第 15 章　構造化技法

開発方法論を構成するもの

構造化技法について議論を始める前に、開発方法論を構成するものに何があるのかをまず考えておきたい。

私は、開発方法論には原則として、下流から上流に向かって、最低でも次の 3 つがあると考えている。

- プログラミングの方法
- 設計の方法
- 要件定義の方法

しかしすでに述べたように、開発方法論とは「考え方」を基にしたものであるから、これらの全てが一貫して揃っていないと開発方法論と呼べないわけではない。しかし構造化技法では、幸いなことにこの 3 つが揃っている。

前置きは以上にして、構造化技法についての議論を始めたい。

構造化技法の考え方

開発方法論の中心は、「考え方」であると述べた。したがって構造化技法の議論を、まずこの「考え方」から始めたい。

ソフトウェア以外の製品を作る一般の「工学」の基本的なアプローチの方法は、「分割と統合（Divide and Conquer）」と呼ばれるものである。

ある製品をこれから開発しようとする場合に、最初に「どういう製品を作るのか」という製品としての全体とのコンセプトを固める。次に、全体としての大きな、複雑な製品を個々の構成要素に分割して、解決するべき問題をシンプルなものに変える。個々の構成要素がまだ簡単に解決できるようなものでない場合は、まだ「分割」が不十分としてさらに分割を繰り返す。そして最初に定めたコンセプトを実現するための解答を容易に得ることができるようになると分割をやめて、対象とする構成要素について、そのコンセプトを実現する上での最良の解答を求める。今度はその解答を、最終製品になるまで順次統合を繰り返す。これが、「分割と統合」の手順である。

例えば今新しい自動車を開発する場合、まずそのコンセプトを定める。例えば、「リッチな若者に受ける、美しくて高性能な中型車」というようなイメージを最初に明確にする。次に自動車を、シャーシ、ボディ、トランスミッション、エンジン、ブレーキ、インテリアといったように、自動車を構成する個々の構成要素に分割してゆく。分割が不十分な場合、さらに分割を繰り返す。そしてそれぞれの自動車の構成要素ごとに、先に定めたコンセプトを実現するための最良の解答を求める。解答が求まると、それを今度は最初の自動車に戻るまで統合を繰り返す、というわけである。

この方法に、全く問題がないわけではない。例えば今挙げた自動車の例で、エンジンの設計チームが決めたエンジンが、ボディのデザイン・チームが決めたエンジン・ルームに収まらない、というようなことがあり得る。その時にどう解決するのかについては、一般的な答えはない。エンジンとボディのどちらを優先するのかで、答えが違うかもしれない。エンジンが大事ならエンジンをそのままにして、ボディを変えるという解答がある。逆にボディはそのままにしておいて、エンジンの設計を変更するという方法もある。両方とも変更するという答えもあ

る。これを決めることは、その自動車の設計に責任を持つ上位の管理者の役割である。
構造化技法とは、この一般の工業製品の開発に適用されている「分割と統合」の考え方をソフトウェアに適用しようとするものである。ソフトウェアではこれを、「トップ・ダウン・アプローチ」と呼んでいる。

構造化技法の経緯

1968 年に、当時オランダの大学に勤めていたエズガー・ダイカストラ（Edsger W. Dijkstra）教授がアメリカのコンピュータ関係の学会誌として著名な CACM（Communication of ACM）の編集長に手紙を書き、「プログラムの中で GOTO ステートメントを使うのは良くない」と主張した[DIJ68]。この雑誌の編集長はこの手紙をそのまま彼の雑誌に掲載し、それをきっかけにして「GOTO ステートメントは有害かどうか」について、世界的な論争が起きた。これが、後で述べる構造化プログラミングへのきっかけになる。
1974 年に、当時 IBM に勤めていたグレンフォード・マイヤーズ（Glenford Myers）がモジュール分割の考え方を取り入れて構造化設計を完成させ、1979 年にトム・デマルコ（Tom DeMarco）がデータフロー図とデータ・ディキショナリを提案して構造化分析をスタートさせた。
結局構造化技法は 10 年以上をかけて、下流から上流へとゆっくりと遡っていったことになる。しかし私は、構造化技法が完成したのはエドワード・ヨードン（Edward Yourdon）が "Modern Structured Analysis" [YOU89b]を出版した 1989 年のことであると考えている。ダイカストラ教授の手紙から、20 年以上も後のことである。

構造化技法の特徴

構造化技法の考え方は、他の工学分野で普通に使用されている「分割と統合」のアプローチをソフトウェアに適用することであると述べた。ソフトウェアの世界ではこの考え方を「トップ・ダウン・アプローチ」と呼んでいることも既に述べた。これが、何よりも大きな構造化技法の特徴である。
もっと端的にいうと、構造化技法ではコンピュータが果たすべき機能に注力し、ソフトウェアでその機能を実現するための手順（プロセス）と方法を実現しようとするもの、ともいうことができる。
例えば、私が長年その開発と保守に当たってきたビジネス・アプリケーションを構成する個々のソフトウェアの目的は、「出力を作成すること」である。ビジネス・アプリケーション自身の目的はもっと高邁なもので、ビジネスの効率の向上とか、より容易に企業内の状況を把握できるようにすること、作業でのミスの減少、などが普通挙げられる。しかしこれはもっと大きな情報システムとしての目的であって、個々のプログラムレベルになると、目的はもっと卑近なものとなる。私は、個々のプログラムの目的は必要な出力を適切に作成することであると考えている。
仮にそうであるとするなら、構造化技法のアプローチは、必要なタイミングで必要な出力を作成するためにコンピュータが何をしなければならないのかを考え、それを明らかにし、それを実現するためのプログラムを用意することである。
何かの出力を作成するためには、当然データの入力が必要になる。しかしデータが発生するタイミング、つまりデータ入力のタイミングと、出力が必要となるタイミングは、一般に一致

しない。このタイミングの差を、ファイルが埋めることになる。つまりデータ発生時点でそのデータを即入力し、必要な処理を行ってファイルに蓄積しておく。そして出力が必要なタイミングになると、そのファイルから必要なデータを取り出して出力する訳である。

ここにあるのはファイルであって、データベースではない。構造化技法には、本当の意味のデータベースの考え方はない。もちろんデータベース管理システムをファイル管理システムの代わりに使い、見かけ上データベースを採用したように見せかけることはできる。しかしそれはあくまで見せかけであって、厳密な意味でこれはデータベースではない。

つまり構造化技法の特徴の 1 つは必要なデータをファイルに格納し、そのファイルをバケツリレーをするように多くのプログラムが順次処理を行って、最後に出力を作成するという目的を達する処理の方式にある。当然このファイル群はこのプロセス専用のものであって、他の情報システムとの共用は難しい。

さらに構造化技法の別の特徴として、ウォータフォール型の開発手順との親和性が高く、プログラム言語として COBOL がよく使われてきたことを挙げることができる。

なお、データ中心アプローチ（Data Oriented Approach：DOA）が普及を始めてから、構造化技法は「プロセス中心アプローチ（Process Oriented Approach：POA）」と呼ばれるようになっている。しかしこの言葉は両方とも和製英語で、欧米では通用しないので注意が必要である。

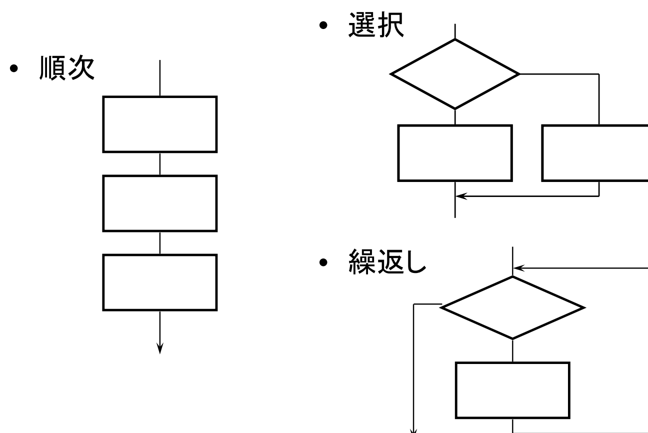
構造化プログラミング

構造化プログラミングは、「GOTO ステートメント有害論」からスタートしたと述べた。この GOTO ステートメント有害論は、すぐに「全てのプログラムは 3 つの制御構造だけで記述することができる」という「構造化定理」に置き換わって、この構造化定理に則ってプログラミングすることが「構造化プログラミング」と呼ばれるようになった。構造化定理はソフトウェアに関わる定理の中で、数学的にきちっと証明された数少ない定理の 1 つである。

この 3 つの制御構造とは、

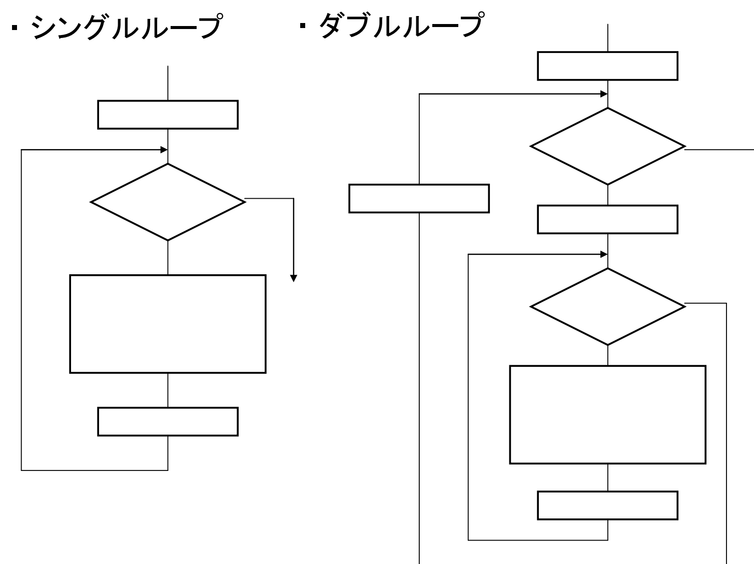
- 順次
- 選択
- 繰り返し

であり、これをフローチャートで図示すると図表 15-1 のようになる。



図表 15-1　3 つの制御構造

この話を情報系の大学で新入生に聞かせると、喜ぶ人が多い。プログラミングとはいたって簡単なものだと思うらしい。しかしこの 3 つの制御構造のそれぞれで「処理」を表す 1 つの四角の中に自分自身も含めて別の制御構造が入り、それが再帰的に、際限なく続くという話をすると、最初はよくは理解してもらえず、その例として繰り返しの中に別の繰り返しを入れる、いわゆる「ダブルループ」を示すと、うんざりしたような顔になる。図表 15-1 の繰り返しの図は簡単すぎるので、それを正規の図にしたシングルループとダブルループのフローチャートを、図表 15-2 に示す。



図表 15-2　シングルループとダブルループ

図表 15-1 で示した 3 つの制御構造だけでプログラミングできるように用意されたプログラム言語を、「構造化プログラム言語」と呼んでいた。最初は COBOL や FORTRAN は、構造化プログラム言語ではなかった。しかしその後の言語仕様の改定で、これらも構造化プログラム言語になった。C 言語は最初から構造化プログラム言語だった。しかし C 言語には、GOTO ステートメントが用意されている。その語彙に GOTO ステートメントを持たなくなった最初のプログラム言語は、Java である。

構造化設計

構造化設計の「心」は、グレンフォード・マイヤーズ（Glanford Myers）が唱えたモジュール化にある[STE74]。モジュール化とは複数の機能を持つ大きなプログラムを、「1 つの入り口、1 つの出口、1 つの機能」である「モジュール」に分割し、それらを組み合わせて、全体としての大きな機能のプログラムを実現することである。

このモジュール化の善し悪しが、設計の善し悪しになる。構造化設計で良い設計とは、「モジュールの強度を『最大』にし、モジュール間の結合度を『最小』にするもの」をいう。ここで「モジュールの強度」とは個々のモジュール内のステートメント間の関連性を意味し、「モジュールの結合度」とは 1 つのモジュールの他のモジュールとの関わり方をいう。

モジュールの強度には 7 つのレベルがあり、モジュールの結合度には 6 つのレベルがある。

このそれぞれのレベルを、モジュールの強度については図表 15-3 に、モジュールの結合度については図表 15-4 に示す。

モジュールの強度についていえば、「機能的レベル」が最も好ましい。機能的レベルとは、例えば C 言語の実行環境にある「平方根」を計算するファンクションのように、全てのステートメントが、ある単一の機能（今のファンクションの例では、「数学的に平方根を計算する」機能）を実現するためだけに存在しているような状態をいう。

図表 15-3　モジュールの強度

好ましさ	強度	レベル	特　　徴
↓	弱	暗号的	複数の、無関係な機能をモジュール化したもの
		論理的	複数の機能を1つのモジュールにし、動かすものを外部から指示する形
		時間的	複数の実行時間が同じ逐次的な機能を1つのモジュールにまとめたもの
		手順的	複数の関連性を持つ逐次的な機能を1つのモジュールにまとめたもの
		連絡的	モジュール内の要素間でデータの受け渡しを行うもの
		情報的	特定のデータ構造を持つ複数の機能をまとめたもの
	強	機能的	全ての要素が1つの機能を実行するために関連しあっているもの

モジュール間の結合度では、「データ結合」が最も好ましい。データ結合とは、2 つのモジュール間のデータの授受が、可能な限りデータ渡しの形で、常にパラメータを通してなされることをいう。

図表 15-4　モジュールの結合度

好ましさ	結合度	レベル	特　　徴
↓	強	内容	他のモジュール内のデータを直接参照したり、直接ブランチしたりする
		共通	共通域のデータ構造を参照する
		外部	外部宣言しているデータを参照する
		制御	制御要素がパラメータとして渡される
		スタンプ	2つ以上のモジュールが、共通域にはない同じデータを共有する
	弱	データ	データ要素のパラメータ受け渡しでインタフェースを取る

モジュールの強度が弱いとモジュールの内部を理解することがより難しくなり、モジュールの結合度が弱いと 1 つのモジュールに入った修正が他のモジュールに修整が伝搬する可能性が高くなる。つまりある理由で 1 つのモジュールを修正しなければならなくなると、それに関連して他のモジュールも修正する必要が生じることがある。

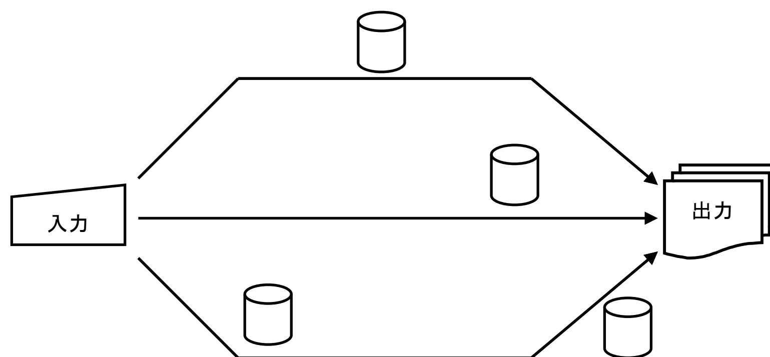
この後構造化設計では、エドワード・ヨードンがラリー・コンスタンチンと書いた本[YOU75]とページジョーンズの本[PAG80]が出版された。私は、これらをいずれも名著と評価している。

構造化分析

構造化分析はトム・デマルコの、やはり今では古典ともなった文献[DEM79]に遡る。彼はここで、データフロー・ダイアグラム（Data Flow Diagram　: DFD）とデータ・ディキショナリを提案した。

その後、チェンが提唱した実体関連図（Entity Relationship Diagram : ERD、ER 図）とコンピュータ・サイエンスの 1 つの中心的な存在であるオートマトンからの流れを持つ状態遷移図（State Transit Diagram : STD）を取り込んで、構造化分析ではこれらの図を用いて、情報システムのイメージを記述してゆくことになる。

この原稿の最初の方で述べたように、構造化分析では最初に出力を確定し、その出力を作るために入力を明らかにする。この入力から出力を作る手順は、特に規定されていない。もちろん効率は大切であり、情報システムやプログラムの変更の容易性も大切である。しかし基本的に、出力が適切に作成されることで、その情報システムは「合格」となる。



図表 15-5　構造化技法で作られる情報システムの広がり

従って構造化技法で作成したシステムは、次に述べるデータ中心アプローチで作成した情報システムと比較して処理の方法が多岐にわたり、設計者の個性が入りやすくなり、処理の内容の幅が広いという弱点がある。この概念を、図表 15-5 に示す。データ中心アプローチについて同じ概念を記した図表 16-2 と比較してみて欲しい。

構造化技法によるシステム構築法

これまで述べてきたことと重複する部分もあるが、構造化技法で情報システムを構築しようとする時、一般に以下の手順に従う。

1. システムの目的（出力の作成）を達成するために、必要な処理（機能）を明らかにする。
2. その機能を満たすために必要なデータ（入力）を明らかにする。
3. プログラムを設計する／ファイルを設計する。
4. プログラムとファイルを作成する。
5. テストする。

構造化技法の功罪

構造化技法の最大の功績は、情報システム構築を行うための 1 つの標準を確立したことである。その対象は、世界中に及ぶ。この功績によって、情報システムの開発を社外のソフトウェア会社などに委託できるようになった。この効果は大きい。

また世界中のソフトウェア技術者がこの技法の概念と技術用語を共有したことによって、お互いに容易にコミュニケーションできるようになったことも大きい。私も現役の技術者だった頃に、ニューヨークやロンドン、チューリッヒなどに出張して何度か現地のソフトウェア技術者と話し合ったことがあった。英語と日本語の問題を抜きにすれば、コミュニケーションは非常にスムーズだった。

一方、構造化技法には大きな問題がある。その 1 つは、「トップ・ダウン・アプローチ」にある。これは構造化技法の基本の考え方であるが、このアプローチになじめない人が多い。私の元同僚で、たいへん優秀なソフトウェア技術者だったにもかかわらず、ついにこのアプローチには馴染めないまま定年を迎えた人がいる。

「トップ・ダウン・アプローチ」の議論は、「そもそも」というような言葉を付けて始めなければならないことが多い。一方の「ボトム・アップ・アプローチ」は今足下にどんな問題があり、それをどう解決したらよいかというところから考え始めればよい。問題解決の手順として、ボトム・アップ・アプローチの方がはるかに易しい。仮にその方法を習得することができても、やはりトップ・ダウン・アプローチは難しいといわなければならない。

さらに構造化技法では、「いかに処理をするか」の意識が強くて、「何を処理の対象にするか」の意識が薄い。「何を」は「いかに」より、重要なことが多い。

構造化技法の将来

構造化技法はまず処理するべき内容を考え、それを素直にコンピュータに実行させるようにプログラムを作るという方法だった。だから昔のようにコンピュータの処理能力が低く、やりたいことが山のようにある時代には最適の方法だった。

別のいい方をすると、ある処理を行うのにいくつかの方法がある場合、IBM のマニュアルで各インストラクションの実行時間を調べ、それぞれの方法での処理時間の合計を求めて比較して、処理時間が短くなる方を採用してアセンブラでプログラムを作成していた時代には、開発方法論としては構造化技法しかあり得なかった[1]。

しかしコンピュータがパワフルになり、コンピュータの処理能力に余裕が生まれると、構造化技法とは異なるアプローチで情報システムを作ることが可能になった。それがデータ中心アプローチであり、オブジェクト指向技法である。

このように考えると、構造化技法は将来的に、着実に廃れて行く方向にあるように見える。1 台ずつのコンピュータはまだまだ強力になり、それらがグリッド・コンピューティングを構成するようになると、コンピュータの処理能力は今の我々の想像を絶するような、巨大なものになるだろう。したがって、コンピュータの処理能力の限界に挑むために構造化技法でソフトウェアを作る必要はなくなるものと考える。

しかし一方で、構造化技法はまだしばらくは生きながらえるだろう。今大学では学生にオブジェクト指向技法を教え、プログラムの演習は C++や Java で行っている。書店に並んでいる

---

[1] 私が 1975 年頃に取り組んだ日興證券（当時）の第 1 次オンラインシステムの開発で、私はまさにここに書いたような方法でプログラムを作成していた。

書籍も、その通りである。COBOL の教材はもう、大学でも書店でもほとんど見ることがない。
それにも関わらず今稼働しているビジネス・アプリケーションの多くは、構造化技法によって、COBOL で開発されたものである。それらはまだ当分の間現役でなければならず、日本のビジネスを支え続ける必要がある。従ってその情報システムは、的確に保守され続けなければならない。そのために構造化技法の理解も、COBOL での開発能力も、まだまだ重要である。私はまだ、そのように考えている。

キィワード

開発方法論、構造化技法、分割と統合、トップ・ダウン・アプローチ、プロセス中心アプローチ、ウォータフォール型開発手順、構造化プログラミング、GOTO ステートメント有害論、構造化定理、制御構造、構造化プログラム言語、COBOL、構造化設計、モジュール化、モジュール、モジュールの強度、モジュール間の結合度、構造化分析、データフロー・ダイアグラム、DFD、データ・ディキショナリ、実体関連図、ERD 、ER 図、状態遷移図、STD

略語

CACM：Communication of ACM
DOA：Data Oriented Approach
POA：Process Oriented Approach
DFD：Data Flow Diagram
ERD：Entity Relationship Diagram
STD：State Transit Diagram

人名

エズガー・ダイカストラ（Edsger W. Dijkstra）、グレンフォード・マイヤーズ（）、トム・デマルコ（Tom DeMarco）、エドワード・ヨードン（Edward Yourdon）

参考文献とリンク先

[DEM79] Tom DeMarco 著、高梨智弘、黒田純一郎監訳、「構造化分析とシステム化仕様-システム設計者とユーザーのための系統的開発手法」、日経 BP 社、1986 年.
この本の原書は、以下のものである。
Tom DeMarco, “Structured Analysis and System Specification,” Prentice Hall, 1979.
[DIJ68] Edsger W. Dijkstra, “Go To Statement Considered Harmful,” Communication of the ACM, Vol.11, No.3, ACM, March, 1968.
このペーパーは、ACM のデジタル・ライブラリ（http://portal.acm.org/portal.cfm）からダウンロードすることができる。（ただし、ACM のメンバーであることが必要。）
[PAG80] Meikir Page-Jones, “The Practical Guide to Structured System Design Second Edition,” Prentice-Hall, 1980.
[STE74] W. Stevens, G. Myers, L. Constantine, “Structured Design,” IBM System Journal, Vol.13, No.2, IBM, May, 1974.
このペーパーは、ACM のデジタル・ライブラリ（http://portal.acm.org/portal.cfm）からダウンロードすることができる。（ただし、ACM のメンバーであることが必要。）

[YOU75] Edward Yourdon, Larry Constantine, "Structured Design Fundamentals of Discipline of Computer Program and System Design," Prentice-Hall, 1975.
[YOU89b] Edward Yourdon, "Modern Structured Analysis," Yourdon Press, 1989.

（2007 年（平成 19 年）6 月 26 日　初稿作成）
（2014 年（平成 26 年）3 月 21 日　一部修正）