要約

現在 ISO では、ANSI COBOL85 規格をベースに機能改善と新機能の追加を目指して標準化作業中である。COBOL は 1959 年ビジネス用プログラミング言語として開発されて以来、①ビジネスロジック表現に優れ、②既存資産、③要員、④言語処理系の安定性などの理由から現在でも多く利用されている言語である。

一方、ソフトウエアの生産性や保守性を向上させるキーテクノロジーとしてのオブジェクト指向技術が注目されており次世代 COBOL においてもオブジェクト指向プログラミング機能が改訂の目玉となっている。

OOCOBOL のドラフト仕様を元に標準化動向とオブジェクト指向に関する機能を検証してみると、オブジェクト指向分析設計成果物を十分実装できる言語仕様であり、既存資産の維持拡大に加えて段階的なオブジェクト指向プログラミングへの移行も可能な言語処理系となっている。

本稿では、標準化動向とオブジェクト指向に関する機能をサンプルをふまえ概説して、現時点の仕様の評価及び実務適用に際して留意すべき課題を紹介する。

要約 1
はじめに 3
１. 標準化動向 3
２. OOCOBOL 言語構成要素 4
２.1. 概要 4
２.2. プログラム構造 5
２.3. IDENTIFICATION DIVISION. 7
２.4. ENVIRONMENT DIVISION. 7
２.5. DATA DIVISION. 7
２.6. PROCEDURE DIVISION. 8
３. サンプル 9
３.1. サンプル OOCOBOL の解説 11
４. 評価と課題 14
４.1. オブジェクトの粒度問題 14
４.2. 強いカプセル化によるデータベース設計問題 14
４.3. 非オブジェクト指向プログラムとの共存問題 15
４.4. 要員教育問題-構造化 COBOL とオブジェクト指向 15
５. おわりに 15
参考文献 16
付録 A.COBOL2000 と C++機能比較 17
付録 B.COBOL2000 トピックス 19

はじめに

COBOL は 1959 年ビジネス用プログラミング言語として開発され、現在では 1985 年に制定された ANSI COBOL85 規格に準拠した処理系が最新版として提供されている。メインフレームの発展とともに企業の情報システムを構築する開発言語としての地位を確立しいる。現在でも利用される理由としては、①ビジネスロジック表現に優れている、②既存資産、③要員、④言語処理系の安定性などをあげることができるし、細かな改訂が継続されていて完成度の高い言語である。

ソフトウエアの生産性や保守性を向上させるキーテクノロジーとしてのオブジェクト指向技術は、オブジェクト指向分析や設計成果物を重要視しており、モデル変換を最小限にして実装するためにはオブジェクト指向開発言語が最適であり必須である。このため次世代COBOL は、構造化プログラミングを支援する COBOL85 をベースとしてオブジェクト指向プログラミング機能を改訂の目玉としている。

本稿では、OOCOBOL のドラフト仕様を元に標準化動向とオブジェクト指向に関する機能についてサンプルをふまえ概説し、現時点の仕様の評価及び実務適用に際して留意すべき課題を明らかにする。

１.標準化動向

現在提供されている ANSI COBOL85 の次期バージョンとして、多様な機能追加と改善の標準化作業が ISO/IEC(International Organization for Standardization and International Electrotechnical Commission)で 1989 年から進められており、当初は1997 年に publication される予定であったことから COBOL97 という仮称で呼ばれていたものである。

現在の標準化作業の状態は当初の予定から遅延しておりスケジュールが 2000 年にのばされ(COBOL2000)、さらに現時点では ISO の Committee Stage の最終段階にあって CD(Committee Draft)の最終版にあたる FCD(Final Committee Draft)の完成を目指しており、Publication Stage が 2002 年の予定となっている。(表 1) [2)]

新機能のうち最大のトピックスとして COBOL にオブジェクト指向言語構成要素が追加されるため OOCOBOL(Object Oriented COBOL)という別称が有名であり、通常 OOCOBOL を指す場合は現在 ISO で標準化作業中の言語仕様を指している。以降説明の便宜上 COBOL2000 または OOCOBOL と表記する。 [3) 4)]

COBOL2000 の標準化作業には大手コンピューターベンダやコンパイラメーカが積極的に参画しており、各社は CD の仕様に基づくコンパイラ製品をすでに提供しており標準化と同時に最終製品の出荷を可能にしている。

表 1 ISO/IEC ステージスケジュール

Stage 1	Proposal stage	NP(New work item proposal)
Stage 2	Preparatory stage	WD(Working draft)作成(1995 年 3 月～6 月)
Stage 3	Committee stage	CD(Committee Draft)を作成(1996 年 9 月～1997 年 1 月) (FCD のレビュー開始予定が 2000 年)
Stage 4	Enquiry stage	DIS(Draft International Standard)を作成予定(DIS が 2001 年)
Stage 5	Approval stage	FDIS(Final Draft International Standard)により最終採択
Stage 6	Publication stage(2002 年)	International Standard となる

表 2 COBOL2000 の主な新機能

- ビット操作と Boolean 値の提供
- 再起呼出可能な call
- コンパイラ制御指令
- 動的領域の確保(ALLOCATE 文、FREE 文)
- 動的に拡張可能な OCCURS TABLE
- 共有ファイル(共有と占有モード)
- フリーフォーマット(コンパイラ制御指示)
- インラインコメント(*>)
- オブジェクト指向プログラミング
- ポインタ
- TYPE、TYPEDEF 句
- 利用者定義 function

本技術解説では、OOCOBOL の部分すなわち COBOL に導入されるオブジェクト指向プログラミング機能の紹介と実務導入に際しての課題を整理する。

表 2は、COBOL2000 で提供予定の主な新機能であり簡単な解説を付録で紹介することとする。

なお OOCOBOL の仕様に関しては、ISO/IEC CD1.5 を基準にしている。[1)]

また用語に関しては JIS ハンドブックを基準にしている。 [5)]

本文中の COBOL 構文やサンプルにおいて［ ］記号は省略可能、{ } 記号はいずれかを選択、・・・記号は繰り返しを表している。

2.OOCOBOL 言語構成要素

2.1.概要

COBOL へのオブジェクト指向構成要素の実現は、まずプログラム構造としては COBOL85 で提供された入れ子プログラム構造を基礎にしている。クラスメソッドやインスタンスメソッドが複数定義可能であり、それぞれのメソッドが局所変数を宣言可能にするため、それぞれのメソッドを 1 つの入れ子プログラムとして実現している。

COBOL 言語に対する言語拡張方法としては、既存の COBOL プログラム構造とは別に新たにオブジェクト指向プログラミング専用のプログラム構造を用意した。前者は COBOL85 上位互換となるが

後者にはCOBOL85との上位互換性はない。また1つのプログラムにオブジェクト指向と非オブジェクト指向を共存させることはできない。ただし相互に呼び出す機構は提供されている。

COBOLではクラスを表すCLASSという用語が既に予約語となっているため、FACTORYという用語に代替している。インスタンスに対しては OBJECT という用語を割り当てている。通常のCOBOLの大きな構造は4つの部(DIVISION)から構成されるが、OOCOBOLでは一番外側が見出し部だけが存在している。さらにその中は2つの入れ子プログラムから構成され、1つはFACTORY段落を定義するためのプログラム、もう1つがOBJECT段落を含むプログラムとなっている。クラスに関する定義はFACTORY段落で、インスタンスに関する定義はOBJECT段落でそれぞれ定義することになる。

2.2.プログラム構造

COBOLにはコンパイルグループ概念が導入される。コンパイルグループには複数のコンパイルユニットを含むことができる。コンパイルユニットの種類としては以下の6つが定義されている。

- program-prototype
- function-prototype
- program-definition
- function-definition
- class-definition
- interface-definition

通常見かけるCOBOLはprogram-definitionとなり、OOCOBOLの場合はclass-definitionということになる。オブジェクトの部品(クラス定義)とその部品を使用するクライアントプログラム(通常見かけるCOBOL)とは別のコンパイルユニットとなる。

C++のようにクラス定義とメイン処理とが共存する書き方は出来ない。(図1)

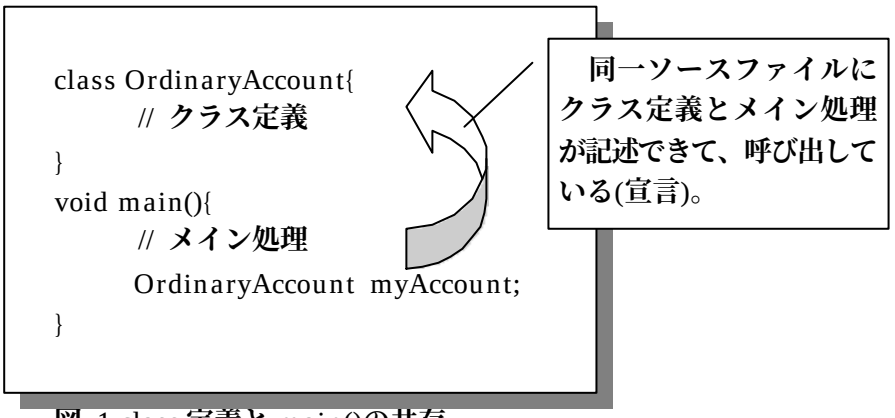

図 1 class定義とmain()の共存

またC++では1つのソースファイルに任意の数のクラス定義が可能だが、OOCOBOLでは1つのプログラム(コンパイルユニット)には1つのクラス定義しか出来ない。

コンパイルグループは

[{program-definitionやclass-definitionなど}･･･]

と定義されているので1つのファイルには、クラス定義を複数とかmainに相当するクライアント

プログラムも複数あわせて記述できる。

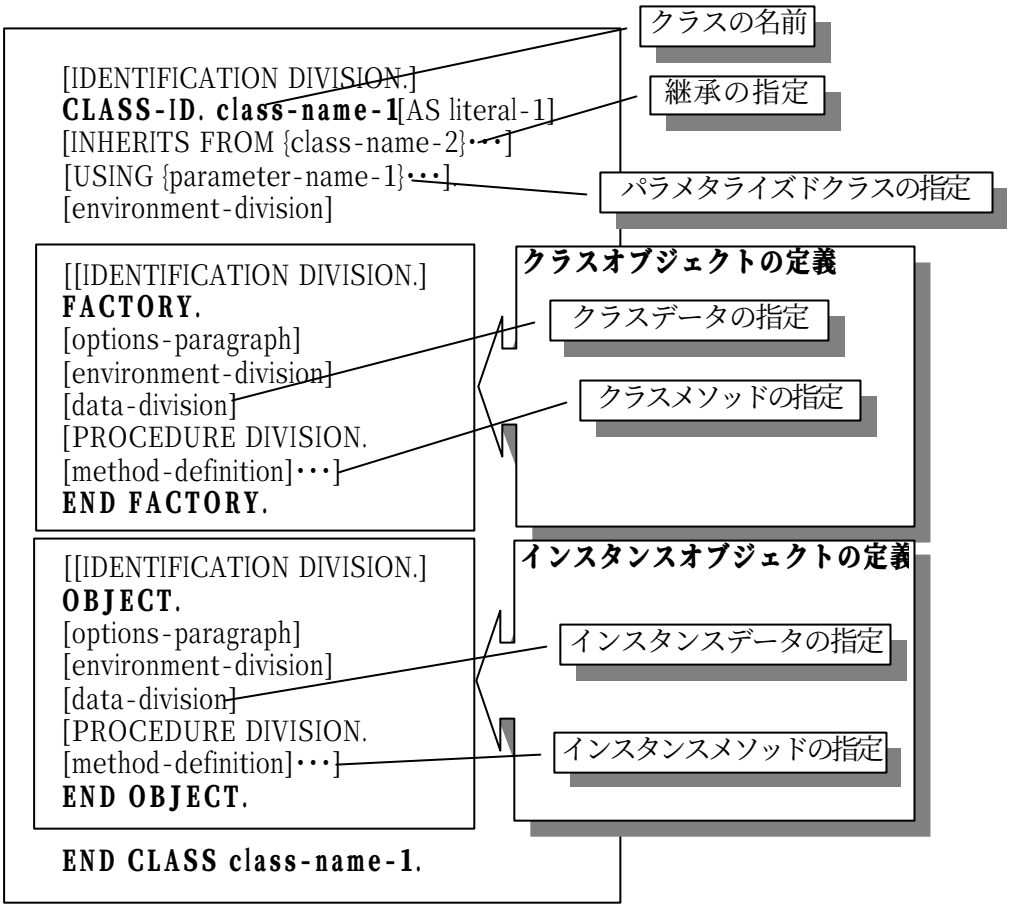

図 2 OOCOBOL 構造

説明上、program-definition のことをプログラム定義COBOL、class-definition のことをクラス定義 COBOL と呼ぶことにすると、
プログラム定義 COBOL のプログラム構造は

```
[IDENTIFICATION DIVISION.]
 PROGRAM-ID. program-name-1 [AS literal-1][IS {[COMMON] PROGRAM].
 ・・・
```

となり
クラス定義 COBOL のプログラム構造を、図 2に示す。

```
[IDENTIFICATION DIVISION.]
```

の[]は省略可能の表記なので、省略して

```
CLASS-ID. class-name-1
```

から書き始めてもよい。すると構文上省略できない OOCOBOL の構造は

```
CLASS-ID. class-name-1.
FACTORY.
END FACTORY.
OBJECT.
END OBJECT.
END CLASS class-name-1.
```

となる。クラス定義にはプログラム開始点は必要がないため PROCEDURE DIVISION が存在していない。

次はクラス定義の構造を DIVISION ごと概観する。

2.3.IDENTIFICATION DIVISION.

PROGRAM-ID の代わりに CLASS-ID を使用する。ということは C++のようにクラス定義と main()とが同一プログラムには記述できないことを意味している。

継承の場合は INHERITS で指定する。多重継承は可能で、クラス名の指定順序は同一のメソッド名が表れた場合に先に記述してあるクラスのものが有効となる。

```
CLASS-ID. 普通預金 INHERITS 流動性預金 .
```

2.4.ENVIRONMENT DIVISION.

クラスを再利用する場合は CONFIGURATION SECTION の REPOSITORY 段落にクラスを記述する。HASA 関係のクラスだけではなく ISA 関係としての親クラスも指定する必要がある。

```
REPOSITORY.
CLASS 流動性預金.
CLASS 顧客 IS 'CIF'.
```

顧客が internal name で、CIF が external name となる。この機能によって外部のクラス部品を内部に取り込むことができる。

通常の COBOL やオブジェクトのクライアントアプリケーションがクラス定義 COBOL のオブジェクトを使用したい場合も REPOSITORY 段落を記述しなければならない。この RIPOSITORY は IR(InterfaceRepository)と呼ばれている。

2.5.DATA DIVISION.

DATA DIVISION を定義できる場所は 3 か所あり、それぞれ意味が異なる。

DATA DIVISION の定義場所	データ項目の意味
FACTORY 段落の DATA DIVISION	クラス変数
OBJECT 段落の DATA DIVISION	インスタンス変数
METHOD 定義の中の DATA DIVISION	メソッドの中での局所変数

また METHOD 定義での DATA DIVISION の場合、SECTION の種類によって変数の寿命が異なる。

SECTION の種類	データ項目の意味、寿命
LOCAL-STORAGE SECTION	メソッドを EXIT すると値は保証されない。
WORKING-STORAGE SECTION	メソッドを EXIT しても値は残っている。再度同じメソッドに ENTER した場合前回の値が保証される。
LINKAGE SECTION	引数や戻り値を定義に使用する。

●オブジェクトの利用(宣言)

```
01 W普通預金 USAGE IS OBJECT REFERENCE.
01 W当座預金 USAGE IS OBJECT REFERENCE 当座預金.
```

データ項目としてオブジェクトを利用する場合、OOCOBOL では OBJECT REFERENCE を使用する。すべてのオブジェクトは少なくとも 1 つの OBJECT REFERENCE がある。C++のようにインスタンスそのものを直接埋め込んで使うことはできない。オブジェクトは FACTORY の new メソッド(インスタンス生成メソッド)によって作成しておき、そのインスタンスをさすように OBJECT REFERENCE を初期化する。

OBJECT REFERENCE には typed と untyped があり、typed では指定されたクラスだけではなく下位のクラスを指し示すことが可能である。untyped の場合はどのようなクラスでも指し示すことができる。

あらかじめ定義されてい特殊な OBJECT REFERENCE として次の 4 つが用意されている。

- EXCEPTION-OBJECT
- NULL
- SELF
- SUPER

EXCEPTION-OBJECT はシステムが例外発生時に設定するオブジェクトである。NULL は OBJECT REFERENCE がオブジェクトを指し示していない状態を表している。SELF は C++の this 相当で自分自身のオブジェクトを指している。SUPER は自分の上位クラスを指している。

2.6.PROCEDURE DIVISION.

●メソッド定義

OOCOBOL のメソッドは C++で言うところの virtual かつ public 宣言されたものとなり動的結合を支援する。OOCOBOL では多態性のモデルを実装することができる。

```
METHOD-ID. 入金 OVERRIDE.
```

OOCOBOL では強いカプセル化を実現するためデータ項目について公開/非公開という指定ができない。そのためにメソッドの種類としてプロパティが提供される。個々のデータ項目に対するアクセスは GET PROPERTY や SET PROPERTY を使用する。

```
METHOD-ID. GET PROPERTY 口座番号.
```

上位クラスのメソッドを書き換える場合(遮蔽定義)は、OVERRIDE を指定する。また各メソッドはメソッド名(例では「入金」)と自分自身の DATA DIVISION と PROCEDURE DIVISION を持っている。DATA DIVISION は当該メソッドの局所変数を定義する。メソッドへの引数や戻り値もここで定義する。

メソッドは呼び出す対象オブジェクトの種類に応じて、次の 2 種類となる

・FACTORY メソッド
・オブジェクトメソッド

●オブジェクト参照(OBJECT REFERENCE)の設定方法
大きくは 2 通りの方法が用意されている。
(1)FACTORY の new メソッドの戻り値として設定する。

```
01 W普通 USAGE IS OBJECT REFERENCE 普通預金.
・・・
INVOKE 普通預金 "new" RETURNING W普通.
```

W普通には生成された普通預金クラスのインスタンスへのオブジェクト参照が代入されている。

(2)SET 文で設定する。

・SET A TO B.

オブジェクト参照 A をオブジェクト参照 B が指しているものに設定する

・SET A TO NULL.

オブジェクト参照 A を NULL に設定する(どこも指し示していないようにする)

・SET A TO SELF.

オブジェクト参照 A が自己を参照するように設定する

●MessagePassing request 方法(オブジェクト参照の呼出方)
2 つの記述方法がある。

INVOKE 文
・INVOKE 普通預金 "入金" USING PARAM.
inline メソッド呼出
・普通預金::入金(PARAM).

呼び出したメソッド名が対象オブジェクトに定義されている場合はそのメソッド特定される。対象オブジェクトに定義されていない場合、INHERITS 句に指定されている上位クラスを左から右に探索する。見つからない場合は EXCEPTION-OBJECT が返される。
invoke ではメソッド名情報を変数に格納することも可能でリテラルであれば static link となり、データ項目であれば dynamic link となる。データ項目の場合は、データ項目変数に格納されている文字列値をメソッド名とみなす。

３.サンプル

金融業務の普通預金口座をモデル化した普通預金クラスを例に C++と OOCOBOL との簡単なサンプルを元にして解説する。(表 3)

表 3 C++とOOCOBOLによるクラス定義例

【C++によるクラス定義イメージ】

```
#include "流動性預金.h"
#include "顧客.h"
class 普通預金 :流動性預金{
private:
  string 店番;
  string 口座番号;
  int 支払可能残高;
  顧客 myCIF;
  //その他の属性
public:
  int 入金(int IN 入金額);
  //流動性預金の入金を override(遮蔽定義)
  int 出金(int IN 出金額);
  //その他のメソッド
};
```

【OOCOBOL によるクラス定義イメージ】

```
IDENTIFICATION DIVISION.
CLASS-ID. 普通預金 INHERITS 流動性預金.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
REPOSITORY.
CLASS 流動性預金.
CLASS 顧客.

FACTORY.
PROCEDURE DIVISION.
METHOD-ID. 口座開設.
DATA DIVISION.
LINKAGE SECTION.
01 IN 店番 PIC X(3).
01 IN 口座番号 PIC X(7).
01 INmyCIF USAGE IS OBJECT REFERENCE 顧客.
01 RESULT PIC 9(3).
PROCEDURE DIVISION
                USING IN 店番 IN 口座番号 INmyCIF
                RETURNING RESULT.
口座開設-start.
      EXIT METHOD.
END METHOD.
END FACTORY.

OBJECT.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 INSTANCE-DATA.
      03 店番
      03 口座番号
      03 支払可能残高
    03 myCIF USAGE IS OBJECT REFERENCE 顧客.

PROCEDURE DIVISION.
METHOD-ID. 入金 OVERRIDE.
DATA DIVISION.
LINKAGE SECTION.
01 IN 入金額 PIC 9(12).
01 RESULT PIC 9(3).
PROCEDURE DIVISION USING IN 入金額
                        RETURNING RESULT.
入金-start.
      EXIT METHOD.
END METHOD.

METHOD-ID. 出金.
DATA DIVISION.
LINKAGE SECTION.
01 IN 出金額 PIC 9(12).
01 RESULT PIC 9(3).
PROCEDURE DIVISION USING IN 出金額
                        RETURNING RESULT.
出金-start.
      EXIT METHOD.
END METHOD.

END OBJECT.
```

	END CLASS.

３.1.サンプル OOCOBOL の解説

CLASS-ID. 普通預金 INHERITS 流動性預金.

C++ではクラス定義と同時にプログラムの開始点となるメインプログラム void main() を記述できるが、OOCOBOL ではプログラム開始点を持つプログラムはクライアントプログラムと呼ばれていてクラス定義のプログラムとは区別されている。PROGRAM-ID ではなく CLASS-ID と書くとクラス定義だけがされているプログラムということになる。

CLASS-ID に続く 普通預金 がクラス名となる。

CLASS-ID. 普通預金.

と書けば 普通預金クラスをゼロから定義することができる。例では INHERITS とあるので 流動性預金 というクラスから継承して 普通預金 クラスを定義している。

REPOSITORY.
CLASS 流動性預金.
CLASS 顧客.

REPOSITORY 句は C++の#include に相当する。普通預金クラスで再利用するクラス部品をすべて REPOSITORY 句で宣言しなければならない。対象となるのは HASA 関係として包含するクラス部品(顧客)だけではなく ISA 関係の上位クラス(流動性預金)も書かなければならない。どのクラスを部品として再利用するかという情報をコンパイラに入力しなければ、流動性預金がどのような主属性を定義しているのかとかどのようなメソッドがあるのかがわからず、コンパイル時に検査することができないためである。

FACTORY.
OBJECT.

CLASS-ID で始まったクラス定義は END CLASS. で終わるが、さらにその内部は大きく REPOSITORY 段落 と FACTORY から END FACTORY.までの FACTORY 段落 と OBJECT から END OBJECT までの OBJECT 段落 の 3 つの段落から成り立っている。

OOCOBOL ではクラスに関わる属性や操作の定義と インスタンス(=オブジェクト)に関わる属性や操作の定義ができる。前者クラスに関する部分が FACTORY で、インスタンスに関する部分が OBJECT となる。CLASS はすでに予約語となっていたため新たに FACTORY という予約語を導入した。

例ではクラス属性の定義がないが必要ならば

FACTORY.
DATA DIVISION.
WORKING-STARAGE SECTION.

01 普通預金口座総数 PIC 9(5).

のように記述する。例では

FACTORY.
PROCEDURE DIVISION.

となっているのでクラスメソッドが定義されていることになる。

METHOD-ID. 口座開設.

メソッド名はMETHOD-IDで指定し、ここでは「口座開設」となる。

DATA DIVISION.
LINKAGE SECTION.
01 IN店番 PIC X(3).
01 IN口座番号 PIC X(7).
01 INmyCIF USAGE IS OBJECT REFERENCE 顧客.
01 RESULT PIC 9(3).

続くDATA DIVISIONではメソッドの引数や戻り値、口座開設メソッド内で使用される局所変数を定義する。
INmyCIF は顧客クラスのインスタンスへの参照が引数で渡されることを意味している。OOCOBOLではC++とは異なり顧客インスタンスそのものを埋め込むのではなく オブジェクト参照(OBJECT REFERENCE)を使用する。オブジェクト参照はオブジェクトそのものではなくオブジェクトを指し示すものとなる。オブジェクト参照として定義したからと言って顧客インスタンスが存在するのではなく、他の箇所で顧客クラスのクラスメソッドを呼び出してインスタンスを生成して、オブジェクト参照が生成されたインスタンスを指し示すように初期化する必要がある。

01 INmyCIF USAGE IS OBJECT REFERENCE 顧客.

OBJECT REFERENCE の後ろにクラス名として 顧客 が指定されているが、これは INmyCIF が顧客インスタンス専用のオブジェクト参照であることを宣言している。INmyCIF を普通預金クラスや明細クラスのオブジェクト参照としては使えないように、つまり型付けを制限することができる機能を提供している。クラス名が指定されていない場合、

01 INmyCIF USAGE IS OBJECT REFERENCE.

INmyCIF は任意のインスタンスのオブジェクト参照になることができる。その場合型付けの検査は行われない。

PROCEDURE DIVISION USING IN 店番 IN 口座番号 INmyCIF RETURNING RESULT.

ここでは、IN店番、IN口座番号、INmyCIFが入力引数となり、RESULTが戻り値となる。

END METHOD.

ここまでが 1 つのメソッドとなる。

END FACTORY.

ここまでが FACTORY で、クラスのインスタンスやメソッドの定義が終わる。

OBJECT.にはじまって
END OBJECT.まで

この囲まれた範囲がインスタンスに関する定義部分となる。インスタンス変数とインスタンスメソッドが定義可能である。

DATA DIVISION.

例では、DATA DIVISION からはじまっているが、これがインスタンス変数を記述する場所となる。その 9 行あたり下にも DATA DIVISION があるが、これはメソッドの内部にある DATA DIVISION でメソッドの局所変数やメソッドの引数を定義する場所となる。「入金」「出金」では LINKAGE-SECTION なので引数のサンプルということになる。

WORKING-STORAGE SECTION.
01 INSTANCE-DATA.
03 店番
03 口座番号
03 支払可能残高
03 myCIF USAGE IS OBJECT REFERENCE 顧客.

インスタンス変数としては店番、口座番号、支払可能残高、myCIF が定義されている。これは普通預金クラスのインスタンスごとにこれらのインスタンス変数領域が確保されることになる。

PROCEDURE DIVISION.

これがインスタンスメソッドの定義の始まりで

METHOD-ID. 入金 OVERRIDE.

OVERRIDE とは上位クラスにあたる流動性預金にも入金メソッドが定義されていて、流動性預金から派生して作成した普通預金クラスはなにも定義しなくても流動性預金の入金メソッドを継承することになるが、普通預金クラスにおける処理が異なるため OVERRIDE と指定することで遮蔽定義を可能とする。

END CLASS.

ここで CLASS 定義が終わる。

以上が、OOCOBOL で 1 つのコンパイルユニットとして普通預金クラスを定義するサンプルとなる。

４.評価と課題

C++と OOCOBOL での実装イメージ比較から、OOCOBOL では表現にかなり冗長度はあるもののオブジェクト指向設計における成果物を表現する能力は十分だと言える。

標準化の特徴として次にあげる観点から説明を加える。

- ●オブジェクトの粒度を大きくとらえている
- ●カプセル化の強化
- ●最新のオブジェクト指向技術対応
- ●上位互換性と段階的再構築

COBOL に対してオブジェクト指向プログラミング機能を取り入れる場合に、ISO ではデータ型を拡張する方式ではなく既存のプログラムとは独立して別のプログラム構造としてクラスを位置づけている。クラスがプログラムや関数と並列に定義されていることからも、想定するオブジェクトの粒度は大きくとらえていることがわかる。日付型とかカウンタ型というクラスの部品を否定するものではないが処理効率や管理負荷を考えればあまり小さな部品を再利用するよりも、ビジネスアプリケーション指向の COBOL としてはモジュールやファンクションレベルでの適用を想定している。

カプセル化において特筆するのは継承関係であってもカプセル化を徹底させている点である。これは継承構造において上位クラスの構造を修正すると下位クラスに対して修正が及んでしまう可能性が高く、多段継承による巨大な業務部品構造には致命的となる。このため継承関係に対してもカプセル化を徹底させて属性参照のためにプロパティ機能を導入している。

ISO は UML や CASE ツールなどの最新技術動向にも対応するべく、パラメタライズドクラスやプロパティ、インタフェースといった概念を OOCOBOL の中に取り入れている。しかし CD の各バージョンでは INTERFACE の取扱に対して Java と同様に独立させるかどうか判断が揺れているようである。

既存の COBOL と上位互換をとらなければならない要求に対して ISO は現実的かつ有効な仕様としてまとめ上げている。この仕様の方向であれば既存資産の維持拡大も段階的なオブジェクト指向プログラミングへの移行も可能な言語処理系として期待できる。

次にいくつかの課題と留意点を紹介する。

４.1.オブジェクトの粒度問題

COBOL 以外のハイブリッド型 OOP の場合では、コンパイラが提供する基本データ型に抽象データ型として利用者定義型を定義できる方向で実現されている。C++、Object PASCAL や JAVA などはカウンタ型とか残高型のようにかなり粒度の小さなデータ型を定義できるし、実用上問題がない。COBOL に対する拡張においてはデータ型という概念が本来存在しないため、PICTURE 句や USAGE 句に拡張あるいは代替する言語構成要素が新たに定義されるのではないかとも想像できたが、ISO 標準化では別の実装方法を選んでいる。

OOCOBOL ではインスタンスを直接定義する実現形式ではなく、オブジェクトへの参照という方式を選んだ。このためすべてのインスタンスはクラス(FACTORY)での生成メソッド呼出を必要とする。結果、属性レベルで適用するのはコード量や CPU 負荷に影響を及ぼすため、エンティティに近い粒度として分析設計するのが妥当である。

４.2.強いカプセル化によるデータベース設計問題

COBOL2000 ではカプセル化が厳しく適用されており、継承関係にあっても上位クラスで定義された属性を直接参照する手段が提供されていない。すなわち継承は C++や JAVA における private 宣言されているものとみなされている。

ここで問題になるのはデータベースの物理設計である。流動性預金と普通預金とのように上位クラスと下位クラスとを分けて設計すると各クラスに永続機能を実現させるためデータベースへの mapping を行うことになる。つまり下位クラスである普通預金の属性に対する参照は、上位で定義された属性を直接参照できないため必ずメソッド呼出を必要としてしまう。これは継承関係と委譲関係との混同である。永続属性に対してはさらに深刻でクラス単位でデータベースとの mapping がカプセル化されているため、下位クラスへの参照はそれ自身のデータベースに加えてすべての上位クラスのデータベースに対する I/O 動作が入ってしまい効率上の問題が発生する。

ISO が FDIS においてこの機能を標準化案として採択すると、設計上あるいは効率上の問題からデータベースの物理設計は最下位クラスで mapping することになるので、①上位クラスは抽象クラスとなり、主に存在理由は多態性を実現するためと見るか、②上位クラスで多数のプロパティを定義しておき下位クラスでは自分の領域へ転記を行う方式のいずれかとなる。

４.3.非オブジェクト指向プログラムとの共存問題

COBOL2000 では 1 つのプログラム(正確にはコンパイルユニット)に COBOL85 でのプログラム構造と拡張されたオブジェクト指向構造を同居させることはできない。ただしコンパイルユニットの種類で定義されている 6 つのプログラムはその他のプログラムを相互に呼び出すことが可能となっている。

共存問題はレガシーラッピング構造とも関わるが、非 OO 部には状態が存在しない。しかし OO 部では永続性を含めてインスタンスやクラスの状態が存在する。両者状態を持たないレベルで共存させる場合には問題は発生しない。問題が発生するのは、状態が存在するというレベルで両者を共存させる場合であり、その場合 2 つの対応策が考えられる。

・OO 部(状態管理あり)から状態管理のない非 OO 部を呼び出す
・OO 部から状態管理ができる非 OO 部を呼び出す

後者の場合には OO 部と非 OO 部との間に状態を管理する機能を実現するための機構が必要となる。簡単には OO 部側に非 OO 部のプロキシを用意すればよい。

４.4.要員教育問題-構造化 COBOL とオブジェクト指向

COBOL2000 のオブジェクト指向機構は COBOL85 をベースとして機能拡張している。このためプログラムを入れ子構造にすると言うモジュール化構造を前提にしている。このため現在の COBOL85 を実際使用してはいても COBOL74 構造をベースに COBOL85 で提供された新機能(EVALUATE 文)を用いている要員は、本来 COBOL85 で提供されていたモジュール化構造を理解する必要がある。またオブジェクト指向プログラミング技術に加えてオブジェクト指向分析・設計技術を習得する必要もある。

５.おわりに

オブジェクト指向技術が普及定着するにつれて C++や Java をはじめとするオブジェクト指向開発言語が台頭してきて COBOL が過去の言語であるという評価を目にすることがある。既存資産や要員数の観点から COBOL の改訂になる OOCOBOL に期待するという考え方もあるが、ビジネスアプリケーションに注目した場合、COBOL 以外のオブジェクト指向開発言語が OOCOBOL よりも保守性・生産性・安定性において優れているのかどうか比較検証するべきであろう。

オブジェクト指向技術による分析や設計成果物は、OOCOBOL で実装することができるし、既存の COBOL を段階的に再構築する場合にも分析や設計技法としてオブジェクト指向技術を適用することができるのである。ただし OOCOBOL も道具としての開発言語の 1 つに過ぎない。採用したからと言って無条件に効果が期待できるものではない。実務適用で成功させるためには、顧客の要求を正しく表現したオブジェクトモデルの作成と、その設計成果物を実装するためのオブジェクト指向開発

言語の利用技術が重要となる。OOCOBOL の標準化を目前に、COBOL という開発言語とオブジェクト指向技術の再認識・再評価につなげていきたい。

なお執筆中にISO CD 1.6を入手したが時間的制約から今回の技術解説には取り込むことができなかった。

参考文献

[1] Committee Draft 1.5 Proposed Revision of ISO 1989:1985 April 1999、
Information Technology Programming Language, - their environments and system software interface - Programming Language COBOL，1999

[2] ISO/IEC Directives Part 1，ftp://ftp.iso.ch/pub/out/directives/en/dirp1.html，1992

[3] 西村恕彦 編著，電子計算機プログラム用言語 JIS COBOL 全釈，オーム社，1972

[4] 加藤 昭，COBOL85 プログラミング 新規格の機能と使い方，啓学出版，1987

[5] 日本規格協会，JIS ハンドブック情報処理 プログラム言語編，1997

OOCOBOL に関する有益な情報は以下の URL サイトから入手することができる。

□オブジェクト指向全般への INDEX

http://mini.net/cetus/software.html

□COBOL 標準化

http://anubis.dkuug.dk/jtc1/sc22/wg4/ (JTC1/SC22/WG4 - COBOL)

□COBOL 一般情報

http://www.infogoal.com/cbd/cbdhome.htm (THE COBOL CENTER)

付録 A.COBOL2000 と C++機能比較

(注記 表中の ― 印は該当する機能が無いことを表している)

機能	COBOL2000	C++
プログラム構造	コンパイルユニット 1 つのクラス定義は 1 つの CLASS-ID に対応	1 つのクラスは class{}で定義
再利用	REPOSITORY 段落	#include
クラス定義	FACTORY 段落 OBJECT 段落	class{}で定義する
クラスメソッド	FACTORY 段落内の PROCEDURE DIVISIONで定義	メンバ関数に static
クラスデータ	FACTORY 段落内の DATA DIVISIONで定義	メンバに static
インスタンスメソッド	OBJECT 段落内の PROCEDURE DIVISIONで定義	メンバ関数
インスタンスデータ	OBJECT 段落内の DATA DIVISIONで定義	メンバ
アクセス制御	指定できない。すべて PRIVATE 扱い	public、private、protected
プロパティ	METHOD として GET PROPERTY SET PROPERTY	―
データ抽象	粒度は大きなものに向いている	粒度は小さなものにも対応できる
カプセル化	メソッド、プロパティ経由以外データ構造にはアクセスできない。 継承関係であってもカプセル化は機能する	カプセル化は実現できる。 例外として friend が用意されている
継承	INHERITS 指定 多重継承可能	: 継承指定 多重継承可能
多態性	メソッドは動的結合する	静的結合と動的結合をサポート
クラスライブラリ	BASE class が提供される	言語処理系では規定せず BASE class 相当はない
多重定義	できない METHOD-ID で一意となる	できる 引数のシグニチャ
オブジェクトの宣言	参照方式 OBJECT REFERENCE 指定	変数の宣言(埋め込み) ポインタ型 参照型
オブジェクトの呼出方法	INVOKE 文 ::(インラインメソッド呼出)	ポインタ型の場合は->演算子 参照型などの場合は.演算子
オブジェクトへの代入	SET 文	= 通常の代入文
特殊なオブジェクト	EXCEPTION-OBJECT NULL SELF SUPER	this 上位のクラスへの参照は::で直接指定する
コンストラクタ(自動呼出)	―	指定可能

デストラクタ(自動呼出)	―	指定可能
オブジェクトの生成	FACTORY の NEW メソッド	変数として宣言した場合は暗黙に処理系が生成 ポインタ型の場合は new 演算子による
オブジェクトの破壊	自動	変数として宣言した場合はブロックを exit する際に処理系が暗黙に破壊 ポインタ型の場合は delete 演算子による
パラメタライズドクラス	可能 USING 指定	可能 < >

付録 B.COBOL2000 トピックス

COBOL2000 の新機能のうち重要なものを紹介する。

新機能	概略
ファイル共有とレコードロック機能	ファイルを複数のプログラムで同時に共有使用できる(OPEN 文で SHARING 指定)。 LOCK 機能は使えるが COMMIT とか ROLLBACK といったトランザクション処理機能ではない
ダイナミックテーブル	テーブルサイズを必要に応じて動的に増加できる。最大値を設定することも可能 (例)OCCURS 最小値 TO 最大値 DEPENDING ON テーブルサイズを決定するデータ項目 DYNAMIC
ソートテーブル	SORT 文はテーブルをソートできる SEARCH ALL 文と組み合わせて高度な検索機能を実現できる
コンパイルグループとランユニット	同一コンパイルグループの中で定義されているコンパイルユニット(プログラムや関数やクラスなど)は相互参照することができる (例)factorial 関数を FUNCTION-ID で定義しておけば、同一コンパイルグループ内のランユニットから COMPUTE 文の中で関数名 factorial で呼び出すことができる
プログラム間通信機能	4 形態のプログラム間通信機能を提供 ・制御のトランスファ ・パラメタの送出 ・共有データ領域 ・ファイル共有
MCS(Message Control System)－通信資源－	ローカルやリモートのデバイスに対して Queue を介在させて、プログラムからは SEND 文、RECEIVE 文で電文入出力のやりとりができる。次期の規格で削除される予定。
イントリンシック関数提供	汎用的なイントリンシック関数(引数を受け取り結果を返す関数)を提供。 例えば算術関数であれば、算術式の中で算術項として使用することができる。
デバッグライン	7 カラム目に D マークを付けることでその行をデバッグラインととすることができる。デバッグラインは SOURCE-COMPUTER 段落で WITH DEBUGGING MODE 句を指定することで実行プログラムとして翻訳されるが、指定しない場合はコメント行として扱われる。
アドレスとポインタ	システムプログラミング用に提供。データとプログラムへのアドレス参照が可能となる(USAGE POINTER)
BOOREAN(論理値)と BIT 操作機能	ビット操作が可能、各ビットは BOOLEAN 値として操作できる また B-AND、B-OR などの論理演算子も提供。 (例) USAGE BIT VALUE B"0110" MOVE B"1" TO データ項目(1:1)
コンパイラ指令	コンパイラに対する制御指令。 例えば、DEFINE,EVALUATE,IF 指令を組み合わせることによってコンパイル時にコンパイル対象となるソースコードを変更することができる。 (例) >>DEFINE test AS 1 >>IF test IS DEFINED >>END-IF >>EVALUATE test

	>>WHEN 1
フリーフォーマット	コンパイル指令で、>>SOURCE FORMAT IS FREE を指定することでフリーフォーマットにすることができる。また任意の場所で FIXED フォーマットに切り替えることも可能。
インラインコメント	行中の任意の場所で *> を指定することで行末までをコメントとすることができる。
その他 強化改善される機能	●TYPE と TYPEDEF 句 ●FILE connector による論理と物理ファイル割り当て ●call 文による再帰呼出 ●マルチランゲージやカルチャ対応(ローカル対応) ●例外処理 ●REPORT WRITER ●VALIDATION 機能 ●Conditional Expression(条件評価のショートカット) ●キャラクターセットの国際化対応 ●利用者定義関数(FUNCTION-ID) ●SCREEN 操作