The ThoughtWorks Anthology

ThoughtWorks® アンソロジー

アジャイルとオブジェクト指向による ソフトウェアイノベーション

ThoughtWorks Inc. 著
株式会社オージス総研
オブジェクトの広場編集部 訳

O'REILLY®
オライリー・ジャパン

5章
オブジェクト指向エクササイズ

Jeff Bay ● テクノロジプリンシパル

5.1 ソフトウェア設計を改善する9つのステップ

誰でも、理解やテスト、保守に苦労するようなひどいコードを目にしたことがあるでしょう。オブジェクト指向プログラミングによって、古い手続き型コードからの解放が約束され、再利用してインクリメンタルにソフトウェアを開発することが可能になりました。それにもかかわらず、これまでC言語で行ってきた複雑で結合度の高い設計を、同じようにJavaでも追求しようとしていることがあります。このエッセイは、新米プログラマがコードを書く過程でベストプラクティスを学ぶ、よいきっかけとなるでしょう。また、経験豊かな熟練プログラマは、ベストプラクティスの復習や、同僚に教える際の題材としてこのエッセイを利用できるでしょう。

優れたオブジェクト指向設計は、複雑さの排除に絶大な効果をもたらすものですが、簡単に習得できるものではありません。手続き型の開発からオブジェクト指向設計へ移行するには大きな思考の転換が必要であり、これは思っているよりずっと難しいのです。多くの開発者は優れたオブジェクト指向設計ができていると思い込んでいますが、実際には手続き型の習慣に無意識のうちにとらわれて、なかなか抜け出せていないのです。多くの手本やベストプラクティス（Sun JDKのコードでさえも）が、性能上の理由や単に歴史的な経緯から、質の悪いオブジェクト指向設計を助長しており、状況を悪くしています。

優れた設計を支える中心的な概念はよく知られています。例えば、凝集度（cohesion）、疎結合（loose coupling）、重複なし（zero duplication）、カプセル化（encapsulation）、テスト容易性（testability）、可読性（readability）、フォーカス（focus）の7つのコード品質特性は有名です[†]。しかし、これらの概念を実践に移すのは簡単ではありません。カプセル化とは「データ、実装、クラス、設計、実体化

の隠蔽」であると理解することと、カプセル化を適切に実現するコードを設計することは、まったく別のことなのです。そこで、優れたオブジェクト指向設計の原理を自分のものにして、実際に使えるようになるためのエクササイズを紹介します。

5.2　エクササイズ

これまで使ってきたものよりずっと厳しいコーディング標準に従って、簡単なプロジェクトを実施してみてください。このエッセイでは、経験に基づく9つのルールを紹介します。これは、ほぼ必然的にオブジェクト指向になるコードを書くように強制するものです。このエクササイズによって、日々の業務で問題に直面したときに、より適切な判断を下せるようになり、さらに選択肢の量と質が向上するでしょう。

まずは騙されたと思って、1,000行程度の小さなプロジェクトで9つのルールを厳密に適用してみてください。そうすれば、ソフトウェア設計に対するまったく異なるアプローチに気がつくでしょう。一度1,000行のコードを書き終えてしまえば、このエクササイズは終了です。今度はリラックスし、ルールを緩めてガイドラインとして使うことができます。

これはハードなエクササイズです。その主な理由は、ルールの多くが普遍的には適用できないからです。実際、クラスが50行を少し超えることもあります[††]。しかし、重複し散在してしまっている責務を移動して、その責務を持つ適切なファーストクラスオブジェクト[‡]へと作り替えるためには何をする必要があるか、を考え

† 訳注：これらのコード品質特性は、著者の経験と、『Code Complete 第2版〈上/下〉』（日経BPソフトプレス）や多くのXP指導者による資料などに基づくものである。また、著者のサイトで公開されている本章の草稿（http://www.xpteam.com/jeff/writings/objectcalisthenics.rtf）では、Alan Shallowayが7つの特性を提唱していることが書かれており、著者が彼のアイデアも参考にしていることがわかる。なお、7つの特性のうち「フォーカス」はそれほど知られていないが、これは概念の集中度を表す特性であり、かかわりのある概念が1つのクラスに集まっていれば高く、多くのクラスに散らばっているほど低くなる。「凝集度」はクラスやメソッドの中で扱われる概念の関連の強さを表すので、「フォーカス」は「凝集度」と直交する特性である。例えば、なんらかのかかわりを持つたくさんの概念を1つのクラスで扱うと、「フォーカス」は高くなるが、「凝集度」は低くなる。

†† 訳注：場合によっては、ルールが互いに矛盾することや、ルールを適用できないこともある。そのため、状況に応じて優先すべきルールを選択し、どのようにルールを適用するのかを判断しなければいけない、というのが著者の意図である。

‡ 訳注：プログラミング言語の文脈では「扱いに関してきわめて制限の少ない言語の構成要素」を表す用語として使われるが、本章では「プリミティブ型やコレクションなどをラップする独自のクラスが定義されたオブジェクト」を表す。具体例は、ルール3やルール8の説明で示されている。

ることに大きな価値があります。このエクササイズの本当の価値は、この種の考え方を身につけることなのです。したがって、これ以上できないという思い込みを捨てて、自分のコードについて新しい見方ができるようになっているかを意識するようにしてください。

5.2.1 9つのルール

以下に、エクササイズのルールを示します†。

1. 1つのメソッドにつきインデントは1段階までにすること
2. else句を使用しないこと
3. すべてのプリミティブ型と文字列型をラップすること
4. 1行につきドットは1つまでにすること
5. 名前を省略しないこと
6. すべてのエンティティを小さくすること
7. 1つのクラスにつきインスタンス変数は2つまでにすること
8. ファーストクラスコレクションを使用すること
9. Getter、Setter、プロパティを使用しないこと

5.2.2 ルール1：1つのメソッドにつきインデントは1段階までにすること

昔に書かれた、どこから手をつけてよいのかわからないほど大きいメソッドを見たことはありませんか。巨大なメソッドは凝集度が欠けています。メソッドの長さを5行までに制限するのもガイドラインの1つですが、500行もある化け物のようなメソッドでコードが散らかっていると、そうした変更は大変なものになるでしょう。代わりに、各メソッドが厳密に1つの仕事を行うこと、つまりメソッドごとに制御構造またはコードブロックを1つだけにすることを徹底しましょう。もし1つのメソッド内にネストされた制御構造があれば、複数の抽象レベルを扱っていることになり、それは1つ以上の仕事を行っていることを意味します。

厳密に1つの仕事を行うクラスで、厳密に1つの仕事を行うメソッドを書くようにすれば、コードが変わってきます。アプリケーションにおける処理単位が小さくなれば、再利用性が格段に上がります。責務を5つも持ち、100行に及ぶようなメ

† 訳注：著者のサイトで公開されている本章の草稿（http://www.xpteam.com/jeff/writings/objectcalisthenics.rtf）では、「ファクトリメソッド以外のスタティックメソッドは作らないこと」というルールもある。

ソッドを再利用できる機会はまずありません。さまざまな文脈で利用できるのは、ある文脈において1つのオブジェクトの状態を管理する3行のメソッドなのです。統合開発環境の「メソッドの抽出（Extract Method）」[†]機能を使って、例に示すようにメソッド内のインデントが1段階になるまで振る舞いを抜き出してください。

リファクタリング前[††]

```
class Board {
        ...
        String board() {
                StringBuffer buf = new StringBuffer();
                for(int i = 0; i < 10; i++) {
                        for(int j = 0; j < 10; j++)
                                buf.append(data[i][j]);
                        buf.append("\n");
                }
                return buf.toString();
        }
}
```

リファクタリング後

```
class Board {
        ...
        String board() {
                StringBuffer buf = new StringBuffer();
                collectRows(buf);
                return buf.toString();
        }

        void collectRows(StringBuffer buf) {
                for(int i = 0; i < 10; i++)
                        collectRow(buf, i);
        }

        void collectRow(StringBuffer buf, int row) {
                for(int i = 0; i < 10; i++)
                        buf.append(data[row][i]);
                buf.append("\n");
        }
}
```

† 訳注：リファクタリング手法の1つであり、メソッドの一部を抜き出して別のメソッドを作り、それを呼び出すようにすること。

†† 訳注：このサンプルコードは、チェスや碁のようなボードゲームをイメージしたものである。Boardクラスは盤面を表しており、board()メソッドは盤面の状態を出力するメソッドである。

このリファクタリングには、もう1つメリットがあることに注目してください。個々のメソッドは小さくなって、実装がメソッド名と一致しました。たいてい、このような小さなコード片の中からバグを発見するのは非常に簡単です。

最後に、このルールを繰り返し適用すればするほどその強みが得られる、ということを付け加えておきます。初めてこの方法で問題を分解しようとするときは、やりにくい感じがして、なんの得があるのかわからないかもしれません。しかし、ルールを適用するにもスキルが必要で、これは次のレベルに進んだプログラマの技なのです。

5.2.3 ルール2：else句を使用しないこと

プログラマなら誰でもif-else構文を知っています。この構文は、ほとんどのプログラミング言語に組み込まれており、単純な条件ロジックなら誰でも簡単に理解できます。しかし、ほとんどのプログラマは、うんざりするほどネストされた追跡不可能な条件文や、何度もスクロールしなければ読めないcase文を見たことがあるでしょう。さらに悪いことに、既存の条件文に単に分岐を1つ増やすほうが、もっと適切な解決方法を考えるよりも楽なのです。また、条件文は重複の原因になることがよくあります。例えば、ステータスフラグはこの種の問題をよく引き起こします。

リファクタリング前

```
public static void endMe() {
        if (status == DONE) {
                doSomething();

        } else {
                ... その他のコード ...
        }
}
```

このコードをelse句を使わずに書き直す方法がいくつかあります。コードが単純な場合には、次のようにします。

リファクタリング後

```
public static void endMe() {
        if (status == DONE) {
                doSomething();
                return;
        }
        ... その他のコード ...
}
```

リファクタリング前

```
public static Node head() {
        if (isAdvancing()) { return first; }
        else { return last; }
}
```

リファクタリング後

```
public static Node head() {
        return isAdvancing() ? first : last;
}
```

このように、単語をたった1つ追加しただけで4行が1行になります。ただし、メソッドからの早期returnは、使いすぎるとわかりにくくなってしまうことに注意してください。ステータスに基づく分岐を避けるためにポリモフィズムを利用する方法については、GoFデザインパターン[GHJV95]のStrategyパターンを参照してください†。Strategyパターンは、ステータスに基づく分岐が複数箇所で使われている場合に特に有効です。

オブジェクト指向言語には、ポリモフィズムという、複雑な条件分岐を扱うための強力なツールがあります。単純なものであれば、ガード節††と早期returnに置き換えられます。また、ポリモフィズムを用いれば、可読性や保守性がもっと高く、意図をより明確に表す設計にすることができます。しかし、else句を使いこなしていると、この変更は簡単ではありません。そのため、このエクササイズの一環として、else句の使用を禁止します。NullObjectパターン‡も試してみてください。これが役に立つ状況もあるでしょう。同様にelse句を使わずに済ませるツールは他にもあります。選択肢をいくつか考えてみてください。

† 訳注：Stateパターンを想起させるような説明だが、実際、GoFのStrategyパターンに条件文を排除する例が掲載されている。

†† 訳注：ある条件を満たしていない場合に直ちにreturnするか例外を投げることで、以降の処理の事前条件を守るif文のこと。『ケント・ベックのSmalltalkベストプラクティス・パターン』（ピアソン・エデュケーション）で、「Guard Clauseパターン」として示されている。また、『リファクタリング』（ピアソン・エデュケーション）で、ネストされた条件記述をガード節に置き換える例が示されている。

‡ 訳注：通常の仕事をするオブジェクトと同じインターフェイスを持ちながら何もしない空オブジェクト（Null Object）を利用するパターン。『プログラムデザインのためのパターン言語』（ソフトバンククリエイティブ）に収録されている。

5.2.4 ルール 3：すべてのプリミティブ型と文字列型をラップすること

int 型は、それだけではなんの意味も持たない単なるスカラ値にすぎません。メソッドが int 型をパラメータとして受け取る場合、メソッド名で意図を表現するしかありません。もし同じメソッドが「時間」オブジェクトをパラメータとして受け取るなら、それが何を指すのかずっとわかりやすくなります。このように小さなオブジェクトを使うことによってプログラムの保守性が高まります。なぜなら、「時間」オブジェクトをパラメータに取るメソッドに「年」オブジェクトを渡すことは不可能だからです。プリミティブ型の変数を使っていると、意味的に正しいプログラムかどうかをコンパイラは教えてくれません。たとえ小さくてもオブジェクトを使うことで、それが何の値でなぜそこで使うのかという情報をコンパイラとプログラマに伝えることができます。

「時間」オブジェクトや「金額」オブジェクトのような小さなオブジェクトを使うと、どこに振る舞いを配置すべきかが明確になります。そのようなオブジェクトがなければ、振る舞いはいろいろなクラスに散らばっていたでしょう。後で紹介する Getter と Setter に関するルール（ルール 9）を適用して、小さなオブジェクトを厳密にカプセル化した場合に、これは特に当てはまります。

5.2.5 ルール 4：1 行につきドットは 1 つまでにすること

あるアクティビティに対する責務をどのオブジェクトが持つべきなのか、判断が難しいことがあります。複数のドットを使っているコードが何行かあれば、責務の配置を間違っている箇所がたくさん見つかるでしょう。1 行の中に複数のドットがある場合、そのアクティビティは間違った場所で実行されようとしています。おそらく、そのオブジェクトは同時に 2 つの異なるオブジェクトを扱おうとしているのでしょう。この場合、そのオブジェクトは仲介役で、多くのオブジェクトを知りすぎています。そのアクティビティを他のオブジェクトに移すことを検討してください。

ドットがつながっているということは、オブジェクトが他のオブジェクトの中を深く掘り進んでいるということです。つまり複数のドットは、カプセル化に違反していることを示しています。自らオブジェクトの中をいじり回るのではなく、そのオブジェクトにしかるべき仕事をさせるようにしましょう。カプセル化の主な役割は、クラスの境界を越えて知るべきでない型にたどり着かないようにすることです。

デメテルの法則（「直接の友人にだけ話しかけよ」）[†]に従うことから始めるとよい

† 訳注：すべてのメソッドは、自分自身、パラメータのオブジェクト、自分で生成したオブジェクト、直接の関連を持つオブジェクトのメソッドのみを呼び出すべきであるという原則のこと。

でしょう。ただし、これを次のようにとらえてみてください。「遊んでよいのは、自分のオモチャ、自分で作ったオモチャ、誰かがくれたオモチャのみである。自分のオモチャのオモチャでは決して遊んではいけない。」

リファクタリング前†

```
class Board {
    ...

    class Piece {
        ...
        String representation;
    }
    class Location {
        ...
        Piece current;
    }

    String boardRepresentation() {
        StringBuffer buf = new StringBuffer();
        for(Location l : squares())
            buf.append(l.current.representation.substring(0, 1));
        return buf.toString();
    }
}
```

リファクタリング後

```
class Board {
    ...

    class Piece {
        ...
        private String representation;
        String character() {
            return representation.substring(0, 1);
        }

        void addTo(StringBuffer buf) {
            buf.append(character());
        }
```

† 訳注：このサンプルコードは、ルール1のサンプルコードと同様にボードゲームをイメージしたものである。Boardクラスは盤面、Pieceクラスはコマ、Locationクラスは盤面の1マスを表している。

```
    }
    class Location {
        ...
        private Piece current;

        void addTo(StringBuffer buf) {
            current.addTo(buf);
        }
    }

    String boardRepresentation() {
        StringBuffer buf = new StringBuffer();
        for(Location l : squares())
            l.addTo(buf);
        return buf.toString();
    }
}
```

この例では、アルゴリズム実装の詳細が拡散していることに注目してください。ひと目で理解するのが少し難しくなったかもしれません。しかし、コマ（Piece）を文字列表現に変換するためのメソッドを作り、character() という名前を付けただけなのです。このメソッドは名前と仕事が強く結びついていて、非常に再利用しやすいものになっています。また、プログラムの他の箇所で representation.substring(0, 1) が繰り返し登場する可能性が格段に下がりました。リファクタリング後のプログラムでは、メソッド名はコメントの代わりになります。名前を付けることに時間を割いてください。このような構造のプログラムを理解するのは、決して難しくありません。ただ、少し異なるアプローチが必要なだけなのです。

5.2.6 ルール 5：名前を省略しないこと

クラスやメソッド、変数の名前を省略したくなることがよくあります。その誘惑に打ち勝ってください。省略は紛らわしくなりますし、もっと重大な問題を隠してしまいがちです。

なぜ省略したくなるのか考えてみてください。同じ単語を何度も何度も入力しているせいではありませんか。もしそうなら、そのメソッドは頻繁に使われすぎています。重複を避ける機会を見逃してしまっているのです。もしくは、メソッド名が長くなってきているせいではありませんか。もしそうなら、責務の配置を間違えているか、必要なクラスを抽出できていないのかもしれません。

クラスやメソッドの名前は、1つか2つの単語だけを使うように気をつけ、文脈

が重複する名前は避けてください。Orderというクラスなら、メソッドをshipOrder()とする必要はありません。単にship()と名付けてください。そうすることで、メソッド呼び出しはorder.ship()となります。何が起こるのかはっきりとわかりやすい表現です。

このエクササイズでは、すべてのエンティティの名前には1つか2つの単語だけを使い、省略しないでください。

5.2.7 ルール6：すべてのエンティティを小さくすること

これは、50行を超えるクラス、10ファイルを超えるパッケージは作らないという意味です。

たいてい、50行を超えるクラスは、複数の仕事をしています。それによって、理解や再利用が難しくなってしまいます。50行のクラスはスクロールせずに1画面で見ることができるので、ひと目で理解しやすくなるという利点もあります。

クラスを小さくするのが難しいのは、複数の振る舞いが集まって1つの論理的な意味を表すことがよくあるからです。そこで、パッケージが必要になります。クラスを小さくして責務を減らし、パッケージ内のファイル数も制限すると、パッケージがある目的のために協調して動作するクラスの集まりを表すようになることに、気がつくでしょう。パッケージもクラスと同様に、凝集度の高い、目的を持ったものにすべきです。パッケージを小さくすることで、パッケージが真の存在意義を持てるようになります。

5.2.8 ルール7：1つのクラスにつきインスタンス変数は2つまでにすること

ほとんどのクラスはただ1つの状態変数を扱うことだけに責任を持つべきですが、2つの変数を必要とするクラスも少しはあります。クラスに新しいインスタンス変数を1つ追加すると、途端にそのクラスの凝集度が低下してしまいます。通常、このエクササイズの9つのルールに従ってコーディングしていると、2種類のクラスがあることに気がつくでしょう。一方は、1つのインスタンス変数の状態を管理するクラス、もう一方は、2つの独立した変数を調整するクラスです。原則として、この2種類の責務を混ぜ合わせないでください。

鋭い読者の方は、ルール3と7は同類であることに気づいているかもしれません。一般的に言って、インスタンス変数をたくさん持つクラスが、凝集度の高い1つの仕事をしていることはほとんどないのです。

ここで読者の皆さんに、クラスの解剖実験に取り組んでもらいましょう。

リファクタリング前

```
class Name {
    String first;
    String middle;
    String last;
}
```

このコードは次のように2つのクラスに分解できます。

リファクタリング後

```
class Name {
    Surname family;
    GivenNames given;
}

class Surname {
    String family;
}

class GivenNames {
    List<String> names;
}
```

名前の分解について考える場合、ファミリーネーム（Surname）は多くの法律的な制約に用いられるので、他とは本質的に異なる種類の名前としてその関心事を分離できる、ということに着目してください†。名（GivenNames）オブジェクトは、名前のリストを持っています。これによって、新たなモデルは、ファーストネーム、ミドルネーム、その他の名前を持つ人々にも対応できるようになります。インスタンス変数を分解することで、関連する複数のインスタンス変数の共通点を理解できることがよくあります。また、関連する複数のインスタンス変数が、実際には同一ファーストクラスコレクション††の要素として収められることもあります。

属性をまとめて持つオブジェクトを分解して、協調するオブジェクトの階層構造に作り変えることは、有効なオブジェクトモデルの作成に直結します。筆者は、このルールを見いだす前、大きなオブジェクト全体にわたるデータの流れをたどるの

† 訳注：ファミリーネームは重要な概念であるため、リファクタリング前のコードのように最初（first）や最後（last）という順番で表すべきでないというのが著者の主張である。

†† 訳注：著者は、プログラミング言語で提供されているリストやマップなどのコレクションをプリミティブと見なして、それをラップしたクラスをファーストクラスコレクションと呼んでいる。具体的には、サンプルコードのGivenNamesのようなクラスがファーストクラスコレクションである。

に膨大な時間をかけていました。オブジェクトモデルをどうにか取り出すことはできましたが、関連する振る舞いのグループを理解し、その処理結果を確認するのは骨の折れる作業でした。それに比べて、このルールを繰り返し適用すると、巨大で複雑なオブジェクトから非常にシンプルなモデルへと速やかに分解できるようになりました。振る舞いは、インスタンス変数の後を追って自然と適切な場所に収まります。というのも、コンパイラとカプセル化のルールが、インスタンス変数と異なる場所に振る舞いを置くことを許さないからです。もし行き詰ってしまったら、オブジェクトを二分して関連する2つのオブジェクトを作るようにトップダウンで作業してみてください。もしくは、2つのインスタンス変数を取り上げて、それらからオブジェクトを作るようにボトムアップで作業してみてください。

5.2.9　ルール8：ファーストクラスコレクションを使用すること

このルールを適用するのは簡単です。コレクションを持つクラスには、他のメンバ変数を持たせないようにしてください。各コレクションをそれぞれ独自のクラスにラップすることで、コレクションに関する振る舞いをそのクラスに置くことができるようになります。フィルタ†がこの新たなクラスに含まれるようになるかもしれません。また、フィルタを独立した関数オブジェクトにしてもいいでしょう。さらに、この新しいクラスは、2つのグループの結合††や、グループの各要素に対するルールの適用‡といったアクティビティも扱うことができます。ルール8は、ルール7（インスタンス変数のルール）を少し拡張しただけのものですが、これだけで独立したルールにする価値のある重要なものです。なぜなら、コレクションは非常に便利ではありますが、実際には単なるプリミティブな型の一種にすぎないからです。コレクションは多くの振る舞いを持っていますが、後任のプログラマや保守担当者にプログラム上の意図やヒントをほとんど示せないのです。

† 訳注：コレクションからある条件に該当する要素だけを抽出する機能のこと。
†† 訳注：2つのコレクションをマージして、1つにまとめること。
‡ 訳注：コレクションの要素1つ1つに対して、なんらかのルールを適用すること。

5.2.10 ルール 9：Getter、Setter、プロパティ†を使用しないこと

前ルールの最後の文は、このルールにほぼ直結します††。インスタンス変数の適切な集合をカプセル化してもまだ設計がぎこちないときは、もっと直接的なカプセル化の違反がないかをチェックしましょう。振る舞いがその場で簡単に値を求められるようになっていると、その振る舞いはインスタンス変数の後を付いてきません‡。カプセル化によって強固な境界を築く背景には、プログラマはオブジェクトモデルの中から振る舞いを配置すべき唯一の場所を見つけ、そこに振る舞いを配置すべきであるという考え方があります。これは後に、重複の大幅な削減や、新機能を実現するための修正の局所化、といった多くの効果をもたらします。

このルールは「求めるな、命じよ」として一般的に言われています。

5.3 まとめ

9つのルールのうち7つは、単純に、データのカプセル化というオブジェクト指向プログラミングの究極のテーマを明文化し、実現するための方法です。また、もう1つはポリモフィズムの適切な利用（else句を使わずに、条件ロジックを最小にする）を促すものです。さらに、もう1つは命名戦略で、一貫性がなく発音しにくい省略をなくし、明確でわかりやすい命名標準を促すものです。

このエクササイズ全体の目的は、コードレベル、あるいは概念レベルにおいて重複のないコードを作り上げることです。目標とするのは、日々扱っている複雑さを抽象化したシンプルでエレガントな概念を、簡潔に表現するコードです。

エクササイズを進めていく過程で、場合によってはルールが互いに矛盾したり、ルールを適用することでかえって悪い結果になったりすることが必ずあります。しかし、あくまでエクササイズだと思って、20時間かけて、ルールを100%適用した

† 訳注：C#などにおけるプロパティを含め、インスタンス変数を他のクラスに公開するあらゆる仕組みを使うべきでない、というのが著者の意図である。

†† 訳注：ここは原著の誤植である。著者のサイトで公開されている本章の草稿（http://www.xpteam.com/jeff/writings/objectcalisthenics.rtf）から、実際はルール7の後半にある「振る舞いは、インスタンス変数の後を追って自然と適切な場所に収まります。」という文章を指していることがわかる。

‡ 訳注：インスタンス変数の値が他のオブジェクトから簡単に取得できるようになっていると、振る舞いはそのインスタンス変数とは違う場所に配置されてしまうということ。例えば、クラスAにインスタンス変数aがあり、クラスBでaを2倍した値が必要だとする。このとき、Getterなどによってクラスで簡単にaの値を取得できるようになっていると、aを2倍する処理はクラスBに書かれてしまう可能性がある。

1,000 行のコードを書いてください。古い習慣から抜け出さなければならないことや、これまでのプログラミング人生でずっと使ってきたルールを変えなければならないことに気がつくでしょう。これまでなら通常は明確な（でも、おそらく正しくない）答えがあったのに、ルールに従うとその答えは使えない、という状況に直面するようなルールを選んであります。

努力してルールを守ることによって、これまでより確かな答えに必ずたどり着くことができます。そして、オブジェクト指向プログラミングをはるかに深く理解できるようになるはずです。ルールに従って 1,000 行のコードを書くと、出来上がったものが想像していたものとはまったく異なることに気がつくでしょう。ルールを守り、そして、結果を確認してください。それを続けることによって、特に意識しなくてもルールに従ったコードを書けるようになるでしょう。

最後に補足ですが、このエクササイズのルールを極端なもの、もしくは実際のシステムには適用できないものだと見る人もいるかもしれません。しかし、それは間違いです。この本が出版される頃、筆者はこの方法で書かれた 100,000 行を超えるシステムを完成させる予定です。このシステムを開発しているプログラマたちは、9 つのルールに常に従っています。そして、真にシンプルであることを受け入れたときに開発がどれほど楽になるかを知って、とても喜んでいます。

ThoughtWorks アンソロジー
── アジャイルとオブジェクト指向によるソフトウェアイノベーション

2008年12月24日　初版第1刷発行
2009年 1 月30日　初版第2 刷発行

著　　者　ThoughtWorks Inc.（ソートワークス）
訳　　者　株式会社オージス総研 オブジェクトの広場編集部
（かぶしきがいしゃオージスそうけん オブジェクトのひろばへんしゅうぶ）
発 行 人　ティム・オライリー
制　　作　有限会社はるにれ
印刷・製本　株式会社平河工業社
発 行 所　株式会社オライリー・ジャパン
〒160-0002 東京都新宿区坂町26番地27 インテリジェントプラザビル 1F
TEL　(03)3356-5227
FAX　(03)3356-5263
電子メール　japan@oreilly.co.jp
発 売 元　株式会社**オーム**社
〒101-8460 東京都千代田区神田錦町 3-1
TEL　(03)3233-0641（代表）
FAX　(03)3233-3440

Printed in Japan（ISBN978-4-87311-389-0）
落丁、乱丁の際はお取り替えいたします。