連載解説

パターン

ーソフトウェア開発ノウハウの再利用

第3回　デザインパターンの適用

ー応用編：ドキュメンテーション，フレームワーク，リファクタリング

細谷竜一　　吉田和樹

（株）東芝　SI技術開発センター

前回は，「会議室予約エンジン試験プログラム」を題材に，2つのデザインパターンの適用過程を詳しく見た．今回は，さらにもう1つのパターンの適用を見ていく．そして，デザインパターンとドキュメンテーション，フレームワーク，リファクタリングの関係について議論する．

秘密工作員 ー*Decorator*

前回，*Abstract Factory*と*Singleton*の2つのパターン[1]の適用によって，会議室予約エンジン試験プログラムの設計に，さまざまな試験項目の追加を許す柔軟性を与えることに成功した．しかし，個々の試験項目について見れば，試験プログラムの外見上の振舞いは（試験項目をコマンドラインで指定することを除けば）まったく同じである．このように，その振舞いを変えずにプログラムの設計を改良し，再利用性や柔軟性を高めることをリファクタリング（refactoring）[2]という．これについては後節で触れる．

今度は，試験プログラムに合否判定機能を付け加えることで，その振舞いの拡張を試みよう．とはいっても，1から設計をし直すのではなく，すでに作成した試験プログラムの変更を極力抑えながら機能を追加することにする．

話を簡単にするために，合否判定はすべての試験項目に共通の次の基準に従って行うものとする：

合否判定基準

試験項目が定める一連の会議室予約エンジンの呼び出しが，次のいずれかの結果に終わった場合は，その試験項目の判定は不合格となる：

1. doYoyakuが呼ばれた直後に，予約を行ったはずの日時に，指定された会議室の予約が空いている．つまり，先約もなく，今回の予約も登録されてもいない場合
2. checkYoyakuが，実際の予約状況と矛盾する答えを返した場合
3. cancelYoyakuの直後，キャンセルしたはずの予約が有効な状態で残っている場合

すべての呼び出しが，上に該当することなく完了した場合には，試験項目の判定は合格となる．

この合否判定基準は，前後の呼び出しの文脈にかかわらず毎回の呼び出しの結果によって不合格の条件を判断できる．

最初に約束したように，既存のコードへの変更を最小限に抑えた解決法を探ろう．「デザインパターン」[1]の表紙見返しの一覧をあたると，次のパターンが目につく：

Decorator ーオブジェクトに責任を動的に追加する．*Decorator*パターンは，サブクラス化よりも柔軟な機能拡張方法を提供する．

さらに，「適用可能性」節では，次の一項が目を引く：

個々のオブジェクトに責任を動的，かつ透明に（すなわち，他のオブジェクトには影響を与えないように）追加する場合．

これを次のように読み替えてみよう：

個々のHaichiKeitaiオブジェクトに合否判定機能を動的，かつ透明に（すなわち，試験プログラムの他のオブジェクトには影響を与えないように）追加する場合．

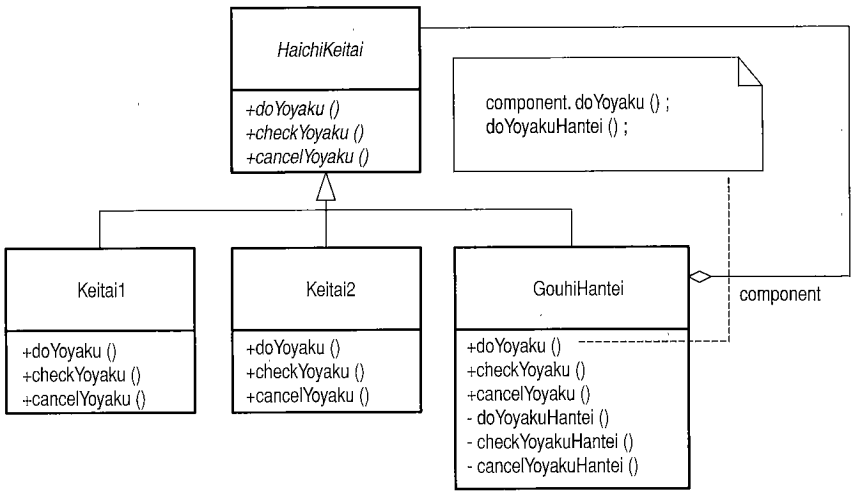

図-1　HaichiKeitaiへのDecoratorの追加

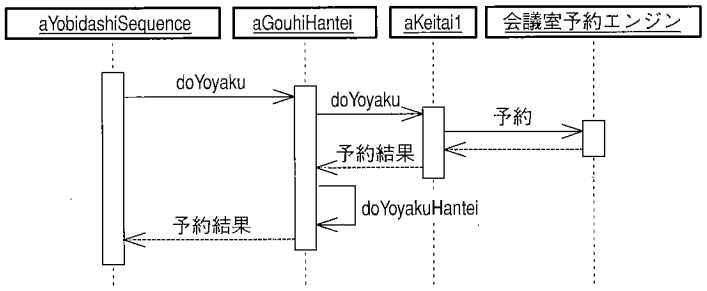

図-2　合否判定付きのdoYoyaku呼び出し

Decorator追加前はaGouhiHanteiがなく，aYobidashiSequenceは直接aKeitai1のdoYoyakuを呼び出していた（UMLシーケンス図で表す）．

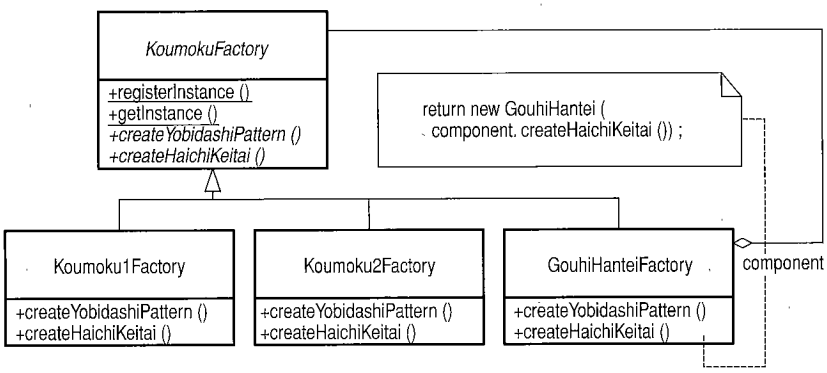

図-3　KoumokuFactoryへのDecoratorの追加

*Decorator*パターンのアイデアは，あるオブジェクトがもともと備えていなかった機能や振舞いを，他のオブジェクトに気づかれずに（つまりそれらを変更せずに），あとから付け加えるというものである．さっそくこのパターンの適用を試みてみよう．**図-1**は，*Decorator*の構造を，我々の試験プログラムに当てはめたものである☆1．GouhiHanteiオブジェクトは，本物のHaichiKeitaiオブジェクト（Keitai1かKeitai2のインスタンス）を「隠し持って」いる．GouhiHanteiオブジェクト自身も，HaichiKeitaiのサブクラスなので，利用する側（つまりYobidashiSequenceオブジェクト）は，それが何かの配置形態に対応するオブジェクトだとしか思っていない．しかし実際には，GouhiHanteiオブジェクトは，会議室予約エンジンの呼び出しを片っ端から「横取り」し，自分の仕事である合否判定を行う機会を得る．しかし，GouhiHanteiオブジェクトは，呼び出しを横取りしたことを周囲に気づかれないように，その呼び出しをきちんと本物のHaichiKeitaiオブジェクトにまわすのである．たとえば，GouhiHanteiのdoYoyakuは，本物のHaichiKeitaiオブジェクト（Keitai1やKeitai2のインスタンス）を呼び出した直後に，自らのdoYoyakuHanteiを呼び出し，直前のdoYoyaku呼び出しが正しく処理されたかを検査し合否の判定を行う（**図-2**）．checkYoyakuやcancelYoyakuについても同様の手法で合否判定を行う．話を簡単にするために，～YoyakuHanteiメソッドは，不合格を判定すると，その瞬間に自らコンソール画面にその旨を表示することにする．

これまでのところ，既存のコードには一切変更を加えていない．ただ単にGouhiHanteiクラスをHaichiKeitaiのサブクラスとして追加しただけである．では，一体誰が，このGouhiHanteiのインスタンスを作るのだろうか．その答えは，やはり*Decorator*パターンにある．今度は，KoumokuFactoryに*Decorator*を追加し，GouhiHantei付きのHaichiKeitaiオブジェクトを生成する細工を施す．

図-3は，*Decorator*型KoumokuFactoryを追加した様子を示している．GouhiHanteiFactoryは，createHaichiKeitaiメソッドを横取りして，本物のKoumokuFactoryオブジェクトを使ってHaichiKeitaiオブジェクトを生成し，それをGouhiHanteiのコンストラクタに渡している．これにより，適切なHaichiKeitaiオブジェクトを「隠し持つ」GouhiHanteiオブジェクトが作られる．このGouhiHanteiFactoryオブジェクトを*Singleton*として登録しさえすれば，試験プログラムは，合否判定

☆1　*ConcreteDecorator*クラスが1つなので，*Decorator*クラスは省いた（文献1）の「実装」節2項）．複数ある場合の設計は，読者への課題として残しておく．

機能が働いていることなど知らずに，試験項目の処理を実行するだろう．

この例に見られるように，*Decorator* パターンによって既存の設計に機能を追加する場合，*Decorator* によって修飾されるオブジェクトを生成するオブジェクトもまた *Decorator* によって修飾される...といった連鎖がしばしば起こる．しかしこれがうまくいくのは，設計が十分に抽象化されて柔軟性を備えている場合だけだろう．我々の試験プログラムの例では，あらかじめ *Abstract Factory* を適用していなければ，*Decorator* による合否判定機能の追加は簡単ではなくなる．

最後の仕掛けは，GouhiHanteiFactory オブジェクトを *Singleton* として登録することである．これには，既存コードに数行分の手直しが必要である．**リスト 1** は，コマンドラインの指定で，合否判定機能の on/off を切り替えられるように変更した Main::main である．

パターンを身につける

ここまで，「会議室予約エンジン試験プログラム」を題材に，*Abstract Factory*，*Singleton* および *Decorator* の適用を通じたデザインパターン活用の具体例を見てきた．3つのパターンが登場したに過ぎないが，いずれも実用的で，プログラム設計では頻繁に用いられるものである．

「デザインパターン」[1] は，初めのうちは，どのパターンを使えばよいのか分かりにくい．しかし，試行錯誤を経るうちに，プログラムの設計にまつわる問題の捉え方が身に付き，それに応えるパターンをすばやく選び出せるようになる．Ralph E. Johnson がよく念を押して言うように，パターンは「試してみなければ，本当には理解できない」[3] のである．

パターンを使うにあたって注意しなければならないことは，どのパターンにも，それを使うことによる副作用があるという事実である．そのため，たまたま使えそうなパターンを見つけたからといって，やみくもに適用してはならない．パターンの適用にあたっては，まず解決すべき問題の発見があって，次に最適なパターンを選び出すという順番を守る必要がある ☆2．

ここまでで見てきた，デザインパターンの適用の手順をまとめると，次のようになる：

1. **解決すべき設計上の問題を認識する．**「デザインパターン」が提示するパターンの多くは，プログラムの設計に柔軟性と再利用性が欠けている場合の解を提供するためのものである．
2. **「目的」節をスキャンし，候補となるパターンを拾い出す．**この段階では，必ずしも1つのパターンに絞り込めない．
3. **候補のパターンの「動機」節と「適用可能性」節を読み，使うべきパターンを決める．**「動機」節で，候補のパターンが自分の直面している問題を取り扱っているかどうかを判断し，「適用可能性」節で，それが十分に効果を発揮するかどうかを見定める．
4. **「構造」節が示すクラス構造を自分の設計に当てはめ，その効果を「結果」節と見比べて，選んだパターンが正しいものかどうかを見極める．**クラス図だけで考えにくい場合は，「サンプルコード」節を研究しよう．ここで大事なのは，そのパターンを組み込んだプログラムが単に機能することを確認するのではなく（間違ったパターンを組み込んでも，あるいは正しいパターンを間違って組み込んでも，プログラムはたいてい機能する），プログラムに対する要求の変化に対して，設計がどのように応じるかを，クラス図あるいはコードの変化として具体的に想像することである．それには「結果」節を参考にしながら，そのパターンが自分の設計にもたらす効果と副作用のバランスを見極めるとよい．
5. **選んだパターンを設計に適用し，実装する．**解決すべき問題を正しく理解し，適切なデザインパターンを選んでいれば，その適用は容易である．しかし一般にパターンは，機械的に適用できるものではない．たとえば，「構造」節で示されるクラス名は，適用するたびにその設計のために変えられなければならない．
6. **「関連するパターン」節や「デザインパターン間の関連」図（裏表紙の見返し）から，次に適用すべきパターンの候補を拾い出し，2に戻る．**これは，設計の中で，すでにパターンを適用した個所に関連した未解決の問題が残っている場合である．1つのパターンの適用が，次のパターンの適用の呼び水となることは，良いパターン集ではよく起こる．たとえば *Abstract Factory* は，しばしば *Pro-*

```
public static void main(String argv[]) {
  int n = Integer.parseInt(argv[0]);
  Client clients[] = new Client[n];
  KoumokuFactory factory;

  try {
   // 指定のKoumokuFactory インスタンスを生成
   Class factoryClass = Class.forName(argv[1]);
   factory = factoryClass.newInstance();
  } catch (Exception e) {
   // 指定された名前のクラスは存在しない...
  }
  if (argv[2].equals("on")) {
    // 合否判定機能 on
    factory = new GouhiHanteiFactory(factory);
  }
  KoumokuFactory.registerInstance(factory);
        // インスタンスを Singleton として登録
  //...
}
```

リスト 1　合否判定機能の切り替え付きの main

☆2　情報サービス産業協会・分散オブジェクト研究会主催特別講演会「デザインパターンを適用して良いソフトウェアを作る方法」（99/12/8）で Ralph E. Johnson が述べたこと．

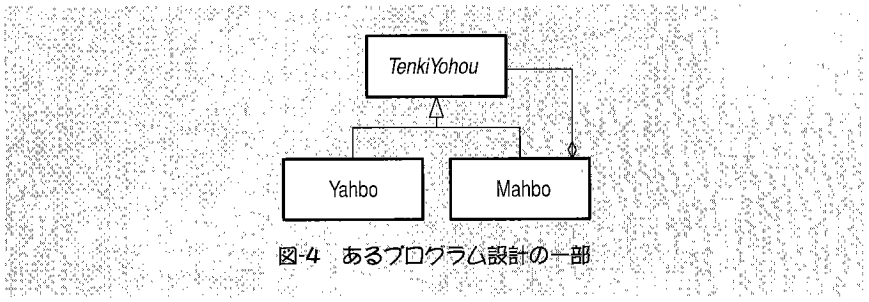

図-4　あるプログラム設計の一部

totype, *Singleton* あるいは *Factory Method* と組み合わせて適用される．また「デザインパターン」[1]は，*Composite* と組み合わせて適用できるパターンの多いことが特徴である．

慣れてくれば，こうした「デザインパターン適用のパターン」を意識的に追うことなしに，自然に問題を見極め，適当なパターンを選び出し，適用することができるようになる．

解説は省いたが，今回取り上げた試験プログラムの中では，他にもいくつかのデザインパターンがすでに用いられている．KoumokuFactory では *Factory Method* が使われている．HaichiKeitai は，試験プログラム側が会議室予約エンジンの呼び出しに使うインタフェースを，実際の会議室予約エンジンのインタフェースの呼び出しに変換する *Adapter* である．YobidashiSequence は *Command* である．そして，Main は *Facade* である．

試験プログラムの質を高めるために有効と思われるパターンもいくつかある．より高度な合否判定を行うために，システムリソース監視や，各クライアントプロセス内のローカルな合否判定を行うオブジェクトを，*Observer* パターンを使って試験項目の合否判定オブジェクトと結ぶことができる．試験項目が使用する予約のためのデータ生成アルゴリズムにはさまざまなものがあるが，これらは *Strategy* 化して，KoumokuFactory に生成させることができる．会議室予約エンジンがリモートマシンに置かれている場合，それはローカル空間内では *Proxy* を使って参照される．ぜひ研究してみて欲しい．

デザインパターンとドキュメンテーション

設計に用いられたデザインパターンを設計図に記載することにより，設計の意味や，保守・拡張の際にどこに手を加えればよいのかといった情報を残すことができる．

通常，プログラムのコードを見て，その設計を完全に理解することは不可能か，あるいは非常に手間がかかる．そのためプログラマはしばしば，ソースコードにコメントを埋め込んだり，変数名やクラス名などを意味のあるものにすることで，コードから欠落しがちな設計の情報を残そうとする．大きなプロジェクトでは，プログラムの設計情報は，設計書としてクラス図やシーケンス図の形で残される．こうしたドキュメンテーションの努力により，プログラムの構造や振舞いは理解しやすいものとなる．

しかし，プログラムを拡張したり，その一部を再利用して新たなプログラムを作成する場合，プログラムの構造や振舞いに関する情報だけでは不十分である．**図-4** を，あるプログラムの設計図の一部で，1つのデザインパターンを適用した個所だとしよう．果たしてそれはどのデザインパターンだろうか．図に示されるクラス構造に着目して，これと一致する構造を持つパターンを文献1) から拾い出してみても，徒労に終わるだろう．なぜなら，一致する構造を持つパターンは，1つどころか3つもあるからである☆3！　結局この図から，使われたパターンを特定することはできない．この例に見るように，クラス図は，設計者がその設計に込めた意図を完全に残せるとは限らない．しかし図-4にひとこと「*Decorator*」と書かれてさえいたら，それが *Decorator* パターンであることを誰が疑うだろうか．そして，*Decorator* によってオブジェクトの振舞いの拡張を容易にしようとした設計者の意図が，いかに明白になることだろうか．

「デザインパターン」[1]に含まれる23個のパターンは，特に再利用を意識したプログラム設計の，再利用のための仕掛けや方法を表現する語彙である．この語彙を共有する者の間では，設計に関する意志疎通は格段に早く，的確なものになる．クラス図を指し示して「そこは *Decorator* だ」というだけで，あなたは振舞いの拡張が可能なオブジェクトがどれで，どのように振舞いを追加すべきかについての主要な情報を伝えたことになる．特に，中・大規模の開発プロジェクトでは，そこで使われるパターンを一覧にまとめ，メンバ間の共通語彙を作ることが意志疎通を高めるうえで効果的である[3]．開発プロジェクトとパターンの関係については，この連載の第5回で詳しく述べる．

クラス図にデザインパターンを記すには，その名前，適用範囲，そして適用範囲内のクラスの構成要素名を書くとよい．これには決まった書き方はないが，たとえば**図-5** のようなものになる☆4．ここで注意しなくてはならないのは，設計のすべてがパターンで埋め尽くされる必要はないし，そうしたことは通常起こらないということである．ソフトウェア技術は，既知のパターンの組合せだけでモノを作れるほどには発達していないし，たぶん永久にそうはならないだろう．

☆3　*Composite*, *Decorator* および *Interpretor*.
☆4　UML 1.3は，媒介変数表示協調図によるデザインパターンの表記を試みている．

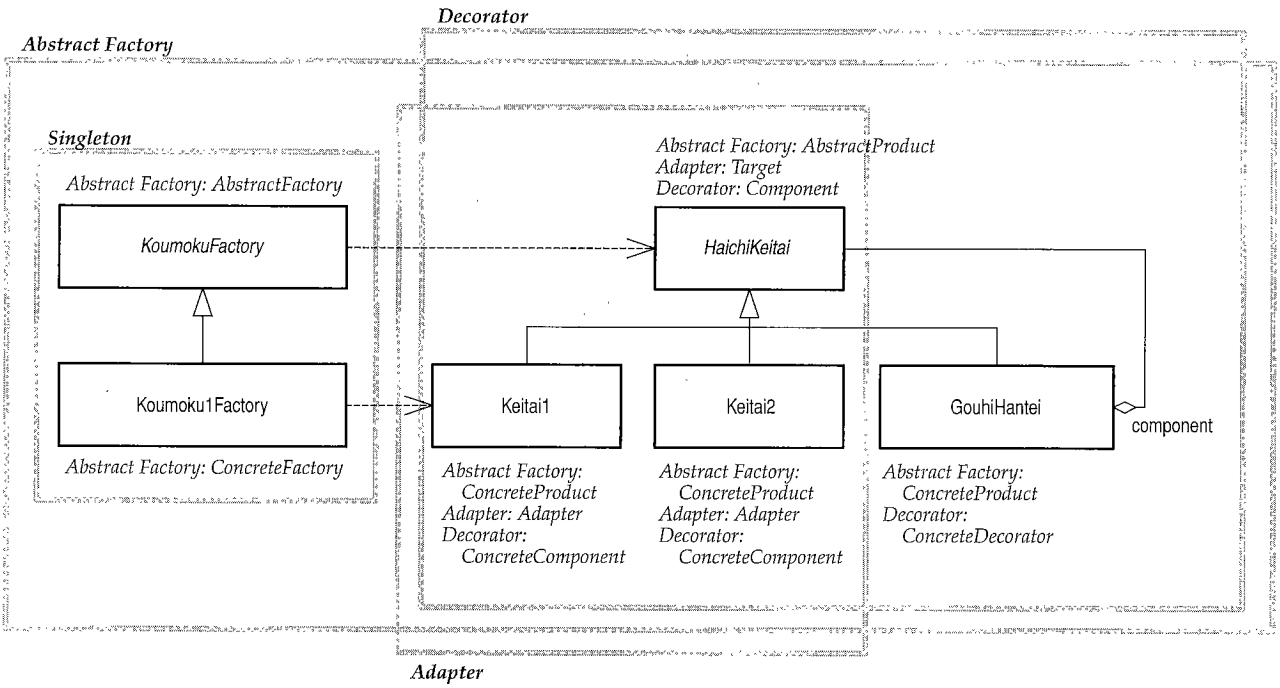

図-5　クラス図でのデザインパターン表示例

パターン適用の範囲とともに，パターン名：構成要素名を各クラスに付記．文脈から明らかな場合は適宜省いてよい．

1つのデザインパターンを他と違うものにするのは，その意図あるいは目的である．文献1）では，各デザインパターンの「目的」節で，そのパターンが解こうとする設計上の問題と解決のための方針を簡潔に述ている．2つのパターン（たとえば*Strategy*と*Template Method*）が解こうとする問題は同じであることもあるが，その場合，解決のための方針は違ったものになる．

プログラムの設計を理解するためには，その各個所がどのような意図をもってそうなっているのかを理解することが必要である．そうしなければ，その個所が解こうとする問題と，解決のための仕組みが理解できないからである．しかしすでに見たように，あるデザインパターンのクラス構造は，他のデザインパターンのそれと必ずしも区別できない．そのため与えられたクラス構造を見て，そのデザインパターン（そこに適用されていたとして）を言い当てるのは容易ではないし，したがってその意図を読み取ることも難しい．

このように，設計の各個所の背景にある意図は，設計を理解するうえで不可欠な情報であるにもかかわらず，それは設計書に十分に記載されないことが多い．その原因は，設計の各個所について，設計者の意図を文章などでいちいち記述することが事実上不可能なことにある．

「デザインパターン」[1]はこの問題に対して，2つの手法をもって改善策を与える．1つは，各パターンに名前を与えたことであり，もう1つは「目的」節の簡潔な記述によって，そのパターンの意図をいつでも素早く確認できるようにしたことである．こうして，設計中でデザインパターンを適用した個所について，その名前を設計書に記しておくことで，設計者の意図を効率よく書きとめることが可能となる．

1つのデザインパターンの適用が適当かどうかの判断は，「構造」節が示すクラス構造を，寸分違わず自分の設計の中に再現できるかどうかではない．むしろそれは設計者が直面している問題と，そのパターンが解こうとする問題の比較照合であり，「目的」節に謳われると同じ目的でそのパターンを使用できるかどうかである．問題と目的が適用の文脈と一致する時にのみ，パターンは即利用可能で，自分の設計に合わせてカスタマイズ可能な良質の解を提供する☆5,6．

デザインパターンとフレームワーク

狭義のフレームワークをありのままに表現すると，それは再利用されるmainプログラムである[3]．フレームワークを用いないコードの再利用では，開発者はライブラリが提供するコンポーネントを再利用して，それを呼び出すmainプログラムはそのつど書き起こす☆7．しかしフレーム

☆5　というよりも，むしろ適用のたびにカスタマイズしなくてはならない．デザインパターンが自動化のためのものではなく，設計者の知的活動を助ける道具[3]といわれる端的な理由はここにある．

☆6　これは，もう1つの重要な結果を暗示している．1つの問題に対しては，潜在的には多数の解決法（パターン）が存在するため，放っておけば異なる設計者は異なる解決法を採り，全体としての質の統一ができなくなる．しかし，1つのパターン集を採用することで設計上の選択肢を減らし，工程の効率化と品質向上がもたらされる．こうした取組みは，伝統的な生産工学では標準化と呼ばれ，近代の工業における最も重要な考え方の1つである．

☆7　Mainプログラムといっているのはたとえである．通常，フレームワークに文字通りのmain関数が含まれているわけではない．

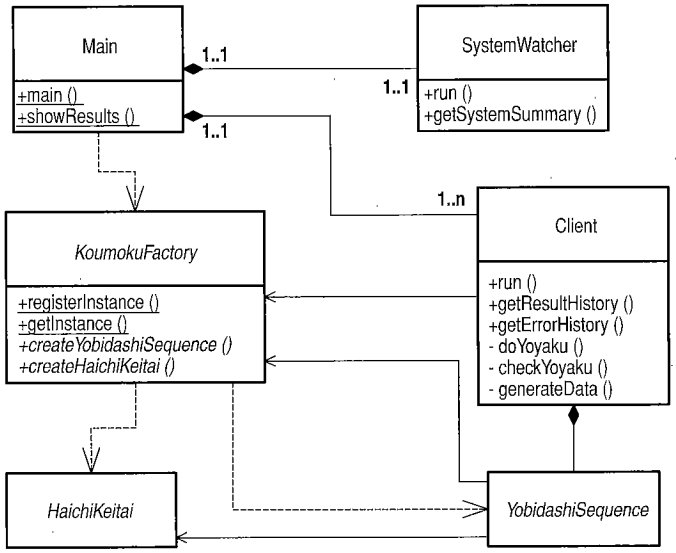

図-6　試験フレームワークの構造

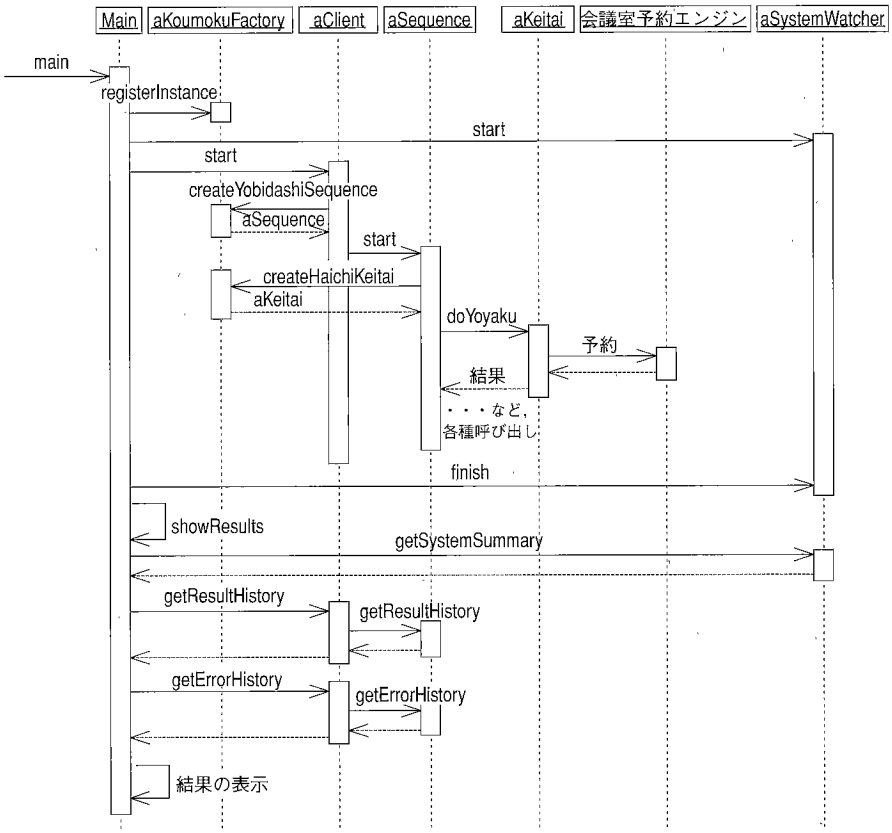

図-7　試験フレームワークの制御のシーケンス図

ワークによる再利用では，この関係が逆になる☆8．開発者はあらかじめ提供されるmainプログラムが呼び出すコンポーネントを書くのである．

前回の記事で作成した会議室予約エンジン試験プログラムは，小規模ながら典型的なフレームワークである．このプログラムのフレームワーク部分は，KoumokuFactory，Yobidashi SequenceおよびHaichiKeitaiの各サブクラスを除いたすべてである（**図-6**）．このフレームワークを利用した試験プログラムの作成では，開発者はプログラム全体の設計と，クライアントプロセスの起動から会議室予約エンジンの呼び出し処理といった一連の流れを実装するコードを再利用する．そして，特定の試験項目や呼び出しシーケンス，そして会議室予約エンジンの配置形態といったコンポーネントを，それぞれに対応する抽象クラス（KoumokuFactory，Yobidashi Sequence，HaichiKeitai）のサブクラスとして実装し，プログラムに追加する．実行時には，これらのコンポーネントは，フレームワークが制御する処理の流れに従って適切なタイミングで呼び出され，特定の試験項目が実行される（**図-7**）．

我々の試験プログラムのように，1つの設計に「デザインパターン」[1]を繰り返し適用すると，やがてそれがフレームワークのように見えてくることがある．実際，このパターン集に収められている23個のパターンは，数々の既存のフレームワークから共通する設計テクニックを抽出した結果[3]なのだから，それは自然な結果といえる☆9．これらのパターンの多くは，設計が持つクラス構造に新たな抽象クラスを導入することで，要求に応じて変化する部分（可変点）とそうでない部分とを分離する．このような作業を繰り返すと多様な状況において利用可能なクラス構造，つまりフレームワークが得られる．しかし，フレームワークの開発は教科書通りにはいかない．開発の出発点となる設計がどうあるべきか，どこに可変点が必要で，どこには不要なのかといった見極めを一度で済ますことはできないため，フレームワークの適用対象に対する理解と試行錯誤，そして改良のための長い過程を経る覚悟が必要である[3]．

デザインパターンとリファクタリング

既存のプログラム設計をより単純で理解しやすいものにしたり再利用性を高めることなどを目的として，

☆8　この現象はinversion of control（制御の逆転）と呼ばれる[3]．

☆9　すべてのデザインパターンが設計の柔軟性や再利用性を高めるためのものとは限らない．文献1）の中でさえも，少なくとも1つは再利用性以外の目的で使用されるパターンが含まれている（*Flyweight*の「目的」節を参照）．

たとえば複数のクラス間で重複しているコードを共通のクラスにまとめるなどといった改良を加えることを，リファクタリング（refactoring）☆10と呼ぶ[2]☆11．特にプログラムの振舞いや外部とのインタフェースを変えないリファクタリングを意味保存リファクタリング（semantics-preserving refactoring）と呼ぶ[2]．

前回の記事では，特定の試験項目にしか対応しない会議室予約エンジン試験プログラムの設計から始まり，段階的にデザインパターンを適用しながら，新たな試験項目の追加を許す設計を導いた．最初の試験項目を実施することだけを考えた場合，デザインパターンの適用の前後で得られるプログラムの機能や振舞いには変化はないので，これは意味保存リファクタリングである．振舞いに変化はなくても，たとえば*Abstract Factory*を適用した際には，設計を1つの試験項目に特化した部分と，すべての試験項目に共通した部分とに分けることに成功した（前回記事参照）．このようなリファクタリングを，パターンに基づいたリファクタリング（pattern-based refactoring）と呼ぶ．

リファクタリングには，たとえば「長いメソッドの中身を複数のメソッドに分割する」といったクラス内で閉じた小さな整形[2]から，複数のクラス構成を一度に変更する大きな整形まである．デザインパターンに基づいたリファクタリングは後者の例で，リファクタリングの中でも最も積極的な部類に属する．積極的であるということは，それだけ設計の多くの部分に手を入れることを意味する．それは果たして奨励されるべきことなのだろうか．むしろ，設計の最初の段階でデザインパターンを積極的に採り入れ，後の設計変更を未然に防ぐべきではないのだろうか．答えは否である．すでに議論したように，デザインパターンは設計のより上位の情報であるにもかかわらず，その適用は設計中に解決すべき問題の発見後，すなわち設計工程の後半で始まる傾向がある☆2．現在，設計の過程で，問題の発生以前に適用すべきパターンを教える「パターン言語」[4]の域に達するソフトウェアのパターン集はないといわれる[3]．設計の前段階でやみくもにデザインパターンの導入を図ることは，クラス数を増大させ，設計を必要以上に複雑なものにする．このような状況で現在得られるソフトウェア設計の最良の知恵を有効活用するためには，むしろリファクタリングを開発工程の主要な部分の1つと捉え，正面から取り組むべきであろう[5]．言い換えれば，最初から完璧な設計を得ようとするのではなく，むしろ時々設計を振り返り，デザインパターン適用の余地があるか等を考え，必要に応じてリファクタリングする態度が大切である☆12．

パターンは思考を助ける道具である

2回にわたり，デザインパターンの適用過程を詳しく見てきた．そして，デザインパターンがプログラム開発にもたらすインパクトについて議論した．もしもあなたがプログラム設計者で，この記事を読み，近いうちにデザインパターンの適用を試みようと考えていただければ，この記事は成功である．もしもあなたがプロジェクト管理者で，この記事を読み，現在のあるいは次回の開発プロジェクトでデザインパターンの導入を図ろうと考えていただければ，やはりこの記事は成功である．

デザインパターン1つ1つをとってみれば，当たり前のことをいっているようにしか見えないかもしれない．実際，よく知られていたり，あるいは誰が思いついてもおかしくないようなテクニックだからパターンと呼ばれるのである．しかし，良いパターン集は，単なる既知のテクニックの寄せ集めではなく，開発の1つの工程において直面する問題を分析し解決する一貫した視点を与えるパターンの集まりである．そこには，個々の当たり前のテクニック以上のものが含まれる．

ソフトウェアにおけるパターンの取組みは始まったばかりで，課題も多い．パターンはまだ当分の間不完全なものだが，自ら考え，主体性を持って利用すれば，開発者の思考を助ける良い道具となる．

参考文献

1) Gamma, E., Helm, R., Johnson, R. and Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, Reading, Massachusetts (1995).（本位田真一，吉田和樹監訳：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，ソフトバンク，東京（1999））．
2) Fowler, M., Beck, K., Brant, J., Opdyke, W. and Roberts, D.: Refactoring: Improving the Design of Existing Code, Addison-Wesley, Reading, Massachusetts (1999).
3) Johnson, R. E., 中村宏明，中山裕子，吉田和樹：パターンとフレームワーク，共立出版，東京（1999）．
4) Alexander, C., Ishikawa, S., Silverstein, M., Jacobson, M., Fiksdahl-King, I. and Angel, S.: A Pattern Language: Towns, Buildings, Construction, Oxford University Press, New York (1977).
5) Beck, K : Extreme Programming Explained: Embrace Change, Addison-Wesley, Reading, Massachusetts (1999).

（平成12年2月1日受付）

☆10 「再分解」などと訳せるだろうか．良い日本語訳があったら教えていただきたい．
☆11 狭義にはパターン化されたプログラム設計の改良テクニックの集合を指すが，オブジェクト指向プログラミングの文脈では，単に既存の設計やコードに手を入れて改良する行為を指すことが多い．
☆12 現実には，実装工程に入って初めて改良可能な設計個所を見出すことが多い．そのため，分析－設計－実装－試験の各工程を後戻りすることなしに進む伝統的な開発モデルにリファクタリングを採り入れることは難しい．近年では，工程の反復を前提とした開発モデルが盛んに提唱されている．

本連載の第4回では，ソフトウェア開発における分析工程に目を向け，2つの分析パターン集－Fowlerの「アナリシスパターン」とHayの「Data Model Patterns」－を比較し，両者に共通する分析テクニックを紹介する．