頑健なJavaプログラムの書き方 (Writing Robust Java Code)

The AmbySoft Inc. Coding Standards for Java v17.01d

著:Scott W. Ambler,　訳：高橋徹

Sun Aug 03 01:16:29 JST 2003

概 要

この文書は、Scott W. Ambler 氏によって記述された Writing Robust Java Code - The AmbySoft Inc. Coding Standards for Java v17.01d(2000.1.15) を訳出したものです。Writing Robust Java Code は、Java 言語のコーディングに関する標準、指針をまとめた文書です。翻訳に関する誤り・ご指摘等ありましたら、訳者へ連絡いただけると幸いです。(訳者：高橋徹 email:torutk@alles.or.jp)

目 次

序章

はじめに

この文書の目的

この文書は、頑丈で良質な Java プログラムを書くための標準・規約・指針について記述したものである。良い結果をもたらす実証されたソフトウェア工学の原理に則っているので、理解しやすく、保守しやすく、拡張しやすいソースコードを作成するのに役立つ。さらに、このコーディング標準に従うことによって Java プログラマーの生産性は著しく向上する。経験によれば、最初から高い品質のコードを書くことに時間を割けば、以降の開発工程の間コードを修正することが実に容易になる。結論として、共通のコーディング標準に開発チームが従うことによって一貫性が増し、極立って生産性が高まる。

この文書の重要な特色

- 産業界で作られた既存の標準を可能な限り使う-コードだけでなくもっと多くのものを再利用できる。
- なぜその標準に従うのかを理解できるようにするために、おのおのの標準の裏にある論拠を説明した。
- 他に代案がある場合は、それらの利点・欠点を示し、それらの間でトレードオフができるようにした。
- この文書で示す標準は、実際に行われたたくさんのオブジェクト指向開発プロジェクトにおける経験に基づいている。理論だけでなく、実践である。
- この標準は実証されているソフトウェア工学の原則に基づいており、開発生産性を高め、保守性を高め、拡張性を大きくする。

対象とする読者

以下の事柄に興味を持っているプロのソフトウェア開発者

- 保守しやすく、拡張しやすい Java コードを書くこと
- 自分自身の生産性を向上させること
- Java 開発チームにおいて生産性の高い一員としての役割を果たすこと

この標準をよりよくする助力の要請

読者からのフィードバックは歓迎なので、コメント・提案を scott@ambysoft.com[1]まで遠慮せずに送ってほしい。共に仕事し、お互いに学び合いましょう。

謝辞

以下の人々からこの標準を開発し改善するにあたり価値のある助力をもらったことをここに明記する。

Stephan Marceau, Eva Greff, Graham Wright, Larry Allen, John Pinto, Bill Siggelkow, Kathy Eckman, Kyle Larson, Mark Brouwer, Lyle Thompson, Wayne Conrad, Alex Santos, Dick Salisbury, Vijay Eluri, Camille Bell, Guy Sharf, Robert Marshall, Gerard Broeksteeg, David Pinn, Michael Appelmans, Kiran Addepalli, Bruce Conrad, Carl Zimmerman, Fredrik Nystrom, Scott Harper, Peter C.M. Haight, Helen Gilmore, Larry Virden, William Gilbert, Brian Smith, Michael Finney, Hakan Soderstrom, Cory Radcliff

[1]日本語ならば訳者 torutk@alles.or.jp

本文

第1章　一般概念

この文書を始めるにあたり、コーディング標準について重要と考えるいくつかの一般的概念について議論する。もっとも重要なコーディング標準として推奨する「最優先規範」[1]から始め、続いて良い命名・良いドキュメントへ導く要素を述べる。この章では、以降の Java コーディングのための標準・指針を述べていく文書のための舞台を整える。

1.1　なぜコーディング標準が重要なのか

Java のコーディング標準が重要であるのは、これが自分自身や仲間が作り出すコードの間に高い一貫性をもたらすからである。高い一貫性はコードを理解しやすくする。理解しやすさは言い換えると開発容易性と保守容易性である。これはアプリケーションの開発コスト全体を低減する。

あなたが書いた Java のコードは、あなたが他のプロジェクトへ移った後もずっと長い期間存在し続けることを忘れてはならない。開発を通じてもっとも重要な目標は、あなたの作業を他の人または開発チームに移管できることを保証することである。その結果、彼らはあなたの書いたコードを解析するという不当な労力を払わずに保守や拡張を続けることができる。理解し難いコードは、捨てられ書き直されることになる。もし皆が自分勝手にコーディングしていたら、開発者の間でコードを共有することができなくなるため、開発費用、保守費用を高騰させてしまうだろう。

経験の浅い開発者や改善の余地のないカウボーイプログラマーはしばしば標準に従うことに抵抗する。彼らは自分のやり方ならば早くコードを書けると主張するだろう。まったくナンセンスである。彼らは早くコードを仕上げると思い込んでいるが、疑わしい。カウボーイプログラマーはいくつか発見困難なバグに遭遇するとハングアップしてしまい、そして彼らの書いたコードを改良する必要が生じる度に自分の手でその大半を書き直すことになりがちである。なぜなら自分だけしか理解できないコードを書くからである。これが望むやり方だろうか？私はそうではない。

1.2　最優先規範

完全な標準など存在しないし、すべての状況に適用できる標準もない。大抵の場合、一つや二つは標準を適用できない状況がある。標準における最優先規範と考えるものを紹介する。

標準に従わないときは、それをドキュメントに書く　この最優先規範を除いたすべての標準は破られる。もし破られるのであれば、なぜ標準を破るのか、標準を破った結果内在する問題、どういう状況であればその標準が適用できるのかその条件を記述せよ。

好むと好まざるとに関わらず、それぞれの標準を理解し、いつ適用すべきかを理解し、同じく重要なことはいつ適用しないべきかを理解することが必要である。

[1]訳注：原語では The Prime Directive とある。この言葉は有名な SF 作品「スタートレック」において、他種族の命運を左右する行為を禁じた宇宙艦隊の規則である最優先規範として使用されている。

1.3 よい命名がもたらすもの

命名規約についてはこの文書を通して議論していく。そこでいくつかの基本を述べる。

1. 正しくフルスペルで英語[2]によって変数/フィールド/クラス/などを命名する。

   例えば firstName, grandTotal, CorporateCustomer のような命名を用いる。x1,y1,fn のような名前は短いためタイプしやすいが、それが何を表現するかについての指標がなにもないので、結果としてコードが理解・保守・拡張困難になる (Nagler,1995; Ambler,1998a)。

2. 業務分野の用語を適切に使用する。

   もしユーザが clients のことを customer と定義しているなら、そのクラスには Client ではなく Customer と命名する。　多くの開発者は、既にその産業/分野に相応しい用語があるにもかかわらず、一般的な用語を作り出してしまう誤りをおかす。

3. 大文字・小文字を使って読みやすい名前をつける。

   一般的には小文字を使う。複数の単語で一つの名前を構成するときは、名前の最初の 1 文字以外の単語の先頭を大文字とする。ただし、クラス名・インタフェース名については最初の 1 文字を大文字とする (Kanerva, 1997)。

4. 略語は使うなら控えめに、注意深く。

   略語を使用するときは、省略形の標準リストを作成し、賢く選び、一貫して使用すること。例えば、”number”という単語の省略形を使いたいなら、nbr, no, num から一つ選び、どれを選んだかをドキュメントし（実際どれを選ぶかは重要ではない)、それだけを使う。

5. あまりに長すぎる名前は使用しない (<15 文字はよい考え)。

   クラス名 PhysicalOrVirtualProductOrService はその時点ではよい名前に見えるかもしれないが、あまりに長すぎるので、何かより短いものに、Offering といったような名前に変更することを考えた方がよい (NPS,1996)。

6. ほんの些細な違いしかない名前を併用しない。

   例えば、persistentObject と persistentObjects とは同時に使うべきでない。同様に、anSqlDatabase と anSQLDatabase とも同時に使わない (NPS,1996)。

7. 標準的な頭字語の先頭文字だけ大文字にする

   名前には Standard Query Language の場合の SQL のように標準的な略語がある。属性名については sQLDatabase ではなく sqlDatabase と命名し、クラス名については SQLDatabase ではなく SqlDatabase と命名する方が読み易い。

1.4 よいドキュメントの仕方

ドキュメント規約についてもこの文書を通して議論していく。そこでいくつかの基本を述べる。

[2] この文書全体を通して使用しているが、英語の部分は開発チームが使用している言語に置き換わる

1. コードを明確化するドキュメントを書く

   コードにドキュメントを書く理由は、自分自身、一緒に仕事をしている人、後に関わる開発者にとってコードをより理解しやすいものにするためである (Nagler,1995)。

2. ドキュメントする価値がないプログラムならば、実行するに値しない (Nagler,1995)

   Nagler 氏の言は正鵠を射ている。

3. 過剰な装飾は使うな（例、見出し状コメント）

   1960 年代から 1970 年代の典型的な COBOL プログラマーにはアスタリスク (*) でコメントを囲った箱を書く習慣があった。彼らの芸術的な主張を表わしているのかもしれないが、率直に言えばそれは製品に加わるちょっとした価値に比べれば大きな時間の無駄である。かわいいコードではなくきれいなコードを書くはずである。さらに、コードを表示するディスプレイや印刷するプリントに使われるフォントはプロポーショナルだったりそうでなかったりして、箱をきれいに整列させることは難しい。

4. コメントはシンプルに

   かつて見たもっとも最良のコメントは、シンプルな要点をまとめた注釈であった。なにも本を書く必要はなく、他の人がコードを理解するに十分な情報を提供するだけでよいのである。

5. コードを書く前にコメントを先に記述する

   コードをドキュメント化する最良の方法は、コードを書く前にドキュメントすることである。それが、コードを書く前にコードがどのように動作するかについて考えるよい機会となり、同時にコードを書く前にドキュメントされることになる。そうでなければ、少なくともコードを書いた時にドキュメントするべきである。ドキュメントによってコードが理解しやすくなることで、コードの開発中にアドバンテージを得ることができる。コードにドキュメントを書く時間を費やすなら、少なくともそれによって得られるものがある (Ambler,1998a)。

6. コメントには、なぜそうなのかを書く。コードを読めば分かることを書かない

   基本的に、コードの一部分を見ればそれが何かを理解することはできる。例えば、以下の例 1.1 のコードを見て、$1000 以上の注文については 5%ディスカウントされることは理解できる。なぜそうなのか？大きな注文ではディスカウントがつきものだというビジネスルールがあるのだろうか？大きな注文に時間限定サービスがあるのか、それともずっとサービスがあるのか？これを書いたプログラマーの気前がよかったのか？　どこかソースコード中か別な文書にドキュメントされていない限り、なぜなのかを知ることはできない。

   リスト 1: 例 1.1

```
if (grandTotal >= 1000.00) {
    grandTotal = grandTotal * 0.95;
}
```

1.4.1 3 種類の Java コメントの使い分け

Java では 3 種類のコメントが使える。ドキュメント化コメントは/**で開始され、*/で終わる。C 風コメントは/*で開始され*/で終わる。単一行コメントは//で開始され、そのソースコード行が終わるまで続く。以下の表では私の提案するコメントの使い方とその例の抜粋を載せる。

コメント種類	使用方法	例
ドキュメント化コメント	ドキュメント化したい interface, class, メソッド, フィールドの直前に書く。ドキュメント化コメントは javadoc によって処理され、クラスの外部ドキュメントとして生成される。	/** 顧客（Customer）- 顧客はわれわれがサービスまたは製品を売った人物もしくは組織のいずれかである。 @author S.W.Ambler */
C 風コメント	特定のコードを無効化したいが、後で使用するかもしれないので残しておくためにコメント化する時や、デバッグ時に一時的に無効化するときに使用	/* このコードは J.T.Kirk によって 1997.12.9 に前述のコードと置き換えたためコメント化した。 2 年間不要であるならば、削除せよ。 ... (ソースコード) */

単一行コメント	メソッド内にて、ビジネスロジック、コードの概要、一時変数の定義等を記述	// 1995 年 2 月に開始された
		// サレク氏の寛大なキャン
		// ペーンで定められた通り
		// 1000$を超える請求には、
		// 全て 5%割引を適用する。

もっとも重要なことは、自分の組織で C 風コメントと単一行コメントをどのように使うべきかの標準を決め、その標準に一貫して従うことである。ひとつをビジネスロジックのドキュメントに使い、もうひとつを古いコードのコメントアウトに使うというように。私は、コードと同じ行にドキュメントできる（インライン）という理由から、単一行コメントをビジネスロジックに使うことにしている。そして C 風コメントを古いコードのコメントアウトに使用する。なぜならば、複数行を一度にコメントアウトでき、また C 風コメントはドキュメント化コメントと非常に似ているので、誤用を避けるためである。

1.4.2 javadoc の概要

Sun の Java 開発キット (JDK) に含まれている javadoc と呼ばれるプログラムは、Java コードが書かれたファイルを処理し、その Java プログラムについての HTML 形式の外部文書を生成する。javadoc はすばらしいプログラムだが、これを書いている時点においては、いくつかの制限がある。まず第 1 にタグの種類が少ないことである。既存のタグは確かによいものだが、コードを適切にドキュメントするには十分とはいえない。以下に javadoc タグの簡単な概要を示す。さらに詳しく知りたい場合は JDK javadoc のドキュメントを参照せよ。

コードにドキュメントする方法は自分自身の生産性と後にコードを保守し拡張する他のすべての人との両方にとって非常に大きな影響をもたらす。開発プロセスの早期にコードにドキュメントすることで、ロジックをコードに書きとめる前に十分考えさせられるため、生産性が向上する。さらに、数日もしくは数週間前に書いたコードを見直している時にも、コードを書いていた頃に何を考えていたか容易に判る。コードに既にドキュメントされているのである。

1.5 Ambler の標準の法則

可能な限り標準や指針を再利用し、再考案をしないこと。標準や指針の適用範囲は広ければ広いほど望ましい。工業標準は組織の標準より望ましいし、組織標準はプロジェクト標準よりも望ましい。プロジェクトは真空の中では開発できないし、組織は真空の中では運営できない。それゆえ、標準の適用範囲が大きいほど誰か他の人が後に続いてきて、ともに仕事するのが容易になるという可能性が大きくなる。

Tip-行末コメントにご用心

McConnel(1993) はインラインコメント（行末コメント）の使用を強硬に反対している。それはコードの右側にきちんと揃っておかれ、コードの視覚的な構造とは干渉していない点を指摘している。その結果、フォーマット作業が大変であり、たくさん使っていればそれだけ整列作業に時間を費やすことになる。そんな時間を費やしてもコードについて理解が深まるわけではなく、ただスペースキーやタブキーを叩くくだらない作業にささげるだけである。また、行末コメントは保守が大変である点を指摘している。ある行のコードの記述が増えて行末コメントの位置を超えてしまった場合、行末コメントの位置を揃えるためには残りの行末コメントについても同じ作業をしなければならない。そうではあるが、私からのアドバイスは、行末コメントを揃えることに時間を使うなということである。

タグ	使用対象	目的
@author name	クラス、インタフェース	コードの該当部分の著者を示す。著者ごとに1つのタグを使う。
@deprecated	クラス、インタフェース、メソッド	クラスの API が deprecated であり、今後は使うべきでないことを示す。
@exception name description	メソッド	メソッドが投げる例外を記述する。例外毎に1つのタグを使い、その例外のクラス完全名を書く。
@param name description	メソッド	メソッドに渡されるパラメータをその型と使用例を交えて記述する。
@return description	メソッド	メソッドの戻り値について記述する。型についてと、戻り値がどのように使われるかを示す。
@since	インタフェース、クラス、メソッド	この要素がいつから登場しているかを示す。すなわち since JDK1.1
@see ClassName	クラス、インタフェース、メソッド、フィールド	文書中に指定されたクラスへのハイパーリンクを生成する。完全限定名を使えるし、使うべきである。
@see ClassName#メソッド名	クラス、インタフェース、メソッド、フィールド	文書中に指定されたメソッドへのハイパーリンクを生成する。完全限定名を使えるし、使うべきである。
@version text	クラス、インタフェース	コードの該当部分についてのバージョン情報を示す。

Ambler の標準の法則

工業標準＞組織標準＞プロジェクト標準＞個人標準＞標準なし

第2章　メソッドに関する標準

私はシステムの専門家が生産性を最大なものにすると固く信じている。アプリケーションは存在する期間の大半が開発ではなく保守期間であるとの認識から、私のコードを開発しやすくするだけではなく、保守しやすく、拡張しやすくするのに役立つ事柄に非常に興味を持っている。忘れてはならないことは、今日書いたコードが今から何年もの後まで使われ続け、きっと他の誰かによって保守され、拡張されていることである。コードは可能な限りきれいに（clean) そして理解しやすいように作る努力をしなければならない。なぜならば、これらのことは保守と拡張を容易にするからである。

この章では 4 つの話題について扱う。

- 命名規約
- 可視性
- ドキュメント規約
- クリーンな Java コードを書くための技術

2.1　メソッドの命名

メソッドは正しくフルスペルの英語で、先頭の単語が小文字で，先頭以外の各単語の頭は大文字となるように命名しなくてはならない。先頭の単語は能動態の動詞とするのが一般的な実践である。

リスト 2: 例）

```
openAccount(), printMailingLabel(), save(), delete()
```

この規約によってメソッド名を見ただけで、目的が分かるようになる。この規約によって名前が長くなるため開発者のタイプ量が増えるが、コードの理解しやすさが向上することによって埋め合わせ以上のものがもたらされる。

2.1.1　アクセッサ・メソッドの命名

フィールド（クラスの属性・プロパティ）に値を設定したり（Setter) 読み出したり (Getter) するためのメソッドに関する命名規約であり、詳細は後の章で述べるが、ここにまとめを載せる。

読み出しメソッド (Getter)

フィールドの値を返却値とするメソッドである。フィールド名の前に'get' を付け加える。ただしフィールドの型が boolean であるときは、get の代わりに'is' を付け加える。

リスト 3: 例

```
getFirstName(), getAccountNumber(), isPersistent(), isAtEnd()
```

この命名規約に従うことで、メソッドはフィールドのオブジェクトを返却していることが明確になり、boolean 型の getter については true か false を返却することが明確になる。この標準のさらなる利点は、JavaBeans 開発キット (BDK) の getter メソッドの規約に則っていることである。不便な点は、せいぜい'get' が余分なタイプ量を必要とすることくらいである。

英語表現としては、'is' ではなく、has や can とすることが正しい文法である命名がある。例えば、hasDependents() とか canPrint() とかがそうである。ただし、JavaBeans の仕様では has や can をアクセッサメソッドとは認識しない点が問題である。その場合、isBurdenedWithDependents() や isPrintable() のような命名に変更する方法もある。

設定メソッド (Setter)

フィールドの値を変更するメソッドであり、mutator とも呼ぶ。フィールドの型にかかわらずフィールド名の前に'set' を付け加える。

リスト 4: 例)

```
setFirstName(String aName)
setAccountNumber(int anAccountNumber)
setReasonableGoals(Vector newGoals)
setPersistent(boolean isPersistent)
setAtEnd(boolean isAtEnd)
```

この命名規約に従うことで、メソッドはフィールドのオブジェクトを設定していることが明確になる。この標準のさらなる利点は、JavaBeans の setter メソッドの規約に則っていることである。不便な点は、せいぜい'set' が余分なタイプ量を必要とすることくらいである。

コンストラクタ

オブジェクトを生成するときに必要な初期化作業を行うメソッドであり、常にクラス名と同じ名前になる。例えば、Customer クラスのコンストラクタは Customer() となる。大文字小文字も同一である。

リスト 5: 例)

```
Customer()
SavingsAccount()
PersistenceBroker()
```

この命名規約は Sun によって設けられ、厳格に従わねばならない。

2.2 メソッドの可視性

クラス同士の結合を極小化するというよい設計を行うために、メソッドの可視性を設定するときは可能な限り制限を強くする。public にする必要のないメソッドは protected にする。protected にする必要のないクラスは private にする。

可視性	内容	使用方法
public	どのオブジェクト／クラスからでも呼ぶことができる	そのクラスのクラス継承階層に属さない外のオブジェクトから呼ばれる必要があるときに定義する
protected	このクラスおよびサブクラスから呼ぶことができる	クラス階層の中で必要とされる振る舞いを提供するときに使う
private	このクラスの中から呼ぶことはできるがサブクラスから呼ぶことはできない	クラスに特有の振る舞いを提供するときに使う。しばしばリファクタリングの結果作られ、ある特定の振る舞いをクラス内にカプセル化するときに使用
	可視性が指定されない。デフォルトあるいはパッケージ可視と呼び、時にフレンド可視と引用される。メソッドは同一パッケージに属する他のクラスには public 同様だが、パッケージ外のクラスには private と同様である	興味深い特徴だが注意深く使うこと。顧客といったビジネス概念を実装するクラス集であるドメインコンポーネント (Ambler、1998b) を構築した際に、コンポーネントのパッケージ内のクラスだけがアクセスできるよう制限するために使用した。

2.3 メソッドのドキュメント

メソッドのドキュメントの仕方は、理解しやすいかし難いかによって、保守性と拡張性の決定要因となる。

2.3.1 メソッドの先頭に書くコメント

全ての Java メソッドはメソッドコメントと呼ばれる何らかの種類のヘッダを含むべきである。ソースコードの先頭に、メソッドを理解するのに必須となる情報を書く。この情報の内容として書くべき項目例を以下に示すがこれだけに限らない。

1. メソッドが何をしているのか、なぜそのように書かれたか

   第三者がそのコードを再利用できるか否かを判断しやすいように、メソッドが何をするのかドキュメントする。第三者がそのコードがどういった文脈で使われるか理解しやすいように、メソッドがなぜそうしているのかドキュメントする。また、そのようなドキュメントがあれば第三者が新たに変更が必要となるかどうか判別しやすくなる。（新たに変更する理由は、最初にコードを記述したときの理由と矛盾が生じた結果である）

2. メソッドの引数として渡さなければならないパラメータが何か

   引数を使用しているならば、メソッドに渡さねばならない引数が何であってどのようにメソッド内で使われるかを記述する。この記述は他のプログラマーがメソッドにどんな情報を渡せばよいか知るのに必要となる。これには 1.4.2 節で論じた javadoc の@param タグを使用する。

3. メソッドが何を返却値として返すか

   返却値があるならば、メソッドが返す返却値/オブジェクトを他のプログラマーが適切に使うことができるように記述する。これには 1.4.2 節で論じた javadoc の@return タグを使用する

4. 既知のバグ

   メソッドの未解決の問題点をドキュメントし、他の開発者がそのメソッドの制約を理解できるようにする。もしそのバグが 1 つのメソッドに収まらないものであるなら、メソッドドキュメントではなくクラスドキュメントの方に記述する。

5. メソッドがスローする例外

   メソッド内で発生するすべての例外をドキュメントし、他のプログラマーにどの例外を補足しなければならないか分かるようにする。これには 1.4.2 節で論じた javadoc の@exception タグを使用する

6. 可視性の決定

   メソッドの可視性を選んだ理由が他のプログラマーにとって疑問になるかもしれないと思うなら、その決定理由をドキュメントしておく。例えば他のオブジェクトからは呼ばれないメソッドを public な可視性にしたときかもしれない。こうすることにより、あなたの思考が他のプログラマーにも明白となり、なぜあなたが疑問を生じることをしたのか悩まずにすむ。

7. メソッドがどのようにオブジェクトを変更するか

   メソッドがあるオブジェクトを変更するとき、例えば銀行口座の withdraw() メソッドが口座残高を変更する場合、このことを記述する。これによって他のプログラマーがそのメソッドを起動すると対象オブジェクトにどのような影響を及ぼすかを正確に知ることができる。

8. 変更履歴を記述する

   メソッドを変更するときはつねに、いつ変更したか、誰が変更したか、なぜ変更したのか、誰が変更要求を行ったか、変更結果の試験は誰が行ったか、いつ試験して製品に組み込む検証を行ったかを記述する。この履歴情報は、コードを保守し拡張する責務を帯びた保守プログラマーのために重要である。注記：この情報は本来ソフトウェア構成管理・履歴管理システムに属するもので、ソースコード上に書かれるものではない！こうしたツールを使わないのならば、ソースコードに書く。

9. メソッドの使用例を記述する

   コードがどのように動作するかを理解する簡単な方法の１つが使用例を見ることである。メソッドをどのように使用するか１つか２つの例を書くことを考慮する。

10. 影響ある事前条件・事後条件

    事前条件はメソッドが正しく機能するための制約であり、事後条件はメソッドの実行が完了した後に真となる特性もしくは表明である（Meyer,1988)。事前条件と事後条件によってメ

ソッドを書いているときに下した仮定を記述し、メソッドをどのように使用するのか正確な範囲を定義する。

11. 並行性について記述する

    並行性は大部分の開発者にとっては新しい複雑な概念であり、並行プログラミングの経験を積んだプログラマーにとってもせいぜいなじみはあるものの複雑な問題である。要するに、Java の並行プログラミング特性を使用したならば、徹底的にドキュメントする必要があるということだ。Lea 氏は次のように提唱している (1997)。クラスが同期メソッドと非同期メソッドを両方含んでいる場合、他の開発者がそのメソッドを安全に使えるようにするために、メソッドが想定している実行時のコンテクストをドキュメントする。とりわけそのメソッドが制約無しにアクセスされる必要がある場合はそうである。Runnable インタフェースを実装するクラスにおけるフィールドを更新する Setter メソッドが synchronized でない場合は、なぜそうしたのか理由をドキュメントするべきである。最後に、メソッドをオーバーライドまたはオーバーロードし、その同期性を変更するときは、なぜそうしたのかドキュメントする。

ドキュメントする際に重要な点は、コードが明瞭になることだけを記述することである。上記に挙げたすべての項目をそれぞれのメソッドに記述しないこと。なぜならば、すべての項目がどのメソッドにも適用できるわけではないからである。それぞれのメソッドについて上記の項目のいくつかをドキュメントする。9 章で、上述のリストをサポートする javadoc のタグをいくつか提案する。

2.3.2 メソッドのコード中に書くコメント

メソッドドキュメントに加えて、メソッド内にドキュメントを書く必要もある。その目的はメソッドを理解・保守・拡張しやすくすることである。

メソッド内ドキュメントに使うコメントには 2 種類ある。C 風コメント (/* ... */) と単一行コメント (//) である。1.4.1 節で述べたとおり、ビジネスロジックを記述するために使うコメント種類と不必要なコードをコメントアウトするために使うコメント種類を十分考慮する。私の推奨は、単一行コメントをビジネスロジックの記述に使うことである。単一行コメントは、行全体をコメントするときにもコードの後ろに記す行末コメントにも使えるからである。C 風コメントは不要なコードをコメントアウトするときに使う。一つのコメントで複数行をコメントアウトできるからである。さらに、C 風コメントはドキュメント化コメントに大変類似して見えるので、この使用は混乱を招き、コードの理解容易性を減少させてしまう。それゆえ、なるべく控えめに使うことにしている。

内部コメントとして書くべき項目例を以下に示す。

1. 制御構造

   比較文、ループなどの制御構造が何をしているのか記述する。制御構造の中のソースコードをすべて読まなくても、代わりに 1,2 行のコメントを見てすぐに何をしているか分かるようにする。

2. なぜかを何をしているかとともに書く

   コードのある部分を見れば、何をしているのかは理解できる。しかし、コードがなぜそのように書かれたのかを理解できることはほとんどない。例えば、あるコードを見て注文合計の 5%割引が適用されていることは簡単に分かる。それは実に簡単だ。難しいのは、「なぜ」そ

の割引が適用されるかを理解することである。何らかのビジネスルールがあって割引を適用しているのは確かなので、コード中でそのビジネスルールについて記述し、他の開発者がなぜコードがそうなっているのか理解できるようにする。

3. ローカル変数

   4 章ではるかに詳しく論じるが、メソッド内で定義されるローカル変数はそれぞれ単一行で宣言し、その使用方法を記述するコメントを行う。

4. 複雑だったり難しいコード

   コードを書き直せない、あるいは書き直す時間がないときは、メソッド内の複雑な部分を完璧にドキュメントする。経験に基づく大まかな指針として、コードが明白でないならばドキュメントする必要がある。

5. 処理順序

   コードの文が必ず定義された順序で実行されねばならないならば、その守らねばならない事項をコメントに記述する。コードが機能していないことを調べるためだけにコードに単純な修正を加え、それから何時間も費やして単にコードの順序が違っていたことを見つけるほど最悪なことはない。

2.4 クリーンなコードを書く技術

この節では、質の低いコーダーとプロの開発者との違いをもたらす技術について論じる。

- ソースコードにドキュメントを書く
- ソースコードを段落化する
- 複数行の命令を分解する
- 空白を有効活用する
- 30 秒ルールを守る
- メッセージ送信の順番を定義する
- 簡潔に、一行にはひとつのコマンドを書く

Tip-閉じ括弧にドキュメントする

しばしば制御構造の中にある制御構造の中に制御構造があることを見つけるだろう。大抵はこのようなコードは避けようとするが、こう書くことがよい場合もある。このとき、どの閉じ括弧'}' がどの制御構造のものなのか混乱してしまうという問題が生じる。ある種のエディターは、括弧の対応を自動的に示してくれる機能を持っているが、すべてのエディターが持っているわけではない。そこで、閉じ括弧にインラインコメントとして、//end if, //end for, //end switch,... のように書くことによってコードが理解しやすくなる。
しかし、どちらかを選択するとしたら、私は洗練されたエディタを使う方を選ぶだろう。

2.4.1 コードにドキュメントを書く

「ドキュメントする価値のないコードは維持する価値がない (Nagler,1995)」ことを覚える。この文書で提案している規準、指針を適切に適用するならば、コードの品質を大いに向上することができる。

2.4.2 コードを段落化する

メソッドの読みやすさを向上する方法の一つが段落化であり、言い換えればコードブロックの単位でインデントすることでもある。括弧（'{' と'}'）で囲まれるコードはブロックを形成する。基本となる考え方は、ブロック内のコードは１単位[1]のインデントを行う。一貫性を維持するため、メソッドとクラスの宣言をカラム 1 から開始する。(NPS,1996)

Java のしきたりでは、開き括弧は以降のブロックの主体となる行に置かれ、閉じ括弧は第 1 レベルインデントに置かれる。Laffra 氏 (1997) によって指摘された重要な点は、組織において一つのインデント流儀を決定し、それに従いつづけることである。私のアドバイスとしては、Java 開発環境が生成する Java コードと同じインデント流儀を適用することである。

2.4.3 段落と複数行の命令

コードを段落化する際に、単一の命令を書くのに複数の行を必要とすることがある。例を以下に挙げる。

リスト 6: 例

```
BankAccount newPersonalAccount = AccountFactory.
        createBankAccountFor( currentCustomer, startDate,
        initialDeposit, branch);
```

2 行目と 3 行目を 1 単位のインデントにしたことにより、それらが見た目で継続している行であることに気づくだろう。2 行目の最後のカンマによって次にパラメータが続くことにすぐに気づく。

2.4.4 空白を使用する

幾行かの空白行をコードに使うことで小さな、把握しやすい部分に分解でき、非常に読みやすくなる (NPS,1996; Ambler,1998a)。Vision2000 チーム (1996) は、1 行の空白行を使うことでコードを制御構造等の論理的なグループに分け、2 行の空白行でメソッドの定義を分けることを提唱している。空白がなくては読みずらいし理解しずらい。以下のコードにおいて、カウンタと合計の計算行との間に空白行を追加したことによって 2 番目のものが読みやすく改善されたことに気づく。演算子の前後とカンマの後にある空白が追加されたことによってコードの読みやすさが向上したことが分かる。

リスト 7: コード例 (1)

[1]単位とは何かについては議論が紛糾するだろう。ただし私にとって単位が意味する唯一のことは水平方向の tab である。これは同時に十分なインデントを与えるのにもっとも少ないタイプ量で済む。スペースを使用することは常に問題を招く、ある者は 2 スペース、ある者は 3 スペース、ある者は 4 スペースなどなど。tab は実に簡単である。エディターの中には tab をスペースに変換してしまったり（げぇーっ）、tab を全然サポートしていないものがある。きちんとしたコード用エディタに費やすように。

```
counter=1;
grandTotal=invoice.total()+getAmountDue();
grandTotal=Discounter.discount(grandTotal,this);
```

リスト 8: コード例 (2)

```
counter = 1;

grandTotal = invoice.total() + getAmountDue();
grandTotal = Discounter.discount(grandTotal, this);
```

2.4.5 30 秒ルールに従う

私は常に、他のプログラマーが 30 秒以内でメソッドを読むことができ、何をしているか、なぜそうなっているのか、どのようにしているかを完全に理解できるべきだと信じている。もしそうできなかったとしたら、そのコードは保守するにはあまりに難しいため改善の必要がある。30 秒でである。Stephan Marceau 氏によって提唱されているよいルールに、メソッドが画面に表示しきれないならば、それはおそらく長すぎる、というものがある。

2.4.6 簡潔に、一行にはひとつのコマンドを書く

1 行では 1 つのことだけをする (Vision,1996;Ambler,1998a)。パンチカードの時代に遡れば、1 つの行に可能な限りの機能を詰め込むことに意味はあったが、私が最後にパンチカードを見てから 15 年以上もたっていることだし、この詰め込み方法は考え直した方がいい。1 つの行に 1 つより多いことをさせようとする試みはすべてコードの理解をより難しくする。なぜこうするのか。コードを理解容易にして保守と拡張を容易にしたいからである。メソッドが一つのことを実行し、その 1 つだけを行うようにするのと同様に、1 行のコードでは 1 つのことだけをする。さらに、画面で見られるようにコードを書く (Vison,1996)。インラインコメントを使っているときも含め、編集ウィンドウを右にスクロールしなくても済むようにする。

2.4.7 演算の順番を定義する

コードの理解性を改善する本当に簡単な方法は、正確な演算順序を指定するように括弧を使うことである (Nagler,1995;Ambler,1998a)。コードを理解するためにその言語の演算順序を知らねばならないとしたら、何か重大な誤りがある。大部分が AND と OR をいくつか別の比較と一緒に使ったときに生じる問題である。上述の、”簡潔に、一行にはひとつのコマンドを書く”ルールを使用していれば、この問題は生じない。

2.5 Java コーディングのこつ

本節では、何年かの間に見つけたソースコードの品質を向上させる指針をいくつか記す。

2.5.1 意味的にコードを構成する

以下の 2 つのコードを比較すれば、anObject を含むステートメントがひとまとまりになっている右側の方が理解しやすい。コードは同じものだが、読みやすくなっている。(注：もし aCounter が message3() のパラメータとして渡されるとしたら、この変更はできない)

リスト 9: コード例 (1)

```
anObject.message1();
anObject.message2();
aCounter = 1;
anObject.message3();
```

リスト 10: コード例 (2)

```
anObject.message1();
anObject.message2();
anObject.message3();

aCounter = 1;
```

2.5.2 定数を比較文の左側に置く

以下のコードを考えてみる。パッと見は両者はどちらも等価だが、左側のコードはコンパイルできるが右側はコンパイルできない。何故か？2 番目の if 文が比較ではなく、代入文であり、0 のような定数に新たな値を代入することができないからである。このようなバグをコード中から見つけるのは困難である (少くとも洗練されたテストツールなしでは)。比較文の左側に定数を置くことによって、同じ効果を達成できコンパイラは比較ではなく代入を誤って使ってしまったことを検出する。

リスト 11: コード例 (1)

```
if (something == 1) {...}

if (x = 0) {...}
```

リスト 12: コード例 (2)

```
if (something == 1) {...}

if (0 = x) {...}
```

第3章　フィールド(属性/プロパティ)標準

この文書全体を通してフィールド (field) という用語を、属性 (attribute) と示す語として使用する。Beans Development Kit(BDK) ではプロパティ(property) と呼ばれる (DeSoto,1997)。フィールドはオブジェクトまたはクラスを記述するデータの一部分である。フィールドは、string や float といった基本型かもしくは customer とか bank account のようなオブジェクトである。

3.1　フィールドの命名

3.1.1　フィールドの命名には完全な英語記述を使う

フィールドの命名には完全な英語の記述を使用 (Gosling,Joy,Steele,1996; Ambler 1997) して、そのフィールドが表現していることが明確になるようにする。配列やベクタのような集合を表わすフィールドについては、複数の値があることを示す複数形の名前を与える。

リスト 13: 例)

```
firstName, zipCode, unitPrice, discountRate, orderItems,sqlDatabase
```

フィールドの名前が sqlDatabase のように頭字語[1]で始まるとき、頭字語 (ここでは'sql') は全て小文字とすべきである。sQLDatabase のような名前を使わない。

別な命名-ハンガリアン記法

ハンガリアン記法 (McConnell,1993) はフィールドを以下の方法で命名するという原理に基づいている。xEeeeeeEeeeee、x はコンポーネントの型を示し、EeeeeeEeeeee は完全英語記述を示す。

リスト 14: 例

```
sFirstName, iZipCode, lUnitPrice, lDiscountRate, cOrderItems
```

主な利点はこれが C++コードに関して工業規準共通になっており、多くの人が既にこれにしたがっている点である。さらに、開発者は変数の名前からその型とどのように使われるかを素早く判断できる。主な欠点は同じ型の属性をたくさん使うときに接頭詞の記法が負担になること、完全英語記述という命名規約を破ることになること、およびアクセッサ・メソッドの命名戦略に影響を及ぼすことである (3.4.1 節参照)。

[1]訳注：複数の語の頭文字を並べた語。ここでは SQL。

別な命名-先頭または末尾のアンダースコア

C++コミュニティから来た一般的なアプローチはフィールド名の先頭か末尾にアンダースコアを含めることである。

リスト 15: 例

```
_firstName, firstName_
```

このアプローチの利点はフィールド名を扱っていることが一目で分かるため、パラメータやローカル変数によって隠蔽されることを防ぐことができる (繰り返しになるが、この場合の名前隠蔽はアクセッサ・メソッドを使用するなら問題にはならない)。主な欠点は、これが Sun による標準セットではないことである。

3.1.2 コンポーネント (ウィジェット) の命名

コンポーネント (インタフェースウィジェット) の命名には、ウィジェットの型を名前の後ろに付加した完全な英語記述を使用する[2]。コンポーネントの使用目的と型が名前を見ただけて分かるようになり、またコンポーネント一覧からどれか選びやすくする（多くのビジュアル開発環境ではアプレットやアプリケーションの中で使われるコンポーネントの一覧表示を提供しており、そこに button1, button2... と表示されていたなら混乱してしまう)。

リスト 16: 例

```
okButton, customerList, fileMenu, newFileMenuItem
```

別なコンポーネント命名-ハンガリアン記法

リスト 17: 例

```
pbOk, lbCustomer, mFile, miNewFile
```

利点は 3.1.1.1 節で述べられたものと同様である。主な欠点は同じ型のウィジェットをたくさん使うときに接頭詞の記法が負担になることである。

別なコンポーネント命名-接尾詞ハンガリアン記法

基本的には他の 2 つの案の組み合わせであり、okPb, customerLb, fileM, newFileMi のようになる。主な利点はコンポーネントの名前がウィジェットの型を示すことと同じ型のウィジェットがアルファベット順のリストで互いにグループ化されないことである。主な欠点は完全英語記述を使っていないので、規準から逸脱してしまい、規準を覚えるのが困難になることである。

[2] これは私の規準であり、Sun が奨励しているものではない。

3.1.3 定数の命名

Java では定数は変更できない値であり、典型的にはクラスの static final フィールドとして実装される。よく知られた規約としては、フルスペルの英単語を用い、すべて大文字で、単語間はアンダースコア'_' でつなぐ (Gosling,Joy,Steele,1996;Sandvik,1996;NPS,1996)。

リスト 18: 例）

```
MINIMUM_BALANCE, MAX_VALUE, DEFAULT_START_TIME
```

この規約の主な利点は定数を変数と区別するのに役立つことである。この文書の後で触れるが、定数を定義するのではなく、定数の値を返却値として返すような読み出し (getter) メソッドを定義することで、コードの柔軟性と保守性を高めることができる。

3.1.4 集合（コレクション）の命名

配列、Vector、などの集合を表わすフィールドは、中にしまわれるオブジェクトの型を複数形にした名前を使用する。名前には、完全英語記述で、先頭の単語が小文字、先頭以外の各単語の頭は大文字を使用する。

リスト 19: 例）

```
customers, orderItems, aliases
```

この規約の主な利点は単一の値を保持するフィールドと複数の値を保持する集合を表わすフィールドとを区別しやすい点にある。

別な集合の命名-'Some' アプローチ

標準ではないアプローチだが興味深いものとして、集合名の接頭詞に'some' を付けるものがある。

リスト 20: 例

```
someCustomers, someOrderItems, someAliases
```

Tip-コンポーネント名標準の設定

どの規約を選んだとしても、公式のウィジェット名一覧を必要とするだろう。例えば、ボタンを命名するときに Button か PushButton、それとも b か pb かを使いますか？リストを作成し、組織内の Java 開発者に入手可能なようにする。

3.1.5　名前を隠蔽しない

名前隠蔽とは、ローカル変数、引数あるいはフィールドの命名において、それより大きなスコープの他の変数と同じ (または類似) 名前を付けてしまうことを示す。例えば、firstName というフィールドがある場合、ローカル変数やパラメータに firstName と名前を付けない。また、firstNames や fistName といった近い名前を付けない (もしかすると誰かが自分の拳[3]に名前を付けているかもしれない、僕自身は自分のには単に”右”と”左”と呼ぶけどね（笑）)。他の開発者や自分自身も含めコードを修正しているときに誤読し、エラーを見つけるのが困難になるので、コードを理解するのが難しくバグを招きやすいこのような名前の隠蔽は避けること。

3.2　フィールドの可視性

Vision チーム (1996) はカプセル化の理由からフィールドを public として宣言しないよう提唱している。私はさらにフィールドはすべて private として宣言するべきであると述べる。もしフィールドが protected として宣言されていると、サブクラスのメソッドが直接そのフィールドにアクセスする可能性があり、クラス階層の間で結合度がかなり増加する。これはクラスの保守と拡張をより困難にしてしまうので、避けるべきである。フィールドは直接アクセスしてはならず、代わりにアクセッサ・メソッドを使うべきである。

可視性	内容	使用方法
public	他のどのクラス／オブジェクトに属するメソッドからでもアクセス可能	フィールドは public にしないこと
protected	そのクラス自身のメソッドまたはサブクラスのメソッドからアクセス可能	フィールドは protected にしないこと
private	そのクラス自身のメソッドからのみアクセス可能で、サブクラスからはアクセス不可能	フィールドは private とし getter または setter メソッドによってアクセスすること

フィールドが永続性を持たないときは、static または transient 修飾子を付加する (DeSoto,1997)。これは BDK の規約に適合する。

3.3　フィールドのドキュメント

それぞれのフィールドには、他の開発者が理解できるに十分なドキュメントをコメントに記述すること。ドキュメントとして記述する内容は、

1. それ自身の記述

   どのように使うか他の開発者に分かるように記述する。

2. 適用できるすべての不変条件をドキュメントする

   フィールドの不変条件はそれについて常に真となる条件のことである。例えば、フィールド dayOfMonth についての不変条件はその値が 1 から 31 の間を取る（明らかにこの不変条件

[3]訳注：fist は拳の意味

をより複雑して年と月に基づいたフィールド値となるように制約することもできる)。フィールドの値の取りうる制約事項をドキュメントすることで、重要なビジネスルールを定義するのに役立ち、コードがどのように動作するか理解しやすくする。

3. 使用例

   複雑なビジネスルールに関わるフィールドについては、それを理解容易にするために何通りかの使用例を提供する。この例は絵で書いてもよい。百聞は一見に如かずである。

4. 並行性

   並行性は大部分の開発者にとっては新しくまた複雑な概念であり、並行プログラミングの経験を積んだプログラマーにとってもせいぜいなじみはあるが複雑な問題である。要するに、Java の並行プログラミング特性を使用したならば、徹底的にドキュメントする必要があるということだ。

5. 可視性の決定

   フィールドを private 以外の可視性として宣言したならば、なぜそうしたかを必ずドキュメントする。フィールドの可視性については前に述べたとおりであり、アクセッサ・メソッドを使用してカプセル化を行うことについては後の章で述べる。要点は、private 以外の宣言を行うならば、それを行うだけの十分な理由を持っていること。

3.4 アクセッサメソッドの使用

フィールドについての保守性にとって、フィールドの命名規約だけでなく適切なアクセッサメソッドの使用が不可欠である。アクセッサメソッドは、フィールドの値を更新したり読み出したりするためのメソッドのことである。アクセッサメソッドにはこの２つの目的によって setter(mutator とも呼ばれる) と getter とに分類できる。setter は変数の値を変更し、getter は変数の値を取得する。

アクセッサ・メソッドを使用することでコードにオーバーヘッドが生じていたけれども、今日の Java コンパイラはアクセッサ・メソッドの使用を最適化するので、もはや真ではない。アクセッサは、クラスの実装方法詳細を隠蔽する。変数にアクセスする個所を多くても２つ（setter と getter）に制限することによって、コードを変更する時の影響範囲を極小化することができるため、保守性が向上する。Java コードの最適化については 7.3 節で議論する。あなたの組織において実施できる重要な標準の一つがアクセッサの使用である。開発者の中には余計なタイプ量（例えば getter の場合フィールド名が増えるだけでなくさらに'get' と'()' も）を嫌ってアクセッサ・メソッドを使いたがらない者がいるかもしれない。要点は、アクセッサを使うことで保守性と拡張性が増加するということである。

Tip-アクセッサはフィールドにアクセスする唯一の場所である

アクセッサ・メソッドを適切に使用する鍵となる概念は、フィールドを直接操作できるメソッドはアクセッサ・メソッド自身だけであるということである。確かにフィールドが定義されるクラスのメソッドからは、private なフィールドであっても直接アクセス可能であるが、クラス内の結合度を増大させることになるのでそうしないこと。

3.4.1 アクセッサメソッドの命名

フィールドの値を取り出すGetterメソッドは、「'get'＋フィールド名」と命名する。ただし、フィールドがbooleanを表現する場合は、「'is'＋フィールド名」と命名する。フィールドの値を更新するSetterメソッドは、フィールドの型によらず「'set'＋フィールド名」と命名する (Gosling,Joy,Steele,1996;DeSoto,1997)。フィールド名は大文字小文字混合で構成し、フィールド名を構成する各単語の最初の文字を大文字とする。この命名規約はJDKにおいて使われている方法とBean開発に要求されていることに一致している。

表 3.1: 命名例

Field	Type	Getter名	Setter名
firstName	String	getFirstName()	setFirstName()
address	Object	getAddress()	setAddress()
persistent	boolean	isPersistent()	setPersistent()
customerNumber	int	getCustomerNumber()	setCustomerNumber()
orderItems	Array of Object	getOrderItems()	setOrderItems()

3.4.2 アクセッサ・メソッドの応用技術

アクセッサを単にインスタンス・フィールドの値の読み出し・更新だけに使うだけでなく、コードの柔軟性を高めるより効果的な使用方法がある。

- フィールドの値の初期化
- 定数へのアクセス
- 集合へのアクセス
- 複数のフィールドへ逐次アクセス

フィールドの怠惰な初期化（Lazy Initialization）

変数は使用される前に初期化されている必要がある。初期化の方法は２つの考え方があり、第一の考え方はオブジェクトが生成されるときに全変数を初期化するもの（traditional approach)、第二の考え方はオブジェクトが最初に使われるときになって変数を初期化するというもの。最初の方法はオブジェクトが最初に生成されるときに起動される特殊なメソッド、すなわちコンストラクタを使う。この方法ではエラーを招きやすいことがある。新たな変数を追加する作業時に、コンストラクタを更新することを忘れがちとなる。もう一つの方法はlazy initializationと呼ばれる、フィールドをGetterメソッドによって初期化するものである。下記コード参照[4]。メソッドがbranch numberがゼロかどうかチェックし、ゼロならば適切な初期値を設定する。

[4]注：getterメソッドの中でsetterメソッドが使われる方法

```
/**
  Answer the branch number, which is the leftmost four digits
  of the full account number. Account numbers are in the
  format BBBBAAAAAAA.
 */
protected int getBranchNumber() {
    if (branchNumber == 0) {
        // The default branch number is 1000, which is the
        // main branch in downtown Bedrock.
        setBranchNumber(1000);
    }
    return branchNumber;
}
```

この lazy Initializatoin は、フィールドをデータベースに格納しているオブジェクトの場合などでは非常によく使われる方法である。例えば、新しい在庫品を生成するとき、デフォルトとして設定した在庫品型をデータベースから取り出す必要はない。代わりに lazy initialization を使って、最初にアクセスされたときにその値を設定し、必要なときだけデータベースから在庫品型を取り出す。この方法は、頻繁にアクセスされないフィールドを持つオブジェクトに利点をもたらす。使わないのに永続ストレージから何かを引き出すオーバーヘッドを招くのでしょうか？lazy initializaiton を getter メソッドで使うときは必ずデフォルト値は何なのか、上述コード例で見たようにドキュメントする。ドキュメントすることでコード中のフィールドがどのように使われているかという疑問を払拭し、保守性と拡張性の両方を高める。

定数へのアクセス

一般的な Java の知恵（多分知恵というのは誤った用語だろう）では、定数を static final なフィールドとして実装する。この方法は「定数」が安定していると保証されている場合に限り意味をなす。例えば、Boolean クラスは 2 つの static final フィールドでそのクラス自身のインスタンスでもある TRUE と FALSE を実装する。また、DAYS_IN_A_WEEK 定数の値もおそらく決して変更されない[5]ため意味をなす。

しかしながら、ビジネスに関する定数はビジネスルールの変更によって何度も変更されうる。以下に例を考えてみる。Archon Bank of Cardassia(ABC) では利息を得るには口座に少なくても$500 以上の残高がなくてはならない。この実装として、Account クラスに static なフィールド MINIMUM_BALANCE を追加し、利息を計算するメソッドの中で使うやり方がある。この実装は動作はするが柔軟性がない。ビジネスルールが変更され、別な種類の口座には別な最低残高が適用され、例えば普通預金口座では$500 だが当座預金口座では$200 となった場合はどうなるのだろうか？また、ビジネスルールが変更され、初年度が最低残高$500、2 年目には$400、3 年目には$300、… となったらどうだろう？夏場は$500 で冬場は$250 となる場合はどうだろう？[6]最後にはこれらルールのすべてを実装するはめになるだろう。

定数をフィールドとして実装することは柔軟性を欠くということがポイントである。よりよい解決法は、定数を Getter メソッドとして実装することである。上述の例では、static(クラス) メソッド getMinimumBalance() が static フィールド MINIMUM_BALANCE よりもはるかにずっと柔軟

[5]私はどの文化も一週は 7 日間であると仮定しているが確かかは分からない。十分多くの国際化アプリケーションの開発に巻き込まれてきた結果、この仮定が本当に正しいかは検証が必要であることを知った。

[6]やあ、私はカナダ人だ。実際起きたんだよ。

である。なぜならば、様々なビジネスルールをメソッドに実装したり様々な種類の口座を適切にサブクラス化することができるからである。

```
/**
  Get the value of the account number. Account numbers are in the following
  format:BBBBAAAAAA, where BBBB is the branch number and
  AAAAAA is the branch account number.
 */
public long getAccountNumber() {
    return ((getBranchNumber() * 100000) + getBranchAccountNumber());
}

/**
  Set the account number. Account numbers are in the following
  format:BBBBAAAAAA where BBBB is the branch number and
  AAAAAA is the branch account number.
 */
public void setAccountNumber(int newNumber) {
    setBranchAccountNumber(newNumber % 1000000);
    setBranchNumber(newNumber // 1000000);
}
```

さらなる定数の getter を使う利点は、コードの一貫性を向上させるのに役立つことである。上述のコードを考えてみると、正しく動作しないことが分かる。口座番号 (account number) は支店番号 (branch number) と支店口座番号 (branch account number) とを結合したものである。このコードを試験すると、setter メソッド setAccountNumber() が支店口座番号を正しく更新しないことが分かった (左 4 桁を取り除かず左 3 桁を取り除いていた)。それは、branchAccountNumber を取り出すのに 100,000 ではなく 1,000,000 を使っていたからである。この値について単一のソースコードとして以下に示すように定数の getter メソッド getAccountNumberDivisor() を適用したら、コードはより一貫性が向上し、正しく動作するだろう。

```
/**
  Returns the divisor needed to separate the branch account number for the
  branch number within the full account number.
  Full account number are in the format BBBBAAAAAA.
*/
public int getAccountNumberDivisor() {
    return ((long)1000000);
}

/**
  Get the value of the account number. Account numbers are in the following
  format:BBBBAAAAAA, where BBBB is the branch number and
  AAAAAA is the branch account number.
 */
public long getAccountNumber() {
    return ((getBranchNumber() * getAccountNumberDivisor()) +
                                    getBranchAccountNumber());
}

/**
  Set the account number. Account numbers are in the following
  format:BBBBAAAAAA where BBBB is the branch number and
```

```
  AAAAAA is the branch account number.
 */
public void setAccountNumber(int newNumber) {
    setBranchAccountNumber(newNumber % getAccountNumberDivisor());
    setBranchNumber(newNumber // getAccountNumberDivisor());
}
```

定数に関してアクセッサを使うことによってバグの可能性を低減し、同時にシステムの保守性を高める。口座番号の割付が変更になり、それを結果として知ることになったとしても（ユーザとはそうしたものである)、既に口座番号の構築・分解に必要な情報の隠蔽化と局所化を行っているのでコードは容易に変更することができる。

集合へのアクセス

アクセッサ・メソッドの主な目的は、フィールドへのアクセスを隠蔽してコードの結合を弱めることにある。配列や Vector のような集合は単純な値のフィールドに比べて複雑なため、普通の Getter/Setter メソッドよりも多くの処理を行う必要がある。典型的な操作としては、集合には要素の追加や削除があるので、アクセッサ・メソッドにもそのような操作が必要となる。私が使うアプローチは集合のフィールドについて以下のアクセッサ・メソッドを追加する。

メソッド種類	命名規約	例
集合自体の Getter	getCollection() [a]	getOrderItems()
集合自体の Setter	setCollection()	setOrderItems()
集合へオブジェクトを追加	insertObject()	insertOrderItem()
集合からオブジェクトを削除	deletetObject()	deleteOrderItem()
新しいオブジェクトを作成して集合へ追加	newObject()	newOrderItem()

[a]集合の命名規約は集合が包含する情報の型の複数形バージョンを使うことを思い出してくれ。ゆえに order item オブジェクトの集合は orderItems と呼ばれる

このアプローチの利点は集合が完全にカプセル化されており、後に別な構造に置き換えること（例えば linked list とか B-tree に）ができることにある。

複数のフィールドへの連続アクセス

アクセッサ・メソッドの強力な利点の 1 つに、ビジネスルールを効果的に（強制的に）使わせる点がある。例として Shape クラスの階層を考えると、Shape のサブクラスは自身の位置を 2 つのフィールド：xPosition, yPosition を通じて知り、2 次元平面を move（Float xMovement, Float yMovement) メソッドを起動することで移動する。この移動は、x 軸、y 軸両方を同時に（連続して）変更しなければ意味をなさない。(どちらかの引数に 0.0 を与えることは許容できる）したがって、move メソッドは public とし、setXPosition、setYpositoin メソッドは private とし、move メソッド内から適切に呼ばれるようにする。

別な実装として、以下に示すように両方のフィールドを一度に更新する setter メソッドを紹介する。メソッド setXPosition() と setYPosition() は private のままで外部クラスやサブクラスから直接呼ばれることはない（以下のコードに直接呼ばれることはないことをドキュメントしておいてもよい)。

```
/**
  Set the position of the shape
 */
protected void setPosition(Float x, Float y) {
    setXPosition(x);
    setYPosition(y);
}

/**
  Set the x position - Important:Invoke setPosition(), not this method.
 */
private void setXPosition(Float x) {
    xPosition = x;
}
/**
  Set the y position of the shape
  Important:Invoke setPosition(), not this method.
 */
private void setYPosition(Float y) {
    yPosition = y;
}
```

あら捜しのための注：この例を Point クラスの単一のインスタンスによって実装することもできたのだが、これは簡単な例題として実装しているのである。

3.4.3 アクセッサの可視性

常に protected にするよう努力すること。そうすれば、サブクラスだけがフィールドにアクセスできる。外部のクラスがフィールドにアクセスする必要が生じた場合にのみ、対応する Getter、Setter メソッドを public にする。よく使う手に、Getter メソッドを public とし、Setter メソッドを protected にするものがある。

不変性を保証するために、Setter メソッドを private にする必要が生じる場合がある。例えば、Order クラスが OrderItem インスタンスの集合をフィールドを持っており、さらに order 全体の合計を示す orderTotal フィールドを持つと考える。このとき、orderTotal フィールドを変更するメソッドは、oerderItem の集合を操作するメソッドに限定するべきである。これらメソッドが Order クラスに実装されているとすると、setOrderTotal() メソッドは private とするべきである。（getOrderTotal() メソッドは public にすることが多いかもしれない）

3.4.4 アクセッサを使う理由

「よいプログラムの設計はプログラムの各部分を、不必要な、意図しない、その他望ましくない外部の影響から隔離しようと試みるものである。したがって、アクセス制限が言語によって明示的かつチェック可能な手段として提供されねばならない。」（Kanerva,1997）アクセッサメソッドは以下の方法でクラスの保守性を向上する。

1. フィールドの更新

   各フィールドを更新するのは単一個所に限定し、修正・試験を容易にする。すなわち、フィールドをカプセル化する。

2. フィールドの値を取得

   フィールドが誰からどのようにアクセスされるかを完全に制御する。

3. 定数の値とクラスの名前を取得

   定数値、クラス名を Getter メソッド内にカプセル化し、値・名前を変更する際に Getter 内部だけを変更すればよいようにする。決して定数・名前を使用している全ての行を変更しなければならないような羽目に陥らないこと。

4. フィールドの初期化

   lazy initialization を使ってフィールドが常に初期化済み、かつ必要なときにのみ初期化されるように保証する。

5. サブクラスとスーパークラスの間の結合を疎にする

   サブクラスがスーパークラスから継承したフィールドにアクセスする時は必ず対応するアクセッサメソッドを介する。これによって、スーパークラスでフィールドの実装を変更してもサブクラスには影響を及ぼさず、効果的に結合を疎にすることができる。アクセッサによってスーパークラスの変更がサブクラスに波及する、いわゆる「もろい基盤クラス問題 (fragile base class problem)」のリスクを低減する。

6. 複数フィールドへの変更をカプセル化

   複数のフィールドに関係するビジネスルールを変更するとき、変更前と同じ機能を提供するようにアクセッサを修正することで新しいルールに対応できるようにする。

7. 並行性を単純化

   Lea 氏 (1997) が指摘するように、フィールドの値に基づいて wait をかける場合、Setter メソッドによって notifyAll を通知する場面を単一個所に限定する。これによって、並行問題の記述を簡易にする。

8. 名前の隠蔽問題をなくす

   ローカル変数とフィールドと同じ名前を適用しないように注意すべきだが、フィールドへのアクセスをすべてアクセッサ経由にすることで、直接操作することがなくなるため名前の隠蔽を気にすることなくローカル変数を自由に命名できるようになる。

3.4.5 アクセッサを使わない理由

実行時間が極度に重要な場合にはアクセッサを使いたくないかもしれないが、アプリケーション内の結合度が密になることを正当化できるほどの理由はほとんどない。連携する複数のフィールドの値を一貫させる場合、単一のフィールドのアクセッサを提供することはよくないという指摘（Doug Lea,1996）がなされ、まさにそのとおりであるが、全てのアクセッサを public にする必要はないという点を忘れている。他のフィールドの値と密接に関わるときは、一貫する正しいメソッドを提供し、アクセッサメソッドを protected や private にすればよい。すべてのアクセッサを public にする必要はない。

3.5　静的フィールドは常に初期化する

Doug Lea(1996) による指摘：静的フィールド（クラスフィールド）は、そのクラスのインスタンスが作られる前にアクセスされる可能性があるため、妥当な値を持つことを保証しなければならない。これには、static initializer（static{...} ブロック）を使えば (Grand,1997) クラスがロードされた時点で自動的に実行される。

これは、静的フィールドにアクセッサを使用しない場合にのみ生じる問題である。アクセッサメソッドを使うことにより、lazy initialization によってフィールドに常に値が設定されていることを保証できる。フィールドをカプセル化するアクセッサメソッドの使用により、フィールドがどのように使用されるかを完全に制御でき、コード内の結合を疎にすることができる。

第4章　ローカル変数標準

ローカル変数はブロックスコープの中で定義されたオブジェクトまたはデータ要素であり、その多くはメソッドである。ローカル変数の有効範囲は、それが定義されたブロックである。ローカル変数にとって重要なコーディング標準は以下に着目する。

- 命名規約
- ドキュメント規約
- 宣言

4.1　ローカル変数の命名

一般にローカル変数はフィールドの命名と同じ規約に従って命名される。すなわち、最初の単語を除く残りの単語の最初の一文字を大文字とする完全英語記述である。

しかし、下記の用途において慣例的に用いられる命名については、便宜上本規約を緩めて適用する。

- ストリーム
- ループカウンタ
- 例外

4.1.1　ストリームの命名

メソッド内において、一つの入力/出力ストリームだけしかオープンし、使用し、終了しないならば、in や out をそのストリームそれぞれに命名する (Gosling,Joy,Steele,1996)。一つのストリームを入出力両方に使用するときは、inOut という命名を行う。

この命名規約の一般的な代替は、in、out、inOut に対してそれぞれ、inputStream、outputStream、ioStream と命名することである。実を言えば私は後者を薦めるが、in や out といった名前は Sun が提唱しているという事実があるため、おそらくそれに固執するべきだろう。

4.1.2　ループカウンタの命名

ループカウンタは非常によく使われるローカル変数であり、C/C++でもよく使われたので、Java でも i, j, k といった命名をループカウンタに使用してもよい (Gosling,Joy,Steele)。これらを使用するときは、一貫性を保つこと。(多重ループのときは、外側から i,j,k と付けるなど)

一般的な代替策は loopCounter か単に counter といった名前を使うことだが、このやり方の問題点は 1 つ以上のカウンタを必要とするメソッドにおいて counter1 や counter2 のような名前を付けることにある。i,j,k をカウンタとして使えば、タイプしやすいし、一般的にも受け入れられる。

カウンタなどに 1 文字の名前を使う主な欠点は、コードファイル内で使用場所を検索しようとするとたくさんの誤ヒットに見舞われることである。文字 i を探すより loopCounter を探すことを考えよ。

4.1.3 例外オブジェクトの命名

Java のコーディングでは例外処理の記述が非常に多用されるので、一般に例外を'e' という名前で扱ってもよい。

4.1.4 ローカル変数命名の悪しき考え

不幸にも、いくつかの「共通的な」短縮名がローカル変数に認められてきているが、卒直に言うとこれらは不適切であると考える。以下の表に、Sun によって既に使われているオブジェクトや変数の命名規約をまとめた (Gosling,Joy,Steele,1996)。私の意見では、これらの規約はある一線を超えており、ソースコードの可読性を損なっている。使いたいならば使えばよいが、私はこの使用を薦めない。

変数、型	（Sun により）薦められる命名規約
offset	off
length	len
byte	b
char	c
double	d
float	f
long	l
Object	o
String	s
Arbitrary value	v

4.2 ローカル変数の宣言とドキュメント

Java におけるローカル変数の宣言とドキュメントに関する規約は下記のとおり。

1. ローカル変数一つにつき一行で宣言する

   コード一行につき１ステートメントという一貫性と、インラインコメントによって各変数毎にドキュメントを記述することができる (Vision,2000)。

2. ローカル変数は行末コメントでドキュメントする

行末コメントは、//で始まる単一行のコメントで、コマンドの直後に同じ行に続いて記述する（これが行末コメントと呼ばれる）。ローカル変数が何のために使われるか、どこが相応しいか、なぜ使われるかを記述し、コードを理解しやすくする。

3. ローカル変数は変数を使用する直前に宣言する

   ローカル変数の最初に必要とする個所で宣言することによって、他のプログラマーがメソッドの先頭へスクロールで戻ってそのローカル変数が何であるか見つけ出す手間を無くすことができる。さらに、コード（の実行）が到達しない時は変数が割り当てられれないため、コードが効率的になる (Vision,1996)。このやり方の欠点は、宣言がメソッド中のあちこちに分散してしまい、大きなメソッドではすべての宣言を見つけ出すことが難しくなることである。

4. ひとつのことだけにローカル変数を使う

   １つのローカル変数を複数の目的に使用すると、コードの凝集性（cohesion）を弱め、理解を難しくしてしまう。さらに、前に使われた際の副作用によってバグを生み出しやすくしてしまう。確かにローカル変数を再利用することでメモリ使用量を減らすことにはなるが、コードの保守性が低く脆弱なものになってしまう。より多くのメモリを割り当てることをほんの少しだけ避けたとしても、普通はそれには価値はない。

4.2.1 宣言についての一般的な注意点

例えば if 文のスコープ内で宣言されるようなコードの間に宣言されるローカル変数は、そのコードを熟知していないと見つけるのが難しい。

代替え案としてローカル変数を最初に使用する直前で宣言するかわりに、コードの先頭で宣言するものがある。2.4.6 節で見たようにメソッドは極力短く記述することになるので、ローカル変数の型が何かを確認するためにコードの先頭を見る作業はそんなに悪いものにはならない。

第5章　メソッドのパラメータ(引数)標準

メソッドのパラメータ（引数）については、どのように命名するか、そしてどのようにドキュメントするかが重要な標準である。なお、この文書を通してメソッド引数のことをパラメータと呼んでいる。

5.1　パラメータの命名

パラメータの命名は、ローカル変数の命名規約に従う。ローカル変数のときと同様名前の隠蔽問題が生じる (アクセッサを使わない場合)。

リスト 21: 例）

```
customer
inventoryItem
photonTorpedo
in
e
```

5.1.1　代替案-'a' や'an' を接頭辞とするパラメータ名

価値ある代替案として、Smalltalk から来たやり方で、名前の先頭に'a' や'an' を付けるローカル変数の命名規約がある。この'a' や'an' を付加することによって、パラメータがローカル変数やフィールドとはっきり区別されるため、名前隠蔽問題を防ぐことができる。これは私が好んで使うアプローチでもある。

リスト 22: 例）

```
aCustomer
anInventoryItem
aPhotonTorpedo
anInputStream
anException
```

5.1.2　代替案-型に基づくパラメータ名

別な変数の命名として、全く推奨できないが、型に基づく名前を付けるというやり方もある。これはいくつかの点で悪い命名である：第一に、型は既にメソッドの定義部分において示されている。第二に、**aString** のような名前は、**accountNumber** や **firstName** に比べて実に貧弱な情報しか伝えない。

5.1.3 代替案-対応するフィールド (があれば) と同じパラメータ名

第三の代替案 (Gosling, The Java Programming Language) は、対応するフィールドがあればそれと同じ名前をパラメータ名に付ける。例えば、**Account** が **balance** と呼ぶ属性を持ち、それに新しい値を与えるパラメータを渡す必要があるとき、パラメータは **balance** と呼ばれる。フィールドはコード内では **this.balance** として参照され、パラメータは **balance** として参照される (適切なアクセスメソッドを呼ぶこともできる)。これは価値あるやり方だけれども、経験では"this"を忘れやすい。私はこのやり方を避ける。

5.2 パラメータのドキュメント

メソッドのパラメータに関するドキュメントは、メソッドのヘッダードキュメント内において、javadoc の@param タグを使って記述される。ドキュメントには以下の内容を記述する。

1. 何に使われるか

   他の開発者がそのパラメータをどのように使えばよいか前後関係を完全に理解できるように、パラメータが何に使われるかを記述する。

2. 制約や事前条件

   メソッドにとって、パラメータがその取りうるすべての値を受け入れることができないのであれば、そのことをメソッドを呼ぶ側に知らせなければならない。例えば、メソッドは正の数値しか受け入れないとか、5 文字以内の文字列しか受け入れない等。

3. 使用例

   パラメータが何か十分に明らかではないならば、いくつかの使用例をドキュメントに記述する。

Tip-パラメータの型にはインタフェースを使え

パラメータの型には、適用できるならば **Object** のようなクラス型を指定せずに **Runnable** のようなインタフェース型を指定する。状況にもよるが、この利点はパラメータをより具体化できる (Runnable は Object より具体的である）し、ポリモルフィズムを提供するよい方法である（パラメータがあるクラス階層の中のクラスのインスタンスであることを示すよりも、必要としているポリモルフィズムを満足できるインタフェースを提供していることを示す)。

第6章　クラス・インタフェース・パッケージ・コンパイル単位に関する標準

この章ではクラス、インタフェース、パッケージそしてコンパイル単位に焦点をあてる。クラスはオブジェクトがインスタンス化（生成）されるときの雛形 (テンプレート）となるもので、フィールドとメソッドの定義から構成される。インタフェースはメソッドとフィールドに関する共通のシグネチャを定義したもので、シグネチャはこれを実装するクラスはかならず提供しなければならない。パッケージは関連あるクラスを集めたもの。コンパイル単位とはクラスやインタフェースが定義されている 1 つのソースコードファイルである。Java ではコンパイル単位をデータベースに保存してもよいので、個々のコンパイル単位が直接物理的なソースコードファイルでなくてもよい。

6.1　クラス標準

クラスにとって重要な標準は以下に基づく。

- 可視性
- 命名規約
- ドキュメント規約
- 宣言規約
- public、protected なインタフェース

6.1.1　クラス可視性

クラスは 2 つのうち 1 つの可視性を持つ。すなわち public かパッケージ (デフォルト) である。public 可視性はキーワード public によって示され、パッケージ可視性の場合は何も示さない (キーワードがない)。public クラスは他のすべてのクラスから見ることができ、パッケージ可視性のクラスは同一パッケージ内のクラスだけが見ることができる。

1. コンポーネント内部のクラスはパッケージ可視性を使う

   パッケージ可視性によってパッケージ内にクラスを隠蔽することができ、コンポーネント内部に効果的にカプセル化する。

2. コンポーネントのファサードには public 可視性を使う

   コンポーネントはコンポーネントのインタフェースを実装したクラスとコンポーネント内のクラスへメッセージを配信するファサードクラスによってカプセル化される。

6.1.2 クラス命名

Java 標準は正しくフルスペルで、最初の 1 文字は大文字とし、複数の単語をつなげるときは、続く単語の最初の 1 文字も大文字とする (Gosling,Joy,Steele,1996;Sandvik,1996;Ambler,1998a)。

リスト 23: 例)

```
Customer
Employee
Order
OrderItem
FileStream
String
```

6.1.3 クラスドキュメント

以下の情報がクラス定義の直前におかれるドキュメント化コメントに書かれていること。

- クラスの目的

  開発者がそのクラスの大まかな目的を見て、要求に合致しているかどうか判断できるようにする。また、例えばそのクラスがある設計パターンの一部を担っているとか、クラスを使う上での制限事項について書くように習慣付ける。

- 既知のバグ[1]

  そのクラスについて判明している問題があれば記述し、他の開発者にとってクラスの弱点・障害点が分かるようにする。また、そのバグが修正されていない理由を記述する。ただし、もしバグがある一つのメソッドに特定されるならば、直接メソッドドキュメントの方に記述する。

- 開発・保守履歴

  日付・修正者・変更概要を記した履歴表を付けるのは一般的な習慣である (Lea,1996)。この目的は保守プログラマーに過去にどんな修正が、いつ誰によってなされたのかを知らせることにある。メソッドの時と同様、この情報は構成管理システムに含むべきで、ソースファイル自身に含めるべきではない。

- 不変条件をドキュメント

  不変条件はインスタンスあるいはクラスが定常期間 (stable time) の間、満たさねばならない表明の集まりである。定常期間 (stable time) とは、該当インスタンスもしくはクラスのメソッドが起動される前からメソッドが起動された直後までと定義される (Meyer,1988)。クラスの不変条件をドキュメントすることによって、他の開発者にクラスがどのように使われるかを示す。

[1] そう、バグを修正するのはよいことだ。しかし、そうする時間を取れなかったり、作業するのがその時は重要でなかったりする。例えば、負の数を渡したときにメソッドが正しく動作しないかもしれないが、正の数では正しく動作することを知っていたとする。アプリケーションでは正の数しか渡さないならば、問題が存在することをドキュメントし、そのバグを生かしたままにできる。

- 並行性に関する戦略

  Runnable インタフェースを記述したクラスは、その並行戦略を完全に記述しなくてはならない。並行プログラミングは多くのプログラマーにとって新しくて複雑な話題であり、特に時間を割いて何を意図しているのか確実に理解できるようにドキュメントを記述すること。また、なぜその他多くの戦略の中からそれを選んだかを書くことが重要である。一般的に並行戦略 (Lea,1997) には次の事柄が含まれる：同期オブジェクト、balking objects、番兵オブジェクト、バージョンオブジェクト、並行ポリシー制御、アクセプタ等。

6.1.4 クラスの宣言

final キーワードを注意深く適用する

final キーワードを継承できないクラスを示すために使う。これは元の設計者の設計上の判断であり、軽く考えるべきではない。

メソッドとフィールドの順序

クラスを理解容易にするために、一貫したやり方でクラスを宣言する。Java において一般的なやり方は、クラスを最も可視性の高いものから最も可視性の少ない順に宣言していき (NPS, 1996)、最も重要な特性である public なものを最初に見つけやすいようにする。Laffra(1997) 氏は、コンストラクタと **finalize()** を最初に置くことを提唱している。たぶんこれらが他の開発者がクラスをどう使うのか理解するために最初に見るメソッドだからである。さらに、すべてのフィールドは private として宣言するという標準を持っているから、宣言の順番は以下のようになる。

```
コンストラクタ
finalize()
public メソッド
protected メソッド
private メソッド
private フィールド
```

メソッドのそれぞれのグループ内での順番は、アルファベット順とする。多くの開発者は静的メソッドを先に、インスタンスメソッドを次に宣言し、この２つのグループ内のメソッドもアルファベット順にならべる方法を取っている。どちらのやり方も妥当であるので、どちらにするか選んだならばそれを固持すること。

6.1.5 Public と Protected なインタフェースを最小限にする

オブジェクト指向設計の原則の１つは、クラスの公開インタフェースを最小限にすることである。その理由は以下のとおり。

1. 学習容易性

   クラスをどのように使うか習得するには、公開されているインタフェースを理解するだけでよい。公開されているインタフェースを少なくすればクラスを学びやすくなる。

2. 結合度を疎にする

   あるクラスのインスタンスが別なクラスのインスタンスや直接そのクラス自身にメッセージを送っているならば、2 つのクラス同士は結合していることになる。公開インタフェースを最小限ににすることで、結合の可能性を低減することにもつながる。

3. 柔軟性を大きくする

   結合度と直接関係しているが、公開インタフェースの中の実装されているメソッドを変更（例えば戻り値を変更等）すると、そのメソッドを起動しているすべてのコードを修正しなくてはならない。公開インタフェースが少なければ少ないほどカプセル化が大きくなり、その結果柔軟性が大きくなる。

public なインタフェースを少なくすることに価値があるのは明らかだが、protected なインタフェースを少なくすべきかどうかは不透明である。サブクラスの視点でみると、スーパークラスの protected なインタフェースは public と同じ効果を持つため、protected なインタフェースのどのメソッドもサブクラスから起動することができる。それゆえ、public なインタフェースを最小化することと同じ理由でクラスの protected なインタフェースを最小化する。

6.2 インタフェース標準

インタフェースにとって重要な標準は以下のとおり。

- 命名規約
- ドキュメント規約

6.2.1 インタフェースの命名

Java 言語の規約ではインタフェース名を正しくフルスペルで、最初の 1 文字は大文字とし、複数の単語をつなげるときは、続く単語の最初の 1 文字も大文字とする。また、**Runnable** や **Cloneable** のように副詞が使われることが多い。もちろん **Singleton** や **DataInput** のような名詞も一般的に使用される。

代替案

1. インタフェース名に接頭詞'I' を付加する

Tip-Public なインタフェースを最初に定義する

経験を積んだ開発者は大抵クラスの公開インタフェースをコーディングに入る前に定義する。まず最初にクラスがどんなサービス/振る舞いを担うか分からなければ、やらねばならないまだ設計作業がまだ残っている。次に、クラスのスタブを素早く作って、そのクラスを使用する他の開発者が実際のクラスが開発されるまでの間そのスタブを使って作業が進められるようにすることが可能になる。最後に、このアプローチはクラスを構築する際の初期フレームワークとすることができる。

Coad 氏, Mayfield 氏ら (1997) が提案している方法でインタフェース名の先頭に'I' を付け加え、例えば **ISingleton** や **IRunnable** のように命名する。このやり方は、パッケージ名やクラス名とインタフェース名とを一目で区別することができる。この命名則で私がよいと思っているのは、クラス図（オブジェクトモデル）が分かりやすくなるという点である。欠点は **Runnable** のような既存のインタフェースがこの命名則を使っていない点で、これは今後も変わるとはないことである。それゆえ、私は上述の事実上の標準を選択した。このインタフェース命名は、Microsoft の COM/DCOM アーキテクチャで使われている規約でもある。

2. インタフェース名に接尾詞'Ifc' を付加する

   Lea 氏が提案 (1996) している方法でインタフェース名の最後に'Ifc' を付け加え、例えば **SingletonIfc** や **RunnableIfc** のように命名する。インタフェース名は常にクラス名と同様となる (Lea,1996)。この考え方全般はよいと思っているが、私なら名前の接頭詞に'Interface' とフルスペルを使うだろう。この命名則が持つ問題は前述 1. と同じである。

6.2.2 インタフェースのドキュメント

以下の情報がインタフェース定義の直前におかれるドキュメント化コメントに書かれていること。

- 目的

  他の開発者はインタフェースを使用する前に、そのインタフェースがカプセル化しようとしている概念を理解する必要がある。すなわち目的である。インタフェースを定義するべきか否かを判定するよい方法は、その目的が簡潔に記述できるか否かである。簡潔に記述できない場合はインタフェースを使わない。なぜならば、オブジェクト指向の初心者が継承、とりわけ多重継承を乱用しがちであったように、Java の初心者がインタフェースを乱用する傾向があるためである。

- どのように使うか、使わないべきか

  インタフェースをどのように使うかということと、どのように使ってはいけないかの両方を知る必要がある (Coad & Mayfield,1997)。

インタフェースでは、メソッドのシグネチャが定義される。個々のメソッドのシグネチャ定義については前の章で述べたメソッドのドキュメント規約に従う。

6.3 パッケージ標準

パッケージにとって重要な標準は以下のとおり。

- 命名規約
- ドキュメント規約

6.3.1 パッケージの命名

パッケージについての命名規約を順番に示すと

1. 識別子の間はピリオドで区切る

   パッケージ名を読みやすくするために、Sun はパッケージ名の中で識別子をピリオドで分割することを提唱している。例えば **java.awt** は、**java** および **awt** の 2 つの識別子を含む。

2. Sun による Java 標準パッケージは、'java' または'javax' の識別子で始まる

   Sun がこの権利を予約しており、開発に使っている Java 開発環境に関わらず、標準 Java パッケージは一貫した方法で命名される。

3. ローカルなパッケージ名は小文字で記述する

   ローカルなパッケージとは組織内部で使用され、他の組織へは配布されない。こうしたパッケージの例としては、**persistence.mapping.relational** とか **interface.screens** とかがある。

4. グローバルなパッケージ名は組織について割り当てられているインターネットドメイン名にもとづいて命名する

   他のいくつかの組織へ配布されるパッケージは、組織のドメイン名で始まる名前を含むものでなくてはならない。トップレベルのドメイン種類は小文字で記述する。例えば前項のパッケージを配布するならば、私は **com.ambysoft.www.persistence.mapping.relational** と **com.ambysoft.www.interface.screens** と命名する。接頭辞 (.com) は小文字とし、インターネット標準のトップレベルドメイン名 (現時点では com, edu, gov, mil, net, org など) でなくてはならない。

5. パッケージ名は単数形とする

   共通の規約は、パッケージ名に **interface.screens** のような複数形ではなく、**interface.screen** のように単数形の名前を使用する。

6.3.2 パッケージのドキュメント

パッケージの目的を記述した 1 つ以上の外部文書を記述し保守する必要がある。個々のパッケージ毎に以下のことをドキュメントする。

1. パッケージの理論的説明

   他の開発者が、そのパッケージがいったい何であるかを知り、それが使えるかどうか判断できる必要がある。また、パッケージが共有されるならば、それを改善・拡張してよいかどうか分かる必要がある。

2. パッケージに存在するクラス

   パッケージに含まれるクラス、インタフェースの一覧にそれぞれの簡潔な 1 行説明を記述し、他の開発者がパッケージ内容を知ることができるようにする。

Lea 氏 (1996) は、それぞれのパッケージ毎に index.html という名前の HTML ファイルを作成し、パッケージのディレクトリに置くことを提唱している。もっとよい HTML ファイルの名前は、パッケージの完全限定名に.html 拡張子をつけたものにすることである。こうすれば、あるパッケージのドキュメントファイルを別なもので上書きしてしまう事故を防ぐことができる。Lea 氏の考え方には大枠賛成だが、私の経験上似た名前のファイルは上書きされてしまうので、彼のアプローチに一部修正を加える。

6.4 コンパイル単位標準

コンパイル単位の標準と指針は以下のとおり。

- 命名規約
- ドキュメント規約

6.4.1 コンパイル単位の命名

コンパイル単位はこの場合ソースコードファイルであり、そのソース内で定義されている public なクラスまたはインタフェース名によって自動的に決まる。すなわちこのクラス名またはインタフェース名に接尾詞**.java** を付加したものである。

リスト 24: 例

```
Customer.java
Singleton.java
SavingsAccount.java
```

6.4.2 コンパイル単位のドキュメント

ファイルにつき１つのクラスまたはインタフェースだけを記述しているかもしれないが、場合によっては複数のクラスやインタフェースを同一ファイルに含めることもある。一般的に、クラス B の唯一の目的がカプセル化されてクラス A だけに必要とされる場合、クラス A のソースコードファイルにクラス B を記述する[2]。結論として、以下のドキュメント規約がソースコードファイルに適用される。（クラスには適用されない）

1. 複数のクラスを含むファイルについては、クラスのリストを記述する

   ファイル中に複数のクラスが含まれるなら、クラスの一覧と短い説明をそれぞれにつける (Lea,1996)。

2. [オプション] ファイル名と識別情報

   ファイル名が、ファイル先頭に記述されていること (Lea,1996)。この利点はコードを印刷したとき、そのコードのファイル名が何か分かる。欠点はソースファイル名を変更したときに、ドキュメントも更新しなければならない。それゆえ、洗練されたソースコード管理システムが使えるならソースファイル名を含めなくてすむだろう。

3. 著作権表記

   適用できるならば、そのファイルに関する著作権表記を行う (Lea,1996)。著作権の年と著作権を保持する個人／組織名を記述するのが一般的である。なお、コードの作成者（author）は著作権保持者ではないことがよくある。

[2]JDK1.1 以降は inner クラスの機能を利用可能であり、これは上記クラス B の実装方法の１つである

第7章　さまざまな標準、考え

この章では、重要ではあるが独立した章として記述するほどには一般化されていないいくつかの標準、指針を紹介する。

7.1　再利用

外部から購入・再利用する Java クラスライブラリやパッケージは 100 % Pure Java として認証されているべきである (Sun,1997)。この標準を施行することによって再利用しようとするパッケージが運用するプラットフォーム上で動作する保証が得られる。Java ライブラリ専門のサードパーティ開発会社や組織内の他部門・他チームといった様々なところを入手源とする Java クラス、パッケージ、アプレットを利用できる。

7.2　クラスの import にワイルドカードを使う

import 文ではクラス名の部分にワイルドカードを使うことができる。例えば、

```
import java.awt.*;
```

は **java.awt** のなかのクラスをすべて取り込む。正確には正しくはない。実際に起きることは、**java.awt** パッケージから使用するクラスだけをコンパイル時にコードに取り込み、使用しないクラスは取り込まない。

7.2.1　代替案-明示的に import するクラス名を指定

もう一つのアプローチは、使用するクラス名を完全に記述することである (Laffra,1997; Vision,1996)。例は以下のとおり。

```
import java.awt.Color;
import java.awt.Button;
import java.awt.Container;
```

このやり方の問題点は、メンテナンスが重荷になることである。新しいクラスを追加したり (コンパイラがそうすることを強制するだろう)、クラスを使うのを止めたり (自身でする必要がある) したときに常に import 文のリストを正確に維持する必要がある。

7.3 Java コードの最適化

誰も気にかけないコードの最適化に時間を浪費してはいけない

私は Java コードの最適化に関する議論を本文書の最後に持ってきた。その理由は、最適化はプログラマーが考える項目のうち、最後の１つであり、決して最初の項目ではないからである。経験上、コードの必要最小限部分だけに限定して最適化するから、最適化作業は最後にのこす。ほとんどの場合、コードのほんのわずかな部分が処理時間の大部分を占めており、最適化すべきなのはこのわずかな部分だけである。経験の浅いプログラマーが犯す典型的な誤りは、すでにコードが十分速く動いているときでさえ、コード全体を最適化しようとすることである。個人的には必要な最適化だけを実施して、それ以後は CPU サイクルを搾りだすよりもっと興味深いことに取りかかっている。

図 1 はソースコードをイテレーティブなプロセスで開発するときのプログラム工程のプロセスパターン (Ambler,1998b) を示す。このプロセスパターンではコードの最適化はプログラミングの一部ではあるが、多くの部分の一つにすぎないことを表わしている。

コードを最適化するときには何を探すべきか？Koenig 氏 (1997) はもっとも重要な要素はオーバーヘッドを固定することと、大きな入力に対する性能だと指摘している。この理由は単純である。固定したオーバーヘッドは小さな入力における実行速度の支配的要素であり、アルゴリズムは大きな入力において支配的要素である。彼の骨子は小さな入力と大きな入力との両方でうまく動作するプログラムならば、中くらいの入力についてもよく動作するということである。

複数のプラットフォームや OS で動作するソフトウェアを記述する開発者は、それぞれのプラットフォーム毎の特異性を意識する必要がある。メモリやバッファを扱かう方式のように特定の時間を要する操作では、プラットフォームによって実に違うことがある。プラットフォームによって異なる最適化が必要なことは一般的である。

ユーザの目から見れば、常に最適化してより速くコードを実行する必要はない

他の意見は、ある遅延に対して敏感になるか否かに依存するので、いつコードを最適化するかはユーザの優先度による。例えば、ユーザは、直ちに画面が描画されてそれから 8 秒間データをロードする方が、データをロードするのに 5 秒かかった後で画面が描画されるよりも幸せを感じるだろう。言い換えれば大部分のユーザは直ちにフィードバックが得られるのであれば、少しくらい長く待っても平気であるということは、いつコードを最適化するかにとって重要な知識となる。

最適化がアプリケーションの成功と失敗の違いを意味するかもしれないが、コードを正しく動作させることがはるかに重要であることを決して忘れてはならない。遅いが動くソフトウェアは速いが動かないソフトウェアよりほとんどの場合常に望ましいことを決して忘れてはならない。

7.4 Java のテストハーネスを記述

オブジェクト指向テストは全ての者にとって重大なトピックであるがオブジェクト指向開発コミュニティには無視されている。現実にはあなたが書いたソフトウェアはどの言語を選んだかに関わらず、誰かによってテストされねばならないだろう。テストハーネスとは、クラス自身に組み込まれているメソッド (ビルトイン・テストと呼ばれる) やテスト専用用クラスにおかれたメソッドの集合であり、アプリケーションをテストするために使用する。

1. 全てのテストメソッドの命名は、'test' を接頭詞として付加する

   コード中からテスト用メソッドを素早く見つけることができる。また、テスト用メソッドに

Scott の演説台-100%Pure は 100%正確に

私の意見では、Sun の 100%Pure 運動は Java にとって必要である。私の 2 つめの書籍”Building Object Application That Work(Ambler,1998a) では、Java のポータビリティについて厳しい言葉を使った。Java についてはポータビリティに関して 2 つの議論がある。ソースコードのポータビリティとバイトコードのポータビリティである。本を書いていた頃 (1997.5) は、JDK1.0 から JDK1.1 へ移植している開発者は今では私が議論していたことに対してもっとよい批判を持っているだろう。Sun は 100%Pure の試みを、ポータビリティは Java をただ使うだけでは得られず、コードがポータブルであることを保証するように作業しなければならないと認識している。いくつかのベンダーは Java を独自のイメージに押し固め、100%Pure の試みをせずにいるが、それでは Java のコードは C のコードとそう大差ないポータビリティしか持ち得ない。
Sun から Java ベンダーや開発者へのメッセージは明らかである。Java のポータビリティは容易ではない。

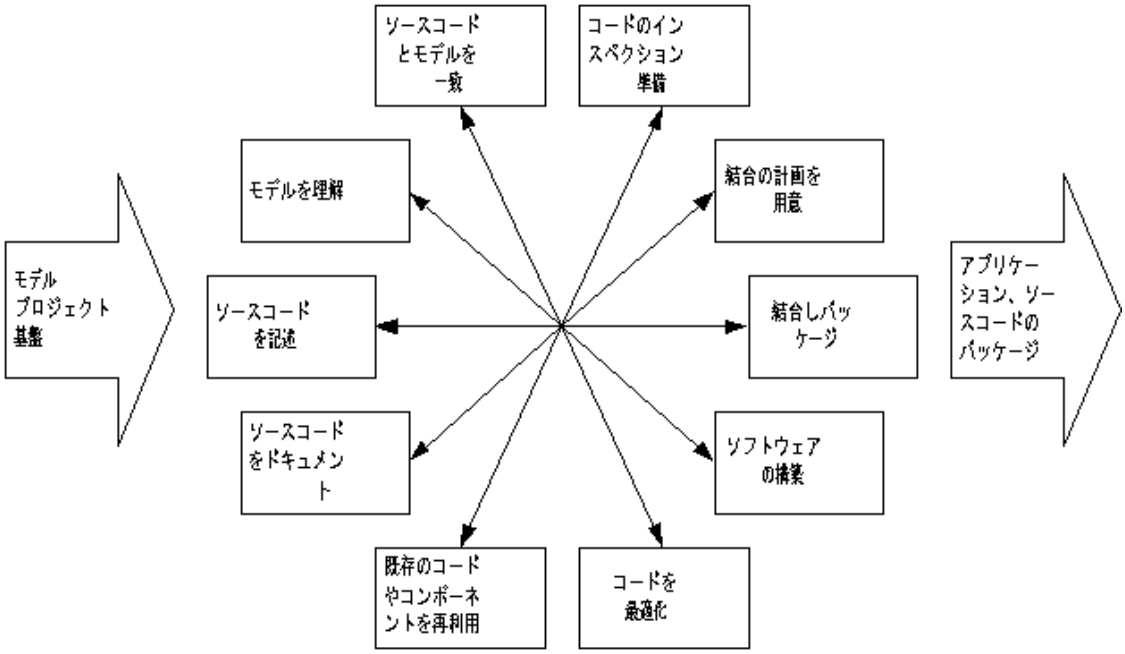

図 7.1: プログラム工程パターン

Tip-プロジェクトにおける開発優先度を定義せよ

すべての人は何が重要であるか自分自身の考えを持っており、それはソフトウェア開発者といえど例外ではない。プロジェクトまたは組織は、メンバーがビジョンを共有して働けるように、開発の優先度が何であるか定義する必要がある。Maguire(1994) によると組織では以下の要素の優先順序を確立すべきである：サイズ、性能、頑健性、安全性、テスト容易性、保守性、単純性、再利用性、移植性。これらの要素は生産するソフトウェアの品質を決定し、また優先づけすることによってチームの開発目標を決めるのに役だち、開発者間での不一致を減少させる。

接頭辞'test' が付いている利点は、製品版としてコンパイルする前にテストメソッドをソースコード中から簡単に取り除くことができる。

2. 全てのメソッドテスト用メソッドには首尾一貫した名前をつける

   メソッドテストとは単一のメソッドが定義通りに機能するかを検証する活動である。メソッドテスト用メソッドは'**testMethodNameForTestName**' のようなフォーマットで命名する。例えば **withdrawFunds()** メソッドをテストするテストハーネスメソッドには、**testWithdrawFundsForInsufficientFunds()** や **testWithdrawFundsForSmallWithdraw()** といった名前をつける。また、一連のテストを **withdrawFunds()** に対して行うときは、**testWithdrawFunds()** メソッドを設けて上述の個々のテストメソッドをすべて起動するようにしてもよい。

3. クラステスト用メソッドには首尾一貫した名前をつける

   クラステストとは単一のクラスが定義通りに機能するかを検証する活動である。クラステスト用メソッドは'**testSelfForTestName**' のようなフォーマットで命名する。例えば Account クラスをテストするテストハーネスメソッドには、**testSelfForSimultaneousAccess()** や **testSelfForReporting()** といった名前[1]をつける。

4. 各クラスにはテストを実行する単一個所を設ける

   すべてのクラステストおよびメソッドテストを起動する **testSelf()** という名前のクラスメソッドを設ける。

5. テストハーネスのメソッドをドキュメントする

   ドキュメントには、テスト内容と期待されるテスト結果を記述する。テストをマスターテスト/QA 計画 (Ambler,1998b;Ambler,1999) のような外部文書に記述する場合は、ソースコードドキュメントから外部文書の該当個所が分かるようにリファレンスを記述する。

[1]訳注：名前に Self と使用しているので、テスト対象クラス内にテストメソッドがある場合と考えられる

第8章　成功の秘訣

よい知らせは、この文書が Java 開発者がより生産的になるのに役立つように書かれたものである。悪い知らせは、この標準文書を持っているだけで自動的に生産性の高い開発者になりはしないことである。成功するためには、常により生産的であろうと志し、この標準を効果的に適用するべく努力しなければならない。

8.1　効果的な標準の用い方

以下のアドバイスは、この文書に述べた Java のコーディング標準や指針をより効果的に用いるのに役立つ。

1. 標準を理解する

   それぞれの標準や指針がなぜ生産性を高めるのか理解する時間をとりなさい。例えばローカル変数をそれぞれ別の行で宣言しないコードを実際に書いてみれば、それがコードの理解を難しくすることが分かるだろう。

2. 確信する

   おのおのの標準を理解することが始りだが、その標準を確信する必要もある。時間に余裕があるときだけ標準に従うというのでは、何ももたらさない。標準がコードを書く最良の方法だと確信して常に標準に従うならば、その効果を得ることができる。私が徹夜でコードを書かねばならなかった未熟な頃は何年も前のことだ。大部分は自分をより生産的な開発者にするツールと技術によるものである。標準に従うことに確信を持っているのは、私の経験上よく練られた標準が適切に適用されることによって著しい生産性の増加が持たらされるからである。

3. コーディング中は常に標準に従う。書いた後で標準に従うのではない。

   ドキュメントされているコードは、コーディング中だけでなく、その後もずっと理解しやすいものである。首尾一貫して名付けられたメソッドやフィールドは開発中だけでなく、保守期間においても作業を容易にする。また、きれいなコードは開発中だけでなく保守期間においても作業を容易にする。要するに、標準に従うことによって開発中の生産性が向上するだけでなく、コードの保守が容易になる（すなわち保守担当者の生産性も高まる）。多くの人々が開発中に汚いコードを書き、それから検査をパスするまで非常に長い時間を費やしてコードをきれいにするはめになっている。実に嘆かわしい。当初からクリーンなコードを書いていれば作成中から利益を得られるのに。それがスマートなやり方さ。

4. 標準を品質保証活動（プロセス）の一部にする

   コード検査にはソースコードが組織の標準にしたがっていることを保証するという役割がある。標準を、訓練や開発者の訓練と指導の基盤として使い、全体を向上させよう。

5. 標準に適合させることには多くの意味がある

   全ての標準を直ちに適用する必要はない。まず、最も受け入れやすそうな標準をいくつか選んで始めるとよい。標準を段階的に、ゆっくりと確実に導入していくことである。

8.2 コードを成功へ導く他の要素

”Building Object Application That Work”(Ambler,1998a) からいくつかのテクニックを紹介する。より大きな生産性を得るだろう。

1. マシンのためではなく、人のためにプログラムせよ

   開発においては、コードが他の人に分かりやすいものとすることを最大の目標として労力を注ぐ。他の誰にも分からないコードであるなら、そのコードによい点は全くない。命名規約を使い、ドキュメントし、段落に分けよ。

2. まず設計、そしてコーディング

   プログラムしたコードのある部分を変更する必要が生じた状況があるだろう。多分、メソッドに渡すパラメータが新しく必要となったり 1 つのクラスをいくつかのクラスに分割する必要となったろう。修正されたコードで再構成されたプログラムの中で確実に動作するためにやらねばならなかった余分な作業がどれくらいあっただろうか。そのときは幸せだったろうか。元のコードを書くときに、なぜ誰かが一旦立ち止まって考えなかったのかと自分に問いかけることはなかったか。最初に設計されるべきでなかったか。もちろんそうした。もしコーディング開始前に、どのようにコードを書くかを理解する時間を費していれば、コードを書くことにはほとんど時間を費やすことはなかったろう。さらに、コードを将来変更する際のインパクトをコードについて考えることによって減らすことができただろう。

3. 少しずつ開発せよ

   少しずつ段階を踏んで開発する、つまり 2,3 のメソッドを書いて、テストして、またそれからメソッドを書いていくことは、コード全体を一度に全て書いてからそれを修正するよりも、はるかにずっと効果があることが分かっている。10 行のコードをテストして修正することが 100 行のコードよりもはるかに易しいし、実際 100 行のコードをテストして修正する際に 10 行ずつ 10 回に段階的に行う方が同じ 100 行のコードを 1 回で書く時間の半分以下で済むと言える。この理由は簡単である。コードをテストしてバグを見つけるときはいつでもほとんど常にバグはたった今書いたばかりの新しいコードに存在する。もちろんコードの残りの部分は始めるときには十分固まっていると仮定している。コードの小さな部分からバグを見付けるのはコードの大きな部分から見つけるよりずっと早い。小さく段階的に開発することによって、バグを見つけるまでの平均時間を短くすることができ、それがひいては全体の開発時間を削減する。

4. 読んで、読んで、読む

   この産業は栄光に座するには誰にとってもあまりに急激に変化する。実際、Sun にいる私の友人達は、ただ Java において起こっていること、オブジェクト指向分野に起きていること、その他開発全般に起きていることにキャッチアップするだけで 2,3 人分のフルタイムの仕事になると見積っている。このことは、追い付くためには少くてもある程度の時間を調査に必

要とすることを意味している。ことを簡単にするため、私はあなたが読むべきであると考える主要な開発に関する書籍を必読一覧としてオンライン上に作成した。

5. ユーザに接して仕事せよ

   優れた開発者は彼らのユーザと接して仕事をしている。ユーザはビジネスを知っている。ユーザは、開発者がシステムを作り上げてユーザの仕事を支援する理由である。ユーザは開発者の給料を含めて金額を支払う。もしユーザのニーズを理解しなければシステムの開発に成功することはできないし、ユーザのニーズを理解することができる唯一の方法は、ユーザに接して仕事することである。

6. コードはシンプルに保て

   複雑なコードを書くことで知的な満足を得られるかもしれないが、他の人々がそれを理解できないようなら、何一つついいところはない。自分自身を含む誰かが複雑なコードのある部分をバグ修正や機能向上等で変更することになった最初の時こそコードを書き直す実によい機会である。実際に、あまりに理解するのが困難だったため、他の誰かのコードを書き直さねばならないことがあっただろう。そのコードを書き直すときに、元の開発者のことをどう思っただろう。その人物を天才と思ったか、それともとんでもない人物だと思ったか。後になって書き直さねばならないコードを書くことには誇りも何もない。だから、KISS ルール (Keep it simple, stupid) に従おう。

7. 一般的なパターン、アンチパターン、イディオムを学べ

   価値ある分析・設計・プロセスパターンとアンチパターン、プログラミングイディオムが存在し、開発生産性を向上するのに指針することができる。私の経験では [Ambler,1998b]、パターンはソフトウェア開発プロジェクトにおいて非常に高いレベルの再利用を行う機会を提供する。さらに詳しい情報が必要なら、プロセスパターンのリソースページ (http://www.ambysoft.com/processPatternsPage.html) を訪ずれ、key pattern resources and process-oriented web sites へのリンクをクリックせよ。

Scott の推奨読書リスト:オンライン書店

http://www.ambysoft.com/books.html を訪れれば、Java、パターン、オブジェクト指向、そしてソフトウェアプロセスを含んだソフトウェア開発の主要なトピックに関する読書リストがある。アマゾン.com の関連プログラムを通して欲しいと思ったらその場で書籍を注文できるように設定した。欲しい本の表紙をクリックするだけで簡単である。

第9章　メソッドに関して提案するjavadocタグ

この文書で記述された議論から、*javadoc* にはいくつかのタグがさらに必要であることが明かになった。*javadoc* をシンプルに保つことが重要であることには同意するが、同時に手に抱えている作業に十分である必要もある。結果として、以下の新しいタグを今後のバージョンの *javadoc* がサポートするように提案したい。

私が最初にこの節を書いてから後に、Sun は javadoc を拡張する (すなわち新しいタグを追加する) ことができる”doclets”と呼ばれる機構を導入した。doclets に関する詳しい情報と提案されている新しいタグについては、http://java.sun.com サイトの javadoc について記載されている情報を参照されたい。

提案タグ	使用対象	目的
@bug 記述	クラス、メソッド	クラスやメソッドの既知のバグを記述する。バグ毎に 1 つのタグを使用する。
@concurrency 記述	クラス、メソッド、フィールド	クラス/メソッド/フィールドが持つ並行戦略/アプローチを記述する。メソッドが実行されるコンテクストはここで記述される。
@copyright 年 記述	クラス	クラスの著作権を示し、著作権発生年と著作権を保持する個人/組織名といった情報を記述する。
@example 記述	クラス、メソッド、フィールド	クラス、メソッド、フィールドの使い方を 1 つ以上例示する。開発者がクラスの使い方を素早く理解するのに役立つ。
@fyi 記述	クラス、インタフェース、メソッド、フィールド	設計上の決定やコードの断片を知るのによい情報を提供する。
@history 記述	クラス、メソッド	クラス/メソッドが過去に変更されてきたことを記述する。変更毎に変更者、変更時期、変更内容、変更理由、変更要求とあればユーザ要求への参照を記述する。
@modifies no	メソッド	メソッドがオブジェクトを変更しないことを示す。
@modifies yes 記述	メソッド	メソッドがオブジェクトを変更し、どのように変更されるかを示す。
@postcondition 記述	メソッド	メソッドが起動された後で真となる事後条件を記述する。事後条件毎に 1 つのタグを使用する。
@precondition 記述	メソッド	メソッドが起動される前に真となっていなければならない事前条件を記述する。事前条件毎に 1 つのタグを使用する。
@reference 記述	クラス、インタフェース、メソッド、フィールド	関係のあるビジネスルールやソースコードに関連した情報を持つ外部文書へのリファレンスを記述する。
@values	フィールド	範囲や特定の値を含むフィールドが取り得る値を記述する。

第10章　これから先はどこへ

本文書は頑健な Java コードをどのように書くのかを理解する出発点である。しかし、上級 Java プログラマーになりたいと真に望むなら、書籍 The Elements of Java Style(Vermeulen et.al., 2000) を読むことを勧める。この書籍には本文書で示したよりさらに幅広い指針が載っており、さらに重要な点として優れたソースコード例が載っている。本文書と Rogue Wave 社の社内コーディング標準とが合体して The Elements of Java Style に進化したので、この書籍は学習プロセスの次の段階を優れたものにするだろう。詳しくは、http://www.ambysoft.com/elementsJavaStyle.html を訪ずれよ。

10.1　あなた自身の会社内の指針を作成する

10.1.1　この PDF ファイルを使う

10.1.2　このファイルのソース文書を入手する

Microsoft Word 形式のソース[1]は米$1,000 で販売している。あなたの組織で独自の内部指針や独自環境に使用するならば、この文書を開始点の基盤として使うことを検討するべきであろう。詳細については、電子メールで scott@ambysoft.com まで。

[1]訳注：英語の原著

第11章　まとめ

この文書は Java 開発者にとって多くの標準と指針について議論した。この文書はかなり大きいので、利便のためこの章にまとめた。この章を再度印刷して作業場所の壁に貼っていつでも使えるようにすることを推奨する。

この章では Java コーディング標準、トピックの集まりを 1 ページにまとめたものをいくつか用意した。これらのトピックとは、

- Java 命名規約
- Java ドキュメント規約
- Java コーディング規約

この文書で述べた標準や指針の残りをまとめる前に、もういちど最優先規範を繰り返しておこう。

標準に従わないときは、それをドキュメントに書く　この最優先規範を除いたすべての標準は破られる。もし破られるのであれば、なぜ標準を破るのか、標準を破った結果内在する問題、どういう状況であればその標準が適用できるのかその条件を記述せよ。

優れた開発者はプログラミングの他に開発するべき多くのことがあることを知っている。
偉大な開発者は開発の他に開発するべき多くのことがあることを知っている。

11.1　Java 命名規約

以下に述べる少数の例外はあるが、名前を付けるときはフルスペルの英語記述を常に行う。さらに、通常は小文字を使用し、1 つの名前を構成する先頭ではない単語の最初の 1 文字とクラス名とインタフェース名の最初の 1 文字は大文字とする。

11.1.1　一般概念

- フルスペルの英語記述を使う
- 業務分野の用語を使う
- 大文字・小文字を混ぜて使い、読みやすい名前にする
- 略語は控えめに、使うなら賢く
- 長い名前を避ける (15 文字以内がよい考え)
- 些細な違いしかない名前を避ける

以下の規約には賛成できないけれども、Sun が推奨する型によってローカル変数につける短い名前がある。よりよい規約はフルスペルの英語記述を使うことだ、なまけずに。

項目	命名規約	例
引数/パラメータ	渡される値/オブジェクトはフルスペルの英語記述を使い、名前の接頭辞に'a' や'an' があってもよい。1つのやり方を選んだらそれに合わせることが重要。	customer、account、あるいは aCustomer、anAccount
フィールド/プロパティ	フィールドは先頭 1 文字が小文字で続く単語の先頭 1 文字を大文字とするフルスペルの英語記述を使う。	firstName、lastName、warpSpeed
真偽型の読み出しメソッド (getter)	boolean の getter は全て'is' を接頭辞とする。この命名標準に従うなら、'is' に続いてフィールド名を継げればよい。	isPersistent()、isString()、isCharacter()
クラス	フルスペルの英語記述で、構成する単語の先頭 1 文字を大文字とする	Customer、SavingsAccount
コンパイル単位ファイル	クラスやインタフェース名、1 つ以上のクラスがあるなら主要なクラス名に'.java' を付けソースファイル名とする。	Customer.java、SavingsAccount.java、Singleton.java
コンポーネント/ウィジェット	コンポーネントが何に使われるかをフルスペルの英語記述で表現し、コンポーネントの型をその後に続ける。	okButton、customerList、fileMenu
コンストラクタ	クラス名を使う。	Customer()、SavingsAccout()
デストラクタ	Java はデストラクタを持っていないが、代わりにメソッドがオブジェクトをガーベッジコレクションされる前に呼ばれる。	finalize()
例外	文字'e' が例外を表すのに一般的に用いられる。	e
final static フィールド (定数)	全て大文字で単語間をアンダースコアで結ぶ。final static 読み出しメソッド (getter) を使用する方が柔軟性が増すのでよいやり方である。	MIN_BALANCE、DEFAULT_DATE
設定メソッド (getter)	フィールド名に'get' を接頭辞として付ける。	getFirstName()、getLastName()、getWarpSpeed()
インタフェース	インタフェースがカプセル化する概念をフルスペルの英語記述で、構成する単語の最初の 1 文字を大文字とする。名前の接尾辞に'able'、'ible' や'er' を付ける習慣があるが必須ではない。	Runnable、Contactable、Prompter、Singleton
ローカル変数	最初の 1 文字を小文字とするフルスペルの英語記述で、存在するフィールドを隠蔽しない。例えば、'firstName' というフィールドがあれば、ローカル変数に'firstName' は用いない。	grandTotal、customer、newAccount
ループカウンタ	文字、、や" が一般的に用いられる。	i、j、k、counter
パッケージ	フルスペルの英語記述だが、すべて小文字とする。グローバルパッケージはインターネットドメインを逆順にしたものをパッケージ名と継げる。	java.awt、com.ambysoft.www.persistence.mappin
メソッド	メソッドが何をするのかをフルスペルの英語記述で、可能な限り先頭を能動態の動詞で始め、最初の 1 文字は小文字とする。	openField()、addAccount()
設定メソッド (Setter)	フィールド名に接頭辞'set' を付ける。	setFirstName()、setLastName()、setWarpSpeed()

変数の型	推奨する命名規約
offset	off
length	len
byte	b
char	c
double	d
float	f
long	l
Object	o
String	s
任意の値	v

11.2 Java ドキュメント規約

ドキュメントに関してよいルールは、自分自身にもしコードを見たことがないとしたら、それほど時間をかけずに効率的にコードを理解するにはどんな情報を必要とするかを問いかけることである。

11.2.1 一般概念

- コードを明確化するコメントを書く
- ドキュメントする価値がないプログラムならば、実行するに値しない
- 過剰な装飾は使わない（例、バナーコメント）
- コメントはシンプルに
- コードを書く前にコメントを先に記述する
- コメントには、なぜそうなのかを書き、なにをしているかは書かない

11.2.2 Java コメント種類

以下の表は 3 種類の Java コメントと推奨する使い方を述べている。

11.2.3 何をドキュメントするか

以下の表は書いたコードの各部分ごとに何をドキュメントするかをまとめたものである。

11.3 Java コーディング規約 (全般)

Java コードの保守性と拡張性に欠かせない規約と標準はたくさん存在する。99.9%の時間をあなたの仲間の開発者である人々のためにプログラムすることが、マシンのためにプログラムすること

コメント種類	使用方法	例
ドキュメント化コメント	ドキュメント化したい interface, class, メソッド, フィールドの直前に書く。ドキュメント化コメントは javadoc によって処理され、クラスの外部ドキュメントとして生成される。	/** 顧客（Customer）- 顧客はわれわれがサービスまたは製品を売った人物もしくは組織のいずれかである。 @author S.W.Ambler */
C 風コメント	特定のコードを無効化したいが、後で使用するかもしれないので残しておくためにコメント化する時や、デバッグ時に一時的に無効化するときに使用 [a]	/* このコードは J.T.Kirk によって 1997.12.9 に前述のコードと置き換えたためコメント化した。 2 年間不要であるならば、削除せよ。 ... (ソースコード) */

単一行コメント	メソッド内にて、ビジネスロジック、コードの概要、一時変数の定義等を記述	// 1995 年 2 月に開始された
		// サレク氏の寛大なキャン
		// ペーンで定められた通り
		// 1000$を超える請求には、
		// 全て 5%割引を適用する。

[a] これは実際には標準ではなく指針である。重要な点は、組織がどのように C 風コメントと単一行コメントを使うべきか標準を設け、それに一貫して従うことである。

項目	ドキュメントすること
引数/パラメータ	パラメータの型、何に使われるか、制約や事前条件、例
フィールド/プロパティ	その説明、適用できる不変表明、例、並行性、可視性の説明
クラス	クラスの目的、既知のバグ、開発/保守の履歴、適用できる不変表明、並行性の戦略
コンパイル単位	定義される各クラス/インタフェースと簡単な説明、ファイル名と識別情報、著作権表記
読み出しメソッド (Getter)	もし遅延生成を使っていればその理由
インタフェース	目的、どのように使うべきか、使わないべきか
ローカル変数	その使用、目的
メソッド-ドキュメント	メソッドが何をして何故そうするのか、必要とするパラメータ、何を返却するか、既知のバグ、スローする例外、可視性の決定、メソッドがオブジェクトをどう変更するか、コード変更の履歴、正しい起動方法の例、事前条件と事後条件、並行性についての記述
メソッド-内部コメント	制御構造、コードがする事とその理由、ローカル変数、難しく複雑なコード、処理順序
パッケージ	パッケージの説明、パッケージに含まれるクラス

より重要である。他の人にコードを理解しやすくすることこそ最重要である。

規約の対象	規約
アクセッサ・メソッド	データベースに格納されるフィールドについては遅延生成を考慮せよ すべてのフィールドを獲得したい変更するのにアクセッサを使え 定数についてアクセッサを使え コレクションについては要素を挿入・削除するメソッドを設けよ 可能な限りアクセッサは public ではなく protected にせよ
フィールド	フィールドは常に private と宣言せよ フィールドを直接アクセスせずにアクセッサ・メソッドを使え 定数に final static フィールドを使わずアクセッサ・メソッドを使え 名前を隠蔽するな 常に static フィールドは初期化せよ
クラス	public と protected なインタフェースは最小限にせよ コーディングする前にクラスの public インタフェースを定義せよ フィールドとメソッドは以下の順で宣言せよ コンストラクタ **finalize()** public メソッド protected メソッド private メソッド private フィールド
ローカル変数	名前を隠蔽するな 一行には 1 つのローカル変数を宣言せよ 行末コメントでドキュメントせよ 使う直前で宣言せよ ひとつのことだけに使え
メソッド	コードをドキュメントせよ 段落分けせよ 制御構造の前に 1 行の空白とメソッドの宣言の前に 2 行の空白をいれよ 30 秒以内に理解できなくてはならない 簡潔に 一行には 1 コマンドを書く 可能な限りメソッドの可視性を制約せよ 命令の順序を特定せよ

用語

100% pure Java アプレット、Java アプリケーション、または Java パッケージが JavaVM をサポートするどのプラットフォーム上においても実行できることについての Sun からの”おすみつき”。

アクセッサ フィールドの値を変更するか返却するメソッド。Getter と Setter を見よ。

アナリシスパターン ビジネス/ドメインの問題を解決する方法を述べたモデリングパターン。

アンチパターン 共通の問題を解決するやり方のひとつで、間違っていることや非常に効果の悪いことを証明する方法。

引数 パラメータを見よ。

属性 クラスやインスタンスを記述する基本データ型や他のオブジェクトを示す変数。インスタンス・フィールドはオブジェクト (インスタンス) を記述し、static フィールドはクラスを記述する。フィールドは、フィールド変数やプロパティとも呼ばれる。

BDK Beans developing kit

ブロック 括弧で囲まれたゼロ個以上のステートメントの集り。

括弧 文字 { と } で、ブロックの開始と終了を定義するのに使用される。

クラス オブジェクトがインスタンス化される型あるいは雛型

クラステスト クラスとそのインスタンス (オブジェクト) が定義したとおりに動作することを保証する行為。

コンパイル単位 クラスやインタフェースが宣言されている、ディスク上にある物理的なファイルあるいはデータベースに格納される仮想的なファイルであるソースコードファイル。

定数読み出しメソッド (Constant getter) ハードコーディングされているか計算結果としての定数の値を返却する読み出しメソッド (Getter)

コンストラクタ オブジェクトが生成されるときに必要な初期化を実行するメソッド。

包含 (Containment) オブジェクトが、自身の振舞いを実行する際に協調する他のオブジェクトを含んでいること。これは、インナークラスを使うか (JDK1.1 以上)、他のクラスのインスタンスをオブジェクトとして集約する (JDK1.0 以上) ことで実現される。

CPU 中央演算装置 (Central processing unit)

C 風コメント Java コメントの書式/* ... */で C/C++言語から来ている。複数行のコメントを作成するのに使われる。一般にテストの間に不要なコードをコメントアウトするのに使われる。

デザインパターン 設計上の問題を解決する方法を述べたモデリングパターン。

デストラクタ 不要となったオブジェクトをメモリから取り除くのに使われる C++のクラスメンバ関数。Java は Java 自身でメモリ管理を行っているので、この種のメソッドは不要である。しかし、Java は似たようなコンセプトの **finalize()** と呼ばれるメソッドをサポートしている。

ドキュメント化コメント Java コメントの書式/** ... */で、*javadoc* によって処理されクラスファイルに関する外部文書を提供する。インタフェース、クラス、メソッド、フィールドに関する主要なドキュメントはこのドキュメント化コメントで書かれる。

フィールド 属性を見よ。

finalize() ガーベッジコレクションが起動されてオブジェクトがメモリから取り除かれる前に自動的に呼ばれるメソッド。このメソッドの目的はオープンしたファイルをクローズするような必要な後始末を行うためである。

読み出しメソッド (Getter) アクセッサ・メソッドの 1 種でありフィールドの値を返却する。読み出しメソッドは定数を static フィールドとして実装されるような状況において、定数の値を答えるのに使うことができる。なぜならこれはより柔軟性のあるやり方だからである。

HTML

インデント 段落を見よ。

インライン・コメント 単一行コメントをソースコードの一行をドキュメントするのに使い、コードと同じ行にコードの直後に記述する。C 風コメントを使用してもよいけれども、単一行コメントは典型的にはこのような使い方をする。

インタフェース インタフェースを実装するクラスは必ずサポートしなくてはならないメソッドやフィールドを含む共通のシグネチャを定義する。インタフェースは合成 (composition) による多態性 (polymorphism) を促進する。

I/O Input/output

不変表明 インタフェースやクラスが、オブジェクト/クラスのメソッドが起動される前と起動された後の全ての安定状態において真でなくてはならない表明。

Java 様々な種類のコンピュータプラットフォームの上で動作しなくてはならないアプリケーションやインターネットアプリケーションを開発するのに非常に適している業界標準のオブジェクト指向開発言語。

javadoc JDK に含まれるユーティリティで、Java ソースコードファイルを処理してコード中に書かれたドキュメント化コメントに基づいてソースコードファイルの内容を説明する HTML フォーマットの外部文書を作成する。

JDK Java development kit

遅延生成 (Lazy initialization) フィールドに対応する読み出しメソッドが最初に起動されたときにフィールドを初期化する技術。遅延生成はフィールドが常に必要とされず、大きなメモリを必要とするか永続ストレージから読み出す必要があるときに使われる。

ローカル変数 メソッドなどのブロックスコープ内で定義される変数。ローカル変数のスコープはそれが定義されたブロック内部である。

マスターテスト/品質保証 (QA) 計画 テストと品質保証の方針と手順を述べた文書で、アプリケーションの各部分の詳細なテスト計画を含む。

メソッド 実行コードの一部分でクラスやクラスのインスタンスに関連している。メソッドは、関数のオブジェクト指向における等価物と考える。

メソッドシグネチャ シグネチャを見よ。

モデリングパターン モデリング上の共通の問題に対する解決を、典型的にはクラス図の形で描写したパターン。

名前隠蔽 フィールド/変数/引数の名前により上位のスコープにある名前と同じ名前や類似した名前をつけてしまうことを指す。名前隠蔽で最も多い乱用はローカル変数にインスタンスフィールドと同じ名前をつけることである。名前隠蔽はコードが理解しにくくバグの要因になるため避けねばならない。

オーバーロード メソッドがオーバーロードされるとは、同じクラス (またはサブクラス) においてシグネチャだけが違って複数定義されていること。

オーバーライド メソッドがオーバーライドされるとは、同じシグネチャを持つメソッドがサブクラスにおいて再定義されること。

パッケージ 関連するクラスの集まり

段落化 (Paragraphing) コードブロックのスコープにおいてコードを 1 単位 (通常水平タブが使われる) インデントしてコードブロックの内外を区別する技術。段落化によってコードの可読性が高まる。

パラメータ メソッドに渡される引数。パラメータは string、int、object のような定義された型になる。

パターン 特定の問題に対する詳細な解決をパターンとして記述された共通の問題や概念に基づいて決定する。ソフトウェア開発パターンはアナリシスパターン、デザインパターン、プロセスパターンなどを含んだ多くのものから来ている。

事後条件 メソッドの実行が完了した後に真となるべき属性や表明。

事前条件 メソッドが正しく機能するために制約。

プロセスパターン ソフトウェア開発に関して実証された、成功するやり方や行動を述べたパターン。

属性 フィールドを見よ。

品質保証 (QA) プロジェクトの行いが標準に合致するかそれを超えていることを保証するプロセス。

設定メソッド (Setter) フィールドに値をセットするアクセッサ・メソッド。

シグネチャ パラメータがあればその型、メソッドに渡される順序を組み合わせたもの。メソッドシグネチャとも呼ばれる。

単位行コメント Java コメントの書式// ... で C/C++言語から来たもの。ビジネスロジックのメソッド内コメントに使われる。

タグ *javadoc* によって処理されるドキュメント化コメントの特定の部分をマーキングする規約で、特殊な見かけのコメント。タグの例に@see や@author がある。

テストハーネス コードをテストするメソッドの集まり。

UML Unified modeling language。統一モデリング言語。モデリングの業界標準記法。

可視性 クラス、メソッド、フィールドのカプセル化の段階を示すのに使う技術。可視性を定義するのにキーワード public, protected, private が使われる。

空白 コードを読みやすくするために加えられる空行、スペース、タブ。

ウィジェット コンポーネントを見よ。

第12章　著者について

Scott W. Ambler はカナダのトロント北方 45km に位置するオンタリオのニューマーケットに住むソフトウェアプロセスの指導者で、オブジェクト指向アーキテクチャ、ソフトウェアプロセス、EJB 開発のコンサルティングファーム Ronin International 社 (www.ronin-intl.com) の社長である。彼は 1990 年よりオブジェクト技術の様々な役割に従事してきた：ビジネスアーキテクト、システムアナリスト、システム設計者、プロセスメンター、リードモデラー、Smalltalk プログラマ、Java プログラマ、C++プログラマ。公式なトレーナーとオブジェクト指導者の両者として教育と訓練に積極的に活動している。

Scott は、トロント大学のコンピュータサイエンス学士と情報科学修士の学位を持っている。書籍 The Object Primer, Building Object Applications That Works, Process Patterns, More Process Patterns の著者であり、The Elements of Java Style の共著者である。これらはすべてケンブリッジユニバーシティプレス (www.cup.org) より出版されている。Scott は 2000 年に出版される RDD Books(www.rdbooks.com) の The Unified Process シリーズの編集者でもある。Scott は Software Development 誌 (http://www.sdmagazine.com) の編集者とコラムニストであり、Computing Canada(http://www.plesman.com) のコラムを書いている。

彼に e-mail で連絡する

scott@ambysoft.com

scott.ambler@ronin-intl.com

個人のウェブサイトを訪ずれる

http://www.AmbySoft.com

会社のウェブサイトを訪ずれる

http://www.ronin-intl.com

関連図書

[1] Ambler,S.W. *Building Object Applications That Work:Your Step-By-Step Handbook for Developing Robust Systems with Object Technology.* New York:Cambridge University Press, 1998.

[2] Ambler,S.W. *Process Patterns:Building Large-Scale Systems Using Object Technology.* New York:Cambridge University Press, 1998.

[3] Ambler,S.W. *More Process Patterns:Delivering Large-Scale Systems Using Object Technology.* New York:Cambridge University Press, 1999.

[4] Ambler,S.W. *The Object Primer 2nd Edition:The Application Developer's Guide to Object Orientation.* New York:Cambridge University Press, 2000.

[5] Ambler,S.W. *The Unified Process Elaboration Phase.* Gilroy,CA:R&D Books, 2000.

[6] Ambler,S.W. *The Unified Process Inception Phase.* Gilroy,CA:R&D Books, 2000.

[7] Ambler,S.W. *The Unified Process Construction Phase.* Gilroy,CA:R&D Books, 2000a.

[8] Arnold,K. and Gosling,J. *The Java Programming Language 2nd Edition.* Reading MA:Addison Wesley Longman Inc., 1998.

[9] Campione,M and Walrath,K. *The Java Tutorial Second Edition: Object-Oriented Programming for the Internet.* Reading MA:Addison Wesley Longman Inc., 1998.

[10] Clan,P. and Lee,R. *The Java Class Libraries:An Annotated Reference.* Reading MA:Addison Wesley Longman Inc., 1997.

[11] Coad,P. and Mayfield,M. *Java Design:Building Better Apps & Applets.* Upper Saddle River,NJ:Prentice Hall Inc., 1997.

[12] DeSoto,A. *Using the Beans Development Kit 1.0 February 1997:A Tutorial.* Sun Microsystems., 1997.

[13] Gosling,J.,Joy,B.,Steele,G. *The Java Language Specification.* Reading MA:Addison Wesley Longman Inc., 1996.

[14] Grand,M. *Java Language Reference.* Sebastopol CA:O'Reilly & Associates,Inc., 1997.

[15] Heller,P. and Roberts,S. *Java 1.1 Developer's Handbook.* San Francisco:Sybex Inc., 1997.

[16] Kanerva,J. *The Java FAQ.* Reading MA:Addison Wesley Longman Inc., 1997.

[17] Koenig,A. *The Importance -and Hazards- of Performance Measurement.* New York:SIGS Publications,Journal of Object-Oriented Programming,January,1997,9(8),pp.58-60, 1997.

[18] Laffra,C. *Advanced Java:Idioms,Pitfalls,Styles and Programming Tips.* Upper Saddle River,NJ:Prentice Hall, 1997.

[19] Langr,J. *Essential Java Style:Patterns for Implementation.* Upper Saddle River,NJ:Prentice Hall Inc., 1999.

[20] Larman,C. and Guthrie,R. *Java 2 Performance and Idiom Guide.* Upper Saddle River,NJ:Prentice Hall Inc., 1999.

[21] Lea,D. *Draft Java Coding Standard.* http://g.oswego.edu/dl/html/javaCodingStd.html, 1996.

[22] Lea,D. *Concurrent Programming in Java:Design Principles and Patterns.* Reading,MA:Addison Wesley Longman Inc., 1997.

[23] McConnell,S. *Code Complete-A Practical Handbook of Software Construction.* Redmond,WA:Microsoft Press, 1993.

[24] McConnell,S. *Rapid Development:Taming Wild Software Scedules.* Redmond,WA:Microsoft Press, 1996.

[25] Meyer,B. *Object-Oriented Software Construction.* Upper Saddle River,NJ:Prentice Hall Inc., 1988.

[26] Meyer,B. *Object-Oriented Software Construction,Second Edition.* Upper Saddle River,NJ:Prentice Hall Inc., 1997.

[27] Nagler,J. *Coding Style and Good Computing Practices.* http://wizard.ucr.edu/~nagler/coding_style.html, 1995.

[28] Niemeyer,P. and Peck,J. *Exploring Java.* Sebastopol,CA:O'Reilly Associates,Inc., 1996.

[29] Sandvik,K. *Java Coding Style Guidelines.* http://reality.sgi.com/sandvik/JavaGuidelines.html, 1996.

[30] Sun Microsystems. *javadoc - The Java API Documentation Generator.* Sun Microsystems, 1998.

[31] United States Naval Postgraduate School. *Java Style Guide.* http://dubhe.cc.nps.navy.mil/~java/cource% slashstyleguide.html, 1996.

[32] Vanhelsuwe,L. *Mastering Java Beans.* San Francisco:Sybex Inc., 1997.

[33] Vermeulen,A.,Ambler.S.W.,Bumgardner,G.,Mets,E.,Misfeldt,T.,Shur,J.,and Thompson,P. *The Elements of Java Sytle.* New York:Cambridge University Press, 2000.

[34] Vision 2000 CCS Package and Application Team. *Coding Standards for C,C++,and Java.* http://v2ma09.gsfc.nasa.gov/coding_stanards.html, 1996.

[35] Warren,N and Bishop,P. *Java in Practice:Design Styles and Idioms for Effective Java.* Reading,MA:Addison Wesley Longman Inc., 1999.