1. はじめに

ソフトウェアのデザインパターン本が流行っている。まだ、日本語に翻訳されているものは少ないが、米国では「デザインパターン」をキーワードとした本の出版が続いている。題名から直感的に内容を推し量ることができるのがその原因だろう。

しかし、私の目から見ると、この流行には大きな誤解がある。すなわち、「デザインパターンを採用すればすぐに生産性が上がる」という幻想である。ある前提の下ではこの命題は成り立つのだが、多くの読者はこの「前提」を忘れている。

その前提とは、「ソフトウェア工学・ソフトウェア科学の成果を土台としてデザインパターンを適用すれば、ソフトウェアの信頼性が向上する」というものである。結果として、欠陥による手直しのフィードバックが減少し、ソフトウェアの生産性が上がる。つまり、デザインパターンの本は「ソフトウェア工学・ソフトウェア科学の採用」を前提として書かれているのだ。

ところが、日本のソフトウェア開発の現場では「ソフトウェア工学・ソフトウェア科学の知識」は前提とされていない。このため、開発現場のプロジェクトでは本来なら避けられるはずの様々な問題が発生している。例えば、私がコンサルタントとして参加した「すべてのプロジェクト」でこのような問題が発生し、プロジェクトは遅れ、ソフトウェアの信頼性は低かった。

このような混乱した状態で、救いをデザインパターンやその前提となるオブジェクト指向技術に求めるのは間違いである。まず、「ソフトウェア工学・ソフトウェア科学」の成果を利用し、その上でオブジェクト指向技術を適用し、さらにそれらをベースとしたデザインパターンを採用すべきなのである。

では、ソフトウェア工学・ソフトウェア科学の成果とは何だろう？　デザインパターンに関連する成果に絞ってみれば、それは情報隠蔽やカプセル化を行う構造化技法をベースとした分析・設計方法論（すなわちオブジェクト指向分析・設計方法

論)、What あるいは「どうあるべきか」を記述する宣言的な仕様記述言語、アルゴリズムとその最先端を行くコンパイラー理論、そしてオブジェクト指向言語技術、さらに人間が主役であるというピープルウェアの概念に基づくチーム編成とプロジェクト管理、ということになる。

そこで本書は、分析・設計段階でデザインパターンの適用を目指す読者を対象として、特定のオブジェクト指向言語は想定せず、実践的に分析・設計が行えるように、分析・設計法と仕様記述法の説明を重視することにした。

また、実際に分析・設計を行う上での思考過程をなるべく明らかにしようと努めた。特に、分析工程の細かい思考過程を書いた本はあまりないので、そのあたりの記述に重点を置いた。

そこで、本書の構成を以下のようにした。

2 章ではデザインパターンを概説する。特に、デザインパターンの評価基準となる「良い設計」の要因について説明する。

3 章は、4 章以後で必要となる個々のデザインパターンの要約である。ここにはデザインパターンだけでなく、分析パターンやアーキテクチャパターンといったものも記述する。

4 章は、本書の核となる部分であり、デザインパターンを意識しながらどうやって分析・設計を進めていくかを述べる。

5 章は、制御系システムの分析・設計をどう進めていくかを述べる。

6 章は、導入の課題と今後の動向を述べる。

クラス図などの記法としては標準になる可能性が強い UML（Unified Modeling Language）[1] を用い、仕様記述言語も UML で定義されている OCL（Object Constraint Language）[2] を使う。UML の記法や OCL の文法は、最初に出てきたところで概要を説明する。

7 章は、付録であり、UML と OCL の概要説明を行う。また、参考文献についても記述する。

---

1. 『UML Notation Guide version 1.1, Rational Software, 1 September 1997』及び付録 7.1 参照。
2. 『Object Constraint, Language Specification,Rational Software, 1 September 1997』及び付録 7.2 参照。

2. デザインパターン入門

この章では、ソフトウェア開発の現状と問題点を述べ、なぜオブジェクト指向技術とデザインパターンが必要かを明らかにする。また、デザインパターンを適用したシステムの評価基準となる「よい設計」要因についても説明する。

さらに、デザインパターンを概観し、詳細仕様を記述するための仕様記述言語について概略を説明する。

2.1 ソフトウェア開発の現状と問題点

デザインパターンが登場した背景として、ソフトウェア開発の現状と問題点を見ていこう。ここで述べる問題点があなたの会社や部署では発生していないとすれば、オブジェクト指向技術やデザインパターンを敢えて採用する必要はない。流行に従って、何でも新しい技術を採用する必要はないのだ。

2.1.1 動かないソフトウェア

昔私が所属していた会社で、開発されたシステムの 3 分の 1 は実際に運用するに至らなかった。多くの場合、エンドユーザー自身の要求が曖昧だったり、エンドユーザーの要求を取り違えた仕様になっていて、プログラムとしては動くのだが、実際の運用には適さなかったのである。

例えば、ある資金運用評価システムは、エンドユーザーにも解答のない問題[1]を解決しようとしていた。結果として、ソフトウェアはできたが、エンドユーザーの所

1. 複数の異なる金額・運用期間の資金運用を、ある時点で評価して「運用成績」を決めようという問題であった。高々 100 件ほどなので、評価方法さえ決めればスプレッドシート・ソフトウェアでも十分解決できるが、万人が認める評価方法など無かった。

属する組織が別会社になり、担当者も変わったことで、そのソフトウェアは必要とされなくなった。

あるいは、ソフトウェアの欠陥を直すことができず 1 年余りリリースが遅れているうちに、世の中が変化してそのシステムが不要になってしまったものもある。

しかし、これでもまだましな方で、米国政府関係のソフトウェアは、もっと動かない率が高いという報告すらある。

動かないソフトウェアはお金の無駄遣いの典型的な例だが、なぜかこのようなソフトウェアを作り出す組織ほど、コピー用紙に裏紙を使わせるといった些末なコスト削減手法[2]が採用されている率が高いというのは、私の思い過ごしだろうか。

2.1.2 保守できないソフトウェア

パソコンの OS を解析し欠陥を修正してしまうような技術者と私とで、200 行ほどの C 言語のサブルーチンを解析しようとしたことがある。しかし、解析は不可能だった。大域変数が多用されていて、引数は一切使われていないため、ある大域変数の値がいつどのサブルーチンによって書き換えられるかを解析しきれなかったからである。

このような解析をするには、プログラムのパス（Path）を解析しなければならないのだが、実用的なプログラムの多くはこの組合せが莫大なものになり、到底解析することができなくなるのである。

私がコンサルティングしたファックス制御ソフトウェアや、同僚がコンサルティングした現金自動引出機制御ソフトウェアでも事情は似たようなものだった。要求仕様はなく、設計仕様はあっても最新でなく、あるのは構造化されていないソースプログラムのみという状況だった。

この他にも、構造化[3]の原則に反する作りをしたソフトウェアの多くが、「保守不能」状態、あるいは保守はできるが膨大なコストが掛かるという状態[4]に陥っている。

---

2. 環境保護に少しは役立つが、プリンターの寿命を縮め人件費が掛かるので、実際にはコスト削減にはならない。
3. オブジェクト指向やデザインパターンも、構造化技法の 1 つである。
4. いわゆる「2000 年問題」は、ソフトウェアの「問題」ではなく「欠陥」により修正に膨大なコストが掛かる例である。西暦の処理を 1 つのサブルーチンで行っていれば、何ら問題ではなかったのである。

例えば、ある証券会社の第 2 次オンラインシステムの最初の目標は、異なるコンピュータおよびネットワーク環境で、第 1 次オンラインシステムと同じ機能を実現するという「新規保守[5]」と呼ばれるプロジェクトであったが、半年間 300 人のプロジェクトが一歩も前に進まなかった。毎週、週はじめに各チームに修正依頼が届き、1 週間修正作業をすると、翌週また修正依頼が届き、といった具合で半年経っても修正依頼が減る気配がない状態が続いたのである。

これは、構造化されていずドキュメントも不完全な、第 1 次オンラインシステムの仕様やモデルを理解せず、「何を作るか」がはっきりしない状態でプロジェクトをスタートさせたことが大きな原因であった。結局は、100 万行あった第 1 次オンラインシステムのソースプログラムを読み、要求仕様を再構築して、ようやくプロジェクトは軌道に乗り、1 年遅れで動かすことができた。

2.1.3 再利用できないソフトウェア

私が経験した証券会社のソフトウェア開発の場合、多くは前に作ったソフトウェアと似たソフトウェアであった。しかし、前に作られたソフトウェアのほとんどが再利用を考慮していないため、新たに似たようなソフトウェアを作らなければならなくなった。

この場合、ソースプログラムをコピーして修正するというやり方が多かったが、ソースプログラムのコピーは、「欠陥」もコピーしてしまう。結果として、品質の悪いソフトウェアを拡大再生産することになってしまう[6]。

2.1.4 品質の悪いソフトウェア

動かないソフトウェアの方がまだましに思えるのが、品質の悪いソフトウェアである。動かないソフトウェアは人を殺さないが、品質の悪いソフトウェアは人を殺すことがある。

---

5. エンドユーザーに提供される機能は変わらないため、一種の保守であるということでこのように呼ばれる。このシステムの場合、第 1 次オンラインシステムのソフトウェアは保守不能であったため、新規保守しか選択できなかった。
6. 私はソースプログラムのコピーと修正という手段は決して取らなかったが、そのようにしたチームは少なく、大多数のチームは欠陥を拡大再生産していった。

放射線を照射する医療機器制御ソフトウェアの欠陥で人を殺した例が報告されているし、名古屋で落ちた中華航空機もその典型的な例であろう。この飛行機の制御ソフトウェアは、「飛行機が飛べない状態」を許していた。

人を殺さなくても、社会的に大きな問題を引き起こす場合もある。例えば、ある株式市場のシステムは、過去10年ほどの間に3回取引ができない状態に陥った。株式の売り買いのソフトウェアには一種の組み合わせ算法(Algorithm)が必要で、工夫しないと計算量が指数関数的に増加し、データが一定数を超えると、どんなに速いコンピュータを持ってきても計算が終了しなくなる危険性がある。しかし、このシステムの担当者はこのような危険性に気づいていなかったのである。

結果として、1回目のトラブルで、ソフトウェアの大修正を行い、当初予定の2倍の性能のコンピュータを導入したが、それも対症療法にすぎず、株式市場が加熱したバブルの絶頂期に2回目のトラブルを起こし、これへの対策も施したが、個別銘柄へ取引が集中した際に3回目のトラブルも起こした。実は、このソフトウェアが最近トラブルを起こさないのは、バブルが崩壊して市場の取引量が減っているからにすぎない。

2.2 よい設計とは？

今までに述べた問題の発生を防ぐことはできるのだろうか？

実はできる。100%は無理だが、問題の発生をかなり抑えることはできる。「よい設計とは何か」が明らかになってなって久しいが、未だに日本のソフトウェア開発現場では採用されていないだけなのである。そこで、以下に「よい設計のための指針」を説明しよう。

分析モデルや設計モデルに「間違ったモデル」はあるが、「絶対に正しい」モデルは存在しない。正しいモデルは無数に存在するが、それらは長所と欠点を併せ持っている。従って、以下に述べる「指針」と照らし合わせて、「よりましな」モデルを作成しなければならない。

2.2.1 外的品質要因

よい設計の到達目標として、あるモジュールに要求される品質要因を外的品質要因と呼ぶが、その中で特に重要なものは以下の5つである。

(1) 正確さ

要求された通りにモジュールが仕事を行う能力であり，日本の場合，多くのソフトウェア開発の現場で，この要因だけは達成しようと努力している[7]。しかし，実際には，達成が非常に難しい品質要因である。

(2) 頑丈さ

異常な状態においても機能する能力である。仕様によって明示されない状態は必ず存在するので，破滅的な状況を回避することが必要である。例えば，編集中のドキュメントを保存したり，データベースの回復をするための情報を待避することなどが「頑丈さ」を達成する。

しかし，多くのソフトウェア開発現場では，この品質要因を達成しようとする努力はなされていない。

(3) 拡張性

仕様の変更に容易に適応できる能力である。大規模プログラミングにおいて特に重要な品質要因である。拡張性を向上させるためには簡明さと非集中性が必要である。

(4) 再利用性

新しい応用にどの程度再利用できるかという品質要因である。再利用されるモジュールは，徐々に品質が向上していくので，品質の向上に寄与すると共にコスト削減にもなる。

---

7. 米国では，これすら守られていないという報告がある。