.

Table 2. First Level FRs

FR1	Define element
FR2	Specify drawing environment

b. *Mapping between the Domains and the Independence of Software Functions*

The mapping for the first level can be derived as shown in Table 3. The upper character in design matrix area represents diagonal relationship and the lower character means off-diagonal relationship.

Table 3. Mapping for the First Level

	DP1: Element characteristics	DP2: GUI with window
FR1: Define element	A	0
FR2: Specify drawing environment	a	B

c. *Decomposition of {FRs}, {DPs}, and {PVs}*

The entire decomposition information can be summarized as follows. Figure 3 illustrates the entire design hierarchy.

Table 4. Second Level Decomposition

	DP11: Line characteristic	DP12: Rectangle characteristic	DP13: Circle characteristic
FR11: Define line element	C	0	0
FR12: Define rectangle element	0	D	0
FR13: Define circle element	0	0	E

	DP21: Radio buttons	DP22: Mouse click information	DP23: Drawing area (ie. Canvas)
FR21: Identify the drawing type	F	0	0
FR22: Detect drawing location	b	G	0
FR23: Draw a element	c	0	H

Table 5. Third Level Decomposition

	DP111: Start point	DP112: End point
FR111: Define start	I	0
FR112: Define end	0	J

	DP121: Upper left point	DP122: Lower right point
FR121: Define upper left corner	K	0
FR122: Define lower right corner	0	L

	DP131: Center point	DP132: radius
FR131: Define center	M	0
FR132: Define radius	0	N

	DP211: Line button	DP212: Rectangle button	DP213: Circle button
FR211: Identify line	O	0	0
FR212: Identify rectangle	0	P	0
FR213: Identify circle	0	0	Q

	DP221: Event for push	DP222: Event for release
FR221: Detect mouse push	R	0
FR222: Detect mouse release	0	S

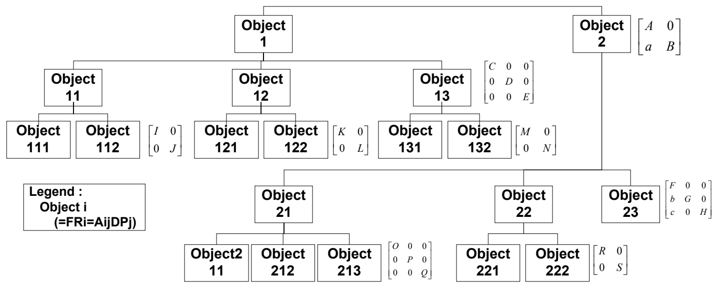

Object 1
Object 2
$\begin{bmatrix} A & 0 \\ a & B \end{bmatrix}$
Object 11
Object 12
Object 13
$\begin{bmatrix} C & 0 & 0 \\ 0 & D & 0 \\ 0 & 0 & E \end{bmatrix}$
Object 111
Object 112
$\begin{bmatrix} I & 0 \\ 0 & J \end{bmatrix}$
Object 121
Object 122
$\begin{bmatrix} K & 0 \\ 0 & L \end{bmatrix}$
Object 131
Object 132
$\begin{bmatrix} M & 0 \\ 0 & N \end{bmatrix}$
Legend : Object i (=FRi=AijDPj)
Object 21
Object 22
Object 23
$\begin{bmatrix} F & 0 & 0 \\ b & G & 0 \\ c & 0 & H \end{bmatrix}$
Object2 11
Object 212
Object 213
$\begin{bmatrix} O & 0 & 0 \\ 0 & P & 0 \\ 0 & 0 & Q \end{bmatrix}$
Object 221
Object 222
$\begin{bmatrix} R & 0 \\ 0 & S \end{bmatrix}$

Figure 3: The design hierarchy

d. *Definition of Modules – Full Design Matrix*

When the decomposition process finishes, inconsistency check should be done to confirm the decomposition. The full design matrix shown in Figure 4 indicates that the design has no conflicts between hierarchy levels. By definition, each row in the full design matrix represents a module to fulfill corresponding FRs. For example, FR 23 (Draw a element) can only be satisfied if all the DPs except DP221 and DP 222 are present.

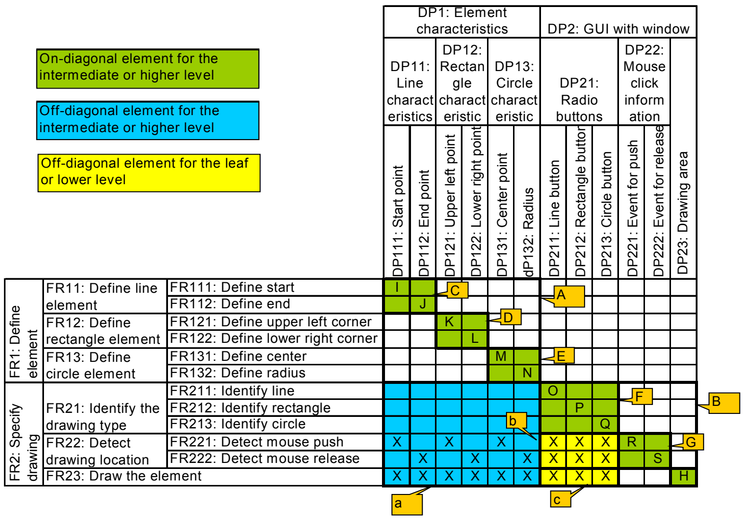
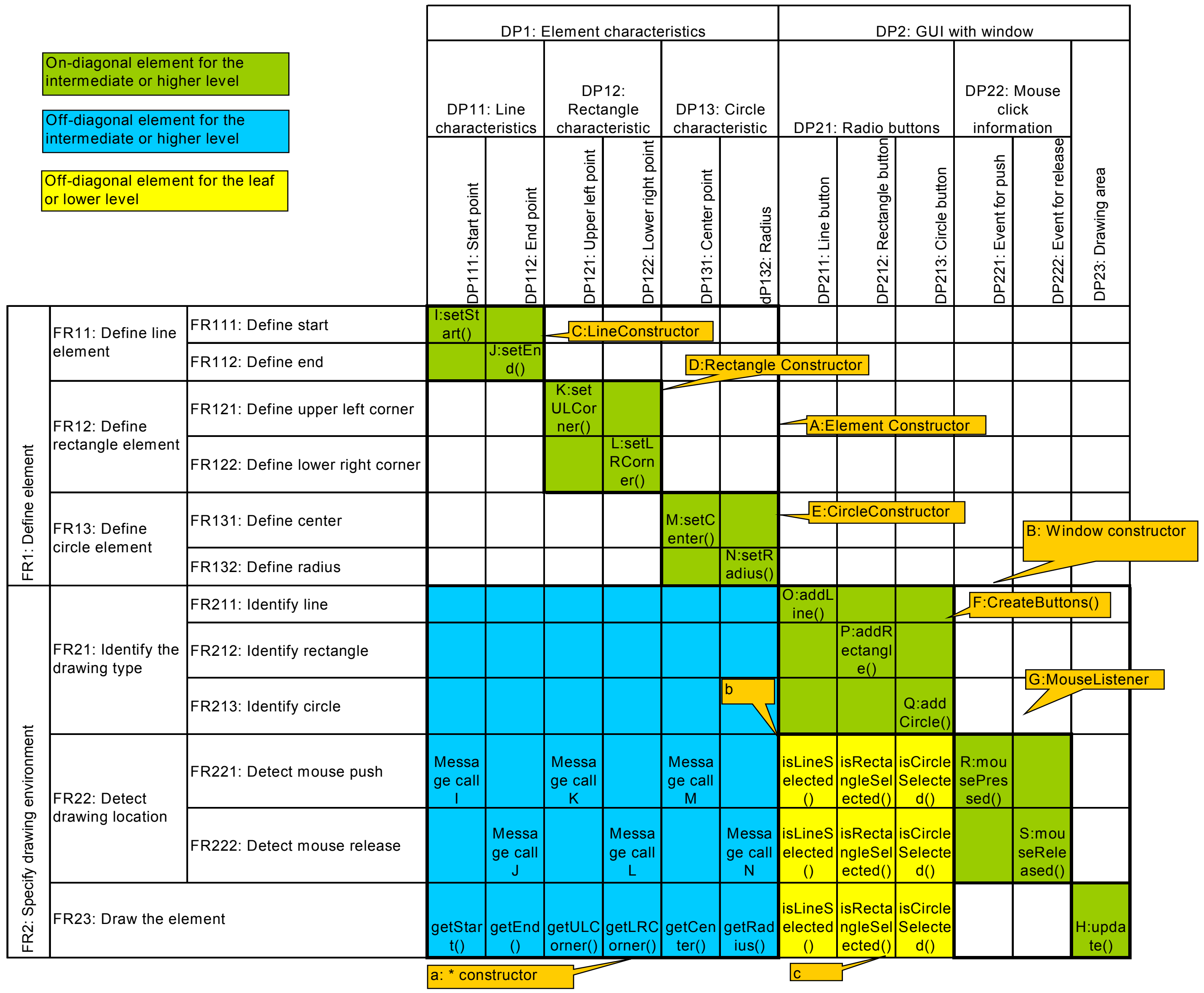

On-diagonal element for the intermediate or higher level
Off-diagonal element for the intermediate or higher level
Off-diagonal element for the leaf or lower level
DP1: Element characteristics
DP2: GUI with window
DP11: Line characteristics
DP12: Rectangle characteristic
DP13: Circle characteristic
DP21: Radio buttons
DP22: Mouse click information
DP111: Start point
DP112: End point
DP121: Upper left point
DP122: Lower right point
DP131: Center point
dP132: Radius
DP211: Line button
DP212: Rectangle button
DP213: Circle button
DP221: Event for push
DP222: Event for release
DP23: Drawing area
FR1: Define element
FR11: Define line element
FR111: Define start
FR112: Define end
FR12: Define rectangle element
FR121: Define upper left corner
FR122: Define lower right corner
FR13: Define circle element
FR131: Define center
FR132: Define radius
FR2: Specify drawing
FR21: Identify the drawing type
FR211: Identify line
FR212: Identify rectangle
FR213: Identify circle
FR22: Detect drawing location
FR221: Detect mouse push
FR222: Detect mouse release
FR23: Draw the element
I
J
C
A
K
L
D
M
N
E
O
P
Q
F
B
b
R
S
G
H
X
a
c

Figure 4: The full design matrix

e. *Identify objects, attributes, and operations*

Figure 5 shows how each design matrix elements were transformed into programming terminology. Unlike the other design cases, the mapping between physical domain and process domain is pretty straightforward in software design case since the process variables for software are the real source codes. These source codes represent each class in object-oriented programming package. Whenever the software designer categorizes module groups as classes using the full design matrix, they define the process variables for corresponded design hierarchy levels. Designers can assume that the design matrixes for DP/PV mapping are identical with those for FR/DP.


🟩 On-diagonal element for the intermediate or higher level
🟦 Off-diagonal element for the intermediate or higher level
🟨 Off-diagonal element for the leaf or lower level
C:LineConstructor
D:Rectangle Constructor
A:Element Constructor
E:CircleConstructor
B: Window constructor
F:CreateButtons()
G:MouseListener
b
a: * constructor
c

			DP1: Element characteristics						DP2: GUI with window					
			DP11: Line characteristics		DP12: Rectangle characteristic		DP13: Circle characteristic		DP21: Radio buttons			DP22: Mouse click information		
			DP111: Start point	DP112: End point	DP121: Upper left point	DP122: Lower right point	DP131: Center point	dP132: Radius	DP211: Line button	DP212: Rectangle button	DP213: Circle button	DP221: Event for push	DP222: Event for release	DP23: Drawing area
FR1: Define element	FR11: Define line element	FR111: Define start	I:setStart()	🟩										
		FR112: Define end	🟩	J:setEnd()										
	FR12: Define rectangle element	FR121: Define upper left corner			K:setULCorner()	🟩								
		FR122: Define lower right corner			🟩	L:setLRCorner()								
	FR13: Define circle element	FR131: Define center					M:setCenter()	🟩						
		FR132: Define radius					🟩	N:setRadius()						
FR2: Specify drawing environment	FR21: Identify the drawing type	FR211: Identify line	🟦	🟦	🟦	🟦	🟦	🟦	O:addLine()	🟩	🟩			
		FR212: Identify rectangle	🟦	🟦	🟦	🟦	🟦	🟦	🟩	P:addRectangle()	🟩			
		FR213: Identify circle	🟦	🟦	🟦	🟦	🟦	🟦	🟩	🟩	Q:addCircle()			
	FR22: Detect drawing location	FR221: Detect mouse push	Message call I	🟦	Message call K	🟦	Message call M	🟦	isLineSelected()	isRectangleSelected()	isCircleSelected()	R:mousePressed()	🟩	
		FR222: Detect mouse release	🟦	Message call J	🟦	Message call L	🟦	Message call N	isLineSelected()	isRectangleSelected()	isCircleSelected()	🟩	S:mouseReleased()	
	FR23: Draw the element		getStart()	getEnd()	getULCorner()	getLRCorner()	getCenter()	getRadius()	isLineSelected()	isRectangleSelected()	isCircleSelected()			H:update()

Figure 5: The method representation

f. Establish interfaces by showing the relationships between objects and operations

Figure 5 represents the additional information for FR/DP mapping. The same rule can be introduced to represent the interface information such as aggregation, generalization and so forth in the design matrix for DP/PV mapping. The flow diagram in Figure 6 guides through the developing process showing how the software can be programmed sequentially.

Table 6 categorizes the classes, attributes, and operations from the Figure 5 using this mapping process. The first row in table 6 represents the PV. The sequences in Table 6 i.e. left to right, also show the programming sequences based on the flow diagram. Figure 7 shows classes diagram for this example based on the matrix for DP/PV mapping.

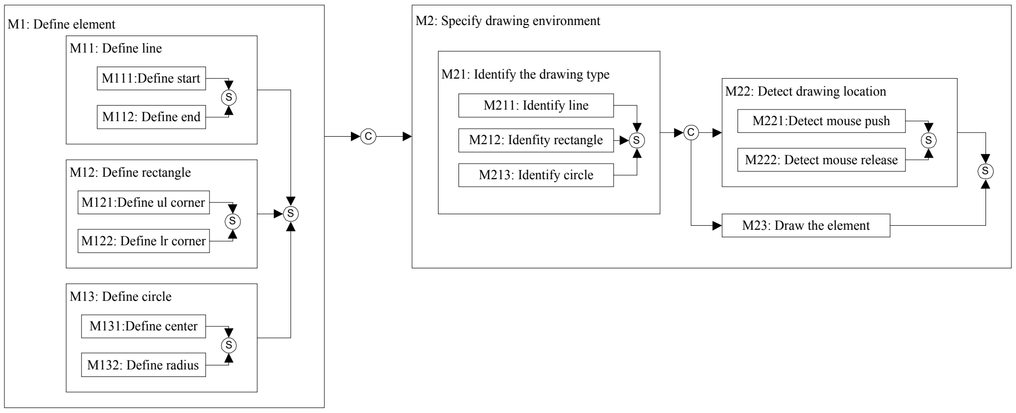


Figure 6: Flow diagram for the simple drawing example

Table 6. Class Identification

PVs	Class for Object 11		Class for Object 12		Class for Object 13		Class for Object 1		Class for Object 2		Class for Object 1*	
Name	Line_d		Rectangle_d		Circle_d		Element_d		Window_d		Element_*	
Attribute	DP111	Point start	DP121	Point upper_left	DP131	Point center	DP11	Line l	DP211	Radiobutton line		
	DP112	Point end	DP122	Point lower_right	DP132	Double radius	DP12	Rectangle r	DP212	Radiobutton rectangle		
							DP13	Circle c	DP213	Radiobutton circle		
									DP22	Mouse m		
									DP23	Canvas c		
Method	C	Line()	D	Rectangle()	E	Center()	A	Element()	B	Window()	a	Element*()
	I	setStart()	K	setULCorner()	M	setCenter()			F	CreateButtons()		getStart()
	J	setEnd()	L	setLRCorner()	N	setRadius()			O	addLine()		getEnd()
									P	addRectangle		getULCorner()
									Q	addCircle()		getLRCorner()
									G	implement MouseLisner		getCenter()
									R	mousePresed()		getRadius()
									S	mouseReleased()		assignLine()
									H	draw()		assignRectangle()
									b/c	isLineSelected()		assignCircle()
									b/c	isRectangleSelected()		
									b/c	isCircleSelected()		

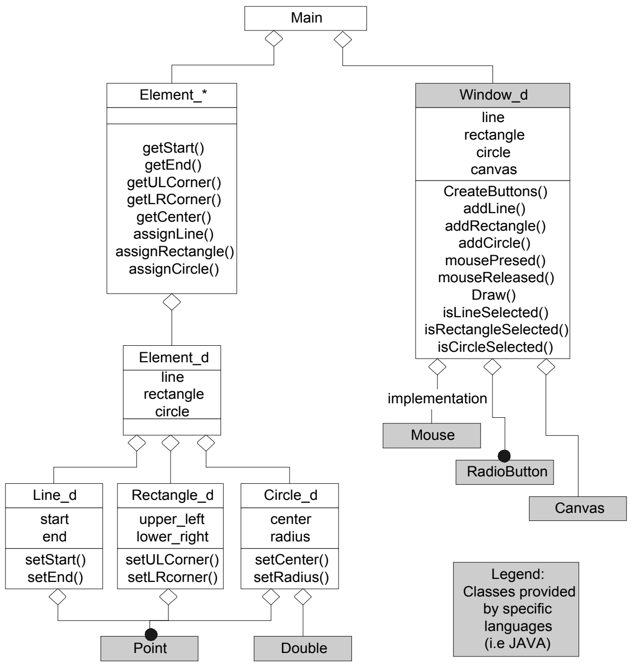


Figure 7: Object-Oriented model generation

4 CONCLUSION

The AD framework has been applied to the design and development of an object-oriented software system. The current software development methodologies demand that each individual module be independent. However, modularity does not mean functional independence, and therefore the existing methodologies do not provide a means to achieve the independence of functional requirements. To have good software, the relationship between the independent modules must be designed to make them work effectively and explicitly. The AD framework supplies a method to overcome these difficulties systematically and ensures that the modules are in the right place in the right order, when the modules are established as the row of design matrix. The axiomatic design methodology for software development can help software engineers and programmers to develop effective and reliable software systems quickly.

5 REFERENCES

[1] Booch G., *Object-Oriented Analysis and Design with Applications*, 2th ed., California: The Benjamin/Cummings Publishing Company, Inc., 1994.

[2] Cox B.J., *Object-Oriented Programming*, Reading, Massachusetts, Addison-Wesley, 1986

[3] DeMarco T., *Structural Analysis and System Specification*, New Jersey: Prentice Hall, 1979.

[4] Do S.H. and Park G.J., "Application of Design Axioms for Glass-Bulb Design and Software Development for Design Automation," *3rd CIRP Workshop on Design and Implementation of Intelligent Manufacturing*, pp. 119-126, June 19-22, Tokyo, Japan, 1996. {also published *in Transactions of the Korean Society of Mechanical Engineers*, Vol. 20, No. 4 , 1996 (in Korean)}

[5] Do S.H., "Application of Design Axioms to the Design for Manufacturability for the Television Glass Bulb*," Ph. D. Thesis*, Hanyang University, Seoul, Korea, 1997.

[6] El-Haik B., "The Integration of Axiomatic Design in the Engineering Design Process*", 11th Annual RMSL Workshop*, May 12, Detroit, U.S.A, 1999.

[7] 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 {*also Robotics & Computer-Integrated Manufacturing*, 3:149-162, 1992}.

[8] Pressman R.S., *Software Engineering*, A Practitioner's Approach, 4th ed., New York: McGraw Hill, 1997.

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

[10] Suh N.P., *The Principles of Design*, New York: Oxford University Press, 1990.

[11] Suh N.P., *Axiomatic Design: Advances and Applications*, New York: Oxford University Press, 2000 (In preparation).