Proceedings of ICAD2011
The Sixth International Conference on Axiomatic Design
Daejeon – March 30-31, 2011
ICAD-2011-20

MANAGING INTRA-CLASS COMPLEXITY WITH AXIOMATIC DESIGN AND DESIGN STRUCTURE MATRIX APPROACHES

Zhen Li
zhen.li@ttu.edu
Department of Mechanical Engineering
Texas Tech University
7th and Boston
Lubbock, TX 79409, USA

Derrick Tate
d.tate@ttu.edu
Department of Mechanical Engineering
Texas Tech University
7th and Boston
Lubbock, TX 79409, USA

ABSTRACT

In Object-Oriented Programming (OOP), classes play an important role as they model the real world through their attributes (features) and methods (behaviours). However, few attempts have been made to help manage complexity within a class (intra-class complexity). In this paper, the authors define, model and manage intra-class complexity using both Axiomatic Design (AD) and the Multiple-Domain Matrix (MDM). By combining the AD and MDM approaches as suggested in this paper, complexity within a class can be managed, and class performance can be improved with little additional effort for developers.

Keywords: Intra-class complexity, Axiomatic Design (AD), Design Structure Matrix (DSM), Multiple-Domain Matrix (MDM), Object-Oriented Programming

1 INTRODUCTION

Object-oriented (OO) programming techniques have been accepted as as the dominant programming paradigm over the past two decades [Misra and Akman, 2008]. According to its proponents, the objected-oriented approach provides better complexity management, improved project quality, and reduced project cycle time compared with the procedural programming paradigm [Booch, 1993]. In object-oriented programming (OOP), classes are used to model a real world concept by incorporating its features (reflected as attributes) and its behaviours (reflected as methods). The concept of a class enables programmers to model the real world and to code more efficiently through OO features such as dynamic patching, encapsulation, polymorphism, and inheritance [Deitel and Deitel, 2004].

However, because the functional requirements of modern software ever increase, the size of software also increases dramatically. In 2001, the popular operating system Windows XP (2001) had nearly 40 million lines of code (LOC) while Windows Server 2003 (2003) contained 50 million LOC. As an indicator of growth in the size of software, Windows NT 3.1 which was released in 1993 contained only 4-5 million LOC, and Windows NT 4.0 released in 1996 contained 11-12 million LOC [Maraia, 2005]. As lines of code and complexity are often correlated, more defects can be expected when the program size increases [Siau and Cao, 2001]. Additionally, while becoming more difficult to detect, the removal of defects before product delivery remains important due to their effect on customer satisfaction [Grady, 1992]. Therefore, reducing design complexity has the potential to play a strong role in improving the quality of software systems through reducing defects in an object-oriented developing environment [Booch, 1993].

To model system requirements, design details, and implementation methods, unified modelling language (UML) was proposed in the mid 1990s and completed in 1996. UML 2.0 revision was released by Object Management Group (OMG) in 2005. Unified modelling language combines concepts from several different methods into object-oriented system development for specifying, visualizing, constructing, and documenting software [Siau and Cao, 2001]. Although UML has received criticism for being complicated, having inconsistent schema, and being ambiguous, it is considered the standard in software development modelling practice. Visualization of the collaborative relationship between classes offers at least one means of complexity management.

At times, supporters of UML even claim that simplicity is the main benefit of UML [Kobryn, 1999]. However, few methods have been proposed to deal with complexity within a class even though several complexity metrics have been proposed by researchers [Subramanyam and Krishnan, 2003]. A lengthy class can not only be hard to understand by simply reading its code line by line, but it is also hard to maintain without a proper complexity management approach. Although much documentation such as class descriptions, usages, and even comments in a class are often available for developers, commonly these documents do not thoroughly explain how the class is organized and what relationships lie between different methods and attributes because the encapsulation feature of OOP demands the hiding of information from users.

Therefore, it's necessary to adopt some techniques from an engineering perspective to reduce intra-class complexity in order to improve the efficiency of coding and maintenance. In this paper, the authors propose a combined method of both axiomatic design and the design structure matrix approach to help manage intra-class complexity.

2 LITERATURE REVIEW

The complexity of the OO methodology has been noticed by researchers and practitioners. Several complexity measures have been proposed in order to predict software reliability and maintainability [Bandi *et al.*, 2003].

Among all the proposed measures, lines of code (LOC) is the simplest because it only counts instruction statements. Research done by Withrow suggests that there is a concave relationship between the number of defects and module size [Withrow, 1990]. As the size of a module increases, complexity may go beyond a developer's comprehension and control and result in a higher defect rate. However, the LOC measure does not take human cognitive capability into consideration, and the LOC measure fails to provide the same results for software written in different programming languages having the exact same functions.

Fundamental concepts and features of OOP include class, object, instance, method, messaging passing, inheritance, abstraction, encapsulation, polymorphism, and decoupling [Noble, 1998]. Other research indicates that structural properties of software components can add cognitive complexity for developers and testers during software development [Briand *et al.*, 1999]. To measure cognitive complexity, Misra introduced a set of complexity metrics based on cognitive weights to model complexity of object-oriented software [Misra, 2007]. According to Misra, the cognitive weights can be calculated as the extent of difficulty or relative time for understanding a given piece of software that was modeled by Basic Control Structures (BCS's). For sequence, branch, and iteration in BCS, the weights were assigned for one, two, and three respectively. The total complexity of a class can be calculated simply by adding all weights together [Misra, 2007].

$$\text{Class Complexity (CC)} = \sum W_c \quad (1)$$

where W_c is defined as

$$W_c = \sum_{j=1}^{q} [\prod_{k=1}^{m} \sum_{i=1}^{n} W_c(j,k,i)] \quad (2)$$

Kim *et al.* describe complexity of object-oriented software in a graphical way [Kim *et al.*, 1995]. Rather than counting the number of methods in a class, their graphs also describe the relationships between attributes and methods. First, they measure the reference probability which is defined by the following equation.

$$P(N_i) = \frac{\sum_{j=1}^{n} W(A_{i,j}) + \sum_{j=1}^{n} W(A_{j,i})}{2 \times \sum_{k=1}^{n} \sum_{j=1}^{n} W(A_{k,j})} \quad (3)$$

where $\sum_{j=1}^{n} W(A_{i,j}) + \sum_{j=1}^{n} W(A_{j,i})$ stands for the total weights connected to node N_i and $\sum_{k=1}^{n} \sum_{j=1}^{n} W(A_{k,j})$ stands for the total weight of all arcs connected to all nodes in the graph. Then, by using the entropy function shown below, the complexity of class can be determined.

$$H(X) = -\sum_{i=1}^{n} p(x_i) \log_b p(x_i) \quad (4)$$

Fothi *et al.* argue in their paper that their complexity measure works well for procedural programs [Fothi *et al.*, 2003]. They extend that measurement to OOP and define the complexity as the sum of complexity of the control structure, data types used, and data handling. Also, this method is based on graph theory. By counting connections between nodes in a graph, the complexity can be determined. Lange *et al.* suggest that the most popular modeling language UML is sometimes inconsistent, incomplete, disproportional, and information-scattered [Lange *et al.*, 2006]. Such features result in an even more complicated situation for programmers and developers. To solve this problem, they introduce a metric for managing defects of UML. Class complexity is ranked in first place as they argue that classes play a critical role in a system. However, they do not propose any measurement for counting complexity in their paper. In addition to defining complexity for software, people also care about how to manage complexity in practice.

Zhao *et al.* describe a dependence-based representation using a graph that is termed a system dependence net (SDN) [Zhao *et al.*, 1998]. They argue that this kind of representation not only represents object-oriented features but can also be used for concurrent object-oriented projects. Figure 1 below shows an example of SDN. Although the dependency of classes is clarified in this graph, it is almost impossible to draw such graph when the number of classes is beyond 100. Also, complexity within class is still not solved.

3 MANAGING INTRA-CLASS COMPLEXITY WITH AXIOMATIC DESIGN AND DESIGN STRUCTURE MATRIX

Although classes are used by developers to model the real world, they are invoked to carry out certain functions. Therefore, managing intra-class complexity must start with understanding functional requirements (FRs) and design parameters (DPs).

Axiomatic design of objected-oriented software systems (ADo-oSS) was introduced by Suh and Do [Suh and Do, 2000]. The ADo-oSS framework starts with defining functional requirements (FRs) of the software system, mapping between the domains and the independence of software functions, selecting the best design based on information content, decomposing functional requirements, design parameters and process variables, implementing object-oriented programming, then finally representing the design with a design matrix, flow chart representation, and system control command (SCC) [Suh, 2005]. These steps are illustrated in Figure 2. Another example of adopting Axiomatic Design approach in software design was made by Cengiz Togay *et al.* Their work combines axiomatic design theory and the component-oriented software engineering (COSE) process [Togay *et al.*, 2008]. They believe this COSE can be matured, supported by the axiomatic design philosophy.

Defining complexity within a class is important as different types of complexity can be eliminated in different ways. According to Suh, there are generally two types of complexity, time-dependent complexity and time-independent complexity. Time-independent complexity can be divided into two parts, real complexity and imaginary complexity. The

Figure 1. System Dependence Net (SDN) [Zhao *et al.*, 1998].

former is defined as measuring uncertainty of achieving functional requirements (FRs) while the latter occurs mainly due to lacking knowledge and understanding of a design. Time-dependent complexity can be separated into combinatorial complexity and periodic complexity [Suh, 2005]. In software design, real complexity is often hard to measure as the nature of software execution is not probabilistic in the same way as mechanical systems. Also, future events in software at the class level are more predictable the outcomes have been determined by programmers in advance. Therefore, the major complexity within a class can be classified as imaginary (cognitive) complexity.

3.1 Considerations of Software Design Versus Axiomatic Design Approach

Before starting to model complexity in a class from an axiomatic design perspective, several considerations from software design practice must be taken into account. These considerations include extensibility, modularity, reusability, readability, and the famous open/close principle.

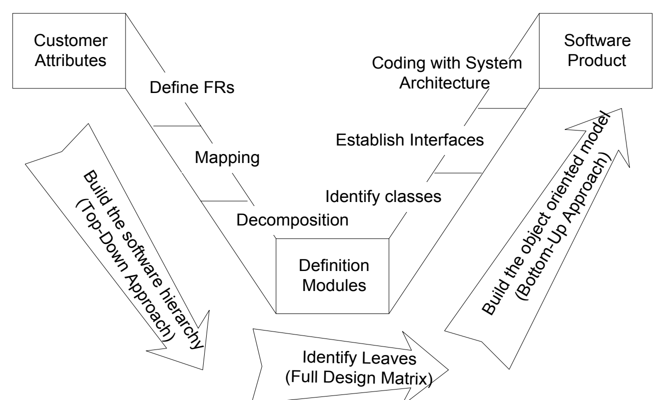

Figure 2. Axiomatic design process for object-oriented software system [Suh and Do, 2000].

Bertrand Meyer first proposed the open/close principle in his book *Object Oriented Software Construction* (1998) [Meyer, 2000]. This principle states that a class should be only changed for correcting error (close for modification) but should be extended to meet new functional requirements (open for extension) [Martin, 1996]. This principle actually is a summary

of extensibility, modularity, and reusability. If a class is closed for modification but open for extension, then this class can serve as an interface for other classes because its underlying structure will not be changed during extension. This characteristic is required by modularity, extensibility, and reusability. Although open/close principle mainly focuses on designing classes, it's also applicable for designing methods within a class. In other words, methods within a class should also conform to open/close principle.

Readability is defined as the ease of human readers to understand purpose, control flow, and operation of source code. It is well known that carefully commented and named source code is easier to comprehend. The concept of modularity facilitates readability as functions are decomposed.

The question is whether those considerations of software design are compatible with philosophy of axiomatic design? The answer is affirmative. Within a class, design parameters (DPs) are methods and that act on attributes (As) of a class. Because methods are created to carry out a certain task such as reading input from a keyboard or outputting a string to a monitor, the concept of method fits the concept of DPs defined in axiomatic design theory [Suh, 2005]. If methods (DPs) are created by obeying the open/close principle, the result should be a decoupled or an uncoupled design. Hence this fits the concept of independence axiom as the axiom requires maintaining independence between different functional requirements.

3.2 Managing and Reducing Complexity with Design Structure Matrix (DSM) and Multiple-Domain Matrix (MDM)

Although the axiomatic design approach helps manage complexity within a class, there are still some issues unsolved. First, users of the class commonly care about how to use the class instead of how the class is developed. Thus, dependency between methods rather than dependency between FRs is more important to them. Second, the axiomatic approach presented above doesn't take conflicts between attributes into consideration. Even if two methods are independent, there may still be a conflict between them as reading and writing attributes can't be done simultaneously (thread interference) without synchronizing or locking mechanism. To check this kind of conflict, mapping between methods (DPs) and attributes (As) must be made in advance.

4 Case Study I

In this section, a class from the authors' previous work is taken as an example to show how to implement axiomatic design and the design structure matrix method to manage complexity within a class. In data-mining practice, acquiring data and indexing data into a database are often done before applying analysis to it. Figure 3 shows a class diagram (WebPageMiner) from UML diagram which is used to search patents from USPTO (United States Patent and Trademark Office) website and then save data to database for further use. In this class, 11 attributes and 21 methods are included which make the class totally more than 600 lines of code in Java. Hence, it is complicated to understand by simply reading its program source code or its UML diagram. To manage the complexity of this class, the first two tasks are defining FRs and mapping them into DPs.

4.1 Step 1: Mapping FRs at Top Level into DPs

The top functional requirements of this class are listed as follows.

FR_1: Create database and index table.
FR_2: Acquire a patent from USPTO web page.
FR_3: Save patent content (title, class, citation, referenced by, abstract, claims, description) into database.
FR_4: Save index information of mined patents
FR_5: Release memory

As stated earlier, DPs should be methods within a class as those methods will carry out tasks to satisfy certain requirements. Therefore, the following methods serve as DPs to satisfy five FRs above.

DP_1: Method createDB and createIndexTable
DP_2: Method extractPatentCotent
DP_3: Method savePatentContentToDB
DP_4: Method saveIndexInfo
DP_5: Method reinitiateObjects

The design matrix can be written as equation 5. The lower case x in the matrix represents that FR_3 can be carried out only after DP_1 and DP_2 have been successfully executed. However, FR_3 mainly depends on DP_3 as each method should maintain its independence to satisfy open/close principle. The design matrix can be written as an uncoupled one if DPs are invoked in the designated sequence. As those FRs and DPs belong to the top level, some of them are declared as public so they can be invoked outside of the class. Exceptions are DP_3, DP_4 and DP_5 which are declared as private methods because it's not necessary to invoke those functions manually outside of the class.

$$\begin{bmatrix} FR_1 \\ FR_2 \\ FR_3 \\ FR_4 \\ FR_5 \end{bmatrix} = \begin{bmatrix} X & 0 & 0 & 0 & 0 \\ 0 & X & 0 & 0 & 0 \\ x & 0 & X & 0 & 0 \\ x & x & 0 & X & 0 \\ 0 & 0 & 0 & x & X \end{bmatrix} \begin{bmatrix} DP_1 \\ DP_2 \\ DP_3 \\ DP_4 \\ DP_5 \end{bmatrix} \quad (5)$$

4.2 Step 2: Zigzagging and Decomposition

Given top level DPs, FRs can be further decomposed into sub-level FRs. It's true that one can decompose top FRs without writing down top level DPs. However, this will result in losing valuable structural information that may eventually cause a logical error. For example, if sub-level DPs of DP_3 were executed before DP_1, the program will invoke an exception handling mechanism as one can't save something to database before it is created. Therefore, it's important decompose FRs with explicitly expressed DPs so that sequence of logical execution can be preserved. As no more sub-function is required for FR_1, decomposition starts from FR_2.

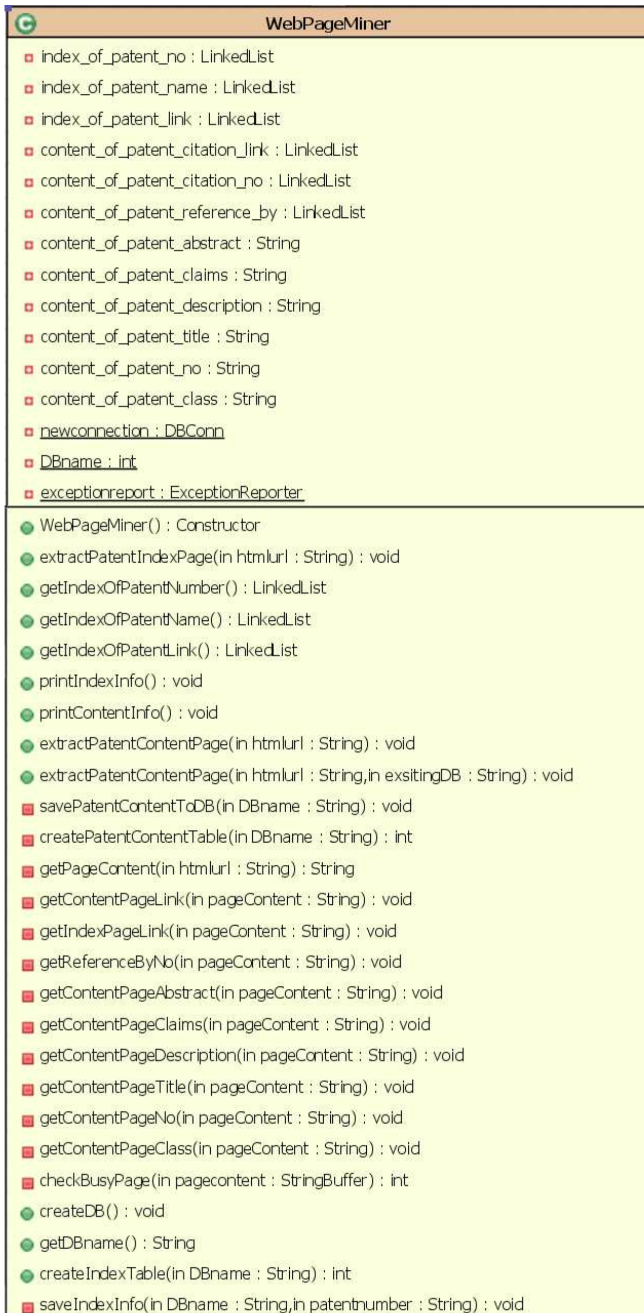

Figure 3. Class WebPageMiner UML diagram.

FR_2 can be decomposed into the following lower-level FRs:

$FR_{2\text{-}1}$: Retrieve page content from USPTO website
$FR_{2\text{-}2}$: Check if page content is correct.
$FR_{2\text{-}3}$: Acquire patent number.
$FR_{2\text{-}4}$: Acquire patent title.
$FR_{2\text{-}5}$: Acquire patent class.
$FR_{2\text{-}6}$: Acquire patent citation number.
$FR_{2\text{-}7}$: Acquire patent referenced by number.
$FR_{2\text{-}8}$: Acquire patent abstract.
$FR_{2\text{-}9}$: Acquire patent claims.
$FR_{2\text{-}10}$: Acquire patent description.

Do the same to FR_3, two low-level FRs must be satisfied.

$FR_{3\text{-}1}$: Create table for saving patent title, class, citation, reference by, abstract, claims and description.
$FR_{3\text{-}2}$: Save corresponding data to created table.

4.3 Step 3: Mapping Lower Level FRs into DPs

For $FR_{2\text{-}1}$ to $FR_{2\text{-}10}$, selected DPs are as follow.

$DP_{2\text{-}1}$: Method getContentPage
$DP_{2\text{-}2}$: Method checkBusyPage
$DP_{2\text{-}3}$: Method getContentPageNo
$DP_{2\text{-}4}$: Method getContentPageClass
$DP_{2\text{-}5}$: Method getContentPageTitle
$DP_{2\text{-}6}$: Method getContentPageLink
$DP_{2\text{-}7}$: Method getReferenceByNo
$DP_{2\text{-}8}$: Method getContentPageAbstract
$DP_{2\text{-}9}$: Method getContentPageClaims
$DP_{2\text{-}10}$: Method getContentPageDescription

As those methods belong to DP_2, they are all declared as private. Hence, invokation of these methods is forbidden outside of the class. The design matrix can be expressed as equation 6.

$$\begin{bmatrix} FR_{2-1} \\ FR_{2-2} \\ FR_{2-3} \\ FR_{2-4} \\ FR_{2-5} \\ FR_{2-6} \\ FR_{2-7} \\ FR_{2-8} \\ FR_{2-9} \\ FR_{2-10} \end{bmatrix} = \begin{bmatrix} X & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ x & X & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ x & x & X & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ x & x & 0 & X & 0 & 0 & 0 & 0 & 0 & 0 \\ x & x & 0 & 0 & X & 0 & 0 & 0 & 0 & 0 \\ x & x & 0 & 0 & 0 & X & 0 & 0 & 0 & 0 \\ x & x & 0 & 0 & 0 & 0 & X & 0 & 0 & 0 \\ x & x & 0 & 0 & 0 & 0 & 0 & X & 0 & 0 \\ x & x & 0 & 0 & 0 & 0 & 0 & 0 & X & 0 \\ x & x & 0 & 0 & 0 & 0 & 0 & 0 & 0 & X \end{bmatrix} \begin{bmatrix} DP_{2-1} \\ DP_{2-2} \\ DP_{2-3} \\ DP_{2-4} \\ DP_{2-5} \\ DP_{2-6} \\ DP_{2-7} \\ DP_{2-8} \\ DP_{2-9} \\ DP_{2-10} \end{bmatrix} \quad (6)$$

For $FR_{3\text{-}1}$ and $FR_{3\text{-}2}$, their corresponding DPs are

$DP_{3\text{-}1}$: Method createPatentContentTable
$DP_{3\text{-}2}$: Method savePatentContentToDB

Again, those two methods are declared as private and design matrix is written in equation 7.

f

$$\begin{bmatrix} FR_{3-1} \\ FR_{3-2} \end{bmatrix} = \begin{bmatrix} X & 0 \\ x & X \end{bmatrix} \begin{bmatrix} DP_{3-1} \\ DP_{3-2} \end{bmatrix} \quad (7)$$

4.4 Step 4: Building Flow Chart

Given equation 5, 6 and 7, a flow chart of methods can be built. As Figure 4 shows, DPs in green box indicates that methods are public and can be invoked outside of class. On the contrary, DPs in a red box suggest that methods are private to the class for information hiding purpose. Although equation 6 states that several DPs (e.g. $DP_{2\text{-}3}$, $DP_{2\text{-}4}$, $DP_{2\text{-}5}$, etc) are independent, each of them must be executed in order to retrieve all information. Therefore, those DPs are executed sequentially.

4.5 STEP 5: MANAGING AND REDUCING COMPLEXITY WITH DESIGN STRUCTURE MATRIX (DSM) AND MULTIPLE-DOMAIN MATRIX (MDM)

The flow chart of the class not only provides invaluable information regarding class structure but also helps reduce complexity and improve readability. Users of the class simply need to know how to invoke DP_1 and DP_2 in this program to achieve the goal as these two DPs are public to all users. Also, the sequence for invoking two DPs is provided in the flow chart. Therefore, the user should first call DP_1 and then DP_2. For the developer of this class, the flow chart helps reduce complexity in several ways. First of all, since the flow chart is built in accordance with the Axiomatic Design philosophy, one can expect that the open/close principle automatically fits. Second, debugging the class is much easier as the developer can perform root cause analysis relatively quickly by following the flow chart. Last but not least, the flow chart greatly improves readability as it can serve as the visualized form of API (application programming interface) documents.

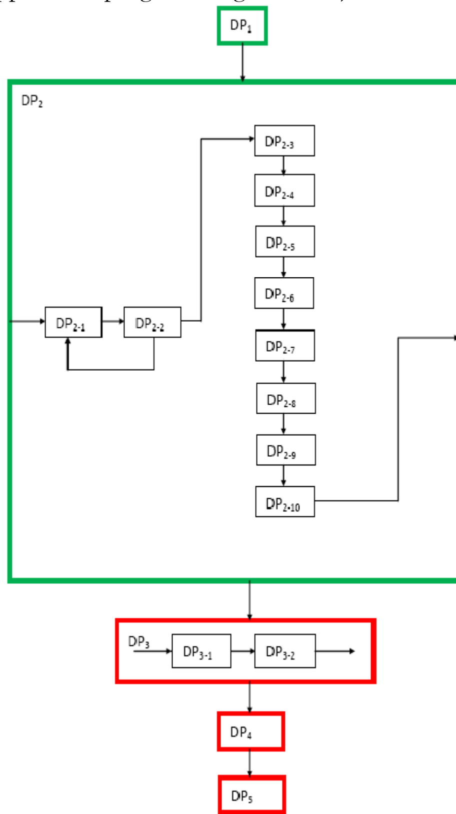

Figure 4. Flow chart of DPs.

However, as stated before, there are some unsolved issues such as undetermined conflicts and dependencies. Fortunately, one can take advantage of DSM and MDM to circumvent those problems. The Design Structure Matrix is a matrix-based complexity management tool which originates from a process focus by Steward [Steward, 1981]. For a system consisting of multiple domains with multiple elements and multiple relationships, the Multiple Domain Matrix enables researchers to analyze the system's structure. Both DSM and MDM are clearly self-explanatory as figure 5 and 6 can thoroughly explain how DSM and MDM works. (For a similar matrix applied in axiomatic design between DPs and components, see [Lee and Jeziorek, 2004].

Table 1 illustrates the DSM of DPs. The red box indicates that there is a loop in the DSM. Note that DP_{2-1} and DP_{2-2} are in a loop to check if the page content is correct. However, FR_{2-1} and FR_{2-2} still maintain their independence. Attributes of class can't be correlated with each other without certain kinds of manipulations. Therefore, constructing the DSM for attributes is not meaningful. As UML class diagram showed previously, there are eight private attributes The left three attributes are private static attributes that are used to invoke other classes. Hence, those three attributes will not be considered in this case.

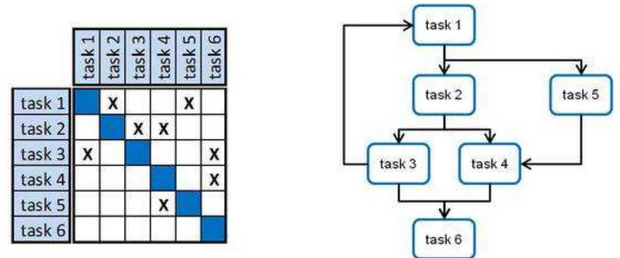

Figure 5. DSM versus flowchart [DSMweb.org, 2009].

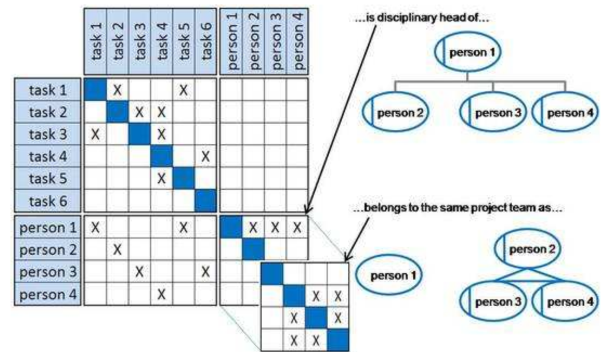

Figure 6. MDM versus organization chart [DSMweb.org, 2009].

A_1: content_of_patent_citation_no
A_2: content_of_patent_reference_by
A_3: content_of_patent_abstract
A_4: content_of_patent_claims
A_5: content_of_patent_description
A_6: content_of_patent_title
A_7: content_of_patent_no
A_8: content_of_patent_class

Manipulations on attributes can be divided into three categories: read, write and initialization. Although initialization is a special type of write, it is different from write since initialization commonly happens at the beginning of the program or the end of program (release resource). Therefore, it should be marked differently for clarification purpose. If initialization happens in the middle of a program, developers and users should be aware of it.

An MDM is shown in Table 2. Unlike some MDM, all of the DPs and attributes (As) are bi-directional in the chart as they are inseparable. Without methods, attributes consume computer storage and memory meaninglessly. On the contrary, if all methods in class don't manipulate any attribute, then MDM is not necessary at all. The most important feature of the MDM is to help programmers to decide whether some methods can be invoked simultaneously via multithreading.

Table 1. DSM of DPs.

			DP_1	DP_2									DP_3		DP_4	DP_5
			1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
DP_1	DP_1	1											X	X	X	
DP_2	DP_{2-1}	2			X	X	X	X	X	X	X	X	X	X	X	X	
	DP_{2-2}	3		X		X	X	X	X	X	X	X	X	X	X	X	
	DP_{2-3}	4												X	X	X	
	DP_{2-4}	5												X	X	X	
	DP_{2-5}	6												X	X		
	DP_{2-6}	7												X	X		
	DP_{2-7}	8												X	X		
	DP_{2-8}	9												X	X		
	DP_{2-9}	10												X	X		
	DP_{2-10}	11												X	X		
DP_3	DP_{3-1}	12													X		X
	DP_{3-2}	13															X
DP_4	DP_4	14															X
DP_5	DP_5	15															

Table 2. MDM of DPs and As.

			DP_1	DP_2										DP_3		DP_4	DP_5	As							
			1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	I	II	III	IV	V	VI	VII	VIII
DP_1	DP_1	1												X	X	X									
DP_2	DP_{2-1}	2			X	X	X	X	X	X	X	X	X	X	X	X									
	DP_{2-2}	3		X		X	X	X	X	X	X	X	X	X	X	X									
	DP_{2-3}	4												X	X	X								W	
	DP_{2-4}	5												X	X	X							W		
	DP_{2-5}	6												X	X										W
	DP_{2-6}	7												X	X			W							
	DP_{2-7}	8												X	X				W						
	DP_{2-8}	9												X	X					W					
	DP_{2-9}	10												X	X						W				
	DP_{2-10}	11												X	X							W			
DP_3	DP_{3-1}	12													X		X								
	DP_{3-2}	13															X	R	R	R	R	R	R	R	R
DP_4	DP_4	14															X							R	R
DP_5	DP_5	15																I	I	I	I	I	I	I	I
As	A_1	I							W						R		I								
	A_2	II								W					R		I								
	A_3	III									W				R		I								
	A_4	IV										W			R		I								
	A_5	V											W		R		I								
	A_6	VI					W								R		I								
	A_7	VII				W									R	R	I								
	A_8	VIII						W							R	R	i								

Step 1, Step 2, Step 3

For example, $DP_{2\text{-}4}$ doesn't depend on $DP_{2\text{-}3}$ when reading the DSM of DPs; therefore, it is possible to arrange them into two threads for faster processing. When checking the MDM, one realizes that $DP_{2\text{-}4}$ and $DP_{2\text{-}3}$ don't manipulate the same A. By performing those two steps, one can ensure that the two DPs will not conflict with each other. However, $DP_{2\text{-}4}$ and DP_5 will generate thread interference or a memory consistency error if the conflict between them is not carefully managed. As the long purple dashed box shows in Table 2, both $DP_{2\text{-}4}$ and DP_5 attempt to write A_6 which results in a conflict if one tries to invoke two methods at the same time without calling synchronized methods. Generally, one can take three steps to decide whether two methods can be arranged for multithreading. The first step is to decide whether those two methods are dependent. If not, step 2 is performed to locate all attributes in one method which will be read or written from MDM table. The third step is to search for conflict between manipulations of attributes. As long as there is no conflict between methods and between attributes' operation, the two methods are safe for multithreading even if no synchronization process is called. In addition to helping to improve class performance via the multithreading technique, the MDM also helps eliminate careless logical errors committed by developers. Take A_6 as the example again, if the green box R appears in front of the red box W, it means the method $DP_{3\text{-}2}$ attempts to read an attribute before another method has written it. This creates a logical error.

MDM can be built only when all functional requirements have been decomposed thoroughly. If there is any un-decomposed FRs, the MDM will not be able to reflect the true structure of the class. Therefore, it's necessary to firstly implement an Axiomatic Design approach for decomposing, zigzagging and mapping followed by constructing a DSM and MDM to further reducing complexity.

5 CASE STUDY II

In case 2, class citMatrixConstruct is another class developed by the author that was used to calculate patent citation measures such as patent originality, generality, forward citation, etc. The result of the program is saved to database for further analysis. Some of methods in the UML chart below share exactly the same name as they are overloaded with different input parameters.

5.1 STEP 1: MAPPING FRS AT TOP LEVEL INTO DPS

The top functional requirements of this class are listed as follows:

FR_1: Create database and index table.
FR_2: Acquire a patent from USPTO web page.
FR_3: Calculate measures and Save them to the database
FR_4: Release memory

As stated earlier, DPs should be methods within a class because these methods will carry out tasks to satisfy certain requirements. Therefore, the following methods serve as DPs to satisfy the five FRs above:

DP_1: Method createDB and createIndexTable
DP_2: Method getPageContent
DP_3: Method StartCitMatrixConstruct
DP_4: Method reinit

The design matrix can be written as equation 8. Again, this design is a decoupled one.

$$\begin{bmatrix} FR_1 \\ FR_2 \\ FR_3 \\ FR_4 \end{bmatrix} = \begin{bmatrix} X & 0 & 0 & 0 \\ 0 & X & 0 & 0 \\ x & 0 & X & 0 \\ 0 & 0 & x & X \end{bmatrix} \begin{bmatrix} DP_1 \\ DP_2 \\ DP_3 \\ DP_4 \end{bmatrix} \tag{8}$$

Figure 7. Class CitMatrixConstruct UML diagram.

5.2 STEP 2: ZIGZAGGING AND DECOMPOSITION

Given the top-level DPs, FRs can be further decomposed into sub-level FRs. FR_2 can be decomposed into following lower level FRs:

$FR_{2\text{-}1}$: Retrieve page content from USPTO website

FR_{2-2}: Check if page content is correct.
FR_{2-3}: Acquire patent number.
FR_{2-4}: Acquire patent class.
FR_{2-5}: Acquire patent citation number.
FR_{2-6}: Acquire patent referenced by number.
FR_{2-7}: Acquire patent year.

Doing the same to FR_3, two low-level FRs must be satisfied:

FR3-1: Create table for saving patent title, class, citation, reference by, abstract, claims and description:
FR_{3-2}: Calculate all measures
FR_{3-3}: Save corresponding data to created table.

5.3 STEP 3: MAPPING LOWER LEVEL FRS INTO DPS

For FR_{2-1} to FR_{2-10}, the selected DPs are as follow.

DP_{2-1}: Method getContentPage
DP_{2-2}: Method checkBusyPage
DP_{2-3}: Method getContentPageNo
DP_{2-4}: Method getContentPageClass
DP_{2-5}: Method getContentPageLink
DP_{2-6}: Method getReferenceByNo
DP_{2-7}: Method getContentYear

$$\begin{bmatrix} FR_{2-1} \\ FR_{2-2} \\ FR_{2-3} \\ FR_{2-4} \\ FR_{2-5} \\ FR_{2-6} \\ FR_{2-7} \end{bmatrix} = \begin{bmatrix} X & 0 & 0 & 0 & 0 & 0 & 0 \\ x & X & 0 & 0 & 0 & 0 & 0 \\ x & x & X & 0 & 0 & 0 & 0 \\ x & x & 0 & X & 0 & 0 & 0 \\ x & x & 0 & 0 & X & 0 & 0 \\ x & x & 0 & 0 & 0 & X & 0 \\ x & x & 0 & 0 & 0 & 0 & X \end{bmatrix} \begin{bmatrix} DP_{2-1} \\ DP_{2-2} \\ DP_{2-3} \\ DP_{2-4} \\ DP_{2-5} \\ DP_{2-6} \\ DP_{2-7} \end{bmatrix} \quad (9)$$

For FR_{3-1} and FR_{3-2}, their corresponding DPs are

DP_{3-1}: Method createCitationMatrixTable
DP_{3-2}: Method getGenerality and getOriginality
DP_{3-3}: Method savePatentCitationMatrixToDB

$$\begin{bmatrix} FR_{3-1} \\ FR_{3-2} \\ FR_{3-3} \end{bmatrix} = \begin{bmatrix} X & 0 & 0 \\ 0 & X & 0 \\ x & 0 & X \end{bmatrix} \begin{bmatrix} DP_{3-1} \\ DP_{3-2} \\ DP_{3-3} \end{bmatrix} \quad (10)$$

5.4 STEP 4: MANAGING AND REDUCING COMPLEXITY WITH DESIGN STRUCTURE MATRIX (DSM) AND MULTIPLE-DOMAIN MATRIX (MDM)

Table 4 illustrates the MDM chart combined with DPs and attributes (As). As in the UML class diagram shown previously, there are private thirteen attributes (As). The left three attributes are private static ones that are used to invoke other classes. Hence, these three attributes will not be considered in this case.

A_1: PatentNo
A_2: Year
A_3: Class
A_4: Forward_cit
A_5: Backward_cit
A_6: Originality
A_7: Generality
A_8: Forward_imp
A_9: Forward_dis
A_{10}: Forward_lag
A_{11}: content_of_patent_citation_no
A_{12}: content_of_patent_reference_by
A_{13}: second_reference_by

To check dependencies between methods, simply follow the three steps again. First, check whether the prerequisite methods have been implemented in DSM (from left to right). If yes, proceed to step 2; if no, one needs to be careful because logical errors may happen. Second, check how many attributes (As) are related in the MDM with the methods that are to be checked (from top to down). The last step is to find out whether the writing and reading sequences for As are correct in the MDM (from left to right again). A variable should be initialized or written with some value first before reading; if the variable is written twice or initialized again before reading, one should be aware of the logical error.

6 CONCLUSION

In this paper, complexity within class is modelled and managed by means of an Axiomatic Design approach and DSM/MDM. Building a design matrix and an MDM does not require laborious work, but the benefits are evident. According to the IDC report in 2008, the estimated cost for fixing software defects was between 5.2 million dollars to 22 million dollars depending on the organization size. Another earlier released report showed that 72% of surveyed companies realized their debugging processes were problematic: 25.5% said they were very often or even all of the time finding serious problems. These statistics suggest that software companies face challenges in patching up the holes they've made. To save costs and build robust software, it is necessary to change the ways in which complex software components are developed. Instead of fixing bugs after software has been released, software engineers should design it in a reliable way and Axiomatic Design can help facilitate the design task. Complexity within classes can be managed with MDM, and complexity between classes can be modelled by UML. Combining those techniques together makes the goal of making software with fewer bugs feasible.

7 REFERENCES

[1] Bandi R. K., V. K. Vaishnavi and D. E. Turk, "Predicting Maintenance Performance Using Object-Oriented Design Complexity Metrics," *IEEE Transactions On Software Engineering*, Vol. 29, No. 1, pp. 77-87, 2003.

[2] Booch G., *Object-Oriented Analysis and Design with Applications (2nd Edition)*, Redwood City, Calif: Addison-Wesley Professional, 1993.

[3] Briand L. C., J. Wüst, S. V. Ikonomovski and H. Lounis, "Investigating quality factors in object-oriented designs: an industrial case study," in ICSE '99 Proceedings of the 21st international conference on Software engineering, 1999.

[4] Deitel H. M. and P. J. Deitel, *Java How to Program (6th Edition)*, Upper Saddle River, New Jersey: Prentice Hall, 2004.

[5] DSMweb.org, *Understand DSM*, http://129.187.108.94/dsmweb/en/understand-dsm/tutorials-overview/descripton-design-structre.html, 2009.

[6] Fothi A., J. Nyeky-Gaizler and Z. Porkolab, "The Structured Complexity of Object-Oriented Programs," *Mathematical and Computer Modeling*, Vol. 38, No. 7, pp. 815-827, 2003.

[7] Grady R. B., *Practical Software Metrics for Project Managment and Process Improvement*, Upper Saddle River, New Jersey: Prentice Hall, 1992.

[8] Kim K., Y. Shin and C. Wu, "Complexity measures for object-oriented program based on the entropy," in Software Engineering Conference, Asia Pacific, 1995.

[9] Kobryn C., UML 2001: A Standardization Odyssey, *Communications of the ACM*, Vol 42, pp. 29-37, 1999.

[10] Lange C., M. Chaudron and J. Muskens, "In Practice: UML Software Architecture and Design Description," *IEEE Software*, Vol. 23, No. 2, pp. 40-46, 2006.

[11] Lee T. and P. Jeziorek, "An Exploratory Study of Cost Engineering in Axiomatic Design: Creation of the Cost Model based on an FR-DP Map," in Third International Conference on Axiomatic Design (ICAD2004), Seoul, Korea, 2004.

[12] Maraia V., *The Build Master: Microsoft's Software Configuration Management Best Practices*, Redwood City, Calif: Addison-Wesley Professional, 2005.

[13] Martin R. C., "The Open-Closed Principle," *C++ Report*, Vol. 8, 1996.

[14] Meyer B., *Object-Oriented Software Construction (2nd edition)*, Upper Saddle River, New Jersey: Prentice Hall, 2000.

[15] Misra S., "An Object Oriented Complexity Metric Based on Cognitive Weights," in 6th IEEE International Conference on Cognitive Informatics, 2007.

[16] Misra S. and K. I. Akman, "Weighted Class Complexity: A Measure of Complexity for Object Oriented System," *Journal Of Information Science And Engineering*, Vol. 24, pp. 1689-1708, 2008.

[17] Noble J., "Objects and constraints," in Technology of Object-Oriented Languages, 1998. TOOLS 28. Proceedings, Melbourne, Vic., Australia, 1998.

[18] Siau K. and Q. Cao, "Unified Modeling Language (UML) - A Complexity Analysis," *Journal of Database Management*, Vol. 12, No. 1, pp. 26-34, 2001.

[19] Steward D. V., "The design structure system: a method for managing the design of complex systems," *IEEE Trans. Engineering Management*, Vol. 28, No. 3, pp. 71-74, 1981.

[20] Subramanyam R. and M. S. Krishnan, "Empirical Analysis of CK Metrics for Object-Oriented Design Complexity: Implications for Software Defects," *IEEE Transactions On Software Engineering*, Vol. 29, No. 4, pp. 297-310, 2003.

[21] Suh N. P., *Complexity: Theory and Applications*, New York: Oxford University Press, 2005.

[22] Suh N. P. and S.-H. Do, "Axiomatic Design of Software Systems," *CIRP Annals - Manufacturing Technology*, Vol. 49, No. 1, pp. 95-100, 2000.

[23] Togay C., A. H. Dogru and J. U. Tanik, "Systematic Component-Oriented development with Axiomatic Design," *The Journal of Systems and Software*, Vol. 81, No. 11, pp. 1803-1815, 2008.

[24] Withrow C., "Error Density and Size in Ada Software," *IEEE Software*, Vol. 7, No. 1, pp. 26-30, 1990.

[25] Zhao J., J. Cheng and K. Ushijima, "A dependence-based representation for concurrent object-oriented software maintenance," in CSMR '98 Proceedings of the 2nd Euromicro Conference on Software Maintenance and Reengineering (CSMR'98), 1998.