三菱総合研究所
知識創造研究会 創造手法分科会 発表

ソフトウェア工学とTRIZ (1)

構造化プログラミング をTRIZの観点から見直す

2004年 9月17日
三菱総研 2階大会議室 (東京・大手町)

中川 徹
大阪学院大学 情報学部

初出: 『TRIZホームページ』 2004. 8.26掲載 研究ノート

はじめに 私とソフトウェア工学とTRIZ

1997年 5月 TRIZを知り、導入普及活動。
富士通研究所、情報科学/ソフトウェア工学を研究。
ハード分野中心に導入。
ソフト関係者:「TRIZはソフト分野に使えないのでないか?」
答え: 「TRIZをハード分野でマスターしてから。」

1998年4月 大阪学院大学 (2000年4月 情報学部新設)
情報科学序説、ソフトウェア工学、科学情報方法論 などを講義

TRIZを、考える方法、情報を整理する考え方、
技術一般での問題解決法、
技術一般への情報科学的な寄与のしかた として教える。

2004年 7月 ソフトウェア分野にTRIZを適用し始める。
基盤ができてきた。
大学のゼミで議論・講義を開始した。

『ソフトウェア工学とTRIZ』 3つのねらい

**(1) TRIZをソフトウェア関連分野に適用した事例を作り、
TRIZの適用分野をソフトウェア分野に拡張する。**

ソフトウェア開発の指針として、TRIZを適用する方法を実証していく。

**(2) ソフトウェア工学にTRIZの観点を導入し、
ソフトウェア工学をより明確にする。**

**(3) ソフトウェ ア工学/情報 科学の知見を、
TRIZにフィードバックする。**

『ソフトウェア工学とTRIZ』の アプローチ

(a) ソフトウェア工学の基本的なトピックスについて、
TRIZの観点を入れて一つ一つ考察する。

(b) 教科書: 『プログラム工学 -- 実装, 設計, 分析, テスト』,
紫合 治 著, サイエンス社 (2002年10月)

(c) 読者対象: プログラミングと TRIZの 基本知識を持つ人。
ただし、両分野とも専門家レベルである必要はない。
ソフトウェアの技術者か、TRIZに詳しい人ならもっとよい。

(d) プログラミング言語での記述を避けて、
本質を図式レベルで議論する。

(プログラミングの記法や書ける/書けないの議論をしたくない。)

「ソフトウェア工学とTRIZ」(1)
構造化プログラミングをTRIZの観点から見直す

目次:

1. はじめに、構造化プログラミングとは
(ソフトウェア工学の立場での説明)

[紫合] 1.4　プログラム工学の歴史

1960年代後半:　プログラム作成技術の研究が始まる。
Dijkstraが**構造化プログラミング**を提唱。
「**ソフトウェア工学** (Software Engineering)」が提唱された。

1970年代:　データの扱いが研究され、
抽象データ型やオブジェクト指向 の概念が発表された。

1980年代:　オブジェクト指向向きのプログラミング言語･環境が出てきた。

1990年代:　インターネット利用が急速に拡大した。

デザインパターンなどノウハウのパターン化が行われた。

本書『プログラム工学』で扱うのは、70年代から90年代までに提唱され、
現在も利用されている (生きている) プログラミング関連技術。

[紫合] 2.1　構造化プログラミ ング

プログラムは命令の列でできている。
この命令はつぎの3つに分類される。
(1) 定義･宣言:　変数や手続きの宣言など。
(2) データ処理:　演算や代入、ファイル入出力など。
(3) 制御命令:　分岐や繰り返し、手続き呼び出し、飛び越し命令など。
プログラムを理解するには、この命令の列を読んでいかなければならない。

プログラムが分かりやすいためには、
原則として上から下へ順に読めるよう になっているべきである。
-- 構造化プログラミングの主張。

当時は、アセンブラやFortran などが多く使われており、goto文が多用された。
goto 文はプログラムを上から下へ順に読んでいくことを妨げ、
思考の連続を中断し、プログラムの理解を妨げる原因になるので、

「goto文を使わずに、分岐や繰り返しだ けでプログラムを作成すべきである」
-- 構造化プログラミングの提唱。

プログラムの制御構造として、つぎの**3つを基本制御構造**という。

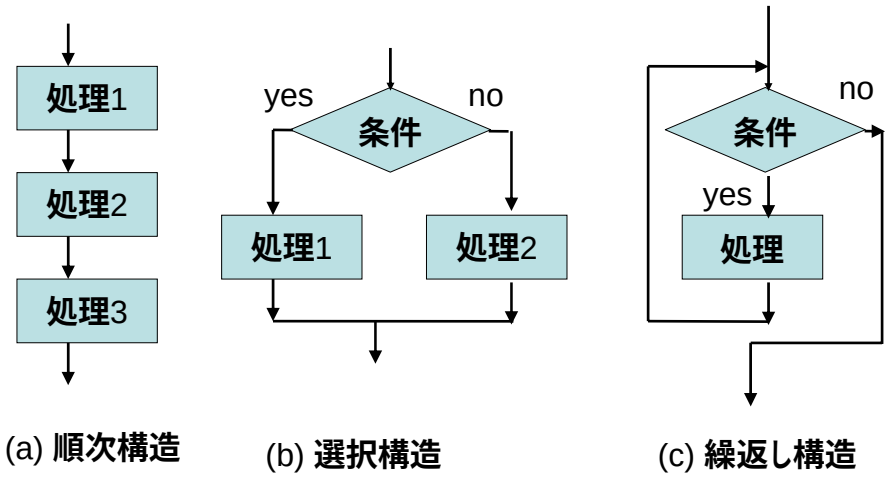
(a) **順次構造**: 上から順に処 理を実行する、

(b) **選択構造**: 条件によって処理を選択して実行する、

(c) **繰返し構造**: 条件が成立する限り処理を繰り返す

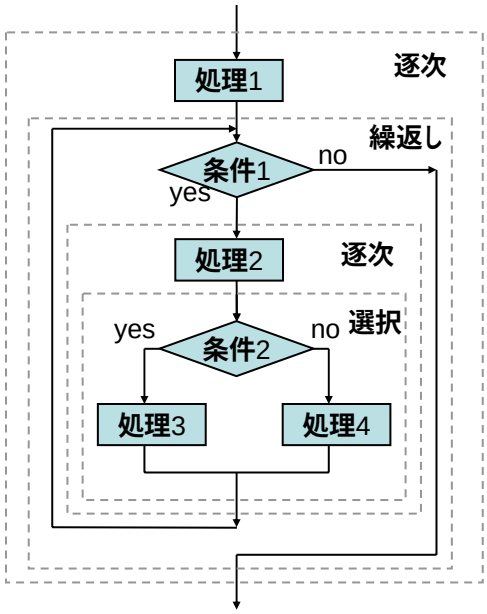
これに、 多分岐選択、
条件判断を後でする繰返し、
繰返しの中断、
例外処理の記述 など、
いくつかの拡張を加えることもある。

構造化プログラミングでは、
この基本制御構造の組合せのみでプログラムを作る。



(a) 順次構造　(b) 選択構造　(c) 繰返し構造

(紫合) 図2.2 3基本制御構造



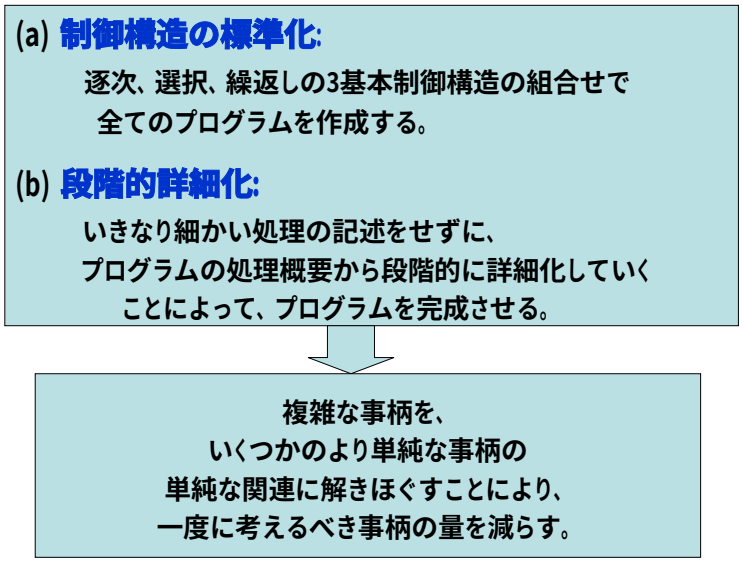
(紫合) 図2.3 **基本制御構造の組合せ (の例)**

構造化プログラミングが提唱された頃は、
普通、 goto 文とラベルを多用してプログラムを書いていたので、
この提唱は大きな論議を呼んだ。

今日のソフトウェア工学、オブジェクト指向設計などの、
いわばはしりと してのエポックであったといえる。

構造化プログラミングでは、
段階的詳細化の原理も合わせて提唱された。
「**複雑な物事を一度に詳細化するのではなく、**
まずは物事の概要をとらえて、徐々に詳細化していく」

これは、プログラムだけでなく、
一般の問題解決にも利用される原理である。



[紫合] 図2.1 構造化プログラミングの原理

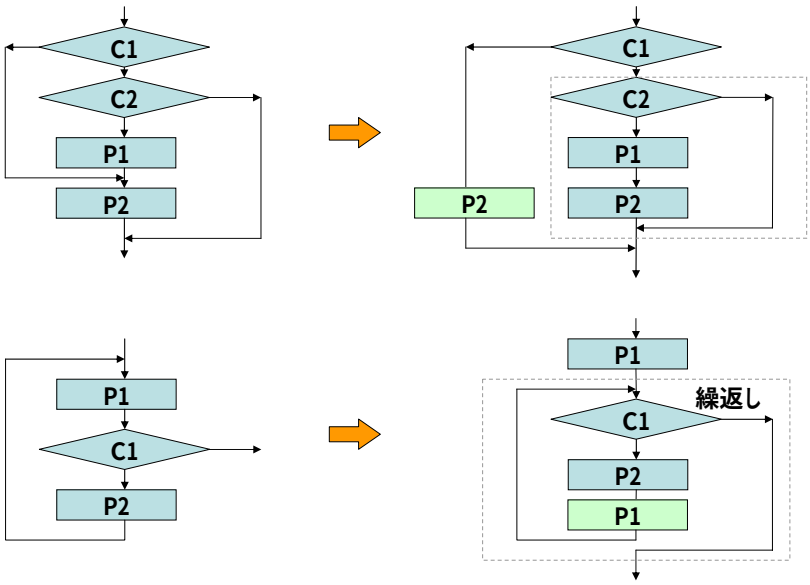
[紫合] 2.2 構造化原理

「どんなプログラムでも基本制御構造の組合せで作れること」を
構造化原理という。

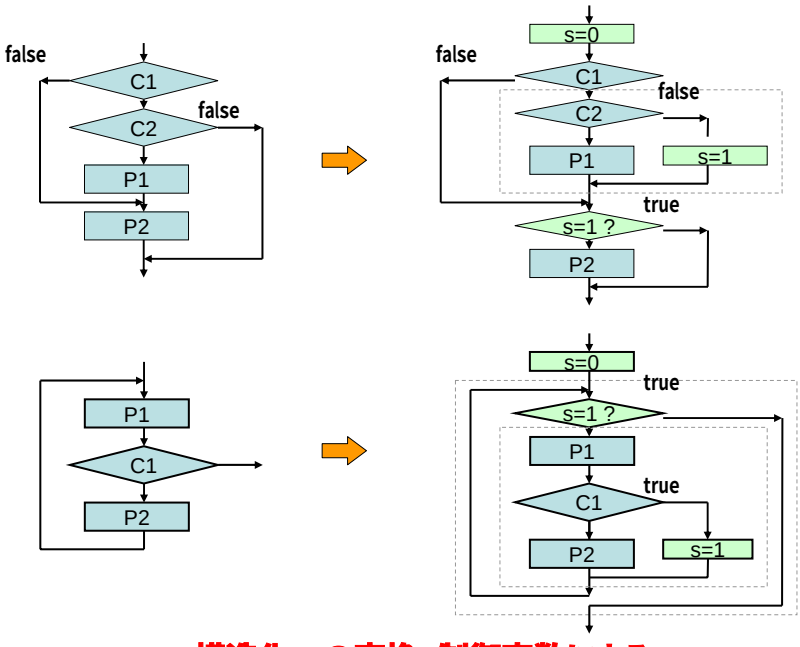
構造化されていないプログラムを構造化する主な手段としては、
処理をコピーする方法と、
新たに制御変数を導入する方法がある。

コピーの方法がよい場合:
コピーする必要のある処理が小さい場合や、
コピーしてもあまりプログラムが大きくならない場合

制御変数の導入がよい場合:
上記以外の場合。



(業合) 図2.6 構造化への変換 (処理のコピーによる)

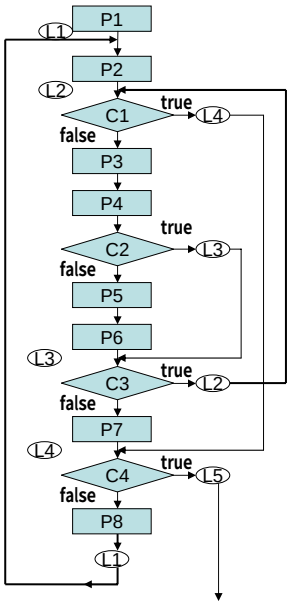


(業合) 図2.7 構造化への変換 (制御変数による)

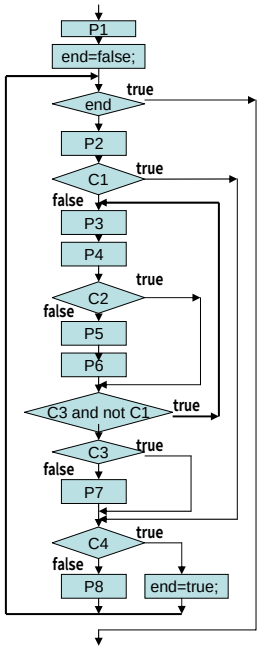
[紫合]

例えば、図2.4のような**goto文を多用した、**
制御の流れが非常に分 かりにくいプログラム
(スパゲティプログラムという) が与えられたとする。

これを構造化プログラミングで定められた
3基本制御構造の組合せだけで表現した
等価なプログラムが図2.5である。



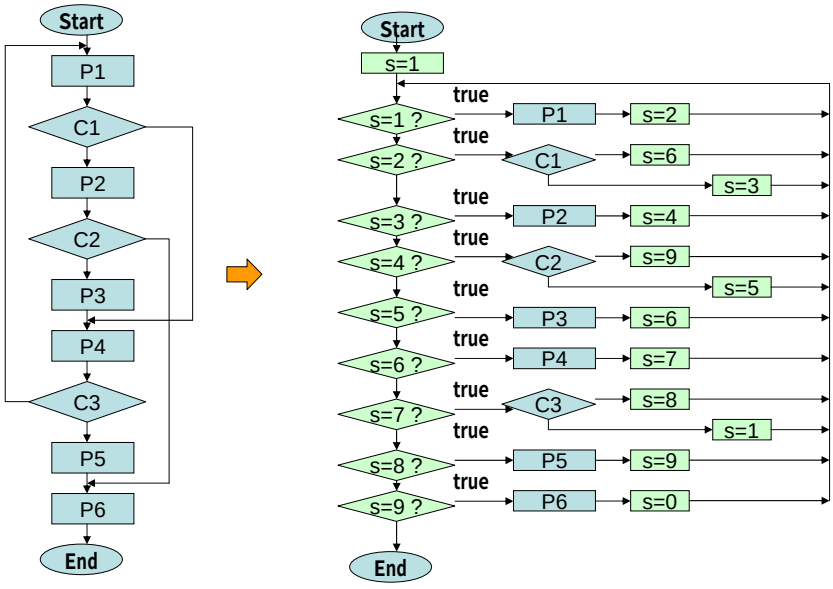
(紫合) 図2.4 Goto文を使った
スパゲティプログラム (の例)



繰返し (前判別)

繰返し (後判別)

(紫合) 図2.5 等価な構造化プログラム(の例)



(集合) 図2.8 **構造化原理: 任意のプログラムは選択の繰返しにできる**

ソフトウェア工学とTRIZ (1)
構造化プログラミングをTRIZの観点から見直す

2. 構造化プログラミングの補足

(ソフトウェア工学の中での説明と議論)

2.1 構造化プログラミングに至るまで

2.2 goto論争

2.3 構造化プログラミングの基本制御構造の意図

2.4 構造化プログラミングにおける追加制御構造の意義

2.5 プログラミング言語における「構文」の役割

2.6 構造化プログラミングが何を禁止 (制限) したのか?

2. 構造化プログラミングの補足
(ソフトウェア工学の中での説明と議論)

以上のようなソフトウェ ア工学の立場からの説明の一部を補足して、
今後の考察のための準備をしよう。

ソフトウェア工学に専門でない人たちのための準備、

専門家の間では常識になってしまっていて
見落とされることがあるものを　予め補う。

2.1 構造化プログラミングに至るまで

goto 文を「多用」したプログラムが作られた時代があったが、
コンピュータの基本原理「フォンノイマンのプログラム内蔵方式」に由来する。

コンピュータにさせたい処理の手順を、命令の列で記憶させておく。
- 命令を順番に読み出して実行する。
- すぐ次でなく、別の所の命令を実行する機能が必要であり、
「ジャンプ命令 (goto 命令)」と、行き先の「ラベル」を使う。
- 条件に応じて ジャンプする　--> 判別命令
- 前に戻る:　　典型的には「繰り返して実行する」場合。
「繰返し」の構造がいろいろ作られていく。

徐々に「プログラミング言語」が高度化していく。
それでも、「goto文と行き先ラベル」の方法が残されていた。

ソフトウェアが大規模・複雑になって、どうしようもなくなってきて、
「構造化プログラミング」が提唱された。

2.2 goto 論争

Dijkstraが構造化プログラミングを提唱したとき、
「**goto 文は有害である**」というキャッチフレーズを使い、
従来のプログラマたちに衝撃を与え、大きな論争になった。

提唱側: 「goto文があると、プログラムが分かりにくい。
だから、goto文を無くそう。
3基本制御構造だけで、任意のプログラムを作ることが可能。」

反論側: 「3基本構造が便利でも、goto 文を無くすと不便でしかたがない。
goto文なしでは、分かりやすく書けない。」

現在の段階で振り返って みると、
提唱側が、「goto文を無くすと、プログラムが便利に書けて、
分かりやすくなる」ことを保証する必要があった。

この論争は、やがて、構 造化プログラミングに、
実際的な機能を追加容認することで決着がついていく。

(d) 多分岐選択
(e) 条件判断を後でする繰返し
(f) 繰返しの中断
(g) 例外処理の記述

これらを追加することによって、
goto文を無くしても、
便利に書けて分かりやすいことが、認められて いった。

2.3 構造化プログラミングの基本制御構造の意図

構造化プログラミングの 3種の基本制御構造

(a) 逐次構造:　　まったく素直に必要･必須のもの。
(b) 判別構造:　　前向きのgoto文を表現する。(分岐構造を作る。)
(c) 繰返し構造:　戻る向きのgoto文を表現する。(繰返しの構造を作る。)

構造化プログラミングの 本質:

ひとまとまりの (複雑な) 処理を、
全体として、入り口一つ、出口一つのブラックボックスとして表現する。

情報科学では「**アルゴリズム**」と呼ばれている。
「処理を行う権利が、決められた一つの入り口から入り、
何らかの処理をして、必ず決められた出口から出てこなければならない。」
一般的な技術用語で「機能」、「システム」といってもよい。

この内部を (もう一段だけ詳しく) 記述する方法として、
3種の基本制御構造「だけ」を許した。
==>「入れ子」による階層的な記述。 ==> 段階的詳細化

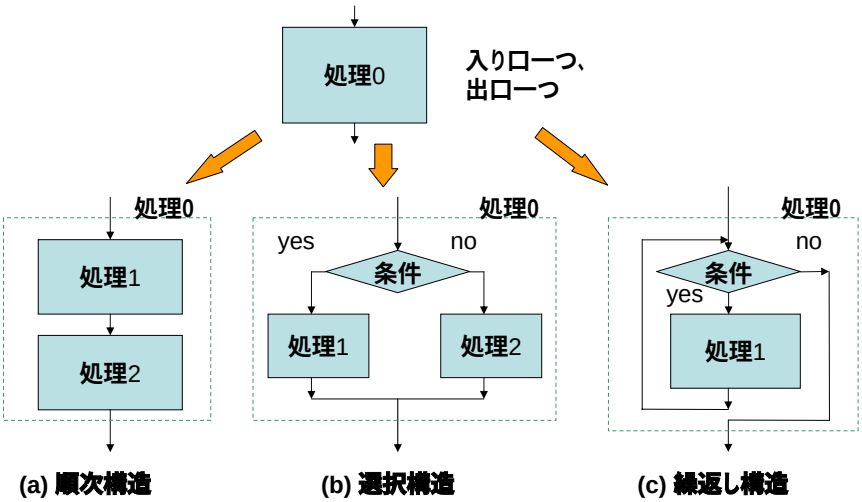
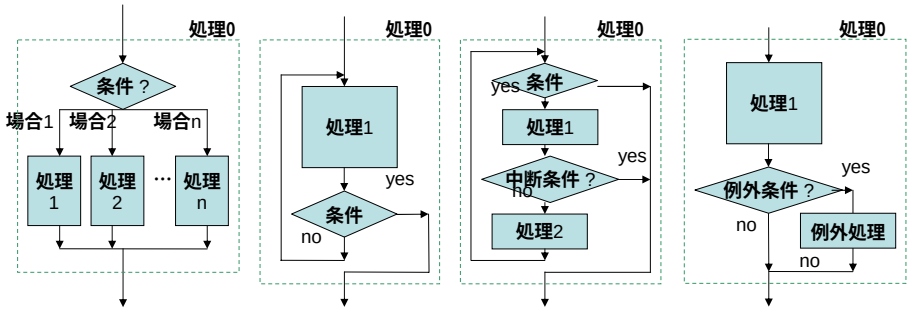


図1 処理と3基本制御構造による (1段階の) 詳細化
処理はいつも、「一つの入り 口から入り、何らかの処理をして、一つの出口から出る。」
内部の処理は「空」でもよい。

2.4 構造化プログラミングにおける追加制御構造の意義

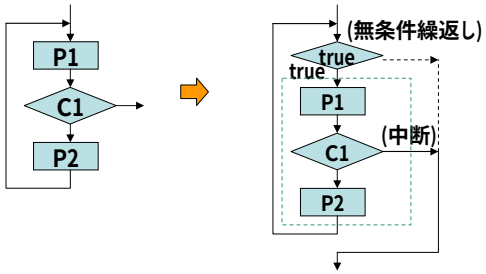
goto論争が延々と行われた後、つぎの機能が追加導入されて、ほぼ決着した。



(d) 多分岐選択構造　(e) 後判断式の繰返し構造　(f) 繰返しの中断　(g) 例外処理

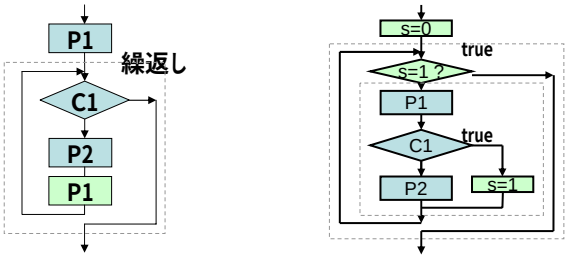
図2 追加の制御構造

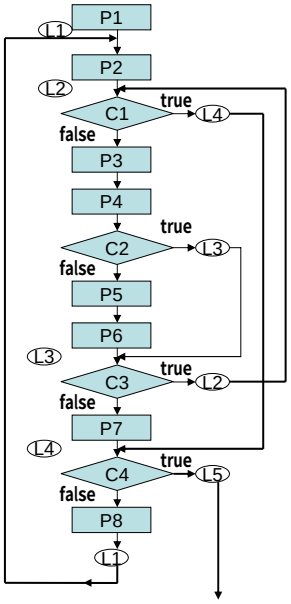
拡張構造化プログラミングを使うと、分かりやすく書ける。



拡張構造化プログラミングによる書き換え例 (繰り返しの中断)

比較: 3基本制御構造による書き換え例 (紫合)





(集合) 図2.4 Goto文を使った スパゲティプログラム (の例)

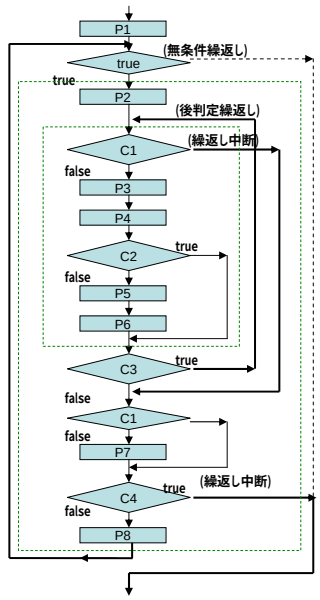
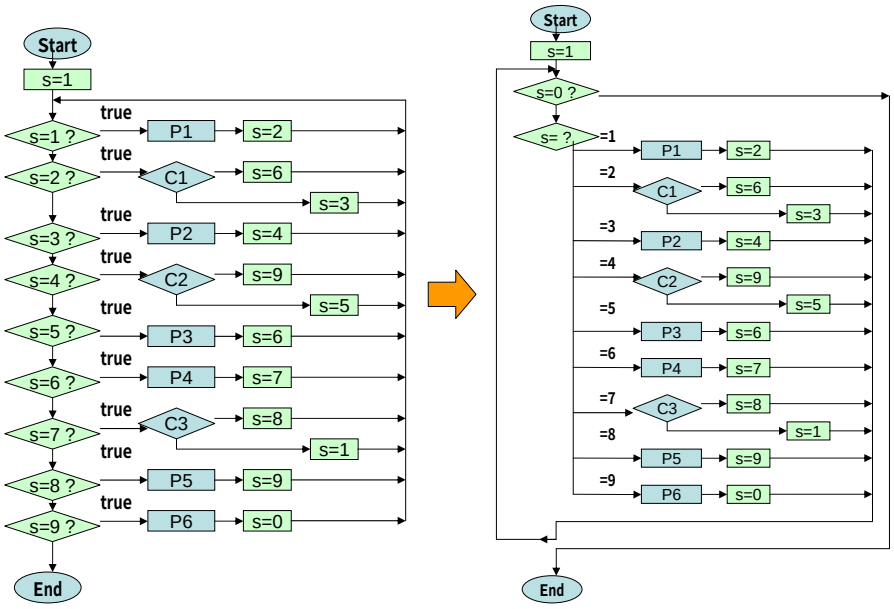


図2b 拡張構造化プログラミング を用いて書き換えた例



拡張構造化プログラミングによる書き換え例 (多分岐選択)

2.5 プログラミング言語における「構文」の役割

図式表現 (フローチャート表現) では必ずしも十分に表現されていず、
プログラミング言語のレベルで考えるべきことがある。

「分かりやすい文章と同様に、プログラムも、
原則として上から下へ順に読めるよ うになっているべきである」

このため、**(拡張) 構造化プログラミングで導入した制御構造は、
すべて「構文」で明示的に表現する。**

逐次構造: P1; P2; 処理P1 のつぎに 処理P2

選択構造: if (C) P1; else P2; 条件C が真なら処理P1、偽なら処理P2

繰返し構造: while (C) P; 条件C を満たしている間中 処理P を繰り返し

言語の「文法」として強制力を持つ。違反する書き方は実行前に排除される。

「**改行と字下げ**」で処理記述の階層を表現することが、
通常「**よい慣習**」として行われる。

2.6 構造化プログラミングが何を禁止 (制限) したのか?

これを考えると、構造化プログラミングの本質を浮き彫りにできる。

**構造化プログラミングが許容しないのは、
「入れ子構造のねじれ」現象である。**

一つの処理構造に途中から割り込んで入ることが許されない。

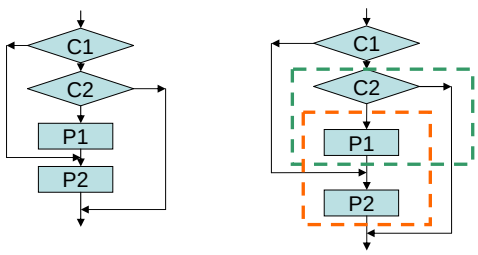


図3. 構造化プログラミングに違反する「入れ子構造のねじれ」の例

3. TRIZから見た構造化プログラミング

3. TRIZから見た構造化プログラミング

「構造化プログラミング」の考えかたやその歴史をTRIZの観点から見よ う。

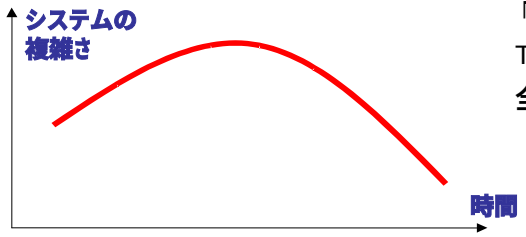
TRIZの
「40の発明原理」、
「76の発明標準解」、
「進化のトレンド」、

また、「システムの考え方」、
「矛盾」、
その他の思想的な要素　　なども考える。

3.1 複雑性の克服

「構造化プログラミング」が目指したのは、
膨大なソフトウェアの「複雑さ」を克服して、「分かりやすく」すること。

「複雑化」の進行と、それを克服して「単純にする」ことによる 一層の発展は、
TRIZにおける重要な認識の一つ。



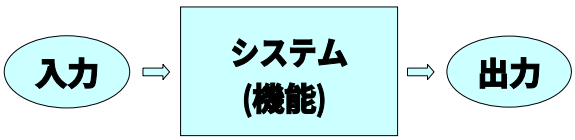
「構造化プログラミング」は
TRIZの観点からは、
全く自然な発展である。

(Mann) 図13.6 **システムの進化において、
複雑さは増大してから後に減少する**

3.2 「システム」の階層性と「入れ子構 造」

TRIZにおいて、「システム」の概念はその理論の根底にある。
「システム」: いくつかの構成 要素を持ち、それらが互いに関係をもって、
一つの機能を実現するもの

「システム」をブラックボッ クスとして見る見方:



システムには、その構成要素として「下位システム」があり、
また当システムを構成要素とする「上 位システム」があって、
「システムの階層」をなしていると考える。

これらの「システム」概念は、TRIZだけでなく、広く認識されている。

TRIZの「発明原理 7. 入れ子の原理」:

A. 一つの物体あるいはシステムを別のものの内部に入れる。
B. 複数の物体あるいはシステムを他のものの内部に置く。
C. 一つの物体あるいはシステムが、
別のものに開いた適当な穴を通り抜けられるようにする。

「構造化プログラミング」は、このサブ原理A, Bにそのまま当てはまる。
「**一つ (あるいは複数) のシステムを、別のシステムの内部に入れる**」

特に「完全に内部に」入れ、
すっきりした「システムの階層性」を実現しようとする。

注: 後述のように、この発明原理をさらに拡張するとよい。

3.3 「goto文はあるべきか、ないべき か?」 TRIZの矛盾の観点から

goto論争は、TRIZでいう典型的な「矛盾」をめぐる問題である。

この論争は、goto文がふつうに多く使われていた段階で、
「goto文はなしにすべ きである」という主張が提示されたために、
「goto文はあるべきか? ないべきか?」という極端な対立 (矛盾) になった。

TRIZでいう「物理的矛盾」の形式。
「物理的矛盾」＝「技術システムの一つの面に対して、
正逆の相反する要求が同時にある状況」

だが、goto論争の主張には (論争の常として)、両者の論理にいくらか飛躍がある。
この論理の飛躍を明確にして、より客観的な議論にすると、
矛盾の解決につながる。

TRIZで「物理的矛盾」を解決するには、要求を再吟味する。
つぎの **3対の質問をする**。

(a) **空間:** 「goto文がある」ことを望むのは、どこか?
「goto文 がない」ことを望むのは、どこか?

(b) **時間:** 「goto文がある」ことを望むのは、いつか?
「goto文 がない」ことを望むのは、いつか?

(c) **場合:** 「goto文がある」ことを望むのは、どんな場合か?
「goto文 がない」ことを望むのは、どんな場合か?

TRIZでは、これらの答えに違いを見つけて、それを利用する。

TRIZの「分離原理」による「物理的矛盾の克服」法:

「物理的矛盾にある両者の要求について、
空間/時間/場合のどれかに関して両者に違いが見出されたら、
その違うそれぞれの状況で各要求を完全に満たす解決策を作り、
その上で両解決策を統合して用いる方法を見出す。」

goto文論争における**両者の当初の主張**は:

(a) 「goto文がある」ことを望むのは、プログラム中のどこでも。
「goto文 がない」ことを望むのは、プログラム中のどこでも。

(b) 「goto文がある」ことを望むのは、プログラムを作るとき、いつでも。
「goto文 がない」ことを望むのは、プログラムを作るとき、いつでも。

(c) 「goto文がある」ことを望むのは、goto文を書きたい場合すべて。
「goto文 がない」ことを望むのは、どんな場合でも。

要するに、
「goto文をなくす」という要求が、goto文の「全否定」の形を取り、
従来の方式を支持する側から、全面的な反対 (反発) が起きた。
両者の主張が極めて広く、細分化されていないことが特徴。

「矛盾の克服」は、本来、**対立しているレベルよりも上位のレベルでの、**
目的/目標の共有がな ければ成り立たない。

goto文論争の場合、**上位の共通の目的/目標は**
「プログラムの書き方を、分かりやすく、間違いを起こしにくいよ うにする」

この目的にとって、「goto文が害になる」というDijkstraの主張が、
「goto文をなくせ」になった点にもっと検討が必要である。

3基本構造だけで表現したために、
制御変数などを多用してプログラムが「分かりにくくなった」のなら、
その主張を再吟味・修正すべきで ある。

goto論争の理解が深まり、**より明確な回答**を両者ができるようになった。

(c)「goto文がある」ことを望むのは、どんな場合か?
--> 後判定の繰返し構造 [を記述するため]、
繰返しからの飛び出し、例外処理の記述の場合。

「goto文がない」こ とを望むのは、どんな場合か?
--> 基本制御構造において「入れ子構造にねじれ現象が起こる」場合。

そこで、**つぎの解決策が作られた**。
「3基本制御構造の他に、つぎの構造を追加する。
多分岐判別、後判定の繰返し、繰返しからの飛び出し、例外処理の記述
これによって、goto文をなくし、「入れ子構造のねじれ現象」を排除できる。」

これが、**「goto論争」で、「矛盾を克服した」後の「構造化プログラミング」**である。

だから、**「構造化プログラミング」**を教える場合には、
「3基本制御構造は 初期の提唱であり、
追加制御構造を含めたものが 最終的な形である」 と教えるべきである。

3.4 汎用性の原理と分割の原理

プログラミングが大規模・複雑になったとき、
それをもっと分かりやすく、簡単にするための指針は何であったろうか?

これをTRIZの中から探すと、汎用性の原理と分割の原理に思い至る。

TRIZの発明原理 6. 汎用性

A. 一つの物体やシステムが複数の機能を実行できるようにし、
他のシステムの必要性をなくす。

プログラミングの方法を考えるときの「汎用性」は、
このサブ原理Aの記述とはややニュアン スが違う。

つぎのように表現しよう。
「**基本的で標準的な機能単位のものを作り、**
広い範囲に一般的に使えるようにする。」

この考え方は、ネジでも、乾電池でも見られる。技術の指針として一般的である。

複雑性を克服するためのもう一つの原理は、TRIZでは分割の原理である。

TRIZの発明原理 1. 分割

A. システムを分離した部分あるいは区分に分割する。
B. 組み立てと分解が容易なようにシステムを作る。
C. 分割の度合いを増加させる。

この原理は、ハードウェア的なシステム、人間の思考に関すること、
社会組織に関連することなど、広範に適用できる。

複雑なプログラム (処理論理) を取り扱いやすくする基本的な方法として、
「分割」の原理は広く適用されてきた。

ソフトウェアの世界では、「**分割統治**」と表現されることも多いし、
「**段階的詳細化**」と表現されることもある。

構造化プログラミングは、
処理論理を階層的に細分化して構成していく方法である。

3.5 信頼性を高めるプログラミング

「構造化プログラミング」の提唱は、ソフトウェアの開発において、
処理の速さや必要記憶容量の小ささなど「性能」を重視する観点から、
「信頼性」を重視する観点への転換を主張したことに 大きな意義がある。

TRIZの「技術システムの進化のトレンド」の中につぎのものがある。

「進化のトレンド (23) 顧客の購入の焦点
性能 --> 信頼性 --> 便 利さ --> 価格」

「性能」: その技術システムが最初に意図した機能を提供すること、
その機能を表現する主要パラメータによりよい値を出すこと。

「性能」に顧客が十分満足して来ると、
つぎに顧客の焦点が「信頼性」に移行していくのだという。

プログラミングにおける「性能重視」から「信頼性重視」への転換

構造化プログラミングを初めとした多くの運動で徐々に進行した。

初期のコンピュータは、
小さい記憶容量、遅い処理速度、単純な処理命令、
--> プログラムを作るには多大の努力を必要とした。

使用記憶容量を少なくし、処理速度を上げることに、努力が集中された。
凝った職人肌のプログラムで、goto文を多用して、縦横無尽に処理する。

広く多数のプログラマ (とその予備軍) のことを考え、
ソフトウェア業界が全体として、
性能から信頼性に重点を移していくための、
一つの大きな運動が「構造化プログラミング」だった。

3.6 「goto論争」と運動

運動として捉えたときの疑問:

初期の提示が適当だったろうか?

あまりにも極端だったのではないか?

これに関して想起されるのが、TRIZの発明原理16である。

「**発明原理16. 部分的な作用または過剰な作用**

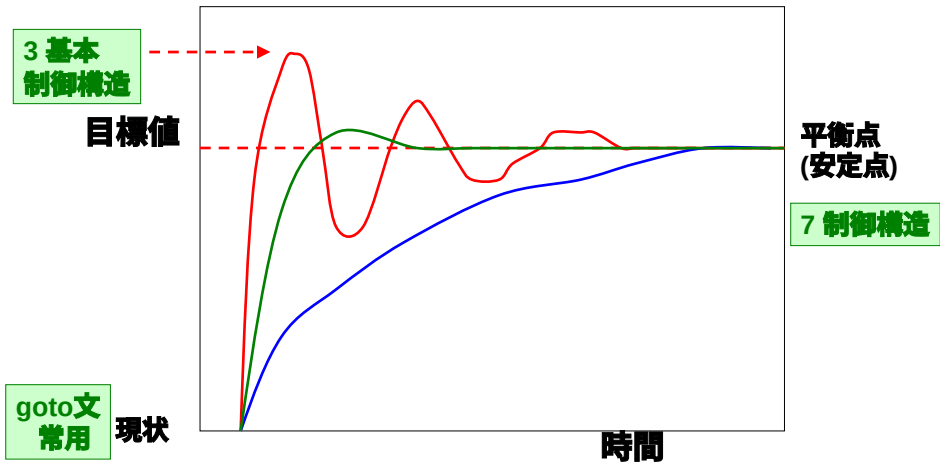
A. 正確に正しい量の作用を達成するのが困難な場合には、

「少し少ない」または「少し多い」作用を施して、

その問題を減少あるいは除去する。」

この発明原理を模式図でつぎに表す。

発明原理16 と システムの応答



「構造化プログラミング」を提唱する戦略を考えてみる。

「徐々にシステムを変えていく」 - 低摩擦戦略

「大きな衝撃を与える」-->「反動があるが急激に変わる」 - ショック戦略

3基本制御構造の提唱は、
衝撃を強くするための「目標値よりも少 し極端な提唱」、
TRIZの発明原理16にいう「過剰な作用」であったと理解できる。

これは、「**追加の制御構造を加えたプログラミングの規約が、**
妥協の産物なのではなく、
本来の最終目標 (望ましい姿) であった」 という理解である。

ただし、提唱者Dijkstraの当初/最近の理解内容には、TRIZは関与しない。

TRIZが扱う技術の進化は、個々の発明者・提唱者の意図を越えた、
はるかに大きなスケールで考えている。

以上、ソフトウェア分野の問題に、TRIZの観点から考察を加え、
「**TRIZが ソフトウェア分野に対しても本質のところで発言できる**」と分かった。

TRIZはあらゆる技術分野を見回して、
そのエッセンスを抽出して一般化した理論だから、
ある意味で当然である。

ソフトウェア分野自体が、
さまざまのハードウェア的な技術分野、
人間や社会に関わるものをモデルとしてきた
からでもある。

4. ソフトウェア工学/情報科学が TRIZにもたらすもの

4.1 ソフトウェアの「処理構造」や「アルゴリズム」の概念の TRIZへの取り込み

TRIZでは「システム」や「機能」の概念を非常に重視してきており、
ソフトウェアにおける「処理構造」や「アルゴリズム」にも同様に適用できる。

ソフトウェア分野での概念が、他の技術的分野と異なることは、
「システム」や「機能」を定義・作成する場合の柔軟性と豊富な構成力にある。

いろいろな「処理機能」を簡単に、かなりの程度思いのままに、作成でき、
その内部構造を随分いろいろな形式で作成できる。
これが新しい考え方や新しい方法に導く可能性がある。

ソフトウェアという対象が論理性を持った対象であり、
「情報科学/コンピュータサイエンス」が構築されてきているから、
ソフトウェア分野の原理や認識をTRIZに取り込むのは、意義深い。

4.2 ソフトウェアの「段階的詳細化」の概念の「分割原理」への導入

ソフトウェア分野、情報科学の分野においては、
複雑な問題 (顧客の要求など) を解明して、
具体的なソフトウェアとして実現していくために、
「段階的詳細化」が中心的な指導原理になっている。

もやもやした問題を、部分部分に分けるこ とによって、
より扱いやすい小さな問題の集まりにして行く。

このときに各部分への分割のしかたが問題になり、
それらの各部分を適切に定義することと、
それらの部分の間の関係を明確にすることが課題になる。

TRIZの「発明原理1. 分割の原理」は、
この段階的詳細化の原理を含んでいるようにも見える。

しかし、もっと踏み込んで、下記のサブ原 理D.を追加するとさら明確になり、
応用性が広いものになるだろう。

「発明原理 1. 分割
A. システムを分離した部分あるいは区分に分割する。
B. 組み立てと分解が容易なようにシステムを作る。
C. 分割の度合いを増加させる。

(追加:) D. 問題をいくつかの部分に分け、各部分とその関係として考える。」

**これは、技術的問題一般、ソフトウェア分野、人間や社会分野一般という、
すべての分野の問題解決にとって、非常に基本的な指導原理を表現している。**

4.3 「入れ子の原理」を「システムの階層性」を表現する原理として捉える

TRIZの「発明原理7. 入れ子の原理」は、
具体的なものを扱う世界では必ずしも大きな位置を占めていない。

「マトリョーシカ (入れ子人形)」は非常に印象的ではあるが、
もっともっと多くの例を含んでいるという印象は少ない。

ソフトウェア分野からは、下記のサブ原理の追加を示唆する。

「**発明原理 7. 入れ子の原理**
A. 一つの物体あるいはシステムを別のものの内部に入れる。
B. 複数の物体あるいはシステムを他のものの内部に置く。
C. 一つの物体あるいはシステムが、別のものに開いた適当な穴を
通り抜けられるようにする。

(追加:) D. システムを入れ子構造の階層的な連鎖の形で実現する。」

4.4 「分かりやすさ」の概念

TRIZでは、システムの「複雑さ」の概念は重要なものであった。
しかし、**「複雑さ」の尺度は必ずしも明確ではない。**

一方、ソフトウェア分野や情報科学の分野では、
「複雑さ」の概念と「分かりやすさ」の概念がともに重視されてきた。

例えば「計算の複雑性」については、
さまざまなアルゴリズムに緻密で定量的な議論がある。

また、「分かりやすさ」の概念は、ソフトウェア分野の中心的な論点である。
「分かりやすくする」さまざまな方法が具体的に提出され、実施されてきた。

しかし、**「分かりやすさ」の概念が情報科学でも十分明確になっていない。**

例えば、「構造化プログラミング」を使って書き換えた紫合の図2.5が、
もとの図2.4よりも「分かりやすい」ことを実証すること はできていない。

「分かりやすさ」の概念をTRIZに導入することは大きな意義を持つだろう。

「分かりやす さの向上」は、
TRIZの技術システムの進化のトレンドの一つとして考えるに値する。

「使いやすさの向上」のトレンドを立て、
その一つの側面として「分かりやすさの向上」がある。

-- ただし、「分かりやすさ」の問題はずっと奥が深いと考えられるので、
本稿ではここまでにしておき、今後、折にふれて考察を加えたい。

4.5 「入れ子構造のねじれ現象」の問題

この問題はまだ検討を要する問題である。

技術システムにおいて、その「下位システム」をどのように区分するか、
(あるいは区分して作る) かには、いろいろな自由度がある。

例えば、システム内を機能に関して区分して作ろうとする場合に、
サブ機能Aに属するものとサブ機能Bに属するものとが、
部品という目で見ると一部に重なりあっていることがあるだろうし、
空間的に見たときにも重なり合うことがあるだろう。

それを「重ならないように作る方がよ い」というべきだろうか?

この**構造化プログラミング**では、
処理の論理のブロック (すなわち機能のブロック) が
「入れ子のねじれ構造にならない」ことを良しとしている。
機能で分けた下位システムが互いに重ならないことを要請している。
それが「分かりやすい」からであるという。

ハードウェアの技術の領域でこれに対応する指針としては、
Nam Suhの「公理的設計」がある。

公理的設計のガイドライン
(1) 良い設計は、異なる機能的要求の間に独立性を実現する。
(2) 良い設計は、機能的要求を最小限の複雑性で実現する。

Mannはこれら には重要な例外があるけれども、
基本的には有用なガイドラインであると評価している。

上記の「構造化プログラミング」における提唱は、
この公理的設計のガイドライン (1) と良く対応する。

「構造化プログラミング」の提唱は、「公理的設計」の提唱よりもずっと早い。

ソフトウェア/情報科学の分野が、技術一般に寄与する一つのしかたが、
このような抽象化したレベルでのシステムに関する考察である。

システムでの「入れ子構造のねじれ現象」の問題はさらに考える必要がある。

4.6 システムの「複雑さ」を克服する具体的な方法

TRIZでは、システムの複雑さを克服していく重要さをよく認識している。

その複雑さを減少させるために、どのような方法を提示しているだろうか?

Darrell MannのTRIZ教科書では、
システム内の部品数の減少に関連する発明原理として、以下の7つを挙げる。

発明原理 2. **分離** (システム内の複数機能のうち、有害/不必要のものを分離する)
発明原理 3. **局所的性質** (既存の一つの構成要素を修正して、複数の機能を実現する)
発明原理 5. **併合** (複数の物体、機能などを、一つに併合する)
発明原理 6. **汎用性** (一つの物体またはシステムに複数の機能を実行させる)
発明原理 20. **有用作用の継続** (無駄な/非生産的な動作あるいは仕事を除去する)
発明原理 25. **セルフサービス** (物体/システムがそれ自体で機能を実行する)
発明原理 40. **複合材料** (複数の構造/機能を組合せ、一体の合成構造にする)

この他にTRIZの方法として考えられるものは:

トリミング: システムのある構成要素をまず無くしたと仮定し、それが持っていた機能を別の (または新しい) 構成要素に担わせて簡単化する。

適応型材料 (賢い材料) のトレンド:

エネルギ変換回数の減少のトレンド:

これらを読んでも、
本稿で扱った「構造化プログラミング」で考えているような、
「分かりやすさ」の向上を鍵として、
標準的な構造を導入することによる簡単化のアプローチは
あまり感じられない。

「汎用性原理」を拡張すべきことは次節に再度述べる。

**「どのよ うにして、システムの複雑さを克服するのか?」は
もっと大きな問題として今後考えていくべきことであると感じる。**

4.7 「汎用性」の原理の拡張と「標準化/規格化」の重要性

標準的なものを作って個別の設計/制作を置き換え、
「分かりやすくする」そして「作 りやすくする」ことが、
構造化プログラミングの大きなねらいであった。

この考え方はTRIZの発明原理6.に含まれているものであるが、
もっと明示的にする、拡張する必要があると考える。

TRIZの発明原理 6. をつぎのように拡張することを提案する。

「発明原理 6. 汎用性
A. 一つの物体やシステムが複数の機能を実行できるようにし、
他のシステムの必要性をなくす。」

**(追加) B. 基本的で標準的な機能単位のものを作り、
広い範囲に一般的に使えるようにする。**
(例: ネジ、乾電池、プログラミング言語などの標準化、規格化)」

この追加は、**TRIZが「特許分析」を土台にしてきたことの盲点**に気付かせる。

「標準化」「規格化」は技術の発展にとって非常に重要なことであるが、
それ自体は「特許」にならない。
特許には標準化・規格化したものを使った利点を書くような部分はない。

このため、技術の発展を支え、技術的問題を解決する大きな要因であるのに、
TRIZの中に適切に取り入れられていない。

Ed Sickafus は、USIT法の適用の重点を**「発明のため」ではなく、**
「技術における創造的な問題解決のため」と規定した。

「発明」に重点を置きすぎると、「新規性」を強調しすぎることになり、
企業現場での実際の問題解決に適当でないからだという。

「標準化」「規格化」を目指すことは、
世界的なスケールで問題を解決するために非常に重要なことであり、
今後のTRIZの発展の方向として、
それをも含めた「創造的問題解決」の目標を掲げるべきである。

4.8 「矛盾」を明確にするプロセスを再考する

「goto論争」の考察は、「矛盾」を明確に定義するプロセスに関して、
TRIZの「矛盾」の扱い方に再検討のヒントを与えた。

「技術の世界」と「人の世界」との扱いの違いに気がつく。

技術の世界では、TRIZを使って矛盾を突き詰めていくとき、
一人の問題解決者 (あるいは共同の意思を持つグループ) が
対立する矛盾を同時に考える形を取っていた。

対立する要求が同時にあるとはいっても、
なんらかの大きな目的があって、その中での要求の対立であった。
「対立」は論理 的に出てきている。

「goto論争」では、
　複数の異なる立場の人々が関与して、対立している。

その対立は、論理的に突き詰めてその「矛盾」に至ったというよりも、
　もっと初期の段階から結論同士は真反対で対立していた。

だから、**「物理的矛盾」の導出が、問題解決の目標地点にあるのではなく、**
　問題解決の随分初期の段階にある。

TRIZの分離原理を使うための3対の質問が、問題状況を明確にするのに役立った。

さらに、**より上位にある共通の目的/目標を明確にすることが有効である。**

ソフトウェア分野の場合には、ソフトウェアを開発する という目的は共通だから、
　(人間や社会分野での対立に比べると) 対立の克服は客観的に行える。

goto論争の場合に、
　対立する要求が、空間/時間/場合について明確になった段階で、
　　その解決策は ほとんど自明であった。

5. おわりに

以上、ソフトウェア工学の初期に提唱され、その後に大きな影響を与えた
　「構造化プログラミング」について、TRIZの観点を加えて考察してきた。

この「ソフトウェア工学とTRIZ」というシリーズのねらいは、つぎの3点である。

(1) TRIZをソフトウェア関連分野に適用した事例を作り、
　　TRIZの適用分野をソフトウェア分野に拡張する。
(2) ソフトウェア工学にTRIZの観点を導入して、
　　ソフトウェア工学自身をより明確にする。
(3) ソ フトウェア工学/情報科学の知見を、TRIZの考え方にフィードバックする。

今回の議論が、**これら3つのねらいに、非常に多くの有用な論点を提供し、**
　新しい認識を要請することが分かった。

今後もこのアプローチを、ソフトウェア工学のさらにいくつかの主題に適用して
　考察を深めていく計画である。