

# **SOFTWARE ENGINEERING AND TRIZ (1) STRUCTURED PROGRAMMING REVIEWED WITH TRIZ**

**Toru NAKAGAWA**

(Osaka Gakuin University, Japan)

## **ABSTRACT**

This is the first report of our research having three-folded purposes as follows: (1) to apply TRIZ to the problems related to software development and to extend the application field of TRIZ into software development and software engineering, (2) to clarify topics of software engineering with the TRIZ views, and further (3) to feed the principles/knowledge in software engineering/computer science back into TRIZ. For these purposes, we are going to make an approach of choosing topics in software engineering one by one and to consider about it with the whole aspects of TRIZ, including Inventive Principles, Inventive Standards, Trends of Evolution, and philosophical elements in TRIZ.

The concept of Structured Programming was examined in this paper. When we see the historical disputes on the "Goto-less" issue from the TRIZ' viewpoint of contradictions, we have found it not appropriate to teach: "the Structured Programming is a proposal to use only three basic control constructs and no Gotos, and it is a compromise with practice to add four more constructs" (as is often taught in computer science classes). It is because only with these additions Structured Programming has its sound bases. On the other hand, the approach of Structured Programming urges TRIZ especially in the following points: (a) TRIZ Principle 1 (Segmentation) should be extended to reflect the idea of Stepwise Refinement; (b) TRIZ Principle 6 (Universality) should be extended to include the idea of establishing/using industrial standards (this has been a blind point of TRIZ due to so much stress on inventions); (c) TRIZ Principle 7 (Nesting) should be regarded more important as reflecting the hierarchy concept of systems.

# 1. INTRODUCTION

It has been desired for a long time to make TRIZ applicable to the issues related to software development and software-based technology systems. Since TRIZ was originally developed in the physical/chemical or hardware-based technology fields, it has not been so clear how to apply TRIZ to software-based technology and how effective it is in such a field. Publications in this direction are still only few:

Kevin C. Rea has published the 'Software Analogies of TRIZ 40 Principles' [1] in the TRIZ Journal and has shown a lot of IT and software examples interpretable in each of Inventive Principles (the table of software analogies was recently extended by Ron Fini [2]). Since he is a software professional in the IT and communication technology, his examples often discuss cutting-edge techniques in such fields in a concise manner (and unfortunately making it hard for ordinary readers in software field or in TRIZ). He has obtained several patents by applying TRIZ to software technology but has not been allowed to disclose them, he writes in his recent short note [3].

The applicability of TRIZ was discussed by Graham Rawlinson [4] especially in the scope of contradictions among speed, reliability, energy used, and complexity of devices, etc. Lately Herman Hartman et al. [5] discusses that TRIZ should be used in the creation of software architectures where the critical problem is to overcome complexity rather than speed.

Darrell Mann has published a paper 'TRIZ for Software?' [6] in the TRIZ Journal, which serves as a summary of his upcoming book "TRIZ for Software Engineers" [7]. In Ref. [6] he writes that he has analyzed around 40,000 US patents in software and has experiences of over 18 months of teaching an in-house 'TRIZ for Software' workshop. For the field of software technology, the paper describes philosophical pillars, newly tailored Contradiction Matrix, reverse-engineering cases of solving contradictions with the Matrix, summary of slightly modified Inventive Principles and their frequencies of appearance, Trends of Evolution, and some other TRIZ tools. Thus the paper goes much further than the common understanding in TRIZ and software communities, but since it is just a short overview article, his book needs to be awaited for better understanding of his methods.

Under these situations, the present author started to publish a series of research notes on 'Software Engineering and TRIZ' [8, 9] in Japanese in the TRIZ Home Page in Japan. The research note series chooses major topics in software engineering one by one for reviewing them from the perspective of TRIZ and for reflecting TRIZ back from the perspectives of computer science and software technology. This style of research is taken because the

method how to apply TRIZ to software field is not clear yet, and is found effective in understanding the deep correspondences between the two disciplines/methodologies. It also has a merit of not being restricted by the non-disclosure agreement as a consultant to industries. The present paper is an English version of the first of the research note series.

For the basis of discussing major topics in software technology, a Japanese textbook "Program Engineering -- Implementation, Design, Analysis, and Testing" [10] is chosen. This textbook of 154 pages is written by Osamu Shigo for undergraduate classes of computer science majors. The author intends to expand students' knowledge of programming step by step towards wider areas of software technology, thus in the reverse order to ordinary texts of software engineering and without handling the issues of project management. In Ref. [8], the texts from [10] are fully quoted first and then explained some more in the context of software engineering for non-specialists in the field before discussing from the perspectives of TRIZ. In the present paper the software engineering section is explained briefly borrowing some examples from [10] on the topic of Structured Programming.

The aims of the present paper are three-folded as follows:

- To apply TRIZ to the problems related to software development and to extend the application field of TRIZ into software development and software engineering.
- To clarify topics of software engineering with the TRIZ views, including all the aspects of Inventive Principles, Inventive Standards, Trends of Evolution, philosophical elements in TRIZ, etc.
- To feed the principles/knowledge in software engineering/computer science back into TRIZ.

## **2. Structured Programming in the Software Engineering Context**

Near the end of 1960s when the scale of software development was getting too large to handle in reliable ways, the discipline of software engineering started. Programming languages like COBOL and FORTRAN were mostly used for business and scientific applications. To make the programs efficient both in time and in memory space, it is typical to use Goto statements to build complicated procedures even though being blamed to be spaghetti programs. E. Dijkstra [11] claimed 'Goto statements harmful' for making the program easy to understand and advocated 'Structured Programming' so as to eliminate Goto statements and use instead only three basic control constructs, including Sequencing, Selection, and (Pre-condition) Loop. The programming method was backed up with the philosophy of 'Step-wise Refinement', with which programmers were encouraged to design/code program modules step by step in a top-down style.

Many programmers at that time argued against Dijkstra's proposal, claiming "Writing programs with the only three control constructs is not easy. Goto statements are still necessary in various cases". Dijkstra and his school showed the proof that any program can be rewritten with the three control constructs. (Roughly speaking, the forward jumps (or Gotos) can be written with the Selection while the backward jumps with Loops.) For rewriting the programs with Goto statements, two techniques are applicable: one is to copy a part and use it in duplicate, and the other to introduce a control variable for regulating the flow. The so-called 'Goto disputes' continued historically for a few years.

Meanwhile on a practical basis, a few additional control constructs were proposed, including multi-path selection, post-condition loop, interruption of a loop, and exception. Thus the proposal of Structured Programming was compromised with the practice by including these additional control constructs, and the disputes diminished gradually. [This is the way typically taught in computer science classes.]

It is understood in the Structured Programming that a procedure having one inlet and one outlet should be refined step by step using the three basic control constructs as shown in Fig. 1 in the style of flowcharts. Additional control constructs are also shown in Fig. 2. [It should be noted that even though Shigo uses the C language to show these constructs, the present author draws the figures in the flowchart style for easier understanding for non-specialists in software and for clarifying the procedural flows.]

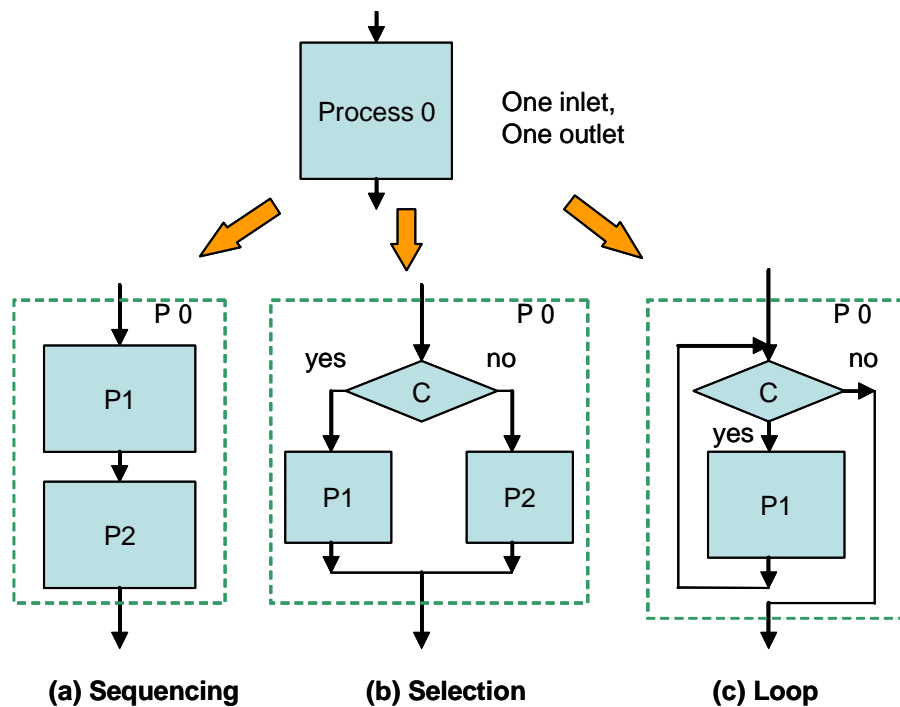


Fig. 1 One stage of refinement of a process with 3 basic control constructs

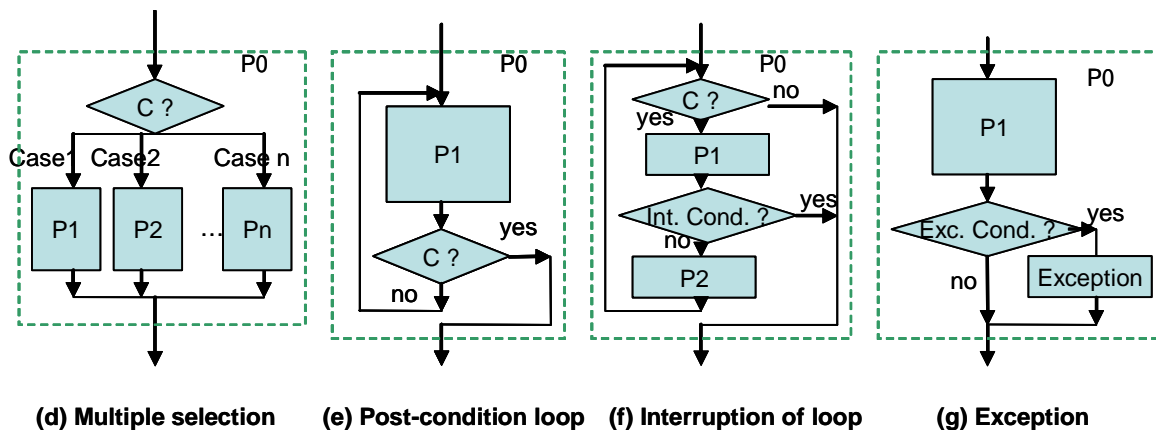


Fig. 2. Four additional control constructs

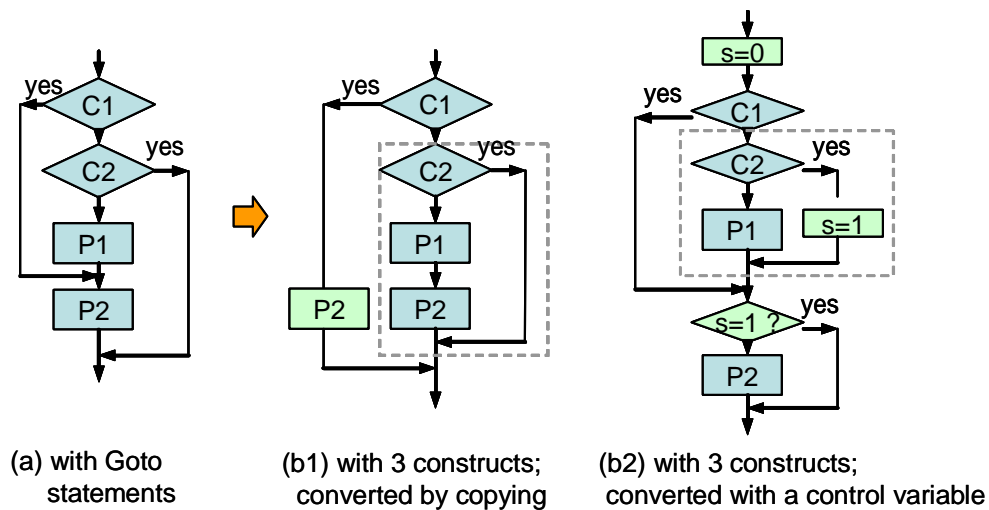


Fig. 3. An example of program fragment written in three different ways

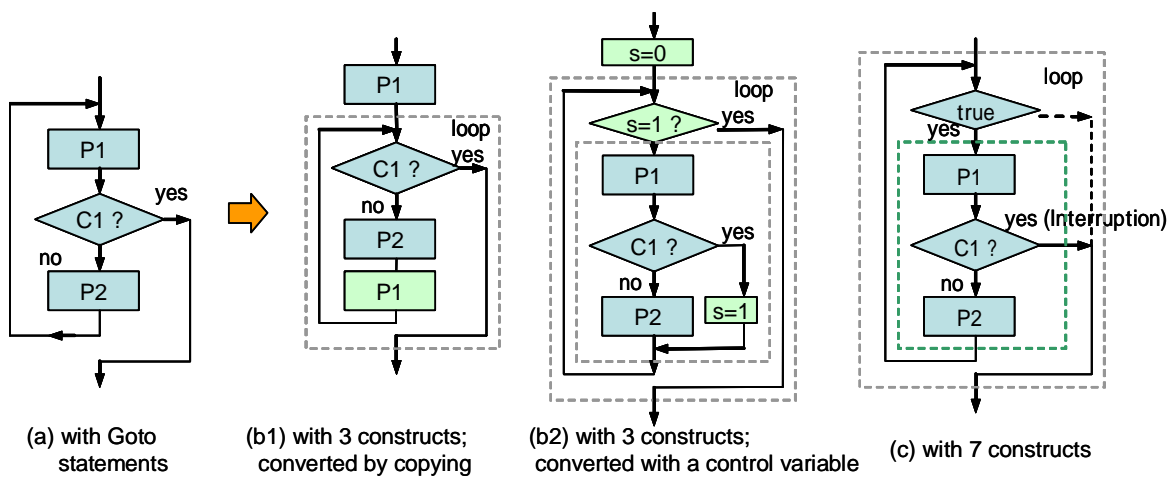


Fig. 4. An example of program fragment written in four different ways

To clarify the arguments, simple cases are demonstrated in Figs. 3 and 4 by comparing the flowcharts drawn in four different styles: (a) Goto style, (b1) original Structured Programming style rewritten with a copying technique and (b2) with the control variable technique, and (c) with additional constructs. [It should be noted that the flowcharts (b1) and (b2) are not easy to read]

It is important to clarify what is the final conclusion of the Goto-less dispute. Unfortunately, this point has not been stated clearly in computer science or computer science education. Four additional control constructs shown in Fig. 4 are really necessary? Are they sufficient? What are the criteria for addition and for rejection? Should we follow to the practice or to any theory? Because of the ambiguity of this final point, computer science education has conventionally taught: "Structured Programming is the theory as originally proposed with the three constructs and is compromised with practice by adding four or so extra constructs." This point will be discussed later in Section 3.5 with TRIZ views.

### **3. Structured Programming Viewed with TRIZ**

Now let us start to view the Structured Programming from the perspectives of TRIZ.

#### **3.1 Needs of Overcoming Complexity for Achieving Reliability**

From the TRIZ perspective of evolution of technical systems, we can clearly see the needs of emergence of software engineering and Structured Programming. Like any other technical systems, the software systems in general grew rapidly by making themselves larger and more powerful in their functionality with the sacrifice of becoming complex. Huge and complex software systems urged software scientists to recognize 'software crisis'. For further evolution of software systems it became essential to develop some methods to make them simpler and easier to handle. Such an era of turning point can be seen in TRIZ [12] as:

- Every technical system evolves by increasing its complexity and then reducing it.
- The focus of customers' (or users') interests shifts in the following four steps:  
Performance --> Reliability --> Convenience --> Price

#### **3.2 Simplifying Systems by Using Hierarchical Structure**

A software system or a software module can be seen as a 'System' in its general meaning in TRIZ (among others). A system is the whole having components working together to

perform a function, which produces some outputs by using some inputs. The concept of hierarchical structure of systems is one of the basic understanding in TRIZ. It is stated in the Inventive Principles as:

- Inventive Principle 7. "Nested doll" or Nesting
  - A. Put one object or system inside another.
  - B. Put several objects or systems inside others.

Even though Principle 7 seems to be relatively limited in use in the physical world, it plays a major role in the software world where hierarchical construction of software systems is easy and prevailing. The concept of Step-wise Refinement encourages to develop a software system in a hierarchical structure in a top-down manner.

Now we should think what TRIZ would suggest/recommend to simplify any system when it becomes too complex. Three methods can be thought of: (a) Inventive Principle 6. Universality, (b) Inventive Principle 1. Segmentation, and (c) Trimming.

TRIZ Inventive Principle states:

- Inventive Principle 6. Universality.
  - A. Make an object or system able to perform multiple functions; eliminating the need for other systems.

The idea of universality for simplifying complex systems, however, has a slightly different nuance, which may be stated as:

- 'Introduce basic and standardized functional units and use them universally in a wide range of applications.'

This may be called 'Universal Standards', and is widely used in the fields of technology (such as screws, batteries, etc.) and society. The proposal of Structured Programming may be regarded as the introduction of standardized control constructs in the representation of procedural flow.

TRIZ Inventive Principle also states:

- Inventive Principle 1. Segmentation
  - A. Divide a system into separate parts or sections.
  - B. Make a system easy to put together and take apart.

This principle can be applied to physical technical systems, social and human issues, as well as software systems. It is quite natural for us to think about a complex issue by dividing it into smaller parts.

### **3.3 Trimming of the Goto Statements**

Trimming technique in TRIZ is applicable to simplify complex systems. The proposal of Structured Programming can be regarded as Trimming of the Goto statements at the level of programming style, instead of the level of individual software programs. When a component of a system is harmful or unessential, it may be trimmed by answering the following two questions:

- Does the component to be trimmed perform any useful function in the system?
- If yes, can the useful function be performed by any object within or around the system, or by anything newly introducible?

Following these steps, we find the Goto statements have useful functions of making the procedural control jump to any place we want in the program. The Structured Programming originally proposed to replace them with the three basic control constructs. But many software professionals argued that the three constructs were not powerful/convenient enough to substitute the Goto statements. And the Goto-less dispute was settled only after a few control constructs (shown in Fig. 2) were added. Thus, in the context of Trimming method in TRIZ, it is clear that the Goto statements were trimmed in the programming style (or in the programming language) not by the three basic constructs in the original proposal of the Structured Programming but by the set of constructs including the additional ones.

### **3.4 Goto-less Dispute Reviewed with TRIZ Contradiction Concepts**

It is also worthy of reviewing the Goto-less dispute with the TRIZ Contradiction concepts. The dispute had the opposite arguments as:

- The Goto statements often make the procedural flow in the programs too complex. They are harmful for writing the programs easy to understand, and hence they should be eliminated.
- The Goto statements are necessary and convenient to represent various procedural flow in the programs. They must be used, and must not be eliminated.

Being typical in the dispute among groups of people, the dispute was elevated to a situation known in TRIZ as a Physical Contradiction:

- Goto Statements should completely be eliminated, in order to make the representations of procedural flow easy to understand.
- Goto Statements are necessary, in order to represent various procedural flows.

For solving such cases of Physical Contradictions, TRIZ offers the Separation Principle:

- Separation Principle:  
Inquire the two opposite requirements in the Physical Contradiction when, where,



and under which condition they apply, and find any difference in the requirements. If any difference is found, set up two separate solutions which fully satisfy each one of the requirements, and then try to find some method to use the both solutions together in combination.

At the initial stage in the Goto-less dispute, the both sides answered that their requirement should be satisfied 'everywhere, every time, and under any situation in the programs'. It is characteristic in this stage that the advocates of Structured Programming requested to eliminate all the usage of Goto statements by replacing them with the three constructs, and the opponents resisted against it. The arguments of the both sides were not examined well in the contexts of place, time, and situations in the programming.

Elimination of any contradiction can only be achieved by sharing some common goal/purpose at a level higher than the level of contradiction. The common goal in this case was found as:

- Common goal: To make the programming style/method easier to understand and more likely to be error-free.

This goal was posted strongly by the Goto-less advocates, and was not opposed in the software community. Thus the logical jump in the advocates' argument from 'Goto statements harmful' to 'No Goto statements' should have been examined more closely.

The advocates argued: "The three basic control constructs are easy to understand and capable to represent any procedural flow, as being proven with the Theory of Structured Programming." However, it was not logically sufficient to show the capability of representing any procedural flow; the focus of argument should have been showing: "The new style of programming is easier to understand/use and more likely to be error-free." If there are cases where the logic of the program rewritten in the Structured Programming style becomes more confusing, then the original proposal should have been revised

Furthering the disputes, the two sides became able to answer to the TRIZ Separation Inquiries in more logical ways such as:

- Goto statements should be eliminated, by substituting them with appropriate control constructs which guarantee proper nesting of procedural blocks. Goto statements which cause skewed nesting of procedural blocks must be prohibited.
- The functions of Goto statements which represent multiple selections, the post-condition loops, interruption of loops, and exception handling, etc. should find some proper constructs in the new programming style.

Reaching this stage, the following solution was formed as the consensus in the software

community:

- In the Structured Programming, the four control constructs shown in Fig. 2 are added to the original three constructs shown in Fig. 1. Goto statements are to be eliminated completely by replacing them with the Structured Programming constructs, which are designed to guarantee not to make skewed nesting of procedural blocks.

This is the final solution to the Contradiction of 'Goto vs. No Goto'. Thus it is clear in the TRIZ view as:

- The original proposal of Structured Programming with three basic control constructs is insufficient.
- The Structured Programming as an important guiding principle in the current computer science is the one typically having seven control constructs as shown in Figs. 1 and 2.

### 3.5 Final Solution. What is Structured Programming, Really?

We are now at the proper position to discuss about the final solution of the Goto-less dispute, and to clarify "What is Structured Programming, as the important guiding principle in the current computer science?"

For making it clear, we should better think what has been really prohibited (or avoided) in the modern programming language established after the dispute. It is the conclusion by the present author that skewed overlapping of blocks of procedural flow is the one prohibited strictly. Fig. 3 (a) is a typical case of using Goto statements in the way which must be prohibited in the Structured Programming. In this case, the blocks of flow, i.e. branches of the two Selection statements, are overlapping in a skewed manner, and violating the tree structure of the hierarchy of procedural blocks (See Fig. 5).

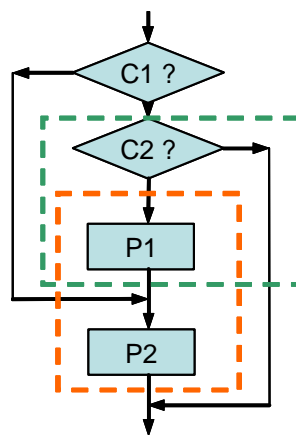


Fig. 5. Example of skewed nesting of procedural blocks prohibited in Structured Programming

It should be noted here that the tree structures of the hierarchy of procedural blocks are recognized to be easier to understand than the skewed overlapping structures or the network-type structures in the hierarchy. They are also interpretable as genuine nesting of blocks. Modern programming languages designed later do not have the Goto statements and have the grammatical restrictions of control constructs for prohibiting the skewed structures. Thus programmers in recent days enjoy the principle of Structured Programming without noticing it.

In this manner, Structured Programming encourages or guarantees the tree structure in the hierarchy of procedural blocks in each program module. Then, is there any method for practically allowing the overlap of procedural flows? There is. Procedural calls of subroutines and functions are the methods for allowing the overlap of procedural flows. Thus we may say division into software modules is the strategy to keep the simplicity inside the modules and to reserve flexible and interchangeable usage of modules.

### **3.6 Structured Programming Viewed As a Movement**

The proposal of Structured Programming gave a strong impact on the software community at that time, because it prohibited from using Goto statements which were thought indispensable for programming. But the original proposal of the three basic control constructs was later found insufficient to replace the Gotos. Then, what kind of role did it play in the history of software engineering? This question reminds me of the following TRIZ Inventive Principle:

- TRIZ Inventive Principle 16. Partial or Excessive Actions
  - A. If exactly the right amount of an action is hard to achieve, use 'slightly less' or 'slightly more' of the action, to reduce or eliminate the problem.

We may now think of the whole software community as a system and plot its behavior in Fig. 6. Before the proposal of Structured Programming, people in the community used Goto statements at their will without restriction. Dijkstra chose a 'drastic strategy' in proposing the elimination of Goto statements by allowing only three control constructs for the substitutes. In response, the community showed oscillating behavior during the Goto-less disputes and it took a few years for them to reach at a balanced level of using control constructs, as shown by curve (a) in Fig. 6. We might imagine now that the movement of Structured Programming could have taken different strategies such as 'mild and slow' and 'just slightly excessive' strategies which might have induced society's responses as shown in two other curves in Fig. 6. A drastic strategy usually has the merit that the issue becomes known widely, in spite of the demerit of necessity of overcoming

severer struggles.

Anyway, Fig. 6 is drawn with the standpoint that the original proposal of Structured Programming was not the goal of the movement nor the final equilibrium position of the software community. The figure asserts that the software community has already reached the equilibrium position and that the position is not a result of compromise between the theory and practice but should be the goal of the Structured Programming movement.

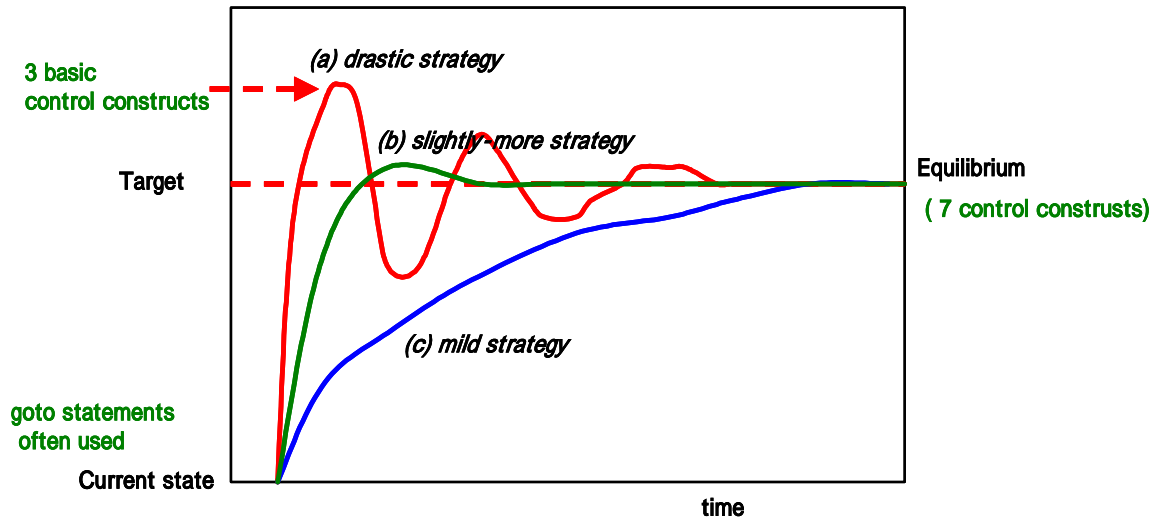


Fig. 6. Hypothetical responses of software community in reaction to three different strategies

## 4. Feedback from Software Engineering to TRIZ

While viewing the topic of Structured Programming in software engineering from the TRIZ perspective as described above, we have been inspired a lot by software engineering or computer science as feedbacks to TRIZ. When we apply TRIZ to social and human issues, we may of course have reflections to TRIZ. In comparison with them, the reflections we obtain back from the software field are unique because software is the objects of logical thinking in the field where computer science has been built already. Thus it must be important to feed the principles in the software field back into TRIZ. Some of such suggestions are rather too big to settle them soon in TRIZ.

### 4.1 Concepts of Procedural Structures and Algorithms

Since TRIZ has much developed the concepts of systems and functions, there should be no surprise when we find them applicable to the concepts of procedural structures and algorithms in software. However, the concepts of systems and functions in software are

unique among other fields of technology in their flexible and constructive ability to define and implement them. Wide variety of procedural functions can be constructed easily with different forms of inner structures. Such a flexible and constructive ability has a possibility of deriving new concepts and methods, for instance hierarchical and recursive structures of systems.

## **4.2 Step-wise Refinement back to TRIZ Segmentation Principle**

As mentioned above in Section 3.2, the concept of Step-wise Refinement is a core guiding principle in software field for clarifying complex problems/requirements and for designing software systems. Though TRIZ Inventive Principle 1 (Segmentation) corresponds partly to this concept, it should better be extended further by adding a sub-principle as follows:

- TRIZ Inventive Principle 1. Segmentation  
D (New): Divide the problem into several parts, and consider the parts and the relations among them.

This principle is applicable to a wide range of problems in the fields of software, society, human, and technology in general.

## **4.3 Extend Nesting Principle for Representing Hierarchy of Systems**

TRIZ Inventive Principle 7 (Nesting) does not play a so big role in the physical world of technology. The concept of hierarchy of systems built up with repeated nesting in software should be introduced into this Inventive Principle as an extension:

- TRIZ Inventive Principle 7. Nesting  
D (New). Implement a system in the multi-layer hierarchy of nesting.

There may be an argument that this extension of Inventive Principle is unnecessary because TRIZ already has a concept of multi-layer hierarchy of systems in its basic philosophy. However, since the Inventive Principles have much popularity in the TRIZ community, especially among novices, the present author believes proper extension of Inventive Principles (as well as other TRIZ tools) is worthy of doing.

## **4.4 Avoiding from Skewed Nesting**

One of the core points of Structured Programming has been found 'Avoiding from Skewed Nesting' in the construction of software, especially inside the modules. In the physical fields of technology this may suggest to avoid from skewed nesting in the construction of technical systems in the sense of space, functionality, etc. For instance it may suggest that subsystems should not occupy spaces in skewed overlapping manners and that subsystems

should not carry functions in skewed overlapping ways.

These suggestions remind us of the guiding principles in Axiomatic Design [13] in the field of technology:

- Good design achieves independence between different functional requirements, and
- Good design achieves the functional requirements with minimal complexity.

It is remarkable that Structured Programming proposed in the abstract level a guiding principle similar to the ones in Axiomatic Design in much earlier time. This encourages us to view theories/principles (like TRIZ) in physical technologies from the eyes of computer science. The present issue of avoiding from skewed nesting should be discussed further in the future as computer science (or software engineering) has made a much further progress.

#### **4.5 Concept of Easiness to Understand**

In TRIZ, the concept of Complexity has been important to understand the evolution of systems. Technical systems evolves at first by increasing its complexity and then after a certain stage by reducing its complexity, or by increasing simplicity. However, the measure of complexity/simplicity in technology is not so clear that the number of parts is used as its rough measure [12].

In the field of software, on the other hand, the concepts of Complexity are much developed. For example, the complexity of an algorithm is evaluated not with the number of lines of code but with the number of actual computations subject to branching and looping depending on the data to process.

The concept of Easiness to Understand has also been an important issue in many areas of computer science, just like here in the discussion of Structured Programming. A wide range of methods for making software systems easier to understand have been developed and implemented successfully, so that people around the world today can use personal computers and Internet without so much trouble. Even so, the concept of Easiness to Understand is not fully developed in computer science; for instance, it is not clear why and how much the actual programs rewritten in the Structured Programming style are easier to understand than the programs written with Goto statements (See Figs. 3 and 4).

Introduction of the concept of Easiness to Understand into TRIZ should have much significance. Increase in the Easiness to Understand should be a part of some Trend of Evolution. The criteria of such concept should contribute much to evaluate the evolution

of TRIZ itself, the present author believes. This concept need to be considered and discussed more in the future.

#### **4.6 Necessity of Guiding Principles for Simplifying Complex Systems**

The main purpose of the proposal of Structured Programming was to simplify complex systems of software programs. It is important for us TRIZ promoters/users to examine what sorts of guiding principles TRIZ provides for simplifying complex systems. Darrell Mann [12] in his textbook relates this issue with the methods for reducing the number of parts in the system and lists the following 7 Inventive Principles in the numerical order:

- TRIZ Inventive Principle 2. Taking out (taking out harmful/useless components)
- TRIZ Inventive Principle 3. Local quality (modifying existing components to achieve several functions together especially in different locations)
- TRIZ Inventive Principle 5. Merging (merging identical or related objects, operations or functions)
- TRIZ Inventive Principle 6. Universality (making an object or structure perform multiple functions and eliminating others)
- TRIZ Inventive Principle 20. Continuity of useful actions (eliminating all idle or non-productive actions)
- TRIZ Inventive Principle 25. Self service (enabling an object or a system to perform functions or organize itself)
- TRIZ Inventive Principle 40. Composite Materials (making a composite structure by joining multiple materials/structures/functions)

TRIZ suggests a few other methods such as:

- Trimming: Try to eliminate a component first and to perform any of its useful function by some other components existing inside or around the system.
- Trend of Smart Materials: Use adaptive or functional materials.
- Trend of Reducing Number of Energy Conversions

In the previous sections of the present paper, the Structure Programming was viewed with Trimming, Segmentation, and Universality in TRIZ. Frankly speaking, however, the present author feels that TRIZ has not yet provided adequate guiding principles for reducing complexity in the system. Making things 'simple yet powerful/effective' should be the way of 'reducing complexity' of a system without losing its power and functionality so much. 'Unification' must be the keyword for this direction of simplicity; this point needs further consideration in the future.

#### **4.7 Necessity of Extending Universality toward Universal Standards**

As discussed in Section 3.2, reviewing Structured Programming with the TRIZ Inventive Principle 6. 'Universality' has strongly suggested the need of extending the Inventive Principle. The extension should be:

- TRIZ Inventive Principle 6. 'Universality'  
B (New). Introduce basic and standardized functional units and use them universally in a wide range of applications.

This may be recognized as the Principle of Universal Standards. Such a principle has been widely recognized and utilized in a wide range of fields and aspects. All industrial products are standardized more or less, and various social systems are also standardized.

Why has TRIZ missed this concept of Universal Standards? This reflection reminds us a blind point of TRIZ which has put so much stress on the analysis of inventions and patents. Even though standardization is a very important method and policy for the sound development of technologies in the world, introduction of standards, i.e. both setting up the standards and using the standards, is not the issue of patents. Thus TRIZ as the 'Theory of Inventive Problem Solving' has not paid attention to the approach of introducing Universal Standards.

Recognition of this fact has a significant effect on our further development of TRIZ, or methodologies for creative problem solving and for systematic innovation. Ed Sickafus [14], while developing USIT, did not put stress on producing inventions but on solving practical industrial problems creatively. This kind of approach must be important for us to promote TRIZ as widely as possible for global use for the happiness of mankind.

#### **4.8 Process of Revealing Contradictions**

Reviewing the process of the Goto-less dispute in software engineering has suggested a new viewpoint on the process of revealing contradictions. Handling contradictions in technology and in human issues makes a difference in the appropriate process of revealing contradictions.

When we try to reveal contradictions in the fields of technology, we assume that a problem solver, or a team of problem solvers having a common will, is considering the both sides of requirements/arguments under the contradiction. Thus, even though opposite conflicting requirements exist, they have arisen under a situation where a common purpose is assumed at a higher level. For instance, "A coffee cup must be hot to keep the coffee hot, but it must not be hot to drink the coffee." The opposite conflicting requirements on an aspect of a system have logically emerged, and hence the contradiction was formed logically.



On the other hand, in the case of the Goto-less dispute, different groups of people were involved in the disputes with different experiences, opinions and requirements. The dispute had two directly opposite requirements, prohibiting vs. using Goto statements, and hence was apparently under the Physical Contradiction in the TRIZ sense. However, the opposite requirements were derived only partially from logical thinking in the dispute but rather existed there as conclusions from the early stage of the dispute. Thus the Physical Contradiction was not derived as the goal of the dispute but occurred at the initial stage of the dispute.

As discussed in Section 3.4, the three inquiries of Separation Principle in TRIZ could have served to help people overcome the Contradiction smoothly. The opponents should have been asked to clarify their requirements in space, time, and conditions, and their answers should have been compared in an objective manner. If both sides of people had realized that their requirements were not clear enough under these inquiries, the dispute would have proceeded one step further. Clarifying a higher-level purpose sharable by the both sides had also been effective in solving the Contradiction. Fortunately in the field of software technology, sharing the common goal of developing good software should not be difficult to achieve in many cases. This sort of process for handling Contradictions may be typical in the application of TRIZ to non-technical, human, and social problems.

## **5. Concluding Remarks**

The present paper discussed on Structured Programming, a major issue in the early days of software engineering, with the perspectives of TRIZ. The paper is the first of the "Software Engineering and TRIZ" series, which have the following three folded aims:

- To apply TRIZ to the problems related to software development and to extend the application field of TRIZ into software development and software engineering.
- To clarify topics of software engineering with the TRIZ views.
- To feed the principles/knowledge in software engineering/computer science back into TRIZ.

The paper has discussed mainly along the second and third aims and has revealed a number of significant findings, especially including:

- The original proposal of Structured Programming with three basic control constructs was improved by overcoming a contradiction through the Goto-less dispute into the one with seven control constructs.
- Concept of Step-wise Refinement should be included in TRIZ Segmentation Principle.

- TRIZ should accept the concept of Universal Standards as a part of its Universality Principle.
- TRIZ Nesting Principle need to be stressed more for representing Hierarchy of Systems.

The approach of this series has been found fruitful, and supposed to work in many other emerging fields of TRIZ applications, such as biology, biotechnology, business and management, etc. The approach is going to be pursued further for achieving the above three aims together in the field of software development and software engineering.

## REFERENCES <sup>1)</sup>

- [1] Kevin C. Rea: 'TRIZ and Software -- 40 Principle Analogies, Parts 1 and 2', TRIZ Journal, Sept. and Nov. 2001 (E); T. Nakagawa (translated), TRIZ Home Page in Japan, Feb. 2002 (J).
- [2] Ron Fulbright: 'TRIZ and Software Fini', TRIZ Journal, Aug. 2004 (E).
- [3] Kevin C. Rea: 'TRIZ for Software: Using the Inventive Principles', TRIZ Journal, Jan. 2005 (E).
- [4] Graham Rawlinson: 'TRIZ and Software', TRIZCON2001, March 2001; TRIZ Journal, Apr. 2001 (E).
- [5] Herman Hartmann, Ad Vermeulen, and Martine van Beers: 'Application of TRIZ in Software Development', TRIZ Journal, Sept. 2004 (E).
- [6] Darrell Mann: 'TRIZ for Software?', TRIZ Journal, Oct. 2004 (E).
- [7] Darrell Mann: "TRIZ for Software Engineers", IFR Press, to appear soon (E).
- [8] Toru Nakagawa: 'Software Engineering and TRIZ (1) Structured Programming Reviewed with TRIZ', TRIZ Home Page in Japan, Aug. 2004 (J)
- [9] Toru Nakagawa: 'Software Engineering and TRIZ (1) Structured Programming Reviewed with TRIZ', Presented at Mitsubishi Research Institute Users' Study Group Meeting, Sept. 17, 2004 (J); TRIZ Home Page in Japan, Sept. 2004 (J)
- [10] Osamu Shigo: "Program Engineering -- Implementation, Design, Analysis, and Testing", Science-Sha, Oct. 2002 (J)
- [11] Edsger W. Dijkstra: 'The Goto Statement Considered Harmful', Comm. ACM, Vol. 11, No. 3, pp. 147-148, 1968.
- [12] Darrell Mann: "Hands-On Systematic Innovation", CREAX Press, Ieper, Belgium, (2002) (E); Japanese edition, SKI, Tokyo, 2004 (J)
- [13] Nam Suh: "The Principles of Design", Oxford University Press, 1990.
- [14] Ed Sickafus: 'A Rationale for Adopting SIT into a Corporate Training Program', TRIZCON99 held by Altshuller Institute, March 7-9, 1999 at Novi, Michigan.

Note: "TRIZ Journal", Editor: Ellen Domb and Michael Slocum, [www.triz-journal.com](http://www.triz-journal.com)

"TRIZ Home Page in Japan", Editor: Toru Nakagawa.

[www.osaka-gu.ac.jp/php/nakagawa/TRIZ/eTRIZ/](http://www.osaka-gu.ac.jp/php/nakagawa/TRIZ/eTRIZ/) (in English),

[www.osaka-gu.ac.jp/php/nakagawa/TRIZ/](http://www.osaka-gu.ac.jp/php/nakagawa/TRIZ/) (in Japanese).

Note: (E): written in English, and (J): written in Japanese.

### **About the author:**

**Toru NAKAGAWA:** Professor of Informatics at Osaka Gakuin University. Since he was first exposed to TRIZ in May 1997, he endeavored to introduce it into Fujitsu Labs for which he was working. After moving to the University in April 1998, he has been working for introducing TRIZ into Japanese industries and academia. In November 1998 he founded the public WWW site "TRIZ Home Page in Japan" and serves as the Editor. He is currently working to present TRIZ in a simple, unified and yet powerful way for solving real industrial problems and for teaching students. -- He graduated the University of Tokyo in chemistry in 1963, studied at its doctoral course (receiving D. Sc. degree in 1969), became Assistant in Department of Chemistry, the University of Tokyo in 1967; he did research in physical chemistry, particularly experiments and analyses in the field of high-resolution molecular spectroscopy. He joined Fujitsu Limited in 1980 as a researcher in information science at IAS-SIS and worked for quality improvement of software development. Later he served as a managing staff in IAS-SIS and then in R&D Planning and Coordination Office in Fujitsu Labs. -- E-mail: [nakagawa@utc.osaka-gu.ac.jp](mailto:nakagawa@utc.osaka-gu.ac.jp)