Ideality, TRIZ and Software Design – Case Study: Software Product for Identity Security

Navneet Bhushan

Hexaware Innovation Labs,
Hexaware Technologies,
Chennai, India
http://www.hexaware.com
Email: navneetb@hexaware.com

Abstract

Ideality as defined by Theory of Inventive Problem Solving (TRIZ) starts with a clear articulation of function to be achieved by the system. Once it is described, ideality is defined as function achieved without cost and with no negative impact on the system. Typically this trigger leads to strong solutions whereby the system achieves the desired function on its own – may be through the use of existing resources. Defining ideality in such terms for software products is inherently difficult due to evolutionary nature of the software systems. It is rarely the case that software system remains to the confines of functions that it was initially designed for. Invariably a software system evolves through addition of multiple functions or capabilities throughout its life. In fact, the graceful degradation of performance specific to the existing function of an evolving software system is more of a norm rather than exception. Further, it is really the evolution of its structure that helps the software system to perform new functions through addition of more and may be different pieces of structure to the system. In such cases, we propose besides the ideality as defined by function, ideality of the structure should also be taken into account. We propose the use of System Complexity Estimator and System Change Impact Model (SCE-SCIM) framework to take care of ideality of software systems. The paper describes the design process of a recent software product for identity securing developed at Hexaware Innovation Labs. The use of ideality from structural perspectives led to much cleaner design which is easier to maintain and evolve then the existing design.

Introduction

According to Brook's [1], the essence of software is a construct of interlocking abstract concepts. Complexity, conformity, changeability and invisibility are inherent properties of the essence of software. Given the radical differences in software development, in comparison to other engineering disciplines, complexity and changeability assumes greater importance. Thus the process of creating software and its very nature makes it prone to complexity. Two main sources of such complexity are functional and structural changes as per evolution of the system.

TRIZ (pronounced TREEZ) is the Russian acronym for the Theory of Inventive Problem Solving. It is a large collection of empirical methods discovered and invented through comprehensive studies of millions of Patents and other inventions for problem formulation and possible solution directions [2]. One of the very strong pillars of TRIZ is the quest for Ideality. TRIZ forces problem solvers to define the ideal system – which is defined as function achieved without resources and harm. It is extremely useful when we know the function that the system being designed need to perform. This is more or less easy for a system whose functionality once defined will not change – typically a hardware system, while it is being developed.

A software system by contrast is developed more in an evolutionary way - no one knows the final fine-grained functionality of the system upfront. We start with something and the system typically ends in something else. In such a scenario, what is the ideal system - do we need to look at the structure of the system rather than the function alone? The structure that should cater for least complex software system can be an ideal system?

In the current practice, there exists many different metrics for software complexity. Researchers have tried to measure the software complexity from the code complexity perspective as well as coupling and cohesion point of view, or even the disorder in the code defined as software entropy [3]. There is reported work studying relationship between software complexity and software reliability in literature [5, 7]. The relationship between software complexity as an interacting process of coupling and cohesion has been studied explicitly in [4]. The System Complexity Estimator (SCE) and System Change Impact Model (SCIM) as described in [16] quantify the structural complexity of the software system.

We propose the ideality of system structure to be an important constituent while designing the system besides the achievement of function. In a way this reflect the wisdom of ancients when they proclaimed *it is not only the end but the means as well that matter*. We believe that *means* in software system design are really the structure of the system that should be ideal besides the function achievement. We further propose in this regard, ideality should be congruent with simplicity or least complex software system. This paper describes our experiments with this line of thinking in specific software development scenario – especially a software product for identity security developed by Hexaware Innovation Labs (http://www.hexaware.com/akiva).

In Section 2 various software complexity measures described in literature are reviewed. In Section 3 the System Complexity Estimator (SCE) as described in [16] is explained as it forms the basis of software system ideality. Section 4 provides a brief discussion of TRIZ and ideality as simplicity or least complex. Section 5 describes a real life case study of development of AKIVA – an identity security software product. Section 6 describes the evolution of a more robust design for AKIVA towards ideality as defined by TRIZ and SCE framework. Conclusions and further areas of exploration are provided in Section 7.

2. Software Complexity Measures

In a software system, the complexity could emanate from unstructured nature of the software, the gap between actual requirements and requirement specifications, gap between requirements specification and design, and finally gap between design and actual implementation, i.e., source or executable code. Functional complexity measurement has been discussed in [6].

The structure of the software system is composed of multiple elements joined together to provide a system level functionality. The elements can be functions, independent modules, procedures, classes or objects. Their interaction is based on the content that they transfer to each other while the software system is executed. This content may be simple data, data structure, control information, functions or programs. The standard design guideline is that such coupling should be minimized. Further, each element of the software system should be cohesive to the extent possible. The complexity emanates from a lack of cohesion in each module and the strength of coupling between various modules [9, 10].

Modularity is central to the design and development of software system. Modular systems incorporate collections of abstractions in which each functional abstraction, each data abstraction, and each control abstraction handles a local aspect of the problem being solved [9, 10]. The coupling-cohesion criteria mean that the system is structured to maximize the cohesion of elements in each module and to minimize coupling between modules.

Coupling refers to the degree of interdependence among the components of a software system. Good software system should obey the principle of Low coupling. The cohesion of a module is defined as a quality attribute that seeks to measure the singleness of purpose of a module. Cohesion seeks to maximize the connections within a module. Composite Module Cohesion has been quantitatively defined in [11].

A measure of complexity called Software Entropy has been defined in [3]. This measure takes into account the disorder in the code. The disorder in the system depends upon the lack of cohesion in the modules, level of coupling between modules and complexity of the modules. Software Engineering Institute (SEI) [8] offers Maintainability Index (MI) which states [12] that a program's maintainability is calculated using a combination of widely-used and commonly-available measures to form a Maintainability Index (MI). Taking a cue from Social network analysis, [16] defines the system complexity estimator as a measure of complexity of the system. This is an integrated metric which combines coupling and cohesion of various components of a system.

3. System Complexity Estimator

Software system complexity is defined as a measure of non-cohesion of its constituent modules, and the interdependencies of modules [16]. This is closer to the design guideline of minimum coupling and maximum cohesion [9, 10]. The System Complexity Estimator defined in [16] computes overall complexity of the system using the centrality measures typically used in social networks analysis [13] to identify the relative importance of different actors based on their connectivity with the rest of the network.

The System Complexity Estimator (SCE) starts with the definition of an ideal software system which is defined as, “ A system with completely independent elements (modules) where each module performs a single function is the least complex architecture – this is the ideal architecture for a system. In such an ideal architecture/design the system complexity is minimized.” Ideally a module should perform only 1 function.

Further the SCE identifies two levels of complexities – one at the element/module level and other at the level of interdependencies between the elements. The non-cohesion is measured by cardinality of functions performed by each module. The more functions a particular module performs leads to less cohesiveness of the module. Second level of complexity is the interdependencies between system elements.

There are two kinds of dependencies that System Complexity Estimator takes into consideration. How much the module depends on the system for its functioning and how much all the modules (the system) depend on the module for their functioning. As an example, the dependency matrix (D) given in Table 1, describes a 4 module software system.

Table 1: System Dependency Matrix

	A1	A2	A3	A4
A1	1.0	0.5	0.0	0.0
A2	1.0	1.0	0.8	0.0
A3	0.0	0.5	1.0	0.0
A4	0.2	0.0	1.0	1.0

Once the dependency matrix (D) and number of functions performed by each module are obtained, next step is to construct System Complexity Matrix (X). Each element of X, i.e., x_{ij} is computed as

$$x_{ij} = d_{ij} \times H_j \qquad (2)$$

where d_{ij} is the i^{th} and j^{th} column element of matrix D; H_j is the non-cohesion of module j which equals the number of functions performed by j; x_{ij} is the i^{th} row and j^{th} column element of matrix X. The System Complexity Matrix for the example above is as given in Table 2

Table 2: System Complexity Matrix

Non Cohesion	3	2	1	6
	A1	A2	A3	A4
A1	3.0	1.0	0.0	0.0
A2	3.0	2.0	0.8	0.0
A3	0.0	1.0	1.0	0.0
A4	0.6	0.0	1.0	6.0

The overall system complexity (Ω) is the sum of all elements of the system complexity matrix. In the above example this comes out to be 19.4. To find out the relative contribution of each module to the overall complexity one need to take into account the dependencies in more detail.

As mentioned there are two kinds of dependencies mapping to compute the two corresponding indices. These indices are called Module Dependency on the System Index (MDSI) and System Dependency on the Module Index (SDMI). The corresponding element of the normalized eigen vector corresponding to principal eigen value of the System Complexity Matrix gives the MDSI for the respective module. Similar element of the vector obtained for the transpose of the System complexity matrix gives SDMI. The MDSI for the X matrix is given in Table 3. Similar computations for the transpose of the matrix will lead to SDMI. These values are shown in Table 4.

As can be seen from Table 4, the average of MDSI and SDMI gives the relative contribution of Module to overall system complexity in percentage terms. If we multiply these percentages (r) with the overall system complexity (C), we get the module complexity. Hence, modules A1, A2, A3, and A4 contribute 4.97, 5.66, 3.34 and 5.43 to the overall complexity, respectively.

Table 3: Normalized Eigen Vector corresponding to Principal Eigen Value

	A1	A2	A3	A4	MDSI
A1	0.45	0.25	0.00	0.00	0.18
A2	0.45	0.50	0.29	0.00	0.31
A3	0.00	0.25	0.36	0.00	0.15
A4	0.09	0.00	0.36	1.00	0.36

Table 4: Relative contribution of modules to system complexity

	MDSI	SDMI	Average (r)	Module Complexity = r x Ω
A1	0.18	0.34	0.26	4.97
A2	0.31	0.27	0.29	5.66
A3	0.15	0.19	0.17	3.34
A4	0.36	0.20	0.28	5.43

A radar plot or Kiviat chart of the complexity contribution of each module gives system complexity map of the software design, which is useful to find out the complexity imbalance in the software system. SCE-SCIM framework has been used to describe an approach for a Robust Inventive Software Design in [15] which combines Analytic Hierarchy Process (AHP) [14], TRIZ and DSM as an integrated framework.

We describe the application of SCE in Section 6 for evolving a software design towards ideality. A review of TRIZ and ideality is given in the next section to explain the conceptual understanding of ideality related to structure of the system besides the focus on the system function.

4. Theory of Inventive Problem Solving – TRIZ

TRIZ (pronounced TREEZ) is the Russian acronym for the Theory of Inventive Problem Solving. It is a large collection of empirical methods discovered and invented through comprehensive studies of millions of Patents and other inventions for problem formulation and possible solution directions. This proven algorithmic approach to solving technical problems began in 1946 when the Russian engineer and scientist Genrich Altshuller studied thousands of patents and noticed certain patterns. From these patterns he discovered that the evolution of a technical system is not a random process, but is governed by certain objective laws. These laws can be used to consciously develop a system along its path of technical evolution - by determining and implementing innovations [2].

TRIZ states that someone, somewhere, sometime has solved the problem that you are facing or a very similar one, it is now a much simpler task to search for the solution rather than thinking about solution with your limited exposure. By abstracting the inventiveness of thousands of inventors, TRIZ brings to the problem solver a plethora of robust techniques and methods that has worked in the past substantially. Solve by exploring in multiple directions but start from the end result – The Ideal Final Result – focus on functionality not features.

TRIZ has variety of techniques for problem formulation and problem solving. There are texts available that describe TRIZ in detail. Reader can refer to large body of knowledge at [17, 18]. Focus on Function – Main Useful function that product needs to deliver to meet a customer/user need. Value is nothing but Function delivered to meet a user need. Ideal Final Result – Value delivered at no cost or resource expenditure and not harming the system in anyway, alternatively the function is achieved on its own – self functioning system.

Given the distinct nature of software systems and their method of development, we propose, besides the ideality of function as defined by TRIZ, structural ideality should also be the objective in designing robust software systems. In this regard, we propose System Complexity Estimator (SCE) –framework for software design.

In the next two sections we describe a specific case study for an identity security software product. We show how the ideality thinking from TRIZ led to focus on structural ideality where in we used SCE to reach the software system towards an ideal product.

5. Case Study – AKIVA – An Identity Security Software product design

AKIVA a GUI driven tool is designed to 'de-identify' personal and sensitive data required for use in a variety of situations such as software development, implementation and testing and outsourcing. It allows creation of disguised copies of production databases and provides realistic and fully functional databases without compromising on confidentiality. It offers additional level of data protection beyond firewalls and encryption. AKIVA is a data scrambling tool to mask Enterprise Database Applications. It maintains Data Consistency, Data Security and Data Integrity while masking data across the enterprise application.

Akiva features

- **Data consistency** – Akiva masks data consistently across the PeopleSoft enterprise, so that the same entity relationship is maintained post masking
- **Ability to choose any data element** – Enables data security officers to choose any of the sensitive data elements across PeopleSoft enterprise online using Akiva. This includes vanilla and customized components.
- **Data security** – Data masking algorithm is not static in nature, Akiva accepts unique 16 digit numeric token key as input for masking
- **Wide coverage** – Akiva supports all modules and pillars for PeopleSoft. Data security officers can use the same tool to mask sensitive information in their HRMS, NA Payroll, Benefits, SCM, financial applications.
- **Data integrity** – Akiva masks PeopleSoft enterprise data without impacting any of the business process validations
- **Secured** – Akiva does not store any of the masking information including the token key in the system.

Algorithms implemented

- **Scramble –** Arithmetically generate new values in required field format based on the input token key
- **Combo Shuffle –** Join a group fields and shuffle together based on a lookup table (e.g.)

Address 1, Address 2, Address 3, City, State, Zip code.

- **Selective Shuffle –** Replace sensitive values with meaningful, readable data based on a lookup table. Shuffle is based on a selection criteria (e.g.) Shuffle female names and male names separately
- **Replacement –** Simply replaces a field value with a static value provided.
- **Blank out –** Simply replaces a field value with a static value provided.
- **Lookup -** Replace employee names and addresses choosing from an inbuilt repository
- **SSN Generator -** Generate valid US Social Security Numbers for all employees
- **Luhn Generator -** Generate numbers satisfying Luhn checksum condition
- **Pattern Generator -** Generates a set of numbers based on user-defined pattern

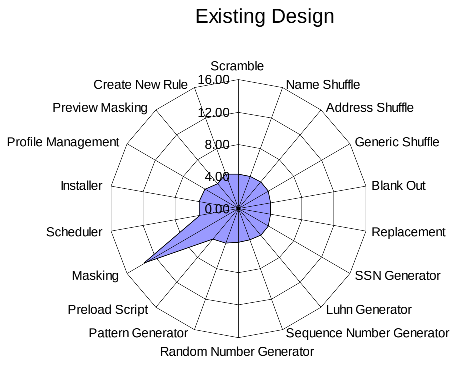
Environment

The web application has been developed using J2EE framework and the masking algorithms have been implemented using PL/SQL Procedures in ORACLE database.

6. Case Study – AKIVA – Evolution towards Ideality

The *existing AKIVA design* has an estimated code size of 8000 lines of code. It is composed of 18 modules performing a total of 54 unique functions. The average number of functions per module comes out to be 3.0. When we study the System Complexity using the SCE, the system complexity came out to be 88.7. The system complexity map is shown below.

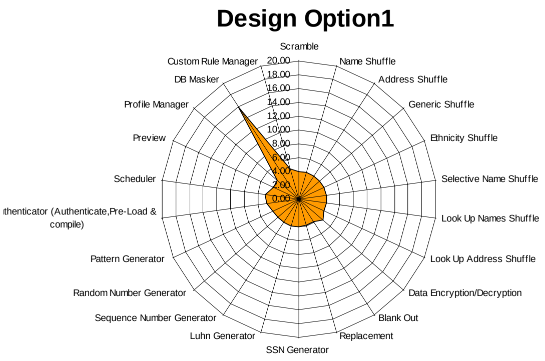
System Complexity Map for Existing AKIVA design

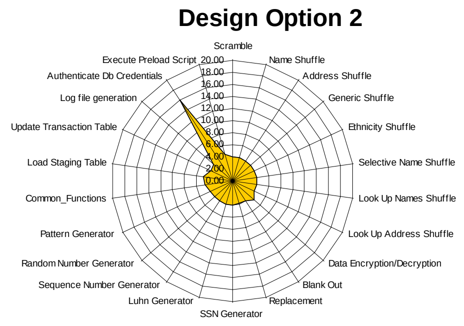


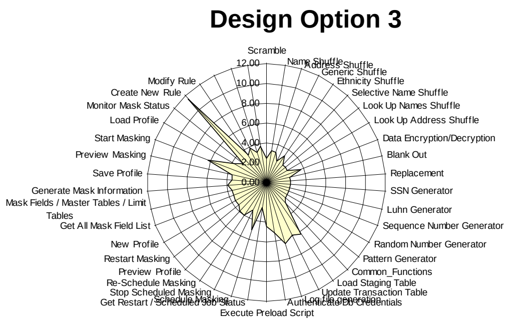
One can see the complexity imbalance created by the module *Masking* which contributes maximum to the overall complexity. In the ideal system, the complexity should be closer to number of functions being performed which is 54.0.

Generating alternative designs based on SCE – moving towards ideality

Looking at the complexity, the team brainstormed to look at alternative designs for minimizing the system complexity. Three alternative designs evolved as shown below. Design option 1 had 22 modules while option 2 and option 3 had 39 and 42 modules respectively.



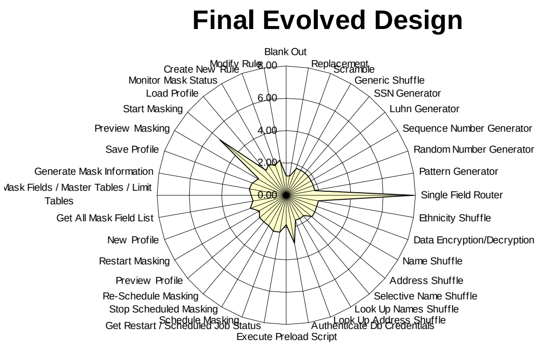




As we increased the number of modules, the functions per module reduced – it came out to be 3, 1.3 and 1.2 respectively for Design option 1, Design option 2 and Design option 3, respectively. Compared to existing design this was definitely an improvement. However, overall complexity of the product in all three Design options actually increased to 102, 174 and 155 respectively. This was definitely way beyond the ideality. It increased because of increase coupling between various modules.

A design that evolved nearer to ideality as defined by its complexity

The team brainstormed further to look at ways and means of reducing the coupling. The team hit upon the idea of a router, which was suggested earlier during the discussions but somehow was not pursued. The system complexity map of final evolved design is shown below.



In the final evolved design, which is the existing design, there are 36 modules performing 45 functions, giving 1.3 functions per module on an average. The overall complexity is reduced to 81 from 89 in the original design. This is a much cleaner design and easier to maintain. *The most useful result however is reduction in lines of code from 7964 to 3866.* This is more than 50% reduction in code size. This helps in creating only the needed coding rather than increasing the code size un-necessarily. The table below gives the lines of code in the modules of original design and final evolved design.

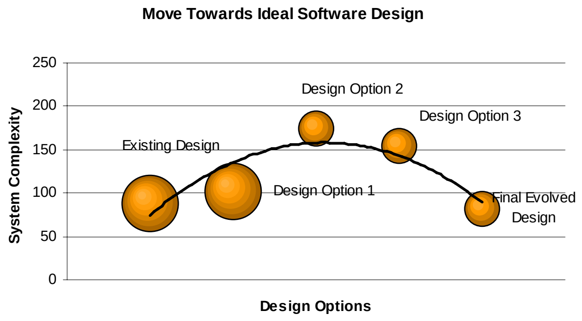
Algorithm	Lines of code	
	Before	After
Blank Out	562	109
Replacement	575	122
Generic Shuffle	1435	211
SSN Generator	1812	542
Scrambling	2453	974
LUHN	1127	168
Log	0	150
Single field Router	0	1590
Total	7964	3866
Difference in Code		4098

The table below summarizes the system complexity analysis of existing, alternatives design options and final evolved design. As one can see the final evolved design is not only closer to design guideline of highly cohesive modules but also coupled to the optimal need.

Complexity Size Design Evaluation

Design	# of functions	Size (# of Modules)	Avg Functions/ Module	Complexity
Existing Design	54	18	3	88.7
Design Option 1	66	22	3	102.3
Design Option 2	51	39	1.3	174
Design Option 3	51	42	1.2	154.6
Final Evolved Design	45	36	1.3	81.2

The chart below plots system complexity of all the design. As one can see the evolved design of AKIVA has least complexity. The bubble size of each design option indicates the average cohesion as defined by number of functions per module. Here also the final evolved design comes out to be 1.3 functions per module.



7. Conclusions and Further Work

The paper describes software product design case study. The design is for a product for identity security. Using the ideality concept objective from TRIZ thinking, we propose that for software systems it makes more sense to look at structual ideality rather than achievement of function alone. We have used the system complexity estimator for evaluating various design alternatives to evolve to a final software system which is closer to ideality. This approach not only produced a more robust and maintainable software product, but reduced the code size by more than 50%. This is a highly desirable result as the demands on software development productivity are becoming intense day by day. Further the SCE framework can be used to minimize the complexity of other non-software products as well.This is the future area of research that we will be conducting.

References

1. Brooks F.P., *The Mythical Man-Month - Essays on Software Engineering*, Addison Wesley, 1995.
2. Bhushan N., Set-Based Concurrent Engineering and TRIZ – A Framework for Global Product Development, *Proceedings of TRIZCON 2007 – The Ninth Annual Conference of Altshuller Institute for TRIZ Studies*, 23-25 April 2007, Louisville, Kentucy, USA.
3. Bhushan N. and Kaushik V., Software Entropy – Definition and Applications, *Workshop series on Empirical Software Engineering*, (Ed.) Bunse C. and Jedlitschka A., Fraunhofer IRB Verlag, 2003, ISBN 381676374X.
4. Darcy D.P. et al, *The Structural Complexity of Software: Testing the interaction of Coupling and Cohesion*, http://littlehurt.gsia.cmu.edu/gsiadoc/WP/2005-E23.pdf, accessed on 15th March 2005.
5. Lew K.S., Dillon T.S. and Forward K.E., Software Complexity and its impact of Software Reliability, *IEEE Transactions on Software Engineering*, Vol 14., No. 11, November 1998.
6. Tran-Cao D., Abran A. and Levesque G., *Functional Complexity Measurement*, Proceedings of International Workshop on Software Measurement, August 28-29, 2001.
7. Bhushan N., *Balancing Reliability and Software Complexity – Can TRIZ help?* International Conference on Quality, Reliability and Information Technology (ICQRIT), December 2003.
8. http://www.sei.cmu.edu
9. Fairley, R., *Software Engineering Concepts*, McGraw Hill, 1985
10. Shooman, M.L., *Software Engineering*, McGraw Hill, 1983.
11. Patel S., Chu W., and Baxter R., *A Measure of Composite Module Cohesion*, ACM conference, 1992.
12. http://www.sei.cmu.edu/str/descriptions/mitmpm.html
13. Bonacich, P.B., Power and Centrality: A Family of Measures, *American Journal of Sociology*, 92, 1170-1182, 1987
14. Bhushan, N. and Rai, K., Strategic Decision Making – Applying the Analytic Hierarchy Process, Springer UK, 2004.
15. Bhushan, N., Robust Inventive Software Design (RISD) – A Framework Combining DSM, TRIZ and AHP, 7th International Conference on Dependency Structure Matrix, Seattle, US, October 2005, http://www.dsmweb.org/workshops/DSM2005/dsm05conf/presentations/day_3/01_Day3_Navneet_Bhushan.pdf (accessed on July 4, 2006)
16. Bhushan N., *System Complexity Estimator - Applications in Software Architecture, Design and Project Planning*, 3rd International Conference on Quality, Reliability, Infocom Technology, ICQRIT, 2-4 Dec 2006, New Delhi, India.
17. http://www.aitriz.org (The Altshuller Institute)
18. http://www.triz-journal.com

*****