解　説

プログラミング言語最新情報　I

Eiffel†
——仕様記述能力をもつオブジェクト指向言語——

酒　匂　　寛††

1. はじめに

この解説記事は近年注目を集めはじめたオブジェクト指向言語 Eiffel の紹介を目的とするものである．Eiffel は元 UC サンタバーバラ校教授の Bertrand Meyer 博士によって設計／実装された言語で，強い型付けをもつコンパイル型言語である．先端的ソフトウェア工学の実践を目指す氏の日頃の意見と態度が，この言語の設計思想に反映されている．

オブジェクト指向設計に対する Meyer 博士の見解を一言で述べるなら，「オブジェクト指向設計とは，(抽象データ型の実装としての)クラスの構造的な集合を構築していくことである」[5] というもので，その意見を反映するように Eiffel はそれぞれのクラスの「抽象データ型としての仕様」を，実装とは区別して定義させることができるようになっている．もちろんここでいう「仕様」とは，システム全体の振舞いをきちんと形式的に定義するような大がかりなものではなく，あくまでも個々のクラスの振舞いを，既存の商用言語よりもより正確に表現できるといった類のものである．

しかし，もちろんこれは，オブジェクト指向設計／言語の目指す目標の一つでもある．オブジェクト指向言語あるいはオブジェクト指向方法論が話題に上るとき，必ずと言っていいほど「再利用性の高さ」といったことが語られる場合が多い．「再利用性の高さ」が意味するところは，インフォーマルな言い方を許していただけるなら，システム構成要素間の依存性の低さ，ということになるだろう．すると個々のクラスの振舞いと，直接関係するクラス間の関係を，きちんと定義できればそれで十分ということになる．別の言い方をするなら，既存のクラス・ライブラリの中から有用な部品を必要に応じて選んで，システム構築に使うというのが理想の利用形態の一つであるとすると，システム全体を記述する制約は個々のクラス定義に及ぶべきでないことは当然であろう．

個々のクラス（のインスタンス）が，他のクラス（のインスタンス）に対してどう振る舞うかということは，Eiffel ではインスタンス自身が常に満たすべき制約と，インスタンスに適用される個個の操作が満たすべき制約によって表現される．あらかじめお断わりしておくが，残念ながらプログラムの正しさをすべて静的に「証明」しようとするほど Eiffel は野心的な言語ではない．ここで述べた制約のチェックは実際には実行時に行われるものである．具体的にどのような記述が可能かは，後述する．

Eiffel の仕様に関して記述されたものとしては，文献 5)，6) が有名であるが，特に，文献 6) は邦訳が存在するため我が国でもより多く読まれているものと思う．残念なことに文献 6) は Eiffel の言語仕様第 1 版に基づくものであり，現在の仕様とは少々異なる箇所がある．最新の言語仕様は第 3 版で，これを解説したものが文献 5) である．本稿は文献 5) の記述に従って最新の言語仕様を紹介する．

次章以降の本稿の構成であるが，まず **2.** で他の言語と比較した Eiffel の設計哲学，用語の違い，実装や環境構築の方針などに触れ，**3.** ではシンタックス，セマンティクス両面から言語の特徴を解説する．**4.** では応用事例に触れ，**5.** では言語標準化の動向について述べる．

† Eiffel: An Object-Oriented Language by Hiroshi SAKO (Software Research Associates, Inc. Software Engineering Lab.).

†† (株)ソフトウェア・リサーチ・アソシエイツ　ソフトウェア工学研究所

2. 他のオブジェクト指向言語との比較

2.1 設計哲学

クラスが抽象データ型の実装であるととらえるなら，その操作の意味は形式的かつ厳密に定義されていなければならないが，そのプログラムの正しさを静的に「証明」するのは至難の業である．なぜなら自動証明それ自身も非常に困難なテーマだが，なにより大部分のソフトウェア・システムとは，理想的な数学の世界を扱っているわけではなく，一瞬後には何が起こるか分からない現実世界の事象を，開発者の限られた経験と予想の下になんとか形式的な世界に写像して，その仮定の世界での動作を記述しようとするものだからである．

とすれば，個々のプログラムは仕様上の正しさだけに注意を払うだけでなく，その実行時にもまた，正しい状態が保たれているか，正しい文脈で自分は実行されているか，要求されている制約を自身は満たすことができそうか（できたか）を常に判断し続けなければならないことになる．

この要求に応えるために，Eiffel は契約によるプログラミング (programming by contract)[6] という考え方を基に，言語が設計されている．これはプログラム構成要素（クラス）間の関係を「契約」という概念でとらえようというものである．たとえばあるオブジェクトAから，別のオブジェクトBのルーチンを呼び出すということは，AからBへの仕事の依頼であり，それには両者の間で，どのような条件で仕事を発注／施工するかという取決めがあらかじめなされていなければならない．この「取決め」は設計時には「仕様」とみなされるし，実行時には確かに契約どおりに進行しているか否かの指標になる．

実際にオブジェクト指向言語を使ってプログラミングを行う者は，意識的にも，無意識的にもこの種の契約／履行の概念でプログラミングを行っているのであるが，商用のオブジェクト指向言語で，このような考えに基づいて設計されているものは，他に見当たらないようである．

Eiffel は純粋なオブジェクト指向言語である．ここでいう「純粋」という意味は，たとえば C++ や Objective-C, CLOS などのように，既存の言語である C や Lisp にオブジェクト指向の拡張を施したというものではなく，プログラムがクラスだけを単位に構成され，クラスから遊離して存在する言語要素がまったくないという意味である (version 2.0 までの Eiffel にはそれでも整数，実数，文字列などのデータ型が「基本データ型」という一応クラスとは別の存在として存在していたが，version 3.0 からはこれらも「クラス」として扱われるようになった)．

2.2 用語の対応

ここでさらに詳しい説明に進む前に Smalltalk, Eiffel それに C++ の用語の対応を示しておこう．

なお Eiffel は下に示したように，いわゆるスーパクラス，サブクラスを，祖先，子孫と呼ぶが，本稿の中ではオブジェクト指向言語一般の話題に触れるときの習慣に従ってスーパクラス，サブクラスと呼ぶ．

Eiffel では属性とルーチンはさらに特性(feature)というカテゴリに統合される．また一般的な意味での「変数」は Eiffel の中ではエンティティと呼ばれる．以下の記述では特に断わらない限り「型」と「クラス」を同じ意味で用いる．

2.3 言語の実装と環境

C++ との比較を考えよう．C++ におけるクラスは struct の拡張である．クラスとそのメンバ関数だけを使う限り，オブジェクトの情報隠蔽は保証されるが，C++ には C 言語としての特性が残っている．すなわちさまざまな手段でカプセル化されたオブジェクトの内部に，アクセスを許すことができてしまう．C としての性格を色濃く残しているため，普通の使い方でも，容易に情報隠蔽の壁を破ることができてしまうのである．Smalltalk は普通の手段では情報隠蔽はきちんと守られているが，メタクラス・システムに自分で

表 Smalltalk, Eiffel, C++ の用語の対応

Smalltalk	Eiffel	C++
クラス	クラス	クラス
スーパクラス	祖先 (ancestor)	スーパクラス
サブクラス	子孫 (descendant)	サブクラス
インスタンス変数	属性 (attribute)	メンバ
クラス変数	*該当なし*	静的メンバ
インスタンスメソッド	ルーチン	メンバ関数
クラスメソッド	*該当なし*	静的メンバ関数(?)
該当なし	総称 (generic) クラス	テンプレート

手を入れはじめると何でもできてしまう．もちろんこの「何でもできる」ところこそが Smalltalk システムの強みであり，さまざまなツールや実験システムの構築が非常に容易なのもそのせいである．

これに比べて Eiffel は，クラスとオブジェクトがきちんと分かれており，環境の内部構造に踏み込むことは通常の手段では不可能である．すなわち Eiffel のクラスは実行の最初から最後まで安定して存在しており，そこに定義されたインスタンス変数や，メソッドの定義も実行中に変化することはない．(とはいえ実用的な言語が常にそうであるように，実行時にクラスの内部構造にアクセスする手段は存在している．これは特にデバッガの作成を容易にするために必要な措置であるが，処理系にある程度依存する部分でもある．もちろん移植時にどの部分が処理系依存かは，はっきりと示されている)．

C++ 同様 Eiffel は基本的にコンパイル型の言語である．Smalltalk はプログラムをいったん中間形式（バイトコード）に変換してそれを解釈実行（インタプリト）するが，Eiffel は機械語レベルまで変換され，実行される．とはいえこの区別は言語の特性というよりも，処理系の問題である．たとえば最新の Eiffel 処理系は，自ら「融けた氷 (melting ice) 手法」と呼ぶ方法を取り込んでおり，そこでは現在開発中のモジュール／ルーチンだけをインタプリタ方式で実行し，安定している部分はコンパイルされた形式のまま実行するといった方式を採用している．

既存のオブジェクト指向言語は，大部分が言語仕様が決まっているだけで，クラスライブラリや開発環境は別途調達しなければならない．C++ は今や非常にポピュラな言語であるが，ライブラリや開発環境に標準的なものが用意されていないため，利用者はまず市販のライブラリのどれを使用して，どのような開発環境を手元にもてばよいのかが分からない．Smalltalk はこの対極にあり，標準的な開発環境と使いきれないほどのクラスライブラリが一緒にやってくる．Eiffel は現在 C++ と Smalltalk の中間地点に存在すると言える．すなわち基本的なデータ構造のライブラリ（リスト，配列，スタック，文字列操作，木，永続オブジェクトなど）やインタフェースのライブラリ（ウィンドウ操作，グラフィクスなど）はすでに存在し，開発環境もきわめてビジュアルなものを目指している．ただし version 2 までの Eiffel の開発環境は，それなりに統合されてはいるものの，テキストベースであり，Smalltalk のような本格的なビジュアル・インタフェースは version 3 の登場を待たなければならない（なおこれらの処理系の話はいずれも Meyer 博士の会社がリリースしている製品に基づいている)．

3. 言語の特徴

3.1 クラスの記述例と使用例

以下に簡単なクラス定義とその使用例を示す．なお "--" から行末まではコメントを表している．例として定義されるクラスは PURSE というクラスである（太字はキーワードを表す)．

```
--クラス PURSE の定義
  class PURSE -- クラス名は PURSE (財布)
  feature  -- 外部に公開される情報
    balance : INTEGER ; -- 属性：財布の残金
    put(sum : INTEGER) is -- put 手続き：財
                          布に sum 円追加
    require
      sum >= 0 ; -- 事前条件は sum が 0 以上
                 であること
    do
      add(sum)
    ensure
      -- 事後条件
      -- 結果の残金が，手続き開始前の残金
                 (old balance)
      -- と sum の和になっていること．
      balance = old balance + sum
    end ;
    get(sum : INTEGER) is -- get 手続き：財
                          布から sum 円取り出す
    require
      sum >= 0 ; -- 事前条件は sum が 0 以上
                 であること
      sum <= balance -- かつ sum が残金以下
                 であること．
    do
      add(-sum)
    ensure
```

```
    -- 事後条件
    -- 結果の残金が，手続き開始前の残金
                          (old balance)
    -- と sum の差になっていること．
    balance=old balance−sum
  end

feature {NONE} -- 外部に公開されない情報
  add(sum : INTEGER) is -- and : 残金
                    (balance) に sum 円追加
  do
    balance := balance+sum
  end
invariant
    balance >= 0 -- クラス不変表明：常に残金
                    は0以上でなければならない
  end -- クラス PURSE 定義の終り
```

Smalltalk の用語を用いれば，balance はインスタンス変数，put，get はメソッドということになるが，Eiffel ではみな同じ feature という項目としてまとめられていて，外部からは類似の情報としてみることができる（**3.2.2**「参照の統一」でさらに解説）．また put と get には require と ensure という句がついているが，これはそれぞれの操作の事前条件（precondition）と事後条件（postcondition）を表している（「表明」の節で解説）．feature と feature {NONE} という二つの表記があるが，前者はそこに定義されるものが全てのクラスのオブジェクトからアクセス可能であることを示し，後者はこのクラス（とそのサブクラス）のオブジェクトからのみアクセス可能であることを示す．一般的に feature {class 1, class 2, …, class N} といった形で，属性やルーチンをある特定のクラスだけに公開することが可能である．これは直接関係するクラスの間だけでインタフェースを公開しあうので，より高いレベルの情報隠蔽を助ける．C++ ではこうしたアクセスコントロールは，private, protected, public というキーワードによって制御されるが，ちょうど Eiffel の feature {NONE} が protected に対応し，単なる feature が public に相当していることになる．feature {class 1, class 2, …} という表記に相当する記述は C++ では直接表現できないが friend というキーワードを使って似たようなことが実現できる．しかしながら friend はたとえ継承上の関係がなくとも他のクラスの内部に直接アクセスすることを許すので，それだけ危険をともなう．

次に示すのが，クラス PURSE の使用例である．

```
acc : PURSE ;  -- PURSE クラスのエンティ
               -- ティ acc を宣言
!!acc ;        -- acc に格納されるインスタン
               -- スを生成
acc.put(5000) ;  -- 財布に 5000 円入れる
acc.get(3000) ;  -- 財布から 3000 円取り出す
```

acc.put, acc.get が適用される場合に，上記の PURSE の定義中の require（事前条件），ensure（事後条件），invariant（クラス不変表明）がチェックされる（もちろん実際の実行時にすべてのチェックを行うと，効率上問題が発生するので，Eiffel のコンパイラは実行時にどの表明を検査するか選択することを許している．ゆえにシステムの開発中や，高い信頼性が要求される場合には全ての表明を検査し，反対にシステムが十分にテストされて表明の検査を省略しても，エラーが起きる可能性が少ない場合には表明の検査を省略する，といったことが行われる）．

3.2 シンタックス

3.2.1 記述単位

見かけの構文上で言えば，Eiffel は Pascal, Ada の流れを汲むものである．純粋なオブジェクト指向言語として，言語の最高位の構成要素はクラスであり，クラス単位のコンパイルが可能である．Ada や C++, Objective-C などはデータ型やクラスの宣言を，実装の部分と分けて記述することができるが，Eiffel では一カ所に記述することを要求する．この一見後退ともみえる選択に関して言えば，Eiffel はあえてその道を選んだ言語である．インタフェースを分離して記述するというやり方には，プログラムの可続性を上げたり，「仕様」と「実装」の部分を別ファイルにして管理することにより作業者の責任範囲を明確にさせたり，記述の冗長性を増して静的チェックの機会を増やしたり，という狙いがあるが，Eiffel の設計者の主張は，こうした狙いは（最後のものを除いて）皆ツールや環境で吸収すべき問題であり，わざわざ長々とした冗長なコードの記述をプログラマに強いるべきではない，というものである．

クラスは既存のクラスをスーパクラスとして指定することができる．多重継承を支援しているた

め，複数の親クラスを指定することが可能である．もし何も指定しない場合は，スーパクラスとしてANYというクラスが指定されたものとみなされる．これはちょうどSmalltalkで言うところのObjectクラスに相当するが，Eiffelの環境ではANYの上にさらにPLATFORM，GENERALというクラスが存在している．GENERALは全てのオブジェクトが必要とするオブジェクトのコピーや，等価性の検査などを定義している．

PLATFORMはGENERALの唯一のサブクラスで，実行プラットフォームに依存した少数の情報（たとえば整数INTEGERの長さなど）を定義する．そしてPLATFORMの唯一のサブクラスがANYである．

3.2.2 参照の統一

あるオブジェクトに情報の問合せを行う場合を考える．問合せの対象となる情報はそのオブジェクト固有のインスタンス変数かもしれないし，内部の計算の結果求まるものかもしれない．仕様作成の段階では毎回計算で求めるつもりだった情報が，実装上の都合でインスタンス変数として取り扱われるようになることは特に珍しいことではない．

簡単な例だが，今「人」というクラスが存在し，そのインスタンスに「年齢」を問い合わせたいとしよう．実装上その「年齢」情報は，インスタンスの中に常に保持しておくやりかたと，毎回「誕生日と現在の日付」から計算して求めるという二つのやり方が考えられる．下のようにaPersonという変数（Eiffelではエンティティと呼ばれる）が人クラスのインスタンスと結びつけられているとすると，C++では実装方法の違いによって，情報の問合せ法が異なる．

```
aPerson. age
    //単純にメンバを参照する方法の場合
aPerson. age()
    //計算により求める方法の場合
```

これに対しEiffelでは，どちらの実現方法をとろうとも

```
aPerson. age -- 内部の実現によらない
```

という形で情報が参照される．これが参照の統一と呼ばれる手法である．実装上必要な性能を保証するために，本来各オブジェクトに分散していたり，計算で求めるべき情報を，内部でキャッシュしておくといった変更はしばしば行われることであるが，Eiffelのような参照の統一があれば，問合せのインタフェースを変える必要がないので，仕様の変更に対する対処がそれだけ簡単になる．もちろん問合せがなんらかの引数を要求する場合にはEiffelの場合でも

```
aPerson.age(arg) -- 引数が必要な場合
```

といった形で引数を指定しなければならないので，単純な問合せの場合しか有効ではないが．

Smalltalkでも，どちらの実現方式をとろうが，問合せの情報は

```
aPerson. age. "age の実現方法によらない"
```

という形で参照できるので，この点ではEiffelとSmalltalkは同じである（とはいえSmalltalkではそもそもメンバを直接参照できないので，この例は少々公平を欠くものであるかもしれない．Eiffelではageが公開された情報なら，そのまま外部から参照できるが，Smalltalkの場合age情報を参照するメソッドを別に定義してやらなければならない）．

3.2.3 総称クラス

オブジェクト指向言語で書かれたクラス・ライブラリが話題になる場合，しばしばコンテナ・クラスと呼ばれるクラス群が登場する．これはたとえば配列構造，リスト構造，木構造などのデータ型を指し，オブジェクト群をある構造にまとめ上げるためのものである．EiffelやC++のように強い型付けをもつ言語では，そうやってまとめられる要素データの型を明示的に指定しなければならない．たとえば「整数のリスト」，「実数のリスト」，「ウィンドウのリスト」といったものを表現する場合，

```
INTEGER_LIST
REAL_LIST
WINDOW_LIST
```

といったクラスをそれぞれのデータ構造に対応して宣言してやらなければならないが，これではデータ構造と要素データの型の数の積だけクラスを用意しなければならない．しかも単にリスト型としてみた場合，リストとしての操作（要素の追加，削除，検索など）はほとんど共通なのであるから，これを別々のクラスとして実現してはアルゴリズムの記述がそれぞれのクラスに分散し，保守性も，空間の利用効率もきわめて悪いものにな

ってしまう．再利用性を高めるという目標がここでは危機に瀕している．もちろん「どんな要素がきてもよい」といったコンテナクラスを定義することによって，上記のような要素型ごとにデータ構造を定義するといったことを避けることはできるが，そうすると今度はせっかく保持されるべきデータ要素を静的にチェックできるという強い型付けの利点が失われてしまう．たとえばSmalltalkでは静的な型チェックがないためにコンテナ・クラスの要素として普通は何を保持させても構わない．これはもちろんシステム構築上非常に柔軟な特性であるが，ある実行文脈のもとである変数が保持し得るオブジェクトの型を制限したい場合，動的に（実行時に）チェックするしか方法がないので，実行効率に影響が及ぶ可能性がある．また動的なチェックが全ての場合を網羅していない場合には，最悪のケースとして実行時に「そのようなメソッドは存在しない」というエラーを引き起こす可能性がある（ただしEiffelの型システムも現時点では完全ではない．詳しくは文献3)，7)参照)．

総称クラス（generic class）は，要素データの型をパラメタとして，新しいデータ構造に対応するクラスを定義させる手段である．たとえば，先の三つの例は，総称クラスを用いると次のようになる．

LIST[INTEGER]
LIST[REAL]
LIST[WINDOW]

これは

LIST[T]

という形であらかじめ定義されていた総称クラスLISTの仮引数Tに，別に定義されたINTEGER, REAL, WINDOWクラスを実引数として与えたものである．何も指定しなければTに与えるクラスは何でもよいが，場合によってはある種の制限を与えたい場合がある．たとえば要素が全てソートされているようなリスト

SORTED_LIST[T]

という総称クラスを定義する場合，その要素Tは互いに順序を決定できるものでなければならない．このような場合

SORTED_LIST[T -> COMPARABLE]

といった宣言を行うことにより，実引数で与えられるクラスがCOMPARABLE（比較可能）というクラスのサブクラスに限ることを示すことができる．このような宣言があるとき，仮にINTEGERがCOMPARABLEのサブクラスでありPOEMがCOMPARABLEのサブクラスではないとすると，以下二つのクラスのうち，

SORTED_LIST[INTEGER]
SORTED_LIST[POEM]

前者は有効であるが，後者はエラーとしてコンパイラに拒絶される．プログラム中の実際の定義を示すと：

aList : SORTED_LIST[INTEGER];

といった形になる（aListはSORTED_LIST[INTEGER]型のエンティティ）．

3.3 セマンティクス

3.3.1 型の適合

「型適合（conformance)」はEiffelの型システムの中でも非常に重要な概念である．もっとも普通の型適合の使用例はエンティティ間の代入時にみることができる．今，下のような式がプログラム中に現れたとしよう：

x := y

ただしxは型Xのエンティティ，yは型Yのエンティティであるとする．このとき，上の式は型Yが型Xに適合するときに限って許されるのである．

型適合は継承に基づいて決定される．基本的な型適合の規則は以下のとおりである．

- 型Yが型Xのサブクラスであること．
- もし型Yが総称クラスであるなら，その引数の型がそれぞれ型Xの対応する実引数に型適合すること．

例を示そう．以下に示す，全ての例で型Yは型Xに型適合していなければならない．

- 代入 x := y を行うとき．
- ルーチン・コール anObj.r(…, y, …) を行うとき．ただしrの宣言時に実引数yの位置の仮引数が型Xとして宣言されていること．
- オブジェクトの生成 !Y!x を行うとき，すなわち型Yのオブジェクトを生成し，xに代入する場合．
- サブクラス中でxを型Yとして再定義するとき．ただしxは属性，関数，ルーチンの引数のいずれか．

- 総称クラスを C[…, Y, …] のように利用する場合．ただし実引数Yの位置の仮引数が型Xに制約されると宣言されていること．すなわちもとの宣言が C[…, T->X, …] であること．

例だけみると複雑なようだが，いずれも先の型適合規則から容易に導かれるものである．この型適合は，ある関数／ルーチンのシグネチャ全体に適用される．型適合による静的型チェックは，設計者の意図がプログラムにきちんと反映されているか否かの手助けとなるし，正しくない文脈にオブジェクトが適用される誤りを，完全ではないが，プログラムの実行以前に検出する役割を果たす．特に多くのプログラムが協力して一つのシステムを構築している場合，この型適合の機構が，クラスの組合せに関する多くの過ちを防ぐ働きをする．

3.3.2 表　明

仕様の，あるいはプログラムの正しさを監視するために，Eiffel はいくつかの表明機構をもっている．それらは以下の4つである．

- ルーチンの事前条件と事後条件
- クラス不変表明
- check 命令
- ループ不変／変更表明

最初の二つがどちらかと言えば，仕様に関する表明であり，後者二つは実装上の表明である．ここでは前者二つについてのみ解説する．なおこの二つの表明は実行時に妥当性の検査を行うか否かを，別個に定義することが可能である．すなわち，非常に高い信頼性が求められる箇所や，開発途中の箇所に関しては常に表明を動的に検査し，性能が重視されるところでは，その動的検査を禁止することができる．

（1） ルーチンの事前条件と事後条件

3.1 で例示したように，各ルーチンには事前条件と事後条件を与えることができる．require に導かれる句が事前条件を表し，ensure に導かれる句が事後条件を表す．仕様上でのこの操作の振舞いをとらえようとするもので，実行時に実際にチェックすることができるだけでなく，仕様記述としても用いることが可能である．代表的な Eiffel の処理系は，ルーチンのシグネチャ部分とコメント，事前条件，事後条件だけを抜き出して技術者に読ませる機能をもっている．

継承を行って，サブクラス内でこのルーチンを再定義する場合，事前条件はより緩い方向（既存の条件と or で結ばれる）へ向かい，事後条件はより厳しい（既存の条件と and で結ばれる）方向へ向かう．これは一見直観に反するようだが，先に触れた「契約によるプログラミング」に基づいて決定された仕様である．たとえば今クラスXとクラスYがあり，YがXのサブクラスであるとしよう．型適合の規則 (**3.3.1**) によれば，以下の代入は正しいものである．

```
x : X ;
y : Y ;
x := y ;
```

クラスXとY双方で，ルーチンfが定義されているとする（すなわちYはXのfを再定義している）．このときさらに以下のような実行をする場合を考えよう．

```
x.f ;
```

実際に実行されるのは Y で定義された f である．このときもしYのfの事前条件がXのfより「厳しい」ものであった場合はどうなるであろうか？　場合によってはこの部分は実行時エラーを引き起こすことになる．しかしここで x.f という表記に期待されているものはあくまでもクラスXが果たすべき責任を全うすることだけである．もしこれが「Yとしての責任」を求めたい文脈であるとするなら，最初から：

```
y.f ;
```

と書くべきなのである．ゆえに，サブクラスでルーチンが再定義される場合はその事前条件を緩めなければならない．スーパクラスで認められる条件は全て受け入れ，なおかつ自身が認める条件も受け入れなければならないのである．

事後条件の場合はスーパクラスが満たすべき事後条件はすべて満たした上で，さらに固有の事後条件を満たさなければならない．

（2） クラス不変表明

クラス不変表明はクラスが定常状態（各ルーチンの実行の狭間）にあるときに各オブジェクトが満たさなければならない条件である．たとえば「人間オブジェクトの年齢属性は常に 0 以上でなければならない」といったものがその例である．先の事前条件，事後条件が，それぞれのルーチンの適用前後に調べられるものであったのに対し

て，クラス不変表明は全てのルーチンの適用後に，事後条件とともに調べられるものである．

事後条件と同じように，クラス不変表明は，そのクラスの中で定義された条件が，上位の条件とand で結合されて成り立たなければならない．なぜなら継承の各階層で定義された不変表明は，サブクラスのインスタンス中でも同じように成り立たなければならないからである．これは型適合規則からも明らかで，上位クラスのエンティティに安心して代入を行えるために必要なことである．

3.3.3 例外機構

前項で説明したような表明が成立しなかった場合，あるいはシステムのエラー（OS レベルのエラーなど）が発生した場合，Eiffel は例外機構に制御を移す．各ルーチンには rescue という句を書くことができ，その中に例外が発生した場合の対応を記述することができる（rescue 句がない場合，そのルーチンの実行はただちに失敗し，それを実行しているさらに上位レベルの rescue が呼ばれる）．rescue 句の中ではエラー回復のための手段を講じることが可能で，必要な措置の後もう一度ルーチンの実行を行うことができる．これは retry という命令の実行によって行われる．rescue 句が retry を実行することなく，その最後の命令を実行し終わると，このルーチン全体が失敗したものとみなされる．

N-version プログラミング[1]がこの機構を使って実現できる．

以下に示す例は rescue と retry の使用例で，実装を変えて実行を試み，それでも失敗した場合には本当の失敗となるものである．

```
-- ルーチン do_something：最初の試みが失敗し
-- た場合，実装を変えてもう一度実行を行い，そ
-- れでも失敗した場合には本当に失敗となる
  do_something is
  require
  …
  local
    tried : INTEGER
  do
  if tried = 0 then
    implementation_1
  elsif tried = 1 then
    implementation_2
  end
  ensure
  …
  rescue
    tried := tried+1 ;
    if tried < 2 then
    -- もし一回目の失敗ならもう一度実行する
      …なんらかのエラー回避策を実行…
      retry
    end
  end-do_something
```

3.3.4 多重継承と多相性

Eiffel は多重継承を支援している．多重継承を行うと名前の衝突が常に問題になるが，Eiffel においては名前の衝突は常に明示的に解決されなければならない．すなわち CLOS のような暗黙的優先順位を用いて名前の衝突が自動的に回避されるというやりかたは「しない」．衝突を解決する手段は，衝突を起こしている名前の改名 (rename) である．たとえば今二つの親A，Bから多重継承を行うクラスCがあるとしよう．両者に衝突する名前がない場合には，Cの定義はたとえば次のように始まる．

```
  -- A，Bから多重継承を行うクラスC
  -- A，B間に名前の衝突はない
  class C
  …
  inherit
      A ;
      B
  feature
  …
```

もしA，Bが共にfという名前のルーチンをもっていたとすると，どちらかのfを改名してCからみた場合の曖昧さをなくさなければならない．もちろんこの改名は既存のクラスであるAもしくはBを変更してしまうわけでは「ない」，それを行っては広範囲に影響が及んでしまう．改名は参照する側であるCの中で行う．下の例ではAから継承したfを Af という名前に改名している．

```
  -- A，Bから多重継承を行うクラスC
  -- A，Bは同じfというルーチンをもつ
  class C
  …
```

```
inherit
    A rename f as Af;
    B
feature
...
```

このようにCが定義されているとき，以下のように二つのルーチンは呼び分けられることになる．

```
c : C ;
...
c.f -- Bのfが呼ばれる
c.Af -- Aのfが呼ばれる
```

もちろんこうした改名作業があまりにも多く発生するようでは，エンティティに保持されているオブジェクトの型によって適切なメソッドの選択が行われるいわゆる多相性（polymorphism）の利点を損なってしまう．また改名はシステムの理解を難しいものにしがちである．多重継承は実り多い手法であるが，改名を駆使してむやみに導入しては，システムの理解性が低下する．両刃の剣であることを，常に肝に銘じておかなければならない．

3.3.5 暫定クラス

クラス階層の中には，それ自身はインスタンスを直接作ることは決してなく，ある共通の性質を複数のサブクラスに対して与えるためだけに存在するものがある．こうしたクラスは抽象クラス(abstract class)[2),8)]と呼ばれるのが普通であるが，通常のオブジェクト指向言語では抽象クラスを明示的に宣言する手段がない．Eiffel は暫定クラス(defferd class) というクラスを明示的に指定することにより，抽象クラスを表現することができる．抽象クラスを明示的に表現できる能力は，特に設計言語として Eiffel を用いたり，再利用のためのクラス検索を行ったりする場合に有効である．

3.3.6 メモリ管理

メモリ管理は言語セマンティクスの一部とはとらえにくいが，オブジェクト指向言語ではオブジェクトの生成消滅が単なるメモリ獲得／解放以上の意味をもちえるので，あえてこの項で記述することにした．Eiffel のメモリ管理は自動ガーベジコレクションによって行われる．参照するものがなくなったオブジェクトは，適当なタイミングで再利用のために回収される．基本的な戦略はインクリメンタルなガーベジコレクションであり，ガーベジコレクションのタイミングに関しては，プログラムが実行効率やクリティカルセクションのために明示的に禁止したり，ガーベジコレクタが一回に走ることができる時間を指定できたりする．ただし言語仕様上は上記のような細かいコントロールができることが望ましいと書かれているだけで，実際の実現は各処理系に依存している（これは CLOS, Smalltalk なども事情は同じで，実際にどのようなアルゴリズムでガーベジコレクションを実現するかは個々の処理系に依存している）．現在市販されている Eiffel の処理系は皆インクリメンタル・ガーベジコレクションを行うようである．

3.4 システム統合のための仕掛け

大規模なシステムを構築しようとする場合，たとえば UNIX の make[4)] に相当するような構成管理システムが必要となる．このシステムは ACE (Assembly of Classes in Eiffel) と呼ばれる．ACE のための記述言語が LACE (Language for ACE) である．LACE は OS 上のどのクラス・クラスタ(ある目的のために構築された互いに関係するクラス群）をロードするか，表明の検査をどのレベルまで行うか，クラス・クラスタどうしでクラス名の衝突が起こったときにそれをいかに解決するかなどの手段を提供する．

特に最後のクラス・クラスタ間の名前の衝突の回避は重要な問題である．複数のクラス・ライブラリ・ベンダから購入したり，別々の部所で独自に開発されたクラス・クラスタの間でクラス名が衝突することは十分考えられる．たとえば C++ では現実にこの問題が起きはじめている．このような場合 LACE は，ちょうど多重継承に対して行った名前の衝突回避に似た手段を使って，クラスタ間のクラス名の衝突を解決する．

4. 応 用 事 例

4.1 Eiffel の処理系

Eiffel 自身で記述された，最も身近なシステムは，Meyer 博士の会社が頒布している Eiffel の処理系そのものである．version 2 までの処理系はC言語で記述されていたが，version 3 の処理系はほぼ全面的に Eiffel 自身を用いて書き直された．こ

の処理系の特徴は，言語仕様が最新のものを支援するようになったということももちろんだが，非常にビジュアルな開発環境を指向しているということである．オブジェクト指向設計を支援するCASE ツール，NeXTStep のインタフェースビルダに相当するもの，システムブラウザ，インタプリタなどが一体となったものである．残念ながらまだ筆者の手元に処理系が来ていないので，詳しい論評は差し控えるが，Meyer 博士本人の話を伺う限り，Smalltalk 並に統合された環境のようである．

4.2 設計言語としての Eiffel

Eiffel を実装言語ではなく，設計言語として用いる例も多く報告されている（実践的オブジェクト指向技術の会議である TOOLS のプロシーディングスなどを参照のこと）．本文中で何回か述べたように Eiffel はクラスの満たすべき制約，各ルーチンの満たすべき制約をさまざまな表明を使って表すことができ，クラスの組合せの妥当性を静的型チェックで解析することができる．このような性質をもつために，Eiffel を設計を表現する仕様記述言語として用いる例が多いのであろう．

4.3 Business Class

ヨーロッパの EC のプロジェクトの中に ESPRIT II と呼ばれるものがあるが，その分科の活動としてビジネス・クラスというプロジェクトが進められている．このプロジェクトはビジネス・アプリケーションのためのオブジェクト指向クラス・ライブラリを作成しようとするもので，実装言語として Eiffel が採択されている．既存の DBMS との連携もきちんと考慮するが，オブジェクト指向 DB に対するインタフェースの定義もその目標に含まれている．

プロジェクトの推進役は，Eiffel の開発協力者でもある，Jean-Marc Nerson 博士である．連絡は marc@eiffel.fr まで．

4.4 ユーザ会

定期的に Eiffel のユーザ会が開かれ，お互いの応用事例を発表しあう機会がある．アプリケーションは，事務処理から電子加速器の制御，数値計算まであらゆる範囲に広がっている．ユーザ会の会員なら，発表資料のコピーを手に入れることができる．ユーザ会に関する問合せは下記まで：

Interactive Software Engineering, Inc.
270 Storke Rd. Suite 7, Goleta, CA 93117 U. S. A.
phone : +1(805)-685-1006,
fax : +1(805)685-6869,
e-mail : users@ciffel.com

5. 標準化の動向

Eiffel の言語仕様は現在 NICE (the Nonprofit International Consortium for Eiffel) という非営利団体が標準化の責任を負っている．言語仕様自体は現在パブリックドメインであり，Meyer 博士の会社の製品だけが独占的に Eiffel という名前を使えるわけではない．NICE が言語仕様上 Eiffel として問題ないと認めれば，だれでも自由に Eiffel の処理系を作って売ることができる．NICE は Eiffel 処理系を提供するベンダ，ユーザなどから構成される団体で，将来的に言語仕様をどうするか，基本的なライブラリの仕様をどうするか，ベンダ相互のライブラリの互換性をどう考えるか，国際化に関してどうすべきかなどを話し合い，決定する．

問合せを先を下に示す：

Nonprofit International Consortium for Eiffel
P. O. Box 6884, Santa Barbara, CA 93160 (USA)
phone : +1(805) 685-1006,
fax : +1(805)685-6869

上記の Business Class に関する情報もここで得られるはずである．

6. おわりに

言語の特徴をということで始めたが，この解説記事は言語 Eiffel 全部の特性を網羅してはいない．細かく紹介しようとすれば，とてもこの紙面では足りなくなってしまう．それは言語仕様がそれなりに大きいということを示唆しているが，通常の言語と際だって異なる特徴は，そうした言語仕様一つ一つに，「なぜそうしたのか」という解説が加えられていることである[5),6)]．この説明があるがゆえに，Eiffel を巡る議論は単なる設計者の趣味の問題を超えて，言語設計の背後にあるより一般的な問題をいかに解決すべきかという方向に進みやすい．またあくまでも実用言語として設計されているため，既存の環境 (UNIX など) との整合性，実行効率もまず満足できるレベルにあると言える．NeXT の上ではインタフェースビル

ダと組み合わせて使うための部品がサードパーティから提供されているようである.

言語処理系を話題にするときに，仕様や実行効率，ソースコードとしての可読性などが中心の問題とされた時代は，オブジェクト指向言語の登場で終りを告げようとしているように思える．伝統的な手続き型プログラミング言語に対して，さまざまな利点が主張されるオブジェクト指向言語は，実は大規模なソフトウェア開発の場面で，かつ豊富なソフトウェア資産（クラス・ライブラリ）が存在するときにこそ，その力を十分に発揮できる.

本稿で触れてきたように，Eiffel は数々の優れた特徴をもつ「筋のいい」言語であるが，それは言語が普及し，生き残っていくための必要条件に過ぎない．Meyer 博士もその点に着目すればこそ，環境とクラス・ライブラリの充実に力を注いでいるのであろう.

商業的な成功という目でみれば，数あるオブジェクト指向言語のうちでもC++が圧倒的に大きなシェアを占めており，その状況はここ2，3年の間はますます顕著なものとなるであろう．しかしながら Eiffel が（すなわち Meyer 博士が）その言語設計を通して主張していることは，全てのオブジェクト指向言語自身の設計者，いわゆるオブジェクト指向分析／設計を行う人々，プログラマたちにあまねく理解されてしかるべきであろうと思う.

OOPSLA などの国際会議に出席して思うことは，現在 Eiffel はオブジェクト指向言語の参照モデルの地位を築きつつあるということである．ちょうどPascalが構造化プログラミングの黎明期に果たしたような役割を，Eiffel がオブジェクト指向プログラミングに対して果たすことになるのだろうか．参照モデルとして使われるのは結構なことだが，できることなら教科書や博物館の中だけに存在する言語で終わって欲しくはない.

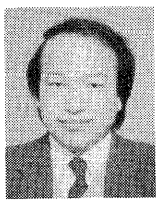
本稿が契機になって，一人でも多くの方が Eiffel に関心をもつようになっていただけるなら，筆者としてこれにまさる喜びはない.

謝辞　本稿執筆の機会を与えていただいた東工大情報処理センターの松田裕幸氏に感謝いたします.

参考文献

1) Avizienis, A.: The N-Version Approach to Fault-Tolerant Software, IEEE Trans. Softw. Eng., Vol. SE-11, No. 12, pp. 1491-1501 (Dec. 1985).
2) Coad, P. and Yourdon, E.: Object-Oriented Analysis, Yourdon Press, p. 233 (1991).
3) Cook, W.: A Proposal for Making Eiffel Type Safe, Proceedings ECOOP '89 (1989).
4) Feldman, S. I.: Make—A Program for Maintaining Computer Programs, Software, Practice and Experience, Vol. 9, pp. 255-265 (1979).
5) Meyer, B.: Eiffel: The Language, p. 594, Prentice-Hall (1991).
6) Meyer, B.: Object-Oriented Software Construction, p. 534, Prentice-Hall (1989)（邦訳「オブジェクト指向入門」二木監訳，酒匂訳，アスキー）.
7) Palsberg, J. and Schwartzbach, M. I.: Type Substitutions for Object-Oriented Programing, ECOOP/OOPSLA '90 Proceedings, pp. 151-160 (1990).
8) Rumbaugh, J. et al.: Object-Oriented Modeling and Design Prentice-Hall, p. 500 (1991)（邦訳「オブジェクト指向方法論 OMT」羽生田監訳，トッパン）.

（平成5年2月5日受付）



酒匂　寛

昭和33年生．昭和56年東京大学農学部獣医学科卒業．昭和57年(株)ソフトウェア・リサーチ・アソシエイツに入社．以後アプリケーション開発を経て，ワークステーション基本ソフト，ソフトウェア開発環境に関する研究開発等に従事．現在同社ソフトウェア工学研究所ボウルダー（米国，コロラド州）にて開発方法論ならびに開発環境に関する調査研究を継続中．ACM，IEEE 各会員.