第 26 章　プログラムの設計

「設計」という作業

スティーブ・マコネル（Steve McConnel）にいわせると、設計とは「ヒューリステック」なプロセスであるという[MCC05]。「ヒューリステック」とは難しい言葉だが、具体的には、確定した方法がなく、ルーズなプロセスで、臨機応変に取り組んでよい、ということを表している。その結果、例えば同じ要件定義書を基に 3 人の設計者がそれぞれ設計を行ったら、3 通りの異なった設計ができあがり、そのいずれもが正しい、ということが起きる。

私にいわせると、設計者が設計を行う場合、設計者の頭の中でプログラムのイメージが作られ、それが頭の中で動く。何通りもの動き方がある場合には設計者がその中で一番良いと思う方法選び、それを具体的に固めるということで設計が行われる。要件定義 [1]でも次の章で述べるプログラミング [2]でも、まず作業をしている人の頭の中で情報システムやプログラムの完成した状態についてイメージができる。そのイメージを具現化するために行う作業が要件定義であり、プログラミングである。

設計も、これらと変わらない。要件定義書を基に、設計者の頭の中でこれから設計しようとしているプログラムがまずできあがり、それが動く。この動いているプログラムをどう作るかについて、文書にまとめる。それが「設計」である。動いているプログラムのイメージが一流で、文書にまとめる技術が一流なら、その人は設計者として一流ということになる。

設計だけに限るものではないかもしれないが、設計とはこのように、充分な経験に裏打ちされるべき作業であると私は考える。

オンライン・システムとバッチ処理のプログラムの設計

オンライン・システムとバッチ処理のアプリケーションプログラムについての一番大きな相違点は、OS とアプリケーションプログラムの間に、オンライン・システムでは一般にアーキテクチャを実現する「ミドルウェア」と呼ばれるプログラムが存在しているのに対して、バッチ処理のプログラムでは一般にアプリケーションが直接 OS の下に位置することである。このことがオンライン・システムとバッチ処理のアプリケーションプログラムの設計に、ある相違を生み出す。

その相違は、オンライン・システムでは入力されたトランザクションをアプリケーションプログラムはミドルウェア経由で受け取るのに対し、バッチ処理では原則として自分で必要なデータを読まなければならないことにある。つまりバッチ処理では、アプリケーションプログラムは全体として 1 つの大きな連続したループを構成し、その中でデータを読み込むこととそのデータを処理することを実現しなければならない。それに対してオンライン処理では、ミドルウェアから渡されたデータについての処理を完結すればアプリケーションプログラムの処理が完結することである。ここには、データを読み込むためのループは必要としない。

データ中心のアプローチでは、処理は発生したトランザクションからデータベースを更新するまでのものと、更新済みのデータベースから各種の出力を作成する部分に、大きく 2 つに分けられる [3]。

---

[1] 要件定義書の作成については、第 21 章で述べた。
[2] プログラミングについては、第 27 章で述べる。
[3] データ中心アプローチについては、すでに第 16 章で述べた。

個々のプログラムは、後で述べる「1 つの機能」を実現するために設計される。この「機能」の中に、全体の処理の流れなどを制御する部分とある項目の計算など個別の処理を実現する部分に分けられる。

モジュールなどを組み合わせたモジュール構造を図示すれば、モジュール構造の上部には処理の流れを制御するものが位置し、下部には個々の計算など個別の処理を行うものが位置することになる。プログラムの設計をトップダウンで行うのか、ボトムアップで行うのかで、処理の流れを先に設計するか、個々の機能を先に設計するかの違いが生じる。トップダウンとボトムアップの問題は、後述する。

この最上位あたりに位置するモジュールの機能の相違を除いて、オンライン処理のプログラムとバッチ処理のプログラムに大きな違いは存在しない。

良い設計とはどういうものか

それでは、良いプログラムの設計とはどういうものであろうか。

要件定義書に記述されたことが的確に実現できており、しかも実行時に敏速に動き、コンピュータ資源（メモリ、CPU の負荷、など）を多用しない、などというのはいうまでもなく「良いプログラム」であるための当然の要件である。これ以外に、以下のような要件がある[MCC05]。

- 分かりやすいこと。（難解でないこと。）
- 1 つの入り口、1 つの出口、1 つの機能を実現していること。
- 他のプログラム（モジュール／クラス）との関係が「疎」であること。
- 情報隠蔽が的確になされていること。
- 変更への対応が適切に図られていること。

以下で、この「良い設計の要件」について議論したい。

分かりやすいこと

設計だけに依存する話ではないが、情報システムが全体として分かりやすいことが重要である。プログラム設計の結果として、モジュールやクラス（以下では総称して「プログラム」と呼ぶ）が設計される。

その 1 つ 1 つのプログラムが果たす処理の内容、つまり機能が分かりやすく、他のプログラムから見てインタフェースが分かりやすく、言い換えると処理を依頼する方法が容易であることが必要である。

この分かりやすさを実現するために、奇をてらった独自の処理の方法や、機能の分割をしてはならない。言い換えると、要件定義書に記載されている要件をコンピュータで実現するためには、多くの人が「こういう方法が適切だ」と考える方法でコンピュータが処理するように設計されていることが望ましい。

それを実現するためには、設計者は自分が設計したプログラムだけではなく、多くの他の設計者の作業の結果にも精通していて、どのようにそれぞれのプログラムを分割し、そのプログラムにどのような機能を持たせ、どのようなインタフェースを具備しているのかなどについて理解していることが必要である。開発組織にレビューの文化があると、ウォークスルーやインスペクションなどのレビューを実施するときに、他の人が書いたプログラムを読む機会を持つことができる。これは、レビューの本来の目的ではない。しかしこのことは必要なことであり、

それが自動的に実現できるということはたいへん好ましいことである[4]。

オブジェクト指向の世界に、「デザインパターン」という名著がある[GAM95]。クラスを設計するときに、「使える」クラスの組み合わせやパターンについて記述されたもので、多くの人に高く評価されている。オブジェクト指向で情報システムを設計する機会を持つことがある人はこの本を精読し、できれば全てのパターンを熟知し、どのような時にどのパターンを使うと効果があるのかを理解しておいてほしい。

構造化プログラムの世界には、このような名著は存在しない。しかしここでも、オブジェクト指向の場合と方法は異なるが、プログラムのパターンを理解し、習得し、活用できることが必要である。

1 つの入り口、1 つの出口、1 つの機能

プログラムは、1 つの入り口、1 つの出口と、1 つの機能を持つことを原則とする。1 つの入り口を実現することは比較的簡単だが、1 つの出口を実現するためには、場合によればある工夫が必要になる。

しかし問題は、1 つの機能である。「機能」とは、何だろうか。これは単純に、「広い意味での 1 つの出力を作成すること」であると考える。例えば次のようなものが、1 つの機能の例である。

- 全体の処理の流れを制御すること。
- あるデータ項目がエラーであるかどうかを判定すること。
- 消費税額など、ある項目を計算すること。
- 帳票の 1 行／単票の 1 ページを編集すること。
- 西暦と和暦の日付を変換すること。

あるデータ項目の計算や西暦／和暦の日付の変換、あるいは特定のデータ項目のエラー判定などのプログラム（モジュール／クラス）は、1 つのプロジェクトの中だけでなく別のプロジェクトや、あるいは別の情報システムとの間で共用することができる[5]。

疎結合

構造化プログラムの世界でモジュール分割を行うときに考慮するべき要件として、強度と結合度という 2 つの概念があった。すぐ後で述べるが、強度は強い方が良く、結合度は弱い方が良かった。

「強い強度」とは、1 つのモジュールの全てのステートメントがそのモジュールが果たすべき機能を実現するために関連していることを意味している[6]。

一方の「結合度」とは、1 つのモジュールが別のモジュールとどの程度関連しているかを意味していて、こちらは前述の通り弱い方が望ましい。お互いに強い結合度を持った複数のモジュールが存在する場合、何らかの理由でその中の 1 つのモジュールを修正しなければならない状況が生まれると、他のモジュールにもその修正が波及してしまうという状況が起きる。

弱い結合度は、1 つのモジュールが別のモジュールをコールするときのデータの受け渡しの方法を、データ項目単位のパラメータを使用することで実現できる。受け渡しするものがデー

---

[4] レビューについては、すでに第 18 章で述べた。

[5] 「ソフトウェアの再利用」については、第 31 章で述べる。

[6]「強度」は、「凝縮度」と呼ばれることがある。

タ項目ではなくレコード単位になると、結合度は少し強くなる。

弱い結合を「疎結合」、強い結合を「密結合」ともいう。

構造化プログラミングの時代に、私はこの強度と結合度の議論は決着を見たと考えていた。しかしオブジェクト指向プログラミングの時代になって、つまりスーパークラスとサブクラスの間に継承を使うことができるようになって、強度も結合度も状況が少し変わってきた。私はこのことに、たいへん驚いた。いったん決着したテーマを最初から覆すようなことは、もう起きないと考えていた。

しかしオブジェクト指向の時代になっても、結合度の重要性は少しも変わらない。プログラム間の結合は、極力「疎結合」でありたい。継承を使わない方が良いとまではいわないが、マコネルは継承を使うときにはその使い方に充分に注意するようその著書の中で力説している[MCC05]。

情報隠蔽

「情報隠蔽」とは、必要のない情報をプログラム（モジュール／クラス）の外部に明らかにしないことをいう。明らかにして良い情報はそのプログラムが持っている機能と、そのプログラムに処理を依頼するときの、その依頼の仕方だけである。それ以外の情報は情報隠蔽の対象になり、明らかにされない。

「情報隠蔽」は、オブジェクト指向の世界の言葉であると考えている人がいるかもしれない。オブジェクトの性格の１つに、「情報隠蔽」が置かれているからである。

しかし情報隠蔽はもう少し古い概念で、構造化技法の時代の途中からすでに存在していた。構造化プログラムでも、情報隠蔽を実現したい。

変更への対応

プログラムの中には、変更されやすい所と変更されにくい所がある。例えば、税金に関わる法律はほとんど毎年改正されて、それに伴って税金の計算方法は変更される可能性が大きい。

プログラムを設計するときに、このような変更されやすい場所が特定できるなら、あらかじめ設計時にその変更への配慮をしておくことが望ましい。あるいは、設計書には「変更される可能性が高い」ことだけを明記し、具体的な変更への対処はプログラミングの段階に譲るという方法もある。

トップダウンの設計か、ボトムアップの設計か

プログラム設計は、トップダウンで進めるのが良いのだろうか、ボトムアップで進めるのが良いのだろうか。結論は、「進めやすい方法で進めるのが良い」というもので、どちらかでなければならないというものではない。

構造化技法は、建前トップダウンで作業を進めることになっていた。しかし、上位のモジュールは処理の流れをコントロールし、下位のモジュールは個々の処理に対応すると述べたように、モジュール設計ではトップダウンの作業とボトムアップの作業で、作業の対象となるものを取り上げる順序が異なる。建前通りのトップダウンで進めることができるならそれでいっこうにかまわないが、ボトムアップの方が進めやすいならボトムアップでも良い。

あるいは、ある範囲をまずトップダウンで進め、その後別のところをボトムアップで進めるということでも良い。トップダウンかボトムアップかということについて、こだわる必要はな

い。

あるいは、そのプログラムの処理の内容で、制御の流れが難しい場合はトップダウンで、個々の計算などが複雑な場合はボトムアップで行う、という考え方もある。

設計はどこまで行うべきか

設計はどこまで行うべきで、何を持って設計終了とするのが妥当だろうか。

最低限の設計とは、個々のプログラムが持つべき機能をどう実現するかについて、誤解がないようにしっかりと明らかにし、その上で他のプログラムとのインタフェースを明確にすることで行われる。つまり前述した情報隠蔽で、隠蔽の対象にはなっていないものを明確に記載することで行われる。

一方の最大限の設計とは、自然言語（日本語や英語など）や図表などを使って書かれた設計書の内容を機械的にプログラム言語に変換することでプログラムが作成されるところまで記載することである。この最低限と最大限の設計の間に、無数の段階が存在する。

最低限と最大限の二者択一であるなら、私は「最低限の設計」で良いと考えている。つまり、そのプログラムについて明確にしておかなければならないことだけを設計段階で明らかにし、後はプログラミングの担当者の自由裁量に任せるのが妥当と考えている。

最低限の設計に何かを付加することで設計内容をより明確にできるなら、最低限の設計にこだわることはない。

それでは、設計はどこまで行うべきだろうか。開発組織には、プログラム設計のための「標準」が定められているだろう。その標準に従って、最低限の記述をし、設計を担当している人がこれで充分と考え、さらにその設計書をレビューする人がここまで記載すれば良いと考える所まで記載がなされていると、それで設計を終わりにして良い。

新人プログラマへの教育をかねて、設計書を懇切丁寧に書く必要はない。教育は設計とは別の切り口で行えば良い。

設計に関わる規格

情報システムの設計に関わる国際規格には、IEEE が定めた IEEE Std 1016-2009 がある[IEEE09]。これは 1998 年に定められたものを 2009 年に改定したもので、設計に関わる国際規格ではこの原稿を書いている時点（2016 年 5 月）では、唯一のものである。

キィワード

ミドルウェア、デザインパターン、機能、強度、結合度、疎結合、情報隠蔽、トップダウン、ボトムアップ、最低限の設計、最大限の設計

人名

スティーブ・マコネル（Steve McConnel）、

規格

IEEE Std 1016-2009

参考文献とリンク先

[GAM95] エリック・ガンマ他著、本位田真一他監訳、「オブジェクト指向における再利用のためのデザインパターン 改訂版」、ソフトウェアバンクパブリッシング、1999 年.
この本の原書は、以下のものである。
Erich Gamma and Richard Helm, “Design Patterns, “Addison Wesley, 1995.
[IEEE09] IEEE Computer Society, “IEEE Standard for Information Technology -Systems Design - Software Design Descriptions IEEE Std 1016™-2009,” IEEE Computer Society, 2009.
[MCC05] Steve McConnel 著、(株) クイーブ訳、「Code Complete 第 2 版」、日経 BP 社、2014 年.
この本の原書は、以下のものである。
Steve McConnel,” Code Complete, Second Edition, “Microsoft Corporation, 2005.

（2016 年（平成 28 年）5 月 3 日　新規作成）