連載解説

パターン

－ソフトウェア開発ノウハウの再利用

第2回　デザインパターンの適用

－基本編：プログラム設計のヒント

細谷竜一　　吉田和樹
（株）東芝　SI技術開発センター

「デザインパターン」」を読んだことはあるが，ピンとこない，といった声がよく聞かれる．本連載の第2回では，設計に関するパターン集として，すでに広く知られている「デザインパターン」の活用の一般的なシナリオを紹介する．ここでは，「会議室予約エンジン試験プログラム」を例に，いくつかのデザインパターンを使ってさまざまな試験方法に対応できる設計を導く過程を見ていく．読者には，「デザインパターン」がプログラム設計にまつわる問題をどのような観点で捉え，解決するのかを理解し，ぜひ次回のプログラム設計の機会にデザインパターンの適用を試みていただきたい．なお，ここで示されるコードは，すべてJavaで書かれている．

最初のデザイン

今回の記事では，会議室予約システムの中核部である「会議室予約エンジン」の動作を試験する試験プログラムの設計を通じて，デザインパターン適用の具体的なシナリオと，それがもたらす効果を見ていく．

次のような状況を考えてみよう：

あなたはプログラマだ．ある日上司がやってきて，あなたに今日の仕事を与えた．上司の指示は，今，作りかけの会議室予約エンジンの動作を確認するプログラムを書けというものだ．どうやら，複数のクライアントから会議室予約エンジンを何回か繰り返して呼び出し，正常に動くことを見せられればよいらしい．動作中のシステムリソースの消費の記録，呼び出しの結果やエラー発生の記録もとれるとよい．

そこで，あなたは**図-1**および**リスト1**に示されるプログラムを書いた．ユーザがコマンドラインからプログラムを起動すると，最初にmain（Mainクラス）が呼び出される．mainはstartを呼ぶことで，所定の数のクライアントプロセスを起動するが，これはClientクラスとして実装されたスレッドとして擬似的に作り出される．1つのクライアントプロセスは，1人の予約業務担当者が使うクライアントアプリケーションのプロセスに対応する．Clientクラスは，会議室予約エンジンとの通信手段を持っており，runメソッドによって起動されると，会議室予約エンジンの持つ機能を所定の回数繰り返して呼び出しながら，発生するエラーや会議室予約エンジンが返す呼び出しの結果を記録する．会議室予約エンジンは今のところ2つの機能を持っている：会議室の予約機能（doYoyaku）と，予約の有無の確認機能（checkYoyaku）である．Clientは，会議室Aを予約し，予約ができたことを確認するように，これら2つの機能を交互に10回ずつ呼び出す．その過程で呼び出し結果や発生したエラーを記録する．

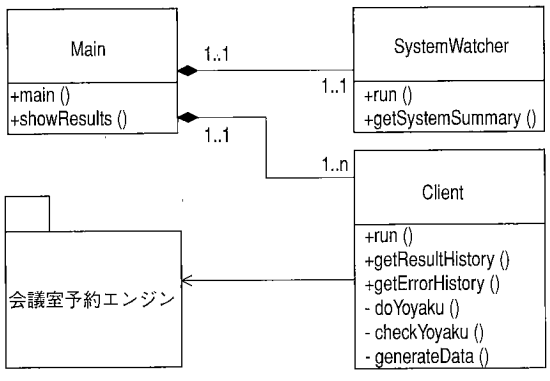

図-1　最初の試験プログラムの構造

```
public class Main {
  public static void main(String argv[]) {
    int n = Integer.parseInt(argv[0]);
    Client clients[] = new Client[n];
    // システムリソース監視プロセスの開始
    SystemWatcher sw = new SystemWatcher();
    sw.start(); // Java ではこの後 run() が呼び出される
    for (int i = 0; i < n; i++) {
      // n 個のクライアントプロセスを起動する
      clients[i] = new Client(i);
      clients[i].start(); // この後 run() が呼び出される
    }
    for (int i = 0; i < n; i++) {
      // 各クライアントプロセスの終了を待つ
      clients[i].join();
    }
    sw.finish();  // システム監視プロセスの停止
    showResults(sw, clients); // 結果の表示
  }
  public static void showResults(SystemWatcher sw,
                                 Client clients[] ) {
    // システムリソース消費状況を表示する...
    // 各クライアントプロセスでの結果の記録を表示...
  }
}
public class SystemWatcher extends Thread {
  public void run() {
    // システムリソース消費状況を監視し，記録する...
  }
  public void finish() {
    // 監視を終了する...
  }
  public SystemSummary getSystemSummary() {
    // システムリソース消費状況の記録を返す...
  }
}
public class Client extends Thread {
  public Client(int clientID) {
    // 会議室予約エンジンと接続する...
  }
  public void run() {
    // 予約，予約の確認　を10回繰り返す
    for (int i = 0; i < 10; i++) {
      try {
        int yoyakuBango = doYoyaku(generateData(i+1));
        boolean result = checkYoyaku(yoyakuBango);
        // 呼び出し結果を記録する...
      } catch (Exception e) {
        // エラー発生．記録をとる...
      }
    }
  }
  public ResultHistory getResultHistory() {
    // 呼び出し結果を返す...
  }
  public ErrorHistory getErrorHistory() {
    // エラー記録を返す...
  }
  private int doYoyaku(YoyakuTarget target) {
    // 会議室予約エンジンの予約機能を呼ぶ...
  }
  private boolean checkYoyaku(int yoyakuBango) {
    // 会議室予約エンジンの予約確認機能を呼ぶ...
  }
  private YoyakuTarget generateData(int n) {
    // n 回目の会議室 A の予約用データを生成する
  }
}
```

リスト1　最初の試験プログラムのコード

また，n回目の呼び出しに使う予約用データ（どの会議室をいつからいつまで予約するか，等）は，generateData(n)によって生成される．mainは，起動したすべてのクライアントプロセスが仕事を終えると，それらから呼び出し結果やエラーの記録を取り出す（Client::getResultHistoryとClient::getErrorHistory）．またmainは，あらかじめ起動しておいたシステムリソース監視のプロセス（SystemWatcherクラス）から，呼び出し中のメモリ使用量の目安や会議室予約エンジンが作り出すスレッドの数といったシステムリソース消費の記録を取り出す（SystemWatcher::getSystemSummary）．プログラムは，これらの情報を画面に表示して終了する（Main::showResults）．

あなたはこのプログラムを使って，会議室予約エンジンが期待通りに動いていることを上司に見せた．上司はあなたの仕事に満足し，あなたはそれっきりこのプログラムのことを忘れていた．

これが我々の出発点となる試験プログラムの第一版である．ここからは，この試験プログラムの設計に，いくつかのデザインパターンを適用して，その柔軟性と再利用性を高める過程を見ていく．

汚れ役 ― *Abstract Factory*

さて，話は次のように続く：

しばらくして，会議室予約エンジンはひと通りのコーディングを終え，本格的な試験の時期を迎えた．上司はあなたに，会議室予約エンジンを試験するプログラムを書けという．あなたは，ちょっと前に書いたプログラムを引っ張り出した．しかし，このプログラムは，その後に追加された会議室予約エンジンの新しいメソッドを呼び出せるようにはなっていないし，呼び出しのシーケンスも固定的なことが分かる．また，会議室予約エンジンとの通信手段も1つに限られている．会議室予約エンジンの試験の仕様には，さまざまな機能をいろいろなシーケンスで呼び出す多数の試験項目が含まれている．また，会議室予約エンジンには2つの配置形態があって，呼び出し側と同じプロセス内に置いて使用す

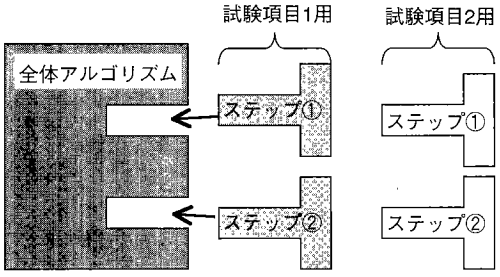

図-2　処理の共通部分と変化部分の切り離し

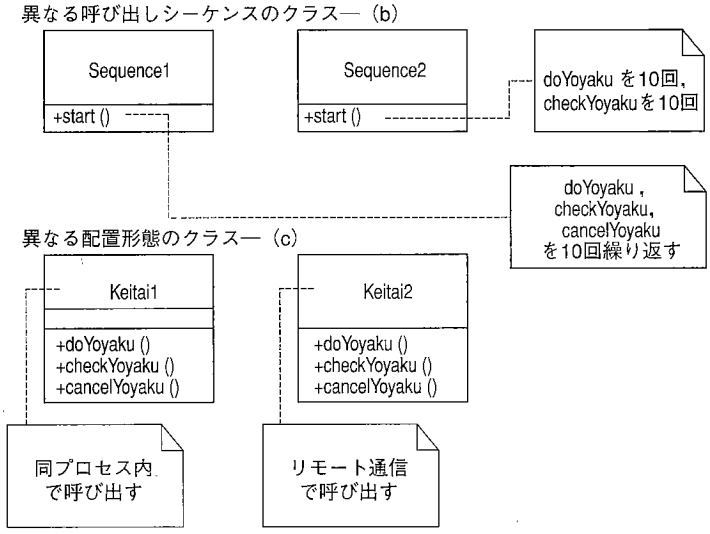

図-3　動作条件のクラス化

る場合と，別のマシン（プロセス）に置いてリモート通信で結ぶ場合とがあり，それぞれの場合で試験をしないといけない．たとえば，2つの試験項目1および2は，それぞれ次のようなものである：

試験項目1

配置形態：呼び出しプロセス内（形態1）

呼び出しプロセス数：3

呼び出しシーケンス：doYoyaku，checkYoyaku，cancelYoyakuを順に呼び出し，これを10回繰り返す（シーケンス1）

予約用データ：クライアントプロセスごとに同じ会議室，同じ予約日時に固定されたデータ（アルゴリズム1）

試験項目2

配置形態：別プロセス（形態2）

呼び出しプロセス数：10

呼び出しシーケンス：doYoyakuを10回呼び出し後，checkYoyakuを10回呼び出す（シーケンス2）

予約用データ：再現性のある乱数により生成される不規則データ（アルゴリズム2）

試験プログラムのアルゴリズムの大筋は，すべての試験項目で同じである．あとは，呼び出しのシーケンスや，試験対象の配置形態に応じた呼び出し手段などの要所を入れ替えることができれば，たくさんある試験項目で互いに違う部分だけを新たにコーディングして，試験プログラムに追加できるはずである．つまり，ここで必要とされる試験プログラムの設計は，すべての試験項目で共通している部分と，項目ごとに変わる部分を区別して，後者の追加と入れ替えを容易にする手段を提供するものである（**図-2**）．こうした設計に基づいて実装されたプログラムの共通部分のコードは，一般にフレームワークと呼ばれるが，これについてはこの連載の第3回で述べる．

すべての試験項目で共通していえる，試験プログラムの処理の手順を整理してみよう：

1. システムリソース監視プロセスを起動する．
2. 試験項目が指定する数（a）のクライアントプロセスを起動する．

   各クライアントプロセスの処理

   2-1．試験項目が定める呼び出しシーケンス（b）に従って特定の配置形態をとる試験対象（c）を呼び出す．

   2-2．呼び出しの結果と発生したエラーを記録する．
3. すべてのクライアントプロセスの終了後，システムリソース監視プロセスとクライアントプロセスから，システムリソース消費の記録，呼び出しの結果およびエラーの記録を取り出し，試験担当者が合否を判定できるよう，画面に表示する．

試験項目ごとにその内容が変化する部分は，上の手順で（a），（b），（c）となっている3カ所である．

この試験プログラムにおける試験項目のような，プログラムの毎回の動作を決める条件をいかにして変更するかは，コンピュータプログラミングに常につきまとう問題である．オブジェクト指向言語では，動作条件が決まったら，最適のコードを持ったオブジェクトだけを実行時に拾い出して，プログラム本体に組み入れる「動的束縛」と呼ばれるテクニックを用いる．これによりプログラマは，プログラム実行の各段階で，条件をいちいち見分けながら分岐をして動作を変えるといった，煩雑なコーディングから解放される．このテクニックを今回の試験プログラムに用いてみよう．先に示した試験プログラムの処理手順で，試験項目ごとに変わる（a），（b），（c）の個所のうち，（a）は単なる整数値の違いなので，何らかの方法でパラメータとしてプログラムに渡せばよい．（b）は，試験項目ごとにまったく違う処理手順を踏むので，呼び出しシーケンスの種類に分けてそれぞれに対応するクラスを用意する．また，（c）でも，会議室予約エンジンの配置形態が異なれば，その通信の手順はまったく異なる．そこで，1つの配置形態に対応して呼び出しの手段を提供するクラスを各配置形態ごとに用意することにしよう．

どこをクラス化すべきかは分かった．つまり，試験プログラムの動作で，試験項目ごとに処理手順が大きく変わる部分をオブジェクト化して，異なる条件をクラス化するのである（**図-3**）．そして，実施する試

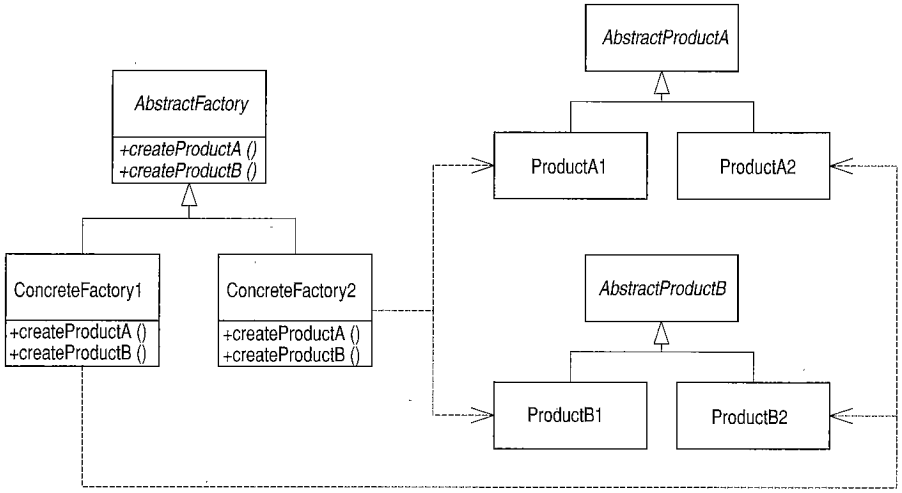

図-4 *Abstract Factory*構造図[1)]

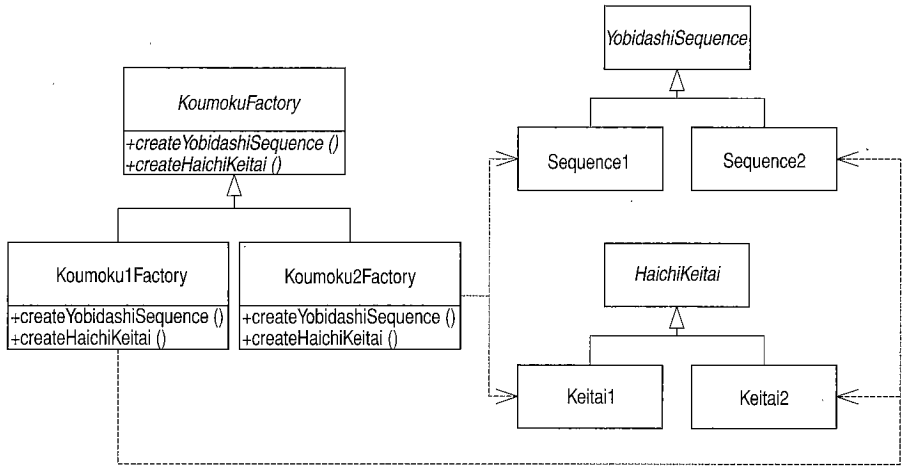

図-5 *Abstract Factory*適用後

験項目が決まると，(b) と (c) の各個所で使用すべきクラスが決まる．さて，ここからが問題である．クラス化すべき個所は分かったが，実際にどのようなクラス階層を構成すればよいだろうか．ここで「デザインパターン」[1)]を紐解いてみよう．この本の表紙の見返しを見ると，そこには23個のパターンが記載されており，「生成に関するパターン」，「構造に関するパターン」，「振る舞いに関するパターン」に分類されていることが分かる．状況に応じて異なったクラスのオブジェクトを使用するときのパターンは，「生成に関するパターン」に分類される．そこで，見返しに書かれた生成に関するパターンの記述から，我々の試験プログラムに関係がありそうなものを探すと，次の3つのパターンが浮かび上がってくる：

- ***Abstract Factory***－互いに関連したり依存し合うオブジェクト群を，その具象クラスを明確にせずに生成するためのインタフェースを提供する[1)]．
- ***Factory Method***－オブジェクトを生成するときのインタフェースだけを規定して，実際にどのクラスをインスタンス化するかはサブクラスが決めるようにする．*Factory Method*パターンは，インスタンス化をサブクラスに任せる[1)]．
- ***Prototype***－生成すべきオブジェクトの種類を原型となるインスタンスを使って明確にし，それをコピーすることで新たなオブジェクトの生成を行う[1)]．

どれもそれらしく聞こえるので，1つに絞ることができない．表紙の見返しに書かれていたのは，各パターンの記述の「目的」節で，文字通りそのパターンを使うことによってもたらされる効果が書かれている．「目的」節から使うべきパターンを1つに絞れない場合は，候補となるパターンについての「動機」と「適用可能性」のそれぞれの節を読むとよい．どのパターンが，自分の直面している問題に1番近いものを取り扱っているかが見えてくるはずである．上に挙げた3つのパターンの「動機」と「適用可能性」を注意深く読むと，*Abstract Factory*の中に，非常に類似した問題を見出すことができる：

- ***Abstract Factory*「動機」節[1)]より**：MotifやPresentation Managerなどの，複数のlook-and-feel規格をサポートしているユーザインタフェースツールキットについて考えよう．（中略）別のlook-and-feel規格に容易に変更できるようにするためには，特定の規格のウィジェットに関するプログラムをアプリケーションに直接書くことは避けな

```
public class Main {
  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...
    }
    // システムリソース監視プロセスの開始...
    for (int i = 0; i < n; i++) {
     // n 個のクライアントプロセスを起動する
     clients[i] = new Client(i, factory); // factory を渡す
     clients[i].start(); // この後 run() が呼び出される
    }
    // ...
    showResults(sw, clients); // 結果の表示
  }
  // ...
}
public abstract class KoumokuFactory {
  public abstract YobidashiSequence createYobidashiSequence(int clientID);
  public abstract HaichiKeitai createHaichiKeitai();
}
public class Koumoku1Factory extends KoumokuFactory {
  public YobidashiSequence createYobidashiSequence(int clientID) {
    return new Sequence1(clientID, this);
  }
  public HaichiKeitai createHaichiKeitai() {
    return new Keitai1();
  }
}
// Koumoku2Factory も同様に定義...
public class Client extends Thread {
  private KoumokuFactory factory;
  private YobidashiSequence sequence;
  public Client(int clientID, KoumokuFactory factory) {
    this.factory = factory;
    sequence = factory.createYobidashiSequence(clientID);
        // factory を使って呼び出しシーケンスを生成
  }
  public void run() {
    // 呼び出しシーケンスを使い会議室予約エンジンを呼ぶ
    sequence.start();
  }
  // ...
}
```

```
public abstract class YobidashiSequence {
  // ...
  public YobidashiSequence(int clientID, KoumokuFactory factory) {
    this.clientID = clientID;
    this.factory = factory;
  }
  public abstract void start();
  public abstract ResultHistory getResultHistory();
  public abstract ErrorHistory getErrorHistory();
}
public class Sequence1 extends YobidashiSequence {
  public Sequence1(int clientID, KoumokuFactory factory) {
    super(clientID, factory);
  }
  public void start() {
    HaichiKeitai keitai = factory.createHaichiKeitai();
        // 配置形態1 のオブジェクトを作ってから,
    // 試験項目1 の呼び出しシーケンスを実行...
  }
  public ResultHistory getResultHistory() {
    // 呼び出し結果の記録を返す
  }
  public ErrorHistory getErrorHistory() {
    // エラーの記録を返す
  }
}
// Sequence2 も同様に定義...
public abstract class HaichiKeitai {
  public abstract int doYoyaku(YoyakuTarget target);
      // 予約をする．予約番号を返す
  public abstract boolean checkYoyaku(int yoyakuBango);
      // 予約が取れていれば true を返す
  public abstract void cancelYoyaku(int yoyakuBango);
      // 予約を取り消す
}
public class Keitai1 extends HaichiKeitai {
  public Keitai1() {
    // 配置形態1の会議室予約エンジンと接続する...
  }
  public int doYoyaku(YoyakuTarget target) {
    // 会議室予約エンジンの予約機能を呼ぶ...
  }
  public boolean checkYoyaku(int yoyakuBango) {
    // 会議室予約エンジンの予約確認機能を呼ぶ...
  }
  public void cancelYoyaku(int yoyakuBango) {
    // 会議室予約エンジンの予約取り消し機能を呼ぶ...
  }
}
// Keitai2 も同様に定義...
```

リスト2　*Abstract Factory*適用後のコード

ければならない．

- **「適用可能性」節**[1]**より**：部品の集合が複数存在して，その中の1つを選んでシステムを構築する場合．

これらの記述を次のように読み替えると，我々の問題との関連性がいっそう明確になる：

- 複数の試験項目をサポートしている試験プログラムについて考えよう．別の試験項目に容易に変更できるようにするためには，特定の試験項目のオブジェクトに関するプログラムをアプリケーションに直接書くことは避けなければならない．
- 試験項目用のオブジェクトの集合が複数存在して，その中の1つを選んで試験プログラムを構築する場合．

どうやら*Abstract Factory*が，ここで必要とする解であることが期待できそうだ．期待を確信に変えるには，「構造」節を見て，自分のプログラムにそのパターンを組み込んだ様子を想像してみるとよい．クラス図（**図-4**）だけで分かりにくければ，「サンプルコード」節を研究して，コードに落とし込まれた形で理解するとよい．*Abstract Factory*の構造を，試験プログラムに当てはめると，**図-5**のようになる．まず，試験項目ごとに入れ替わるオブジェクトの種類，つまり「呼び出しシーケンス」と「配置形態」のそれぞれに*AbstractProduct*クラスを用意する（YobidashiSequenceとHaichiKeitai）．次に，*AbstractFactory*クラスを用意し，これをKoumokuFactoryと命名する．この中でcreateYobidashiSequenceとcreateHaichiKeitaiの各抽象メソッドを定義する．そして，YobidashiSequenceとHaichiKeitaiそれぞれに，実際に存在する呼び出しシーケンスや配置形態に対応する*ConcreteProduct*クラスを追加していく．最後に，1つ1つの試験項目に対応する*ConcreteFactory*クラスを追加することで，試験プログラム版*Abstract Factory*構造が完成する．

リスト2は，*Abstract Factory*を使った試験プログラムのコードである．試験プログラムは，実施すべき試験項目が決まると，それに対応す

```
public class Main {
  public static void main(String argv[]) {
    int n = Integer.parseInt(argv[0]);
    Client clients[] = new Client[n];
    try {
      // 指定のKoumokuFactoryインスタンスを生成
      Class factoryClass = Class.forName(argv[1]);
      KoumokuFactory factory =
        factoryClass.newInstance();
      Koumokufactory.registerInstance(factory);
          // factoryをSingletonとして登録
    } catch (Exception e) {
      // 指定された名前のクラスは存在しない...
    }
    //...
    for (int i = 0; i < n; i++) {
      // n個のクライアントプロセスを起動する
      clients[i] = new Client(i);
      clients[i].start(); // この後run()が呼び出される
    }
    // ...
  }
  // ...
}
public abstract class KoumokuFactory {
  protected static KoumokuFactory instance = null;
  public static void registerInstance(KoumokuFactory factory)
  {
    if (instance == null) instance = factory;
  }
  public static KoumokuFactory getInstance() {
    return instance;
  }
  public abstract YobidashiSequence createYobidashiSequence(int clientID);
  public abstract HaichiKeitai createHaichiKeitai();
}
public class Koumoku1Factory extends KoumokuFactory {
  public YobidashiSequence createYobidashiSequence(int clientID) {
    return new Sequence1(clientID);
  }
  public HaichiKeitai createHaichiKeitai() {
    return new Keitai1();
  }
}
public class Client extends Thread {
  private YobidashiSequence sequence;
  public Client(int clientID) {
    // Singletonを使って呼び出しシーケンスを生成
    sequence =
      KoumokuFactory.getInstance().createYobidashiSequence(clientID);
  }
  public void run() {
    // 呼び出しシーケンスを使い会議室予約エンジンを呼ぶ
    sequence.start();
  }
  // ...
}
//...
public class Sequence1 extends YobidashiSequence {
  private KoumokuFactory factory;
  public Sequence1(int clientID) { super(clientID); }
  public void start() {
    HaichiKeitai keitai =
      KoumokuFactory.getInstance().createHaichiKeitai();
        // 配置形態1のオブジェクトを作ってから,
    // 試験項目1の呼び出しシーケンスを実行...
  }
  // ...
}
// ...
```

リスト3　*Singleton*適用後のコード

る *ConcreteFactory* クラスを使って，適切な組合せの「呼び出しシーケンス」オブジェクトと「配置形態」オブジェクトを生成する．試験プログラム本体は，もはや1つの試験項目に特化したコードを持たない．ユーザがコマンドラインから *ConcreteFactory* クラス名を指定するだけで，試験プログラムは正しいオブジェクトを選ぶことができる．

新しい試験項目が増えたときには，新たな *ConcreteFactory* クラス（KoumokuFactoryのサブクラス）を追加すればよい．同様に，新たな呼び出しシーケンスが増えたときには，YobidashiSequenceのサブクラスを定義すればよい．こうして，試験プログラムは，既存のコードを変更することなく，新たな試験項目に対応できるようになる☆1．これで目標は達成された！

どのパターンにも欠点はある．*Abstract Factory* は，次のような事態への対処は苦手である：

- **新しい *AbstractProduct* が頻繁に追加される場合．** たとえば，会議室予約エンジン以外のソフトウェアを試験できるようにするために，試験項目ごとの変化要因として「呼び出しシーケンス」，「配置形態」の他に，「試験対象」が加わったらどうなるだろう．既存のすべての *ConcreteFactory* クラスにcreateShikenTaishoメソッドを追加しなくてはならない．
- **試験項目数が膨大なとき．** 原則として *ConcreteFactory* クラスは，利用される *ConcreteProduct* の可能な組合せ1つ1つについて定義しなくてはならない．2つの試験項目の違いがわずかであっても，2つの *ConcreteFactory* クラスが必要である．試験項目が1000個もあるとしたら，すべての *ConcreteFactory* クラスを定義するのは骨が折れるだろう．

もしも上に挙げた事態が現実のものとなるようならば，それに対処できる別のパターンを探すか，*Abstract Factory* に別のパターンを組み合わせて問題を解決する必要がある．該当するパターンがなければ，自分で解決策を練り，新しいパターンの候補としてPLoP[2]のような場で報告すれば，歓迎を受けるだろう！　たとえば，*ConcreteFactory* クラスが多すぎる場合は，*Prototype* パターンと組み合わせることにより，クラスの数を減らすことができる（*Prototype* パターン「結果」節4項[1]）☆2．デザインパターンは，条件を誤ると，設計上の問題を解決しないばかりか，かえって設計を悪くすることもある．パターン適用前には，そのパターンの「結果」節を読み，その効果や副作用をチェックして，それが自分のプログラムの条件に合っているかどうかを考えるべ

☆1　クラスを動的に読み込めない場合，使うべき *ConcreteFactory* クラスをハードコーディングで特定する個所が少なくとも1カ所現れるので，試験項目が追加されたときには，コードの書き換えが生じる．しかし，書き換え個所は限定されるので，必ずしも設計の柔軟性が損なわれるわけではない．

☆2　*AbstractProduct* と *ConcreteProduct* をそれぞれ *Prototype* と *ConcretePrototype* にする．唯一の *ConcreteFactory* に適切な *ConcretePrototype* を登録し，各 *createProduct* メソッドはそのコピーを返す．このデザインパターンは *Pluggable Factory*[3] と呼ばれる．これを使って，ファイルから生成すべきオブジェクトのリストを読み込むなどして使えば，*ConcreteFactory* クラスの増加を抑制することができる．

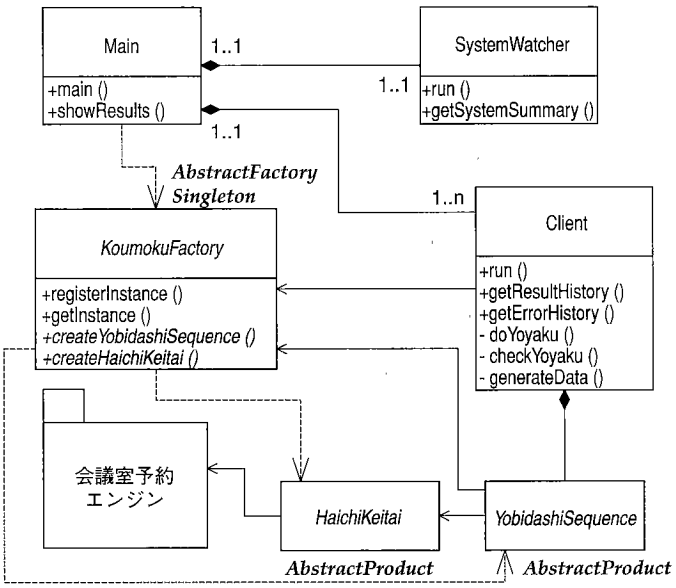

図-6 *Abstract Factory+Singleton* 版試験プログラム構造.
パターンの構成要素となっているクラスには構成要素名を記した.

きである.

さて，これが「会議室予約エンジン試験プログラム」への最初のパターンの適用例となった．*Abstract Factory*は，プログラムの動作を決定する選択肢がたくさんあったり，後から追加されるときに，それらを切り替える手段を提供する. それは，アプリケーションが特定の動作条件に固定されないように，特化した部分をオブジェクト化して，その取り扱いを一手に引き受ける，いわば「汚れ役」である.

孤独なインスタンス ー *Singleton*

リスト2では，Mainが作るKoumokuFactoryのサブクラスのインスタンスを，各Clientインスタンスのコンストラクタに直接渡している．また，同じように，ClientはYobidashiSequenceのサブクラスのコンストラクタにKoumokuFactoryのサブクラスのインスタンスを直接渡している．しかし，KoumokuFactoryを使うすべてのオブジェクトにそのインスタンスを直接渡すのは，少々煩雑である．ところで，*Abstract Factory*パターンの「関連するパターン」節[1]には，次のように書かれている：「*ConcreteFactory*オブジェクトは，しばしば，*Singleton*オブジェクトである」．実は，上で述べた問題に対する解が，この*Singleton*パターンである：

- ***Singleton***ーあるクラスに対してインスタンスが1つしか存在しないことを保証し，それにアクセスするためのグローバルな方法を提供する[1].

一見，上で述べた問題と関係がなさそうである．ヒントは，このパターンの「適用可能性」節にある：

- ***Singleton*「適用可能性」節**[1]**より**：唯一のインスタンスがサブクラス化により拡張可能で，また，クライアントが，そのインスタンスを公開されたアクセスポイントを通してアクセスできるようにしなければならない場合.

一言でいうと，*Singleton*は，あるクラスとそのサブクラスの間で，インスタンスがたった1つしか存在しないことを保証するトリックである．これを使って，KoumokuFactoryとそのすべてのサブクラスの間で，実施中の試験項目に対応する正しいインスタンスだけが常に存在する仕組みを実現できる（**リスト3**）．タネを明かせば，実行時に1度試験項目が決まると，それに対応した*ConcreteFactory*クラスのインスタンスを，親クラスであるKoumokuFactoryのクラス変数に格納するのである．KoumokuFactoryクラス自体はコーディング時に参照可能なので，試験プログラムの各所では，コンストラクター引数などを使わずに，このクラスに格納されたインスタンスを取り出すことができる☆3．**図-6**は，この*Abstract Factory+Singleton*版試験プログラムの全体構造である.

試験プログラムへの*Abstract Factory*パターンの適用は，*Singleton*パターンの適用への流れを自然に作り出した．このように，1つのパターンの適用が，次のパターンの適用の必要性を自ずと明らかにすることは，しばしば起こる．「デザインパターン」[1]の「関連するパターン」節には，そのパターンに組み合わせて使えるパターンが書かれている. また，裏表紙の見返しには，23個すべてのパターンの関連図が書かれている．これらは，これから適用しようとするパターンと併せて使うべきパターンを探すヒントになる.

☆3 複数の試験項目を連続して実行する場合，試験項目が切り替わるたびに，*Singleton*が持つ*ConcreteFactory*を殺し，新しいものと入れ替える手段が必要となる[4].

参考文献

1) Gamma, E. et al.: Design Patterns, Addison-Wesley, Massachusetts (1995).（本位田真一，吉田和樹(監訳): デザインパターン改訂版，ソフトバンク，東京 (1999)).
2) Pattern Languages of Programs (PLoP) Conference Home Page: http://st-www.cs.uiuc.edu/~plop/
3) Lauder, A.: Pluggable Factory in Practice, C++ Report, Vol.11, No.9, pp.27-31 (1999).
4) Vlissides, J.: Pattern Hatching, Addison-Wesley, Massachusetts (1997).(パターンハッチング，ピアソン，東京 (1999)).

(平成11年12月20日受付)

本連載の第3回では，今回Abstract FactoryおよびSingletonの各パターンを適用した「会議室予約エンジン試験プログラム」に，台否判定機能を追加するためのパターン適用を試みる．そして，デザインパターンがソフトウェア開発にもたらす意味について議論し，その中でデザインパターンによるドキュメンテーションやフレームワーク開発などについて触れる.