SOFTWARE ENGINEERING AND TRIZ (2) STEP-WISE REFINEMENT AND THE JACKSON METHOD REVIEWED WITH TRIZ

Toru Nakagawa
Osaka Gakuin University
nakagawa@utc.osaka-gu.ac.jp

Abstract
We have been examining major topics in software engineering (SE) one by one with the aims (1) to clarify them with the views of TRIZ, (2) to feed the principles/knowledge in SE and computer science back into TRIZ, and thus (3) to extend the application field of TRIZ into software development.
The concept of Step-wise Refinement is in harmony with TRIZ principles such as Segmentation and Local Quality, but is more advanced in its logical way of description. We notice a case of 'poor practice resulting in the redesigning' in the sense of SE can be regarded from TRIZ as a good and natural practice of refinement applying the Segmentation of problem in 'Another Dimension'.
The Jackson Method (or Jackson Structured Programming) proposes to build the structure of the processing system in accordance with the data structures of inputs and outputs. Though TRIZ has a similar concept to model the system in accordance with the structure of its objects/environment/super-systems, TRIZ can learn more from the software concepts and techniques. The 'Prior-reading technique' is also discussed.
In these basic areas of software development, TRIZ is in harmony with SE and can learn more from it.
Keywords: Software development, Software engineering, Design method, Data structures, Step-wise Refinement.

1. Introduction

It has been desired long to extend the applicability of TRIZ to the field of software development and software-based systems. A number of pioneers, such as Kevin Rea [1] and Darrell Mann [2], have actually applied TRIZ to software development problems, but they are often not allowed to disclose their case studies because of company's secrecy policies. Thus how to apply TRIZ to software problems has not been demonstrated well.

In this situation, the present author [3] started an approach to review basic concepts in software engineering (SE) and computer science (CS) one by one with the eyes of TRIZ. The approach has the following three aims:

- To clarify the concepts in SE/CS from the TRIZ views,
- To feed the principles and knowledge in SE/CS back into TRIZ, and hence
- To clarify how to apply TRIZ to software development and software-based problems.

For this purpose, we have chosen the textbook "Program Engineering -- Implementation, Design, Analysis, and Testing" written by Osamu Shigo [4] in Japanese as the basic material

for discussion. It is a concise textbook for an undergraduate course of CS majors and is unique in explaining from basic programming concepts to larger-scale software design, i.e. in the reverse order of explanation to the ordinary SE textbooks.

In the first paper of this series [3], the concept of Structured Programming was reviewed with TRIZ. The so-called 'Goto-less dispute', invoked by the proposal of Structured Programming, is an interesting case. It is conventional for SE/CS education to teach the dispute as the conflict between theory and practice and its result as a compromise. With the views of TRIZ, however, we can regard it as a case of Physical Contradiction and its result as a final solution having overcome the contradiction with the introduction of the rule of 'non-skewed nesting of procedures'. We have also obtained several suggestions to extend TRIZ Inventive Principles, such as 'Segmentation of problems in step-wise refinement', 'Establishing and using universal standards', and 'Nesting for constructing hierarchy of systems'.

In the present paper, we will go further ahead to review the concept of Step-wise Refinement in SE and the Jackson Method (or 'Jackson Structured Programming') [5] which is often used in the programming of business applications.

2. Step-wise Refinement Viewed with TRIZ

2.1 Step-wise refinement at the step of detailed design of program modules

Software development is a series of decisions to breakdown/clarify/construct how we realize what we want with the software. For developing larger-scale software, top-down style thinking is much recommended in SE. At the stage when the external specification of program modules are already set up, the internal procedural logic need to be designed in detail. This is the stage where the Step-wise Refinement of program modules is recommended. It is one of the most basic concepts and techniques in SE/CS, and was proposed as the foundation of Structured Programming [3].

To clarify the procedural logic inside a program module, the specification of the module are broken down (or refined) step-wise. Usually, the logics of procedural control are written in some standardized manner reflecting the Structured Programming, while the contents of the more detailed parts of procedure are written in natural language. This process of writing is repeated step-wise until the contents of the whole procedure can be converted smoothly in some programming language. To write the procedural control logically, pseudo-code is often used, where the three basic constructs (i.e. sequence, selection, and loop) in Structured Programming and some additional ones such as database inquiry are used in writing the control logic. For constructing and representing the procedures, various diagrammatic methods, generally called as structured-chart methods, have also been used.

Shigo [4] writes that the criteria of good practice of Step-wise Refinement are:

- A step of refinement can be carried out by refining each procedural instruction individually.
- The whole procedure can be understood with the descriptions up to each stage of refinement without reading any further details.
- Description of unnecessary details is avoided.
- At each stage, the procedures are described at a consistent level of detail.

2.2 Step-wise Refinement viewed in harmony with TRIZ

The general principle of designing a system in a top-down manner has been established also in various fields of hard-technologies. Thus TRIZ also has relevant Inventive Principles, such as Segmentation (Principle 1) and Local Quality (Principle 3). (Nevertheless, as discussed in [3], the Step-wise Refinement principle in SE urges TRIZ to introduce a new sub-Principle of "Segmentation of Problems".)

The concept of using pseudo-code is interesting. In TRIZ terms, it is in accordance with:

- Intermediary (Principle 24): between the natural language for general specification and the programming language for specific implementation.
- Prior Action (Principle 10) and Partial or Excessive Action (Principle 16): in the sense to write the control parts in a standardized programming-like style prior to the details of procedures.
- Homogeneity (Principle 33): to use a pseudo-code which matches with the programming language to be used later.

A key point in the Step-wise Refinement is to distinguish which aspects should be abstracted and described first. SE naturally chooses to write the aspect of the control structure first. TRIZ does not seem to give any suggestion in this point, probably because the issue depends on the specific field, i.e., SE in this case.

2.3 Criteria of Step-wise Refinement examined with TRIZ

The four criteria of good Step-wise Refinement as quoted above from Shigo [4] are important. The last three of them may be rephrased as: "At any stage of refinement, the whole procedure should be understood with the description down to the level, without referring to any further detail and without being forced to read unnecessary details."

TRIZ Law of System Completeness seems to be relevant to this concept; the Law requires an engine, a transmission, a working unit, a target (or product), and a control unit as the essential components of a working system. Since we understand systems always in the hierarchy, the Law is relevant to the hierarchical construction (or understanding) of systems. However, Step-wise Refinement concept in SE states more clearly how to describe or design a (software) system at each stage of refinement.

The first criterion quoted above from Shigo needs further discussion. This criterion recommends to construct a (software) system so as to be able to refine each part of it individually. Hence, Shigo shows an example of a 'poor' case of practice (see Fig. 1):

> "A module for data processing was first structured in the standard way of input-process-output of the whole data. Then it was found better to handle sections of data with the input-process-output procedure repeatedly. Thus the top structure of the module was changed into a loop for sections of input data. This is a case of stepping-back and re-designing due to an inadequate insight beforehand."

The initial design in this example is typical and good if the whole data can be stored in the memory at the same time, and it is efficient or even necessary if the whole data need to be handled in the memory at the same time. The final design, on the other hand, is nice if the whole data is too large to be stored in the memory at the same time while the sections of data can be processed either independently of other sections or in sequence of sections.

In the sense of TRIZ, however, the change from the initial design to the final one may be seen quite natural. It is just a good and reasonable process of refinement by applying the TRIZ Principle of Another Dimension. As shown in Fig. 1, the whole procedure was segmented into the input-process-output sequence and then the procedure was further

segmented in the dimension of data to be processed. This is the case of reasonable refinement (or segmentation) by using 'Another Dimension', i.e. data dimension in addition to the process dimension.

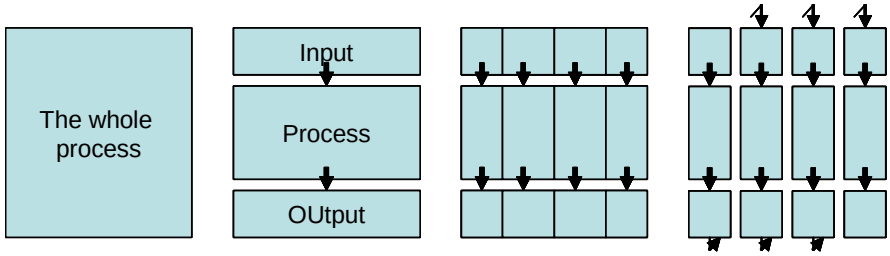

Fig. 1 Step-wise Refinement of procedure in 'Another Dimension'

This observation shows that Shigo's first criterion of 'Refining instructions individually' is just a first approximation. Software engineers should better be prepared to introduce 'Another Dimension' (or a different aspect of view) in the Step-wise Refinement; this is a suggestion from TRIZ to SE.

3. Feedback of Step-wise Refinement from SE to TRIZ

The concept and guidelines of Step-wise Refinement are worthy of being fed back from SE to TRIZ, especially in the analysis and designing of technical systems in mostly hard-technology fields. This may imply various points to discuss, as follows:

For describing a technical system at a level, the following guidelines should be observed:

- Components of the technical system should be selected at an appropriate level of detail consistently.
- With the description of the system at a detail level, the function and structure of the system should be able to be understood (at the level).
- Unnecessary details should not be described.
- Technical systems should be described in a hierarchical manner with multiple levels of detail, for the purpose of better understanding, analysis, and designing.

Concepts and guidelines similar to Step-wise Refinement certainly exist in TRIZ and in hard-technology fields. However, those in SE and CS as discussed here are clearer and more logical, and worthy of feeding back.

In this relation, we should note that the description of the TRIZ methodology itself sometimes does not observe a hierarchical structure. In particular, Inventive Principles in TRIZ do not have a hierarchical structure, as novices in TRIZ very often realize and feel confused. (We have reorganized Inventive Principles and many other TRIZ tools and constructed a system of USIT Solution Generation Operators in a hierarchical manner.)

The concept of the control structure of processing in SE should also be fed back into TRIZ to better describe the functional mechanism of the system. This point will be discussed some more in the next sections.

4. The Jackson Method of Procedure Construction Viewed with TRIZ

The Jackson Method (or Jackson Structured Programming) [5] is a method used for constructing procedures at the detailed design stage of developing software mostly for business applications. The basic principle of the method is to construct a program structure so as to reflect the data structures of input and output data.

4.1 The Jackson Method illustrated with a simple example

Shigo illustrates the method through a simple example [4]:

Suppose we are going to build a program to process the command inputs from the keyboard and making reports of the commands. Fig. 2 shows an example of the input sequence. The input sequence is consisted of a repetition of command parts, which is formed with a command name and a parameter part. The parameter part has one or multiple of parameter data among three kinds of parameters A, B, and C.

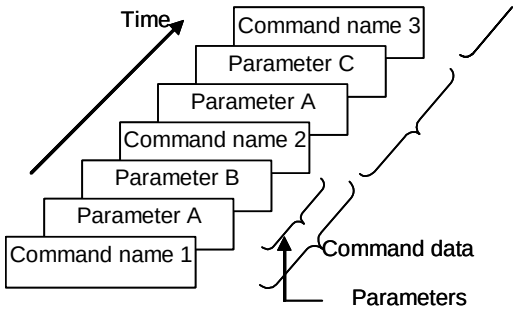

Fig. 2 Example of an input sequence of commands [4]

In the Jackson Method, we first represent the data structure of possible inputs in a tree diagram, as shown in Fig. 3(a). The tree diagram shows the data structure in a hierarchical manner from top to bottom, by using the three constructs, i.e. sequence, selection, and repetition. Each box shows a data structure at a certain level. The * mark in a box represents the repetition; multiple boxes shown at a same level represent sequencing of data; and the circle in a box represent selection among them. In a similar manner, the data structure of the output report is shown in Fig. 3(b).

Most interesting technique in the Jackson Method is to build the control structure of processing by combining the input and output data structures and making a similar tree structure as shown in Fig. 3(c). In case there is a direct matching between the input and the output, it is interpretable that we need a direct conversion from the input into the output. In case of input or output without direct correspondence, there we suppose a handling process of the input or the output. By using the same notation for sequence, selection, and repetition, the tree diagram can represent the framework of the control structure for the procedure to be designed.

Adding some minor instructions, Shigo wrote the program structure in a pseudo-code. Shigo's program structure may be represented in the tree diagram as shown in Fig. 4.

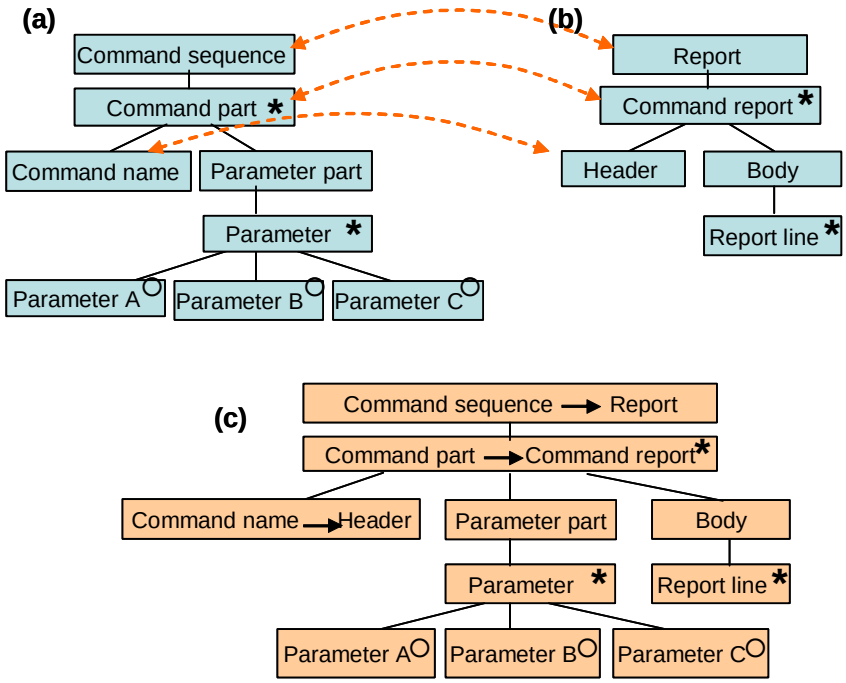

Fig. 3. Basic design process in the Jackson Method: (a) Input data structure, (b) Output data structure, and (c) Skeleton of procedural structure obtained by joining (a) and (b) [4]

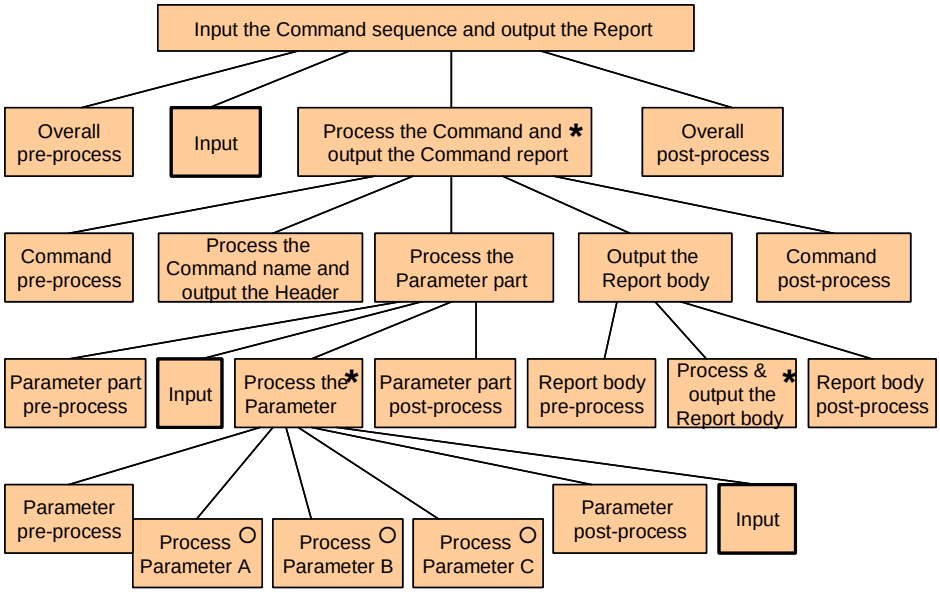

Fig. 4. Tree diagram representation of the program built with the Jackson Method

4.2 Comparison with a program written by the use of a conventional method

To illustrate the merit of the Jackson Method, Shigo [4] has shown another program for the same problem built without using the Jackson Method. Shigo's second program written in a pseudo-code is illustrated in Fig. 5 in a tree representation. It may be obvious that the structure and logic of the second program are rather confusing, in contrast to Fig. 4.

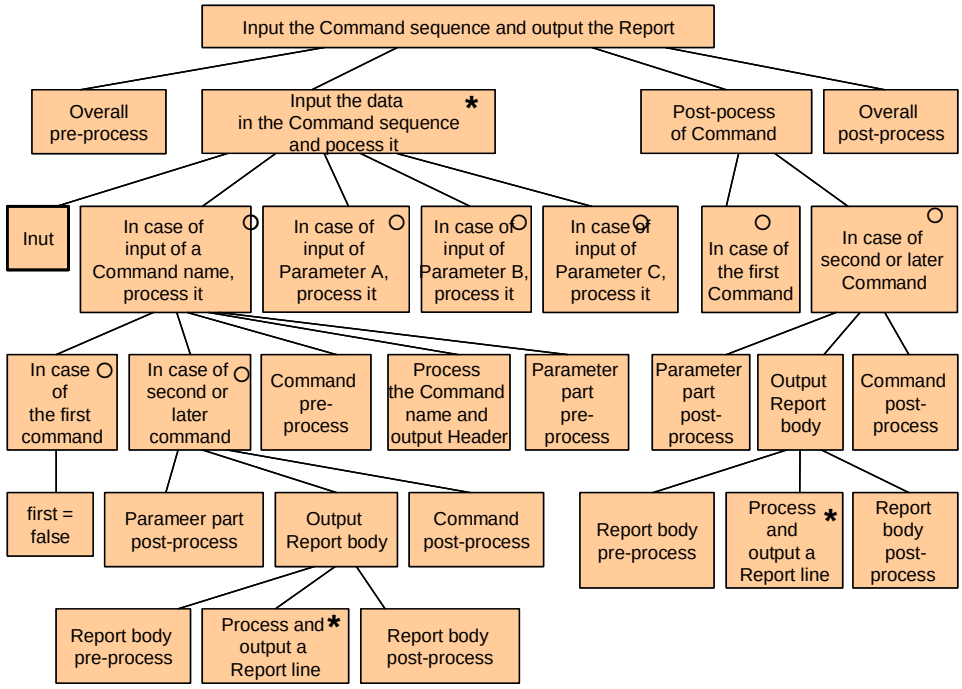

Fig. 5. Tree diagram representation of the program built with a conventional method

Readers may suspect that Shigo has shown a poor example intently. On the contrary, the example has a clear philosophy of reading the input one by one and processing it as much as possible in a quite reasonable way. The control structure of the second program (Fig. 5) may be demonstrated in the flowchart shown in Fig. 6. The framework of the program is the loop of reading an input datum and processing it right away. (The present author prefers to use a simplified flowchart here for clarity, even though many software professionals may think flowcharts are old-fashioned and unsuitable. The truth is: Flowcharts are so flexible that one may write both well-structured and poorly-structured (i.e. spaghetti-like) programs.)

For clarifying the comparison further, the control structure of the program (Fig. 4) built with the Jackson Method is shown in the flowchart in Fig. 7. It has double loops nested, where the outer loop is for processing at the command level and the inner loop for processing at the parameter level.

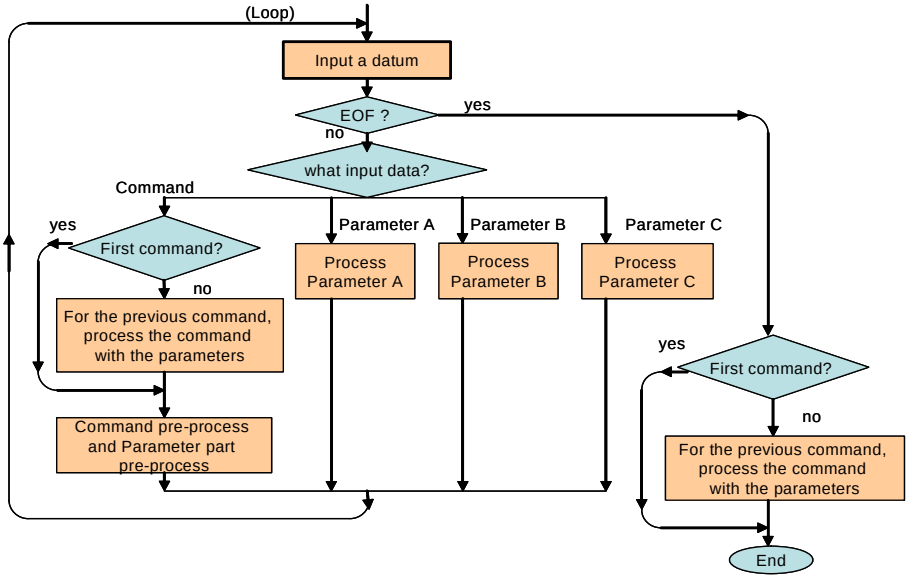

Fig. 6. Flowchart of the program in Fig. 5 (built with a conventional method)

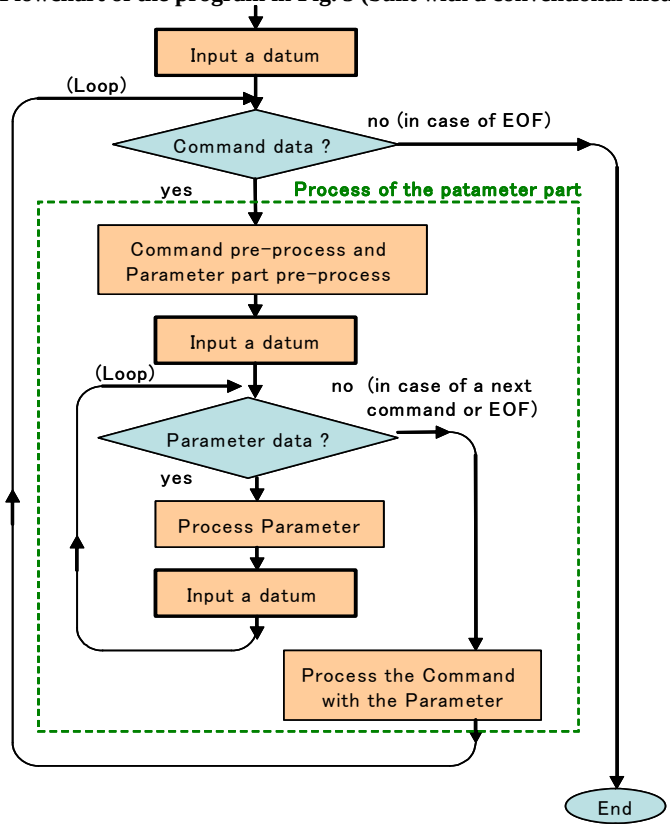

Fig. 7. Flowchart of the program in Fig. 4 (built with the Jackson Method)

As shown in this example, the program built with the Jackson Method has logically clear structure reflecting the data structures of input and output (see Fig. 4), while the one built otherwise does not reflect the data structures of input/output and resulting somewhat confusing in the logic of overall control of process (see Fig. 5).

4.3 The philosophy in the Jackson Method viewed with TRIZ

As descried so far, the principal philosophy in the Jackson Method is to build a software module (i.e. a processing system in the sense of TRIZ) so as to reflect the structure of its input and output. In the sense of TRIZ and hard-technologies, inputs are the materials and information we are going to handle with our technical system while outputs are the products and other materials and information after processing. Thus, the principal proposal in the Jackson Method is 'to build the structure of a technical system so as to reflect the structures of the materials and information which are to be handled with and produced by it'.

The concept of 'the structure of inputs and outputs' of a technical system seems to correspond to TRIZ' basic view for understanding the world in a hierarchy of systems. TRIZ always tries to analyze the technical system in the problem, to analyze its super- and sub-systems, to analyze the environment of the system. And the technical system is always considered as the interaction between the 'tool' (i.e. a working unit) and the 'product' (i.e. an target object of the interaction). In this meaning, the philosophy in the Jackson Method seems to match well with TRIZ.

The philosophy of 'building the structure of a technical system so as to reflect the structures of inputs and outputs' may also be seen in the following parts of TRIZ:

- 9-Windows method: especially to think of the super-systems,
- Asymmetry (Principle 4): to adjust the system's structure in accordance to the asymmetry in the environment,
- Flexible Shell and Membrane (Principle 30): to use flexible shell and membrane in place of solid structure, so as to make the system adaptive to the things to handle,
- Parameter Change (Principle 35): especially to change the degree of flexibility,
- Composite Materials (Principle 40): to change a homogeneous material into composite materials so as to adapt them to multiple functional requirements,
- Selective Introduction (Inventive Standards Cc7 and Dc11 [6]): to introduce a limited amount of new substance selectively to the places where really required,
- Flexible Systems (Inventive Standards Da5): to make the system more flexible and more adaptive,
- Introduction of Field (Inventive Standards Dd3): to introduce a field for which the existing substances in the system or the environment work as a media or a source.

Shigo also mentions that the structure of the program should correspond to the processing order. This point is also suggested by TRIZ in the following ways:

- Law of Rhythm Coordination: All the parts of a system should have the vibrational frequencies (or the periodicity of actions) coordinated well (or off the coordination intently).
- Merging (Principle 5): to merge or connect multiple substances, operations or functions, so as to work together in time,

- Conversion of the Field in Order (Inventive Standards Cb2 and Db4): to convert a homogeneous or non-ordered Field into the one non-homogeneous or ordered Field, which may change in time, be temporary, or be permanent.

4.4 So-called 'Prior-Reading Technique' in the Jackson Method reviewed

As discussed in Shigo [4], a key technique in the Jackson-Method programming is supposed to be the so-called 'Prior-Reading Technique'. To uderstand this technique one should examine the ordinary 'non-Jackson' way of programming as shown in Fig. 6 (or in Fig. 5), where a datum is read and then processed as much as possible before going back to the top of the loop to read the next datum. In the Jackson-Method program shown in Fig. 7 (or in Fig. 4), on the other hand, it appears that a datum is read and processed and then the next datum is read at the currently-processing position without going back to the top of the outmost loop. The multiple input instructions placed at various levels of processing (or within various loops) are the keys to make the program hierarchically structured in accordance to the logic of input/output data structures, and hence to make the program easy to understand.

The technique of inserting additional input instructions inside the inner loops is usually called 'Prior-Reading Technique'. With this naming, programmers often think that the input instruction is inserted to 'read the next datum prior to the timing absolutely necessary so that any preparation can be made in advance to processing the datum'. This contains, however, a mis-understanding that the actual sequences of processing (including the input instructions) of the two program are different. If you take an example of input sequence and trace the program behavior, you will find that the two progams perform the input and processing instructions in the same sequence (disregarding the flow-control instructions). This means that the additional input instructions are inserted at the position absolutely necessay (in the sense of performance timing) to read the next datum and in the manner to keep the hierarchical structure of the logic.

(I am not sure how much the discussion here is related to TRIZ. The fact is that Shigo's original text is not clear in the meaning of the technique, just as I had been for a long time. So I started to think about it and in a month or so I figured out the statements shown here. During this course of thinking, I am sure I used my backround of both CS and TRIZ but am not sure how much TRIZ helped me reach the conclusion. I also do not know if any software engineers have claimed to rename the 'Prior-Reading Technique' to avoid the confusion.)

5. Feedback of the Jackson Method from SE to TRIZ

Now let us convert our standpoint to view from software engineering to TRIZ.

5.1 Distinguishment between 'The Object System' and 'The Objects of the System'

The concept of 'the structure of input and output' has some corresponding parts in TRIZ, as discussed briefly in Section 4.3. However, it should be considered here again.

The phrase 'Objects of the System we are going to handle in our problem' may often cause some confusion in various fields of engineering, I suppose. It may often be understood as '(the components of) the technical system (or device) itself' in the problem and not as the 'object things that we are going to handle or process with the technical system in the problem'.

Thus there are various methods to analyze 'the Objects of the Problem' (in the sense of 'the technical system in the problem'), such as the methods of analyzing the structure, functions,

characteristics, performance, mechanisms, merits, weakness, failures, etc. of the technical system in the problem. However, when we think of methods to analyze 'the Objects which we are going to handle or process with the technical system in the problem', the situation becomes quite different. Engineers who are working for the design/manufacturing etc. of the technical system may or may not be familiar with such methods of analyzing the Objects.

For example, if we take a drilling equipment for civil engineering as the technical system in the problem, the Objects which the drilling equipment is going to handle are rocks and strata; thus the structures of the Objects concern the mineralogical structures of rocks and the layer structures of strata, etc. If we think of a milling machine for metalic products, the Objects of processing with the machine are metal materials and metalic artifacts. Thus the structures of these Objects may concern with the inner structures of metal materials and shapes and mechanical structures of the products, among others. Thus, it is evident that the fields of profession of the technical systems (e.g. a drilling equipment and a milling machine) are much different from those of the Objects (e.g., rocks and metalic artifacts). Because of these differences and far-apart separation, the responsibility for studying and improving the technical systems and that for the Objects of the technical systems are often placed on different people and different fields of profession. Thus the engineers who work for the design and development of a technical system often assume that the study of the Objects which the system is going to handle are already done by someone else at somewhere else.

In the field of software development, such a separation also occurs often. For example, software engineers who are going to develop a banking system are usually not familiar, initially at least, with the details of the bank's accounting procedures, etc. However, the uniqueness in the software field is that the Objects of the technical system (e.g. the inputs and outputs of the banking software) can be represented with the information/data and can be processed with the technical system (e.g. the banking software) in quite the same framework of methodology. Because of this unique feature, software engineers can clearly understand the concepts of 'the structure of the Objects to be handled' and 'the input and output structures of the Objects', and they can analyze/construct/reconstruct those structures in their specific problems. Since this feature is so unique, it is worthy for us to feedback such concepts and methods into TRIZ.

The concept of 'considering the structures of (input and output) Objects in designing the structures of the technical systems' seems to be already known in the hard-technology field (or in TRIZ) as a general principle, as discussed in Section 4.3. Thus a more specific recommendation is shown here as the feedback from SE to TRIZ:

- "Sub-principle of Introducing the Structures of Objects: The Objects to be processed by the technical system should be examined closely in their structures, especially from the aspects of spatial, temporal, causal, and logical structures, and the structues of the Objects should be represented in some proper way and should be introduced in designing the structures of technical systems in the problem."

This recommendation may be regarded as a sub-pinciple of one of the Laws of Completeness of Technical Systems. It should be noted that this recommendation comes not only from the Jackson Method discussed in this paper but also from various parts of SE, CS, and software development.

6. Concluding Remarks

As we have discussed so far, the concept of Step-wise Refinement in analyzing problems and in designing (software) systems is important and useful. The concept of data structures of inputs and outputs is the basis of the Jackson Method, and the structure of processing of the data is constructed in accordance with the input/output data structures. These concepts in SE are basic and general and have some corresponding parts in TRIZ and hard-technologies, in their general meaning. However, these concepts have been much developed in SE/CS in more detailed and logical ways. Thus we have found it valuable to feed these concepts in SE/CS back into TRIZ and hard-technologies.

Since these concepts in SE/CS are so basic and deep, the ways of utilizing them in TRIZ and hard-technologies need further consideration. Generaly speaking, while reviewing the principal topics in SE with TRIZ, we have found that TRIZ can learn a lot from SE/CS, probably more than TRIZ can provide to SE/CS. This point may not be a surprise becasue we are discussing in the original playgrounds of SE/CS in order to find new applicability of TRIZ in the field of software development.

In the basic field of software development discussed in the present paper, TRIZ is found in harmony with the leading principles but TRIZ has found only little chances of giving essential contributions. The discussion of the 'Cases of Refinement in Another Dimension' has contributed to make the Step-wise Refinement more flexible, and the discussion of the 'Prior-Reading Technique' has made some wide-spread confusion clearer. These findings encourage us to go further for reviewing other principal topics in SE with TRIZ and to explore the ways of applying TRIZ to software development.

References

[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] Darrell Mann: 'TRIZ for Software?', TRIZ Journal, Oct. 2004 (E).

[3] Toru Nakagawa: 'Software Engineering and TRIZ (1) Structured Programming Reviewed with TRIZ', Poceedings of TRIZCON2005 held at Brighton, MI, USA, on Apr. 17-19, 2005; TRIZ Home Page in Japan, Jun. 2005 (E); Earlier manuscript in Japanese, TRIZ Home Page in Japan, Aug. 2004 (J)

[4] Osamu Shigo: "Program Engineering -- Implementation, Design, Analysis, and Testing", Science-Sha, Oct. 2002 (J)

[5] Michael A. Jackson: "Principles of Program Design", Academic Press, London (1975)

[6] Darrell Mann: "Hands-On Systematic Innovation", CREAX Press, Ieper, Belgium, (2002) (E); Japanese edition, SKI, Tokyo, 2004 (J)

Note: "TRIZ Journal", Editor: Ellen Domb and Michael Slocum, www.triz-journal.com
"TRIZ Home Page in Japan", Editor: Toru Nakagawa.
www.osaka-gu.ac.jp/php/nakagawa/TRIZ/eTRIZ/ (in English),
www.osaka-gu.ac.jp/php/nakagawa/TRIZ/ (in Japanese).

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