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

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。

所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。

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]。

---

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。
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。

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。

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。

(5) 互換性

ソフトウェア相互の組み合わせやすさを表す品質要因である。例えば、UNIX は、ファイルはすべて文字ストリームであるとすることによって、コマンドを組み合わせて新たなコマンドを作ることができた。Smalltalk は、変数以外のすべてをオブジェクトとすることによって、クラスやプログラムもオブジェクトとなり、モジュール相互の組合せがかなり自由になった。

2.2.2 内的品質要因

外的品質要因は、モジュールを外から眺めたときの品質要因なので、モジュールを作るときどうしたらよいかは示していない。外的品質要因を満たすために必要な、作成するモジュールが備えているべき品質要因を内的品質要因と呼ぶ。内的品質要因を満たせば、外的品質要因を達成できることが分かっている。

(1) モジュールの分解しやすさ

一群のモジュールの中から、必要なモジュールを簡単に取り出すことができる時、モジュールは「分解しやすい」。分解しやすいモジュールは、変更余波（変更したときの影響度）が小さく、再利用しやすい。

(2) モジュールの組み合わせやすさ

よく考えられたインタフェースを持つモジュール群は、組み合わせることが容易である。

例えば、Smalltalk のクラス・ライブラリーは組合せやすいモジュール群からできていて、他のオブジェクト指向言語のライブラリーのお手本になった。また、UNIX OS は、各アプリケーションを組み合わせるために、ファイルの仮想化・標準入出力の切換・パイプラインという機構を用意していた。

(3) モジュールの分かりやすさ

あるモジュールを理解しようとするとき、そのモジュールのソースコードだけを読めば理解できるのが、分かりやすさの基本である。

例えば、大域変数を使っていると、実行時に他のモジュールがその大域変数を変更することに影響されるので、実用的な大きさのソフトウェアでは、モジュールの理解が非常に困難になる。あるいは、命令の実行順序に依存するモジュールも理解困難なモジュールの例である。

(4) モジュールの連続性

変更の影響が小さい場合、モジュールは連続性を満たしているという。

例えば、シンボル定数を使うモジュールは連続性を満たしているが、物理的表記や静的配列を使用しているモジュールは連続性を満たしていない。

(5) モジュールの保護性

異常条件の影響が少ないとき、モジュールは保護性を満たしているという。

例えば、入力パラメータの妥当性検査をしたり、システム・エラーの場合に情報を保存するといった処理は、モジュールの保護性を高める。

2.2.3 モジュール性の原則

内的品質要因はまだまだ抽象的なので、実際にモジュールを作る際に必要な原則を以下に示す。この原則が守られていれば、内的品質要因を満たす。

(1) 言語としてのモジュール単位

言語の適切な支援なしに高いモジュール性を実現することは不可能である。

例えば、COBOL はこの面では欠陥ソフトウェアだし、マイクロソフト Basic もこの面が弱い。幸いなことに、ほとんどのオブジェクト指向言語では高度なモジュール性を実現できる。

(2) 少ないインタフェース

モジュールはできるだけ少ないモジュールと対話すべきである。オブジェクト指向の場合は、あるオブジェクトは少ないオブジェクトとメッセージのやり取りをす

べきであるいうことになる。あるいは、あるクラスは少ないクラスと関連を持つべきであるとも言い換えることができる。

(3) 小さいインタフェース

２つのモジュールが交換する情報はできるだけ少なくすべきである。オブジェクト指向の場合、メッセージの引数はなるべく少なくし、かつ、個々のパラメータはできるだけ単純なものにするということになる。

(4) 明示的なインタフェース

モジュール間に対話がある場合、どちらかまたは両方のモジュールにその事実が記載されなければならない。要するに、引数を使い、大域変数などの共有データは使うなということである。

(5) 情報隠蔽

公にすると宣言されていない限り、モジュールに関する情報はすべて非公開にする。不要な情報を他のモジュールから隠すことにより、変更余波が少なくなり、再利用をしやすくなる。

(6) 開放／閉鎖原則の両立

以下を同時に満たさなければならない

- **モジュールが拡張可能である（開放原則）**
  ソフトウェアは常に修正がある。従って、それに備えておかなければならない。
- **モジュールが他のモジュールから安定して使用できる（閉鎖原則）**
  修正したからといって、従来からあるモジュール同士の関係はできるだけ手を入れずに保持していなければならない。

2.2.4 再利用可能なモジュール構造の要件

再利用に必要なモジュール構造の要件も分かっていて、以下の通りである。

(1) 型の変化に対応できる

例えば、同じモジュールで、整数の表を検索したり文字列の表を検索したりできなければならない。要するに、モジュール呼び出しの引数として型やクラスを渡すことができなければならない。

(2) データ構造と算法の変化に対応できる

データ構造と算法の対を容易に一体のものとして扱うことができなければならない。また、あるモジュールでデータ構造と算法を改良しても、他のモジュールにその影響が及ばないようにできなければならない。

(3) 関連した操作がまとまって定義できる

例えば、表の検索・表の作成・表への挿入・表からの削除などの操作が 1 つのモジュールに定義できることが必要である。

(4) 顧客モジュールが実現方法を知ることなく操作を要求することができる

サービスを受ける顧客モジュールが、サービスの供給者であるモジュールのサービス実現方法を知ることなく、サービスを受けることができなければならない。顧客モジュールが実現方法を知っていると、変更余波が広範囲に及ぶからである。

(5) 実現方法の間の共通部分がうまくまとめらる

異なるモジュール間の共通部分をうまくまとめることができなければならない。

例えば、表検索の場合、順次形式のデータ（順次配列・連結リスト・順次ファイルなど）の扱いは非常に似ている。これらの共通点をうまくまとめられるかが問題になるのだが、従来のサブルーチンではうまくいかなかった[*8]。幸いにも、オブジェクト指向言語は継承機構を使った共通機能の括りだしができ、サブルーチンより強力な共通化が行える。

---

8. C 言語では、関数へのポインターを使えばできなくはなかったのだが、一般にはうまくいかなかった。

2.3 なぜデザインパターンか？

ソフトウェア以外の分野で、専門家達はデザインパターンに相当するものを使っているのだろうか。使っているとすればどれくらい使っているのだろうか。

例えば囲碁の場合、パターンに相当するものとして、定石や手筋あるいは大局観と呼ばれるものがある。このうちの定石だけでも約 2 万個あり、アマチュア初段で大体マスターしていなければならないとされている。

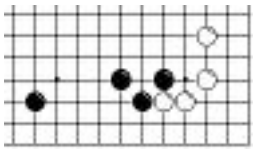

図 1 定石の例

数学の場合、公理や定理あるいは公式がパターンに相当するだろう。数学事典などに載っている数学公式の主なものだけで約 2000 個ある。

ソフトウェアの世界でも、ソフトウェア科学の一分野である算法の基本的なものだけで約 1000 の算法があり、形式仕様記述手法 RAISE の検証公式は約 2000 個ある。また、オブジェクト指向の再利用可能クラス群も、Smalltalk のクラス・ライブラリーや C++ のクラスライブラリーである MacApp や ET++ など、数百から数千のクラスと、数千から数万個の操作を持っている。

つまり、専門家達は数千個から数万個のパターンを使って仕事をしているといってよいだろう。

では、どうしたら専門家になれるのだろう。世界で最も専門家と素人の差が大きいと思われる、囲碁の世界での専門家の養成方法を参考にしてみよう。囲碁の専門家になるステップは以下の通りである。

- **最初にルールと用語を学ぶ**

  例えば、石の名前や打ち方や勝敗の決め方あるいは碁盤の構成など覚える[*9]。

---

9. ソフトウェアの開発現場では、この段階で実際の開発プロジェクトに放り込まれることが多い。たとえて言えば、碁を覚えたての素人が名人戦に出場するようなものだ。

- **定石や手筋を学ぶ**

  定石の手順[*10]や、局面ごとの石の価値[*11]、全局を評価する視点である大局観[*12]などを勉強する。

- **強い人の棋譜を仲間で勉強する**

  定石・手筋・大局観を理解し、覚え、繰り返し適用してみる。1万局ほど打ってアマチュア初段になるのが標準的である。アマチュアは6段が最高だが、それでもプロ10級(プロの見習い）より弱いのが普通である。プロの最高位は9段だからいかに差が大きいかが分かる。

- **定石や手筋を作り出し、実戦で評価し、独自の大局観を磨く**

  この段階がプロである。しかし、プロでも常に勉強会などに出席して、自分の定石や手筋や大局観が通用するか検証していなければ生き残れない。

ソフトウェアの専門家を育てるのも同じ方式が通用するだろう。

- **最初にルールと用語を学ぶ**

例えば、系統的プログラミングを学んで、問題を抽象化してモデル化し、それをプログラムに落としていくという、どのプログラミング言語にも通用するメタな方法を学ぶ。次に、仕様記述言語やプログラミング言語など学んで、実際の仕様を記述し、それを実装することを学ぶ。

- **デザインパターンを学ぶ**

算法やソフトウェア工学方法論あるいは離散数学など、デザインパターンの基礎となる知識を学び、その上で、その応用となるオブジェクト指向技術やデザインパターンを勉強する。

- **優秀なSEの設計(結果と過程)を、仲間で勉強する**

デザインパターンを理解し、覚え、繰り返し適用して評価する。囲碁と異なり、生涯に1万ものシステムを作れる人はいないだろうから、学会やシンポジウムやワー

---

10. ソフトウェア開発では、デザインパターンに相当する。
11. ソフトウェア開発では、モジュール性の原則に相当する。
12. ソフトウェア開発では、外的品質要因に相当する。

クショップなどで、他人の作ったものやアイデアを評価し、自分の作ったものやアイデアを発表して評価してもらうということが一層重要になる。

- **デザインパターンを作り出し、実践して評価する**

デザインパターンやそのアイデアを発表し、世間の批評を受ける。この段階で、ソフトウェア開発のプロと呼べるようになる。それでも、常にシンポジウムなどに参加して自己を磨かなければ、千年以上の歴史を持つ囲碁と異なり、30 年ほどのソフトウェア技術は変化が激しいので、落ちこぼれるのも早い。

以上述べたように、デザインパターンはソフトウェア技術者の養成に役立ち、この養成課程を経た技術者のコミュニケーションを助け[*13]、かつ先人の知恵を再利用する仕掛けといえる。

2.4 デザインパターンとは？

デザインパターンは、ある状況下で開発する際に発生する問題の解決法で、設計の主要項目の静的及び動的構造と協調方式を整理したものである。デザインパターンは、うまくいったアーキテクチャと設計の再利用を促進する。

デザインパターンは、元々、建築家 C.Alexander とその仲間が、建築の設計についてまとめた考え方であった[*14]。彼らは、253 のパターンを使って都市や町や建物を設計すると、かなり快適な生活空間が得られることを「発見[*15]」した。

デザインパターンの多くは、予想される変更に対して、クラス構成の変更やソースコードの変更を、極力無くすか減らすことを意図している。また、各クラスの独立性を高めることで、再利用しやすい構造になるようにも意図されている。

以下に、デザインパターンの例を見ていこう。

13.「これは状態 (State) パターンを使って、オブジェクト内の状態を分離すべきだ」などという会話が成立する。同じことをデザインパターンの概念や用語抜きで説明しようとすると、理解が難しい。

14.『C. アレグザンダー著、パタン・ランゲージ、鹿島出版会、1992 年』参照。アレグザンダーはカリフォルニア大学バークレー本校の建築学科教授で、日本でも、埼玉県の東野高校を設計し高い評価を得た。住宅公団の非人間的設計案に対抗した住民の要望に応えて、名古屋の団地の自然を生かし人間に優しい設計案を作成してもいる。

15. もっとも、東京の下町の町並みや、彼らの研究より前に作られた武蔵野市の明星学園の小学校校舎は、デザインパターンにかなり適合しているので、彼らだけが発見者であるとは言えないだろう。

2.5 デザインパターンの例

ある証券会社の多数のPCに株価情報を提供する下図のようなシステムを考えてみよう。株価は株式市場のコンピュータから提供されるものとする。

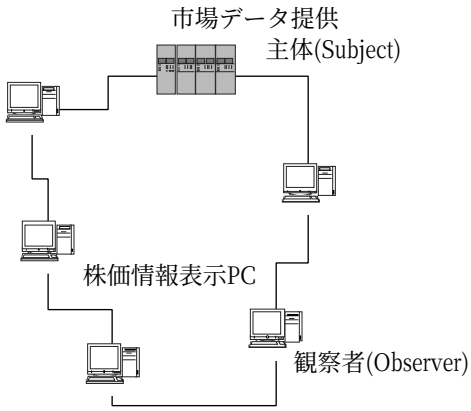

図 2 観察者（Observer）パターンの例

このようなシステムで株価を提供するコンピュータとそれを表示するPCを密に結合したシステムを作ると、以下のような弊害が起こる。

- **変更余波の増大**

  情報の提供側あるいは表示側のプログラムの変更が、他の側のプログラムの変更に結びつく可能性が高い。

- **運用条件の固定化**

  株価表示側の PC の種類の限定や、台数の制限が発生しやすい。

このような状況の時、観察者 (Observer) パターンを使う。観察者パターンのクラス図は以下のようになる[*16]。

16. 以下では、UML のクラス図記法は必要最小限の説明を行う。細かい説明は、第3 章で行う。

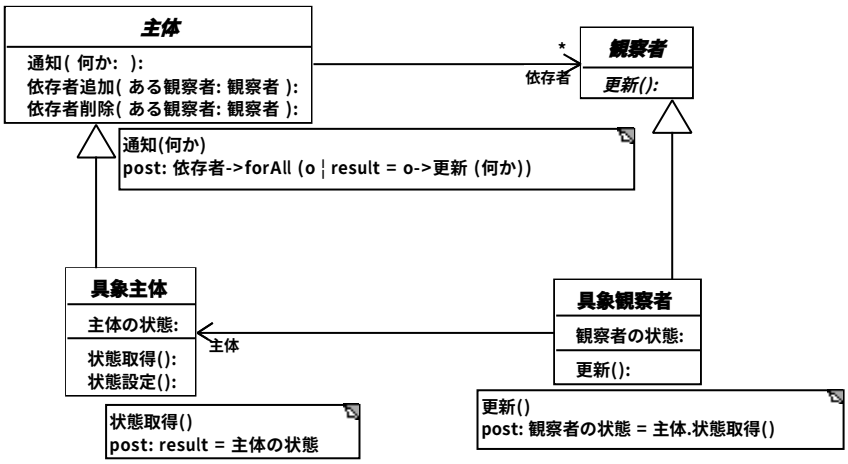

図 3 観察者（Observer）パターンのクラス図

ここで、株価提供コンピュータ上の株価提供オブジェクトを具象主体クラス[*17]のインスタンスとし、株価表示 PC 上の株価表示オブジェクトを具象観察者クラス[*18]のインスタンスとする。ここで、具象主体クラスの属性「主体の状態」が株価である[*19]。

主体側の株価が変わったとき、それを見ている観察者に通知し、自動的に最新状態に更新する。

株価が変わると、具象主体オブジェクトはスーパークラス[*20]である抽象クラス[*21]「主体」の操作である「通知」を自分自身に送信する。通知操作の中で、登録された

17. 実際のアプリケーションでは「株価提供クラス」といった名前が付けられるだろうが、ここでは観察者 (Observer) パターンの役割名をそのまま使った。
18. クラスの記法は、四角形の一番上の欄がクラス名で 2 番目の欄が属性の記述、3 番目の欄が操作の記述である。
19. 右上に折り込みが入った四角形は「注釈」を表す。ここでは、状態取得操作や更新操作の仕様を表している。post は後件の意で、操作実行後の状態を表す OCL の予約語である。「主体 . 状態取得」は、「主体オブジェクトの状態取得操作」を示す OCL の記法である。
20. 白抜きの三角形で継承関係を表す。三角形の頂点の側のクラスがスーパークラスで、三角形の底辺の側のクラスがサブクラスである。
21. 抽象クラスはクラス名が斜体フォントで示される。

[*22] 具象観察者オブジェクトへ更新メッセージが送られる。更新メッセージを受け取った具象観察者オブジェクトは、具象主体オブジェクトへ状態取得メッセージを送り、主体の状態すなわち株価を獲得し、自身の属性「観察者の状態」へ設定する。設定された値を、株価表示PC内の表示用オブジェクト[*23]が、画面上に表示する[*24]。

主体クラスと観察者クラスの間の矢印の付いた線は「関連[*25]」と呼ばれ、この場合、主体オブジェクトの「依存者[*26]」である観察者オブジェクトが複数個ある[*27]ことを示す。

このオブジェクト間のメッセージのやり取りを、以下の順序図に示す。

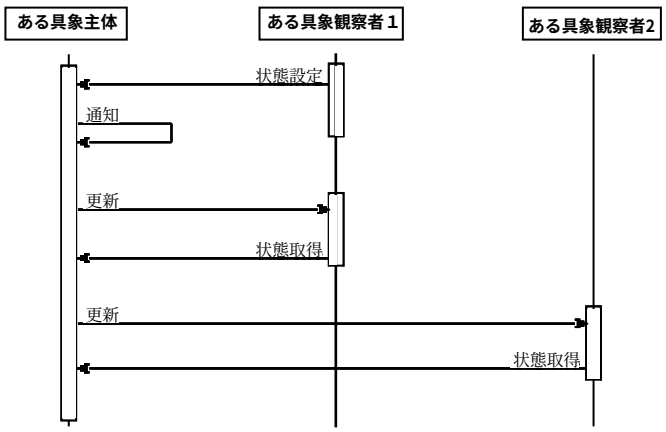

図 4 観察者 (Observer) パターンの順序図

---

22. 依存者追加操作で登録される。依存者追加操作の（）内はパラメータの記述であり、「パラメータ名 : パラメータの型」という形式のパラメータ指定リストが記述される。
23. 実装はいろいろ考えられるので、ここでは、実装よりのクラスはクラス図上に表していない。
24. 以上の説明の一部は、クラス図上の右上が折り返された四角形で表される「注釈」内にOCLで記述しているが、OCLの説明も第3章以下に記述する。
25. 「関連」はメッセージの通り道と考えてよい。矢印付きでなく、関連を線分で表すこともあるが、この場合は両方向にメッセージが飛び交うことを示す。
26. ロール名と呼ばれ、今の場合、観察者の役割を規定している。
27. 星印＊が多重度を表す記号で、「多」であることを示す。＊が無ければ「１」を示す。今の場合、主体オブジェクト「1」に対して、観察者オブジェクト「多」が対応することを示している。

この順序図では、「ある具象観察者１」オブジェクトが状態設定を行うことになっているが、株価設定の場合は、第三者のオブジェクトが「ある具象主体」オブジェクトに状態設定メッセージを送り、株価を設定する方が自然である。

観察者 (Observer) パターンを使うことによって、主体側のオブジェクトと観察者側のオブジェクトの間の抽象的結合が提供される。結果として、以下のような効果が得られる。

- **各観察者が異なるアーキテクチャ・GUI でも問題がない**
- **主体と観察者は独立していて、相手に影響を与えずに変更できる**
- **ソースコードでメッセージの受け手を明確にしておく必要がない**
- **観察者が多くなっても問題が少ない**

観察者 (Observer) パターンは、Smalltalk や C++ や Java の多くのクラスライブラリーで頻繁に使用されているパターンであるので、第 3 章で、さらに詳しい説明を行う。

2.6 デザインパターンの詳細を記述するための仕様記述言語について

オブジェクト指向分析・設計は、クラス図や状態遷移図を書けばよいように思えるが、実際にはそうではない。図はあくまでも概要を示すもので、図がうまく書けたからといって分析や設計が完成したわけではない。そこで、従来の分析・設計技法では、自然言語で詳細仕様を記述していた。

しかし、クラスの制約条件や、操作の定義を自然言語で記述すると、どうしても曖昧さが残り、あまり意味のないモデルができやすい。そこで、形式仕様記述言語と呼ばれる、厳密な仕様を記述できる言語[*28] が開発されてきた。

また、本書では、デザインパターンを使用したオブジェクト指向分析と設計を、なるべく Smalltalk や C++ や Java といった実装用プログラミング言語に依存しないで説明したい。そのために、オブジェクト指向分析・設計のための標準記法UML に 1997 年秋に追加された仕様記述言語 OCL を使うことにした。

---

28. 言語といっても、実行はできず、あくまでも「仕様を記述する」ためのものである。

OCL は、元々、IBM の保険部門で開発されたビジネスモデル作成用の形式仕様記述言語である。従って、制御系システムなどで、「時間」に関する記述が必要な場合には適していないが、その部分の仕様だけは自然言語で書くことで、一応いろいろなシステムの記述に使うことができる。特殊記号を使用せず、文法が簡単で、普通の形式仕様記述言語で要求される数学的バックグラウンドも少なくて済む考慮がされているため、導入しやすい。

OCL[*29] は参照の透過性[*30] が保たれていて副作用がないため、仕様の分析や変換などが容易であるという性質を持っている [*31]。参照の透過性が保たれているため、変数への代入やフロー制御は書けない[*32]。

OCL は、C++ や Java と同じく型のある言語で、文法も似ているので、これらの言語のユーザーが覚えやすい。一方で、型とクラスを同一視していて、オブジェクトの集まりを表す型が Smalltalk の同等のクラスに似ているので、Smalltalk ユーザーなどにも覚えやすい[*33]。

OCL の概要は付録で説明するが、本文中でも、OCL の記法なり機能が初めて出てきた場所で、脚注として説明する。

29.『Object Constraint, Language Specification,Rational Software, 1 September 1997』及び付録 7.2 参照。

30.「式や記述の一部をそれと透過なものに書き換えても、全体の評価値や意味が変わらないという性質」(『情報科学事典、岩波書店、1990 年』より)。参照の透明性ともいう。信頼性の高いプログラムを書くためには必須の性質である。

31. 参照の透過性は、形式仕様記述言語や関数型言語で実現されているものが多い。オブジェクト指向言語や、構造化言語では、この性質はまだ実装されていない（実装できない設計であるとも言える)。

32. デザインパターンの中には、参照の透過性のない言語のためのパターンもあるので、その部分の記述のため、本書では RSL（Raise Specification Language）という形式仕様記述言語から代入やフロー制御のための構文を借りてきて OCL の仕様に追加している。追加部分は、付録に記述した。RSL については、『The RAISE Language Group, The RAISE Specification Language, Prentice Hall, 1992』参照のこと。

33. そもそも、OCL で記述されたシステムのかなりは Smalltalk で実装された。