The Employment of Axiomatic Design in Software Engineering: A Software Development Conceptual Framework

Basem El-Haik
Adnan Shaout
The Employment of Axiomatic Design in Software Engineering: A Software Development Conceptual Framework

Basem El-Haik, Ph.D.
E-Mail: basem.haik@sixsigmapi.com
Six Sigma Professionals, Inc.
www.SixSigmaPI.com
39505 Dorchester Cir.
Canton, MI 48188
USA

Adnan Shaout, Professor
E-Mail: shaout@umich.edu
The University of Michigan – Dearborn
The Electrical & Computer Engineering, Dept.
Dearborn, MI 48128
USA

Abstract:

Software permeates in every corner of our daily life. Software and computers are playing central roles in all industries and modern life technologies including cyber security to enable organizations to practice safe security techniques and minimize the number of successful cyber security attacks. In manufacturing, software controls manufacturing equipment, manufacturing systems, and the operation of the manufacturing enterprise. At the same time, the development of software and IT technologies can be the bottleneck in development of organizations and systems, since current software development is full of uncertainties, especially when new products are designed. Software Design for Six Sigma (Soft DFSS) is a methodology that was proposed by El-Haik & Shaout [2] to tackle the shortcomings of current software development practices. The goals of software DFSS is twofold: first to enhance algorithmic efficiency so as to reduce execution time and 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 [6]. Axiomatic Design is a methodology that was suggested as a conceptual framework for software development within Soft DFSS.

This paper introduces Software DFSS and Axiomatic Design methodology and shows thru a case study the employment of Axiomatic Design as a software conceptual development engine within Soft DFSS.

Keywords: Axiomatic Design, Software Complexity, Six Sigma, Software DFSS, Software Engineering, Software Development, Software Concept, Software Design.

1. INTRODUCTION

Software is designed and implemented by making prototypes based on experience of software engineers. Consequently, they require extensive ‘debugging’ – a process of correcting mistakes made during the software development process. It costs unnecessary time and money beyond the original estimate. The current situation is caused by the lack of fundamental principles and methodologies for software design, although various methodologies have been proposed.

In current software development practices, 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 development approach that ensures high quality, productivity and reliability of software systems a priori. Hence, Software Design for Six Sigma (Soft DFSS)

Soft DFSS combines design analysis (e.g. requirements cascading) with design synthesis (e.g. process engineering) within the framework of the deploying company’s software (product) development systems. Emphasis is placed on CTSs (Critical-To-Satisfaction requirements, a.k.a Big Y’s) identification, optimization and verification using software transfer function and scorecard vehicles. A transfer function in its
simplest form is a mathematical relationship between the CTSs and/or their cascaded Functional Requirements (FRs) and the critical influential factors (called the X’s, the design parameters). Scorecards help predict risks to the achievement of CTSs or FRs by monitoring and recording their mean shifts and variability performance. Soft DFSS is a disciplined and rigorous approach to software design by ensuring that new designs meet customer requirements at launch.

Axiomatic Design is a methodology within Soft DFSS tool box that is used to generate software conceptual alternatives. It is design theory that constitutes basic and fundamental design elements knowledge. Axiomatic Design is a scientific design method, however, with the premise of a theoretic system based on two axioms.

Software development requires the translation of good abstract ideas into clear design specifications. Subsequent delivery of the software product in moderate-to-large scale projects requires effective requirements definition, translation into useful codes and assignments for a team of software belts and engineers to meet deadlines in the presence of resource constraints. In [2] we explore how Axiomatic Design may be integrated into the software DFSS process. An approach to mapping functional requirements and design parameters into code is described. Application of Axiomatic Design to software development was firstly presented at the 1991 CIRP General Assembly (5) and the system design concepts presented in the 1997 CIRP General Assembly [10].

Axiomatic Design is not heuristic in nature and provides basic principles for good software systems. The Axiomatic Design framework for software overcomes many of the shortcomings of current software design techniques: high maintenance costs, limited reusability, low reliability, 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 section has helped software engineers to improve productivity and reliability. This paper presents a new software concept generation framework within Soft DFSS methodology based on Axiomatic Design theory. It also incorporates an application to object-oriented programming software development case study. This methodology overcomes the shortcomings of various software design strategies commonly used resulting in extensive software development and debugging times and the need for extensive maintenance.

This paper is developed as follows: Section 2 is the background section. We will start by listing current software development practices in Subsection 2.1, followed by a brief introduction to Soft DFSS in Subsection 2.2. We review the basic principles of Axiomatic Design1 in Section 2.3. The Axiomatic Design software concept generation approach is presented in Section 3. An application case study is provided in Subsection 3.1. The conclusion section is Section 4. References are listed in the last section.

2. BACKGROUND
2.1 Current Software Development Practices

IT quality engineering, quality improvement methods are constantly getting more attentions from world corporate leaders, all levels of management, design engineers and academia. This trend can be easily seen by the wide spread of ‘Six-Sigma’ initiatives in many Fortune IT 500 companies. For Six Sigma initiative in IT, software design activity is the most important to achieve significant quality and reliability results. Because design activities carries big portion of software development impact, quality improvements done in design stages will often bring most impressive results. Patching up quality problems in post-design phases is usually inefficient and very costly.

Over last 20 years, there are significant software methodologies for quality improvement in software design, those methods include the Waterfall modeling, Personal Software Process (PSP), Team Software Process (TSP), the Capability Maturity Model (CMM), Software Process Improvement Capability Determination (SPICE), the Linear Sequential Model, the Prototyping Model, The RAD Model, and the Incremental Model among others [6]. The historical evolution of these methods and processes, while indicates improvement trends, indicates gaps that each method tried to pick up where its predecessors left while filling the gaps missed in their application.

A software design method is typically defined as a systematic approach for carrying out a design, and describes a sequence of steps for producing a software

1 In the context of DFSS, the topic of axiomatic design was discussed in [2]
design [3]. There are certainly a number of ways to design software, but a designer must use certain types of established practices when preparing software. Different types of approaches to software designs may be used depending on the type of problem being encountered. Moreover, different types of software design methods each have their own unique advantages and disadvantages that are unique to one another. Many people think that software engineering is a creative activity that does not need a structured approach; however it is important to note that an informal approach towards software development does not build a good software system. Dividing software design methodologies into classifications, aids in the understanding of software design methodologies (Khoo, 2009).

Primitive types of software development started around the late 1940s and early 1950s, with the first stored-program computer, the Cambridge EDSAC. By the late 1960s, software had become part of many products. However, there was no real metric to determine the quality of software, which led to many safety issues. This particular situation became known as the software crisis. In response, software manufacturing has to be based on the same types of foundations traditionally used in other types of engineering. During the early 1970s, structured design and software development models evolved. Researchers started focusing on software design to develop more complex software systems. In the 1980s and 1990s, software engineering shifted towards software development processes.

Although object oriented programming was initially developed around the late 1960s, this type of programming did not become especially popular until the late 1980s and 1990s [1], [11]. Object orientation programming can be traced back to the late 1960s with the development of Simula and Smalltalk, which are types of object oriented programming languages. However, object-oriented programming did not become extremely popular until around 1990s, as the Internet became more popular.

During the 1990s, object orientation was also modified with Class Responsibilities Collaborators (CRC) cards. Moreover, methods and modeling notations that came out of the structured design movement were making their way into the object-oriented modeling. During this time, an integrated approach to design was becoming needed in an effort to manage large-scale software systems, and developed into the Unified Modeling Language (UML). UML integrates modeling concepts and notations from many methodologists. UML is a widely used, generalized type of modeling language, and falls under an object oriented approach. The UML approach was started around the early to mid 1990s, and was developed by James Rumbaugh from and Grady Booch of Rational Software Corporation. At that time, Rational was the source for the two most popular object-oriented modeling approaches of the day: Rumbaugh's OMT, which was better for object-oriented analysis (OOA), and Grady Booch's Booch method, which was better for object-oriented design (OOD). Rumbaugh and Booch attempted to combine their two approaches and started work on a Unified Method.

Another popular approach that started to develop around the same time was the use of design patterns. A design pattern is reusable solution used to solve commonly occurring problems in software design. In other words, a design pattern is not a finished design that can be transformed directly into code, but rather a template for how to solve a problem. Originally design patterns emerged as an architectural concept in the late 1970s. It was not until the late 1980s that design patterns were considered in programming. However, design patterns did not start to become extremely popular until around 1994, after the book Design Patterns: Elements of Reusable Object-Oriented Software was published. That same year the first Pattern Languages of Programming Conference was held. In 1995, the Portland Pattern Repository was set up for documentation of design patterns.

Six-Sigma is a methodology to manage process & product variations that use data and statistical analysis to measure and improve any organization operational performance. It works by identifying and eliminating defects in manufacturing, software development/ maintenance and service-related processes. The maximum permissible defects are 3.4 per one million opportunities. While Six Sigma is manufacturing-oriented, its application to software problem solving is undisputable since as you may imagine there are problems that needs to be solved in software and IT domains. However, the real value is in prevention rather than in problem solving, hence, Software Design for Six Sigma (Soft DFSS).
2.2 Software Design for Six Sigma (Soft DFSS)

Design for Six Sigma (DFSS) is a very vital to software design activities that decide quality, cost, cycle time of the software and can be greatly improved if the right strategy and methodologies are used. Major IT US corporations are training many software design engineers, project leaders to become Six Sigma Black Belt, or Master Black Belt, enabling them to play the leader role in organization excellence. Software DFSS approach can be phased into I-identify, C-onceptualize, O-ptimise, and V-erify or ICOV for short. These are defined below:

- **I-identify** customer and design requirements. Prescribe the CTSs, design parameters and corresponding process variables.
- **C-onceptualize** the concepts, specifications, and technical and project risks.
- **O-ptimise** the design transfer functions and mitigate risks.
- **V-erify** that the optimized design meets intent (customer, regulatory and deploying software function).

El-Haik & Shaout (2010) provides an algorithm, a roadmap, of software engineering using the design for six sigma thinking, tools, and philosophy to software design as depicted in Figure 1. The C-onceptualize phase of the algorithm includes conceptual design frameworks using Axiomatic Design for Six Sigma capability upfront to enable software design teams and individuals to disregard concepts that are not capable upfront, leaning the software development cycle and saving developmental costs.

Software Design for Six Sigma (Soft DFSS) offers engineers powerful opportunities to develop more successful systems, software, hardware, and processes. In applying Design for Six Sigma to Software systems, two leading experts offer a realistic, step-by-step process for succeeding with DFSS. El-Haik & Shaout (2010) provides a clear; start-to-finish roadmap designed for successfully developing complex high-technology products and systems. They cover the entire software DFSS project lifecycle, from business case through scheduling, customer-driven requirements gathering through execution. The authors provide real-world experience for applying their techniques to software alone, hardware alone, and systems composed of both. Product developers will find proven job aids and specific guidance about what teams and team members need to do at every stage. The uniqueness of this book is bringing all those methodology under the umbrella of design and give detailed description about how those methods; QFD, Robust Design methods, Software Failure Mode & Effect Analysis (SFMEA), Design for X, Axiomatic Design be utilized to help quality improvements in software development. The book also helps identify the different roles those methods may play in various stages of design and how to combine those methods to form a comprehensive strategy, a design algorithm, to tackle any quality issues in any design stage.

![Software DFSS Project Road Map](image)

**Figure 1. Software DFSS Project Road Map**

2.3 Axiomatic Design

Motivated by the absence of scientific design principles, Suh [7] proposed the use of axioms as the scientific foundations of design. Out of the twelve axioms first suggested, Suh introduced the following two basic axioms along with six corollaries that a design needs to satisfy:

**Axiom 1:** The Independence Axiom

Maintain the independence of the functional requirements

**Axiom 2:** The Information Axiom

Minimize the information content in a design

---

2 The case study in Section 4 is a direct application of Axiom 1.
3 Refer to El-Haik & Shaout [2] for Axiom 2 literature. It will not be discussed in this paper.
In Axiomatic Design (AD) approach, the engineering design process is described in Figure 2, in which the array of functional requirements (FRs) is the minimum set of independent requirements that completely characterizes the design objective based on customer attributes (CAs). Design is defined as the creation of synthesized solution to satisfy perceived needs through the mapping between the FRs in the functional domain and the design parameters (DPs) in the physical domain and through the mapping between the DPs and the process variables (PVs) in the process domain. The physical and process mappings can be expressed mathematically as

\[ \{FR\}_{mx1} = [A]_{mxp} \{DP\}_{px1} \]

\[ \{DP\}_{px1} = [B]_{rnx} \{PV\}_{nx1} \]

Figure 2. The AD Design Process Mappings

Where \( \{FR\}_{mx1} \) is the vector of independent functional requirements with \( m \) components, \( \{DP\}_{px1} \) is the vector of design parameters with \( r \) components, \( \{PV\}_{nx1} \) is the vector of process variables with \( n \) components, \( A \) is the physical design matrix, and \( B \) is the process design matrix. The mapping process can be mathematically abstracted as the following matrix equation: \( \{FR\} = [A]\{DP\} \), where \( FR \) is the array of FRs, \( DP \) is the array of DPs, and \( A \) is the design matrix that contains the sensitivity coefficients of the FRs to the mapped-to DPs. The process mapping is described by: \( \{DP\} = [B]\{PV\} \) The subsequent development uses the physical mapping for illustration purposes. Nevertheless, the results and conclusions are equally applicable to the process mapping as well.

Axiom 1 states that the design parameters (DPs) and the functional requirements (FRs) are related such that a specific DP can be adjusted to satisfy its corresponding FR without affecting the other functional requirements, which will require that \( A \) should be either a diagonal matrix or triangular matrix.

The design team will conceive a detailed description of what functional requirements the design entity needs to perform to satisfy customer needs, a description of the physical entity that will realize those functions (the DPs), and a description of how this object will be produced (the PVs).

The mapping equation \( FR = f(DP) \) or, in matrix notation \( \{FR\}_{mx1} = [A]_{mxp} \{DP\}_{px1} \), is used to reflect the relationship between the domain, array \( \{FR\} \), and the co-domain, array \( \{DP\} \), in the physical mapping where the array \( \{FR\}_{mx1} \) is a vector with \( m \) requirements, \( \{DP\}_{px1} \) is the vector of design parameters with \( p \) characteristics, and \( A \) is the design matrix. Per Axiom 1, the ideal case is to have a one-to-one mapping so that a specific DP can be adjusted to satisfy its corresponding FR without affecting the other requirements. However, perfect deployment of the design axioms may be infeasible due to technological and cost limitations. Under these circumstances, different degrees of conceptual vulnerabilities are established in the measures (criteria) related to the unsatisfied axiom. For example, a degree of coupling may be created because of Axiom 1 violation, and this design may function adequately for some time in the use environment, however, a conceptually weak system may have limited opportunity for continuous success even with the aggressive implementation of a continuous improvement program.

When matrix \( A \) is a square diagonal matrix, the design is called uncoupled, i.e. each FR can be adjusted or changed independent of the other FRs. An uncoupled design is a one-to-one mapping. Another design that obeys Axiom 1, though with a known design sequence, is called decoupled. In a decoupled design, matrix \( A \) is a lower or upper triangular matrix. The decoupled design may be treated as an uncoupled design when the DPs are adjusted in some sequence conveyed by the matrix. Uncoupled and decoupled design entities possess conceptual robustness (i.e. the DPs can be changed to affect specific requirements without affecting other FRs unintentionally). A coupled design definitely results in a design matrix with a number of requirements, \( m \), greater than the number of DPs, \( p \). Square design matrices (\( m=p \)) may be classified as coupled design when the off-diagonal matrix elements are non-zeroes. Graphically, the three design classifications are depicted in Figure 3 [2] for 2x2 design matrix case. Notice that we denote the non-zero mapping relationship in the respective design.
matrices by “X⁴. On the other hand, “0” denotes absence of such a relationship.

Consider the uncoupled design in Figure 3-(a). The uncoupled design possesses the path independence property, that is, the design team could set the design to level (1) as a start point and move to setting (2) by changing DP1 first (moving east to the right of the page or parallel to DP1) and then changing DP2 (moving toward the top of the page or parallel to DP2). Due to the path independence property of the uncoupled design, the team could start from setting (1) to setting (2) by changing DP2 first (moving toward the top of the page or parallel to DP2) and then changing DP1 second (moving east or parallel to DP1). Both paths are equivalent, that is, they accomplish the same result. Notice also that the FRs independence is depicted as orthogonal coordinates as well as perpendicular DPs axes that parallel its respective FR in the diagonal matrix.

Path independence is characterized, mathematically, by a diagonal design matrix (uncoupled design). Path independence is very desirable property of an uncoupled design and implies full control of the design team and ultimately the customer (user) over the design. It also implies high level of design quality and reliability since interaction effects between the FRs are minimized. In addition, a failure in one (FR,DP) combination of the uncoupled design matrix is not reflected in the other mappings within the same design hierarchical level of interest.

Path independence is characterized, mathematically, by a diagonal design matrix (uncoupled design). Path independence is very desirable property of an uncoupled design and implies full control of the design team and ultimately the customer (user) over the design. It also implies high level of design quality and reliability since interaction effects between the FRs are minimized. In addition, a failure in one (FR,DP) combination of the uncoupled design matrix is not reflected in the other mappings within the same design hierarchical level of interest.

For the decoupled design, the path independence property is somehow fractured. As depicted in Figure 3-(b), decoupled design matrices have design settings sequence that need to be followed for the functional requirements to maintain their independence. This sequence is revealed by the matrix as follows: First, we need to set FR2 using DP2, fix DP2, and second set FR1 by leveraging DP1. Starting from setting (1), we need to set FR2 at setting (2) by changing DP2, and then change DP1 to the desired level of FR1.

4 In general, when an alphabetic character is used a matrix entry it represent a non-zero sensitivity between a (FR, DP) pair.
The above discussion is a testimony to the fact that uncoupled and decoupled designs have conceptual robustness, that is, coupling can be resolved with the proper selection of the DPs, path sequence application and employment of design theorems [2]. The coupled design matrix in Figure 3-(c) indicates the loss of the path independence due to the off-diagonal design matrix entries (on both sides) and the design team has no easy way to improve controllability, reliability and quality of their design. The design team is left with compromise practices (e.g. optimization) amongst the FRs as the only option since a component of the individual DPs can be projected on all orthogonal directions of the FRs. The uncoupling or decoupling step of a coupled design is a conceptual activity that follows the design mapping and will be explored later.

The design matrices are obtained in a hierarchy, and result from employment of the zigzagging method of mapping as depicted in Figure 4 [7]. The zigzagging process requires a solution neutral environment, where the DPs are chosen after the FRs are defined and not vice versa. When the FRs are defined, we have to zig to the physical domain, and after proper DPs selection, we have to zag back to the functional domain for further decomposition or cascading, though at a lower hierarchical level. This process is in contrast with the traditional requirements cascading processes that utilize only one domain at a time, treating the design as the sum of functions or the sum of parts or modules.

Software designed based on Axiomatic Design is self-consistent, provides uncoupled or decoupled interrelationships and arrangements among ‘modules’, and is easy to change, modify, and extend. This is a result of having made correct decisions at each stage of the design process, i.e., mapping and decomposition [2], [7].

Based on Axiomatic Design and object-oriented method, Do and Suh [12] have developed a generic approach to software design. The approach presented here is an integration of Axiomatic Design and Object-Oriented Software Systems. The goal 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. The case study 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.

One of the final outputs of AD framework 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 improvement of the proposed design through identification of coupled designs and diagnosis of the impending failure of a complex system.

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

---

3. SOFTWARE CONCEPT GENERATION USING AXIOMATIC DESIGN

The methodology presented in this section 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 [7], [11]. 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.

Software designed based on Axiomatic Design is self-consistent, provides uncoupled or decoupled interrelationships and arrangements among ‘modules’, and is easy to change, modify, and extend. This is a result of having made correct decisions at each stage of the design process, i.e., mapping and decomposition [2], [7].

Based on Axiomatic Design and object-oriented method, Do and Suh [12] have developed a generic approach to software design. The approach presented here is an integration of Axiomatic Design and Object-Oriented Software Systems. The goal 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. The case study 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.

One of the final outputs of AD framework 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 improvement of the proposed design through identification of coupled designs and diagnosis of the impending failure of a complex system.

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

---

5 El-Haik & Shaout (2010)
defined and located in the right place in the right order. 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.

The fundamental construct for the object-oriented method is the object, 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 DPj, i.e., module) in a single entity. Object retains certain information on how to perform certain operations, using the input provided by the data. The 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 \([FR_i = A_{ij} DP_j]\) 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.

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 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. 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 shown in Figure 5 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 (CAs) 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.

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 case study 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.

3.1 Case Study: Simple Drawing Program [12] 

In this subsection, a case study involving simple drawing software designed based on Axiomatic Design will be presented.

a) Define FRs of the Software System: Let us assume the customer attributes as follows:

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 below

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 (2). The upper character in design matrix area represents diagonal relationship and the lower in Table character means off-diagonal relationship.

DP1 = Elements Characteristics
DP2 = GUI with window

\[
\begin{bmatrix}
\text{FR1} \\
\text{FR2}
\end{bmatrix} = \begin{bmatrix}
A & 0 \\
a & B
\end{bmatrix} \begin{bmatrix}
\text{DP1} \\
\text{DP2}
\end{bmatrix}
\]

(2)

c) Decomposition of \{FRs\}, \{DPs\}, and \{PVs\}: The entire decomposition information is listed in El-Haik & Shaout [2] with the following design hierarchy depicted (Figure 6),

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 7 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, FR23 (Draw an element) can only be satisfied if all the DPs except DP221 and DP222 are present.

e) Establish interfaces by showing the relationships between objects and Operations Figure 8 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. Figure 9 shows a class diagram for this example based on the matrix for DP/PV mapping. The flow diagram in Figure 10 guides through the developing process showing how the software can be programmed sequentially.

f) Table 1 categorizes the classes, attributes, and operations from the Figure 9 using this mapping process. The first row in Table 1 represents the PV. The sequences in Table 1, i.e. left to right, also show the programming sequences based on the flow diagram. Figure 10 shows classes diagram for this example based on the matrix for DP/PV mapping.

4. CONCLUSIONS

This paper, we presented a new concept generation methodology to software development within the framework of Software Design for Six Sigma (Soft DFSS) as suggested by El-Haik & Shaout [2]. We reviewed current software development practices and listed their shortcomings that can be easily satisfied by the approach suggested here. A brief introduction to Soft DFSS was presented in Subsection 2.2. We reviewed the basic principles of Axiomatic Design in Section 2.3. The Axiomatic Design software concept generation approach is presented in Section 3. An application case study is provided in Subsection 3.1.
programmers to develop effective and reliable software systems quickly.

In this paper, we formed and integrated several strategic and tactical methodologies that produce synergies to enhance software development capabilities to deliver a broad set of optimized solutions. The method presented in this paper have a wide spread of application to help design teams in different project portfolios (e.g. staffing and other human resources functions, finance, Operations and supply chain functions, organizational development, financial software, training, technology and tools and methods, etc.)

Software DFSS provides a unique commitment to the project customers by guaranteeing agreed upon financial and other results. Software DFSS approach ensures these outcomes through risk identification and mitigation plans, variable (DFSS tools that are used over many stages) and fixed (DFSS tool that is used once)-tool structures and advanced conceptual tools. The DFSS principles and structure should motivate design teams to provide business and customers with a substantial return on their design investment.

Figure 7: The Full Design Matrix

REFERENCES:


Figure 8: The Method Representation

Table 1: Class Identification

<table>
<thead>
<tr>
<th>Class</th>
<th>Object ID</th>
<th>Description</th>
<th>Method</th>
<th>Initial State</th>
</tr>
</thead>
<tbody>
<tr>
<td>Line</td>
<td>1</td>
<td>start</td>
<td>line</td>
<td>0.0</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>end</td>
<td>line</td>
<td>1.0</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>upper_left</td>
<td>line</td>
<td>2.0</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>lower_right</td>
<td>line</td>
<td>3.0</td>
</tr>
<tr>
<td>Rectangle</td>
<td>5</td>
<td>start (left, top)</td>
<td>rectangle</td>
<td>5.0</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>end (right, bottom)</td>
<td>rectangle</td>
<td>6.0</td>
</tr>
<tr>
<td>Circle</td>
<td>7</td>
<td>center</td>
<td>circle</td>
<td>7.0</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>radius</td>
<td>circle</td>
<td>8.0</td>
</tr>
</tbody>
</table>

Figure 9: Object-Oriented Model Generation

Figure 10: Flow Diagram for the Simple Drawing Study