＜実践報告・資料＞

情報技術に必要な記号論理の教育

－UMLその他の現在の設計手法との関係－

職業能力開発総合大学校東京校　　福　良　博　史

The necessity for the knowledge of symbolic logic to the softwa re engineer
－The relation with the software design methodologies (UML and s oon)－

Hirofumi FUKURA

要約　「ものづくり」としてのソフトウェア製作は、ビジネス、工場、組込み製品など、多様な分野に関与している。しかも近年とくにソフトウェアの不具合が社会に与える影響が深刻化しつつある。このことから、より信頼性の高いソフトウェアをつくるためには信頼性の高い設計・実装のための技術が求められる。

現在の東京校のカリキュラムにおける記号論理の取り扱い、プログラミング言語の発展、要件からの論理をまとめるための事例などを説明し、UMLによるオブジェクト指向の設計・実装の方法論などにも記号論理の素養が欠かせなくなってきていることを述べる。そのため今後は、記号論理の教育が情報技術の基礎となることが望まれる。

Ⅰ　緒　言

現在、短大に相当する専門課程と大学3，4年に相当する応用課程の情報関連の情報技術科と生産情報システム技術科におけるカリキュラムの中には、記号論理を主として訓練する科目がほとんどない。情報技術科では、ディジタル工学2単位中に、6項目の分野がありその中の1項目に「論理代数」という分野がある。つまり1時限で半年18日と考えると、大雑把に3日間（つまり3時限）が平均的な「論理代数」に割当てられる日数となる。当校の場合で見てみると、シラバスでは3日程度が割り振られている。そして、生産情報システム技術科では、基礎的な科目は、専門課程で完了しているという前提から、全く教える科目がない。情報技術の分野は多岐にわたるためどこに重点を置く必要があるのかによって基礎となる科目が大きく異なってくると考える。

当校は、「ものづくり」に焦点を合わせた教育訓練施設と位置付けられている。とすると、組込み系システムからビジネスアプリケーションまで含めた応用のための基礎技術として、ソフトウェアについての設計から実装・検証・保守まで含めた総合的なソフトウェア工学を身に付けるような教育が必要だと判断する。

現在のソフトウェア工学の流れから考えるとUML (Unified Modeling Language™)[(1)]による設計から実装までについての技術が社会に出てから欠かせないものとなりつつある。この現在の流れの中で、ソフトウェアのものづくりのために必要な技術として、記号論理を教科として重点を置く必要性が出てきていると考える。以下にその現状を設計・実装技術として説明し、これからの方向性を考え具体的なカリキュラム例を提示する。

Ⅱ　設計・実装技術

1　設計とプログラミング言語

ソフトウェア設計の流れを概観すると、おおよそ以下のようになる。

はじめは、第二次世界大戦の最中にドイツのエニグマという暗号を解読するためにどうすればよいか、というところから高速演算できる機械が模索され、チューリングからノイマン型コンピュータへと発展してく

論文受付日　H17.12.27

る。このコンピュータに仕事の指示をするためには、機械に理解できる'0'と'1'の数字の羅列を与えることから始まる。しかしこれでは非効率で、あとからみても人が理解することが困難を極める。そこで、人が理解しやすいような文字をコードに対応させ（ニューモニック・コード：例えば'0110'がaddのように）、人はそのコードをを利用して指示を書きアセンブラがその指示を機械が理解できる'0'と'1'のコードに翻訳し、コンピュータに入力するようになる。しかし、このアセンブラ言語の場合は、指示のニューモニック・コードと、コンピュータへの命令とは、一対一に対応している。このためやはり人が理解するためにはかなり時間をかけなければならなかった。そこで高級言語と呼ばれる手続き型の言語が登場してくる。COBOLとFORTRANがその初期の代表的な言語として登場する。COBOLは、事務処理に特化した言語で、記述自身に当時としては、画期的と思うようなソースコードにドキュメント性を意識した構造になっている。FORTRANは、科学技術計算に特化した言語として流通する。こうしてアセンブラより構築しやすい言語の出現により爆発的にプログラムが作られるようになる。このため、無秩序に膨大なソースコードが公害のように作り出されたため、ソフトウェア危機という言葉が出現してくる。そして管理しやすく保守が容易なソースコードが求められるようになる。そこで、ソフトウェア工学の必要性が出てきた。プログラミングの設計には、フローチャートが利用されることが多かった。フローチャートには、当時の言語を反映し、goto文に該当する、任意の場所に飛び出せる記述が許されていた。このプログラミングの流れの中で、1966年にBoehmとJacopini[(2)]により、手続き型のプログラミングの表現について、手続き型の制御構造には、（1）順次処理、（2）判断分岐処理、そして（3）ある条件の間繰り返す処理の「3種類」の要素ですべてのことが表現できる、ということがわかった。これは、goto文のような分岐がなくて良いということを意味している。1968年のDijkstra[(3)]に始まるgoto文の是非を問う論争となり、無秩序なコーディングを改め、秩序を持ったプログラミングをしていこうという運動がはじまる。この年の10月に開催されたNATO のSoftware Engineering Conference[(4)]によって、ソフトウェア工学が世間に知られるようになる。フローチャートによる設計が、手続き型の3種類の制御構造のみを用いる形態で記述するような基準化が色々考案された。そして、ソフトウェア工学の中心テーマが構造化プログラミング、構造化設計へと進んでいく。

構造化プログラミング言語が出てきたところで、それまでのフローチャート一辺倒の設計から、よりソフトウェアの開発から保守までが容易になるようにという観点から構造化設計とよばれる設計方法が登場してくる。この構造化プログラミングは、プログラムのデータ管理という観点から弱点が指摘され、情報隠蔽、抽象データ型などの考え方が提起され、その後C++やJavaなどのオブジェクト指向のプログラミング言語に取り入れられ、実用化されてきた。かつ、Javaの場合は、アサーション機能も追加されることになる。このオブジェクト指向の考え方を取り入れた設計手法が各種出現してくるなかで、OMT法などを中心に幾つかの手法を統合したUMLが出現してくる。

2 各種プログラミング言語の例

ここで参考までに1から10までの和を求めるプログラムの例を紹介する。用いる言語は、アセンブラ、C言語およびJava言語とする。C言語では、goto文の有無によるコードを比べる。Javaでは、情報隠蔽と、アサーションを利用した例を示す。

```
ORG    100H
       LD     B,10
       SUB    A
LOOP:  ADD    A,B
       DEC    B
       JR     NZ,LOOP
       LD     (SUM),A
       HALT
       ORG    200H
SUM:   DEFS   1
       END
```

図1　Z80アセンブラによるプログラム例

```
int main(void) {
   int i,sum=0;
   for  (i=1; i<=10; i++)
            sum=sum + i;
   printf ("sum=%d¥n",sum);
}
```

図2　Cによるプログラム例（goto文なし）

図1は、Z80アセンブラで1から10までの和を求め、SUMに結果を格納している。Bレジスタに10をセットし、Aレジスタをゼロクリアする。そして、LOOPとラベルがついている個所で、A=A+Bの加算処理を行う。

次にBレジスタから1を減算する。その結果がゼロに未だならなければ、LOOPに戻る、しかしゼロになったら10から1までの加算が完了したので、SUMに結果を格納し、終了する。このプログラムでは、結果の表示は行っていないので、SUMのアドレスの中身を自分で参照しなければならない。

```
int main(void) {
   int i=0;
   int sum=0;
   LOOP:
      i=i+1;
      sum=sum+i;
      if (i<10)
               goto LOOP;
   printf ("sum=%d¥n",sum);
}
```

図3　Cによるプログラム例（goto文あり）

図2は、同じ機能の処理をC言語で、goto文なしのプログラム例となっている。この場合は、最後に結果の表示も行っている。アセンブラで記述した内容と比較すると、コード量が減少し、しかもアセンブラよりははるかに見やすくなっている。

図3は、同じことをC言語でgoto文を用いたプログラム例を示す。アセンブラよりは見やすい。しかしgoto文なしの例と比較すると見難くなっている。これが大規模なプログラムになると、いわゆるスパゲッティ・プログラムとなり他人が引き継いで理解していくことが難しくなり、保守が困難になってくる。

```
public class Sum{
  private int s;
  public Sum() {
    s = 0;
  }
  public int calcSum(int first, int last){
    int i;
    assert s==0;
    for(i = first; i <= last; i++)
        s += i;
    assert i==last+1
     && s==(((last+1)*last)/2)-(((first-1)*first)/2);
    return s;
  }
  public int getSum() {
    return s;
  }
}
```


図4　Javaによるプログラム例

図4は、Javaによるプログラム例として、情報隠蔽をクラスのフィールドにprivateとすることで実現している。こうすることでこのフィールドは、外部のクラスからは直接アクセスすることができないように保護することが可能となる。もう一方のアサーションは、その地点での前提条件をテストできるもので、アサーション実行時にtrueとなるべき論理式を記述し、ソフトウェアの信頼性に必要とされており、バグの早期発見や保守のためのドキュメントとしても役立つものとなっている。このアサーションはJavaの場合、assert文によって記述し、assertが記述された論理式が偽の場合は、例外処理が発生する。この例では、ループの前後の二箇所にassert文を記述している。

図4のプログラムを実行するためのテストドライバの例を図5に示す。アサーションを利用する場合のコンパイルと実行例を図6に示す。

```
class TestSum{
public static void main(String[] args) {
 Sum s = new Sum();
 System.out.println("s の値を得る＝"+s.getSum());
 System.out.println("和を計算＝"+s.calcSum(1,10));
 System.out.println("s の値を得る＝"+s.getSum());
}
}
```

図5　Javaのプログラムのテストドライバ例

```
C:¥user¥java>javac -source 1.4 *.java
C:¥user¥java>java -ea TestSum
s の値を得る＝0
和を計算＝55
s の値を得る＝55
```

図6　Javaの実行結果

アサーションを丁寧に記述できれば、プログラムの動作環境に誤りが無いことが今まで以上に保障され、品質の良いソフトウェア作成の実装の可能性が高まる。

3　設計と実装の条件表現の行き違い

次にソフトウェアの仕様とプログラムのコードとの関係の危うさを以下に例示する。

この例は、ある部材の品質を検査するものと仮定する。この品質は、厚みと重さにより判定するものとし、具体的な文書は、図7のように示されたとする。

```
厚みが 7mm 未満か
10mm より大きいものは、不良品
また
重さが 5g 未満か
8g より重たいものは、不良品
```

図7　部材の不良品の仕様記述

この図7に基づき擬似コードを図8のように記述し、このままプログラミングしようとした。これははたして正しいコードとなっているか？

```
If (atumi<7 or atumi>10) and
   (omosa<5 or omosa>8)
Then    部品エラー処理。
Else  正しい部品の処理。
```

図8　部材の不良品の擬似コード例（不正）

図8は、正しくないプログラムのコード例を示している。この場合の良品について見直してみると、図9のような定義になっているハズと考えるのが妥当な判断と思う。

```
厚みが 7mm～10mm
and
重さが 5g～ 8g
```

図９　図７から推測した良品の定義

良品とは

（7≦厚み　and　厚み≦10）
and
（5≦重さ　and　重さ≦8）

つまり、不良品は、

not ｛（7≦厚み　and　厚み≦10）　and　（5≦重さ　and　重さ≦8）｝

これにドモルガンの法則[5]を適用し、外側の中カッコをはずすと、

not　（7≦厚み　and　厚み≦10）
or
not　（5≦重さ　and　重さ≦8）

となり、もう一度ドモルガンの法則を適用すると、最終的に以下のようになる。

（7＞厚み　or　厚み＞10）
or
（5＞重さ　or　重さ＞8）

これを擬似コード化すると図10となり、図8とは異なる。

```
If (atumi<7 or atumi>10) or
   (omosa<5 or omosa>8)
Then    部品エラー処理。
Else    正しい部品の処理。
```

図10　部材の不良品の擬似コード例（正しい）

このような設計論理の正しさを検証するためにはツールを利用して検証する方法がある。

図11は、Mizar[6][7]による検証の例を示す。この場合、$aをatumi>=7、$bをatumi<=10、$cをomosa>=5、$dをomosa<=8とし、「良品または不良品」と一つの式で記述し、トートロジーを構成する。そのトートロジーとしての論理の矛盾が無いかあるかを検証した結果で、一方は正しくもう一方は問題の箇所に「*4」と印がつけられている。

```
environ
 vocabulary BOOLE, GATE_1;
 notation XBOOLE_0,GATE_1;
 constructors XBOOLE_0;
 clusters XBOOLE_0;
 requirements BOOLE;
begin
reserve a,b,c,d for set;
::  下記は、論理に矛盾がない
    (($a & $b) & ($c & $d))   or
    ((not $a or not $b) or (not $c or not $d)) ;
::  下記は、検証したところ問題がある
    (($a & $b) & ($c & $d))   or
    ((not $a or not $b) & (not $c or not $d)) ;
::>                                              *4,4,4,4
```


図11　Mizarによる検証例

4　UMLの歴史

UMLは、Rumbaugh、JacobsonおよびBooch[8]の3人が自分たちのオブジェクト指向の設計方法論を持ち寄り、GUIに重点を置いた統一的な設計言語を提唱しそれをOMGが引継ぎ、その他の設計方法論も統合し、標準化・規格化を推進している。現在1.5版ができており、2.0版の原案が一応でき上がったところという状況にある。UMLでの設計ツールは、高価な100万円前後のツールからフリーのツールでは例えば日本製のJUDE等があり、Java等の開発環境としてIBMが数年前にフリーで利用できるように開放したeclipse上で動

くプラグイン等も出現している。

5　UMLによる設計

UMLで簡単なクラス図を描いてみる。従業員管理のための従業員クラスの概要を図12に示す。

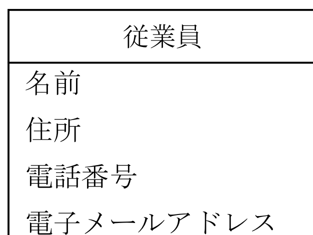

図12　従業員クラス図の例

従業員クラスに電子メールアドレスが付加される場合を考えてみる。この電子メールアドレスは、人それぞれに異なるものでなければいけないとする。しかし、このクラス図からではそのことは明記されていない。そのためこの従業員管理の仕組みを知らない人にとっては、電子メールアドレスに企業のある部署共通のアドレスを登録しても良いなどの誤解を生じる可能性がある。そこでこのような誤解を防ぐための手段が必要になる。このためにOCL[9]（Object Constraint Language）という制約記述用言語が用意されている。

6　OCLによる表現の正確化

OCLは、プログラミング言語ではなくクラスまたはインスタンス間の関連や範囲などの制約条件を記述するものです。UMLの中のMDA（Model Driven Architecture）という考え方のための必須言語となっている。UMLの図だけではモデルとして表現しきれないことを記述するための補助として利用できる。またソースコードを生成するための参考情報としての考え方もあり得る（注：MDAはこの方向を考えている）。ドキュメントとしては保守のための補足情報として利用できる。前述のJavaにおけるアサーションの記述のための情報にもなる。

この記述には、記号論理の∃や∀などを使用した束縛変数を用いる表現方法が採用されている。∃のかわりにexistsが用いられ、∀のかわりにforAllが用いられる。

束縛変数を用いると、たとえば「ｘとｙが実数の場合、任意のｘに対して、ｘ＋ｙ＝０となる或るｙが存在する。」ということを、

$(\forall x \in R)\ (\exists y \in R)\ (x + y = 0)$

のように記述することができる。

例として先ほどの図12の電子メールアドレスを不変条件（invariant）として記述すると図13のようになる。

```
context 従業員  inv  :
  従業員.allInstances()->
    forAll ( e1 , e2 | e1 <> e2 implies
        e1.電子メールアドレス <>
        e2.電子メールアドレス )
```

図13　OCLでの制約記述の例

contextは、従業員クラスに制約があることをあらわす。その制約条件が不変条件(inv)と定義されている。任意の二つのインスタンスe1とe2を取り出した場合、その両者が異なる(＜＞)インスタンス（つまり異なる従業員）ならば(implies)、お互いの電子メールアドレスは異なる（同じではいけない)。

Ⅲ　記号論理の教育の必要性

以上のことから、従来からの設計およびプログラムの論理条件の複雑な記述についての検証および、ソフトウェア設計におけるUMLとOCLの活用とJavaによるアサーションの記述等、ソフトウェアの設計・実装のための各種技術の動向を考えると記号論理を自由に駆使できる訓練を積んでおく必要がある場面が考えられる。これはビジネスアプリケーションのみではなく、埋め込み系のアプリケーションの場合は特にMDAとの関係が今後強くなりつつあると思われる。このような背景から考えると、記号論理をカリキュラムに加える必要があると判断する。しかし現在の過密なカリキュラムの中に新たに追加するのは限界を超えてしまうと思われる。そこで、各校の現場のニーズに合わせた選択制などを考えて、取り入れていくことが必要ではないか。

Ⅳ　記号論理の教育のカリキュラム案

表１に情報系を意識した記号論理関連のカリキュラム案のイメージを示す。時間比は、18回を全体とみた場合を意識した配分を考えた。各々の教科細目の具体的な詳細は、参考文献の(10)、(11)、(12)、(13)を参考とした。

No	教科細目	時間比
1	集合	1
2	写像	1
3	関係	1
4	ブール代数	2
5	関数の再帰的定義	2
6	関数の計算可能性	1
7	帰納法	1
8	命題論理	3
9	述語論理	3
１０	プログラムの検証	1
１１	状態遷移図	1
１２	フォーマルメソッド	1

表1　情報系の論理数学の教科細目案

V　あとがき

ソフトウェアのものづくりの現場を考えると、信頼性の高いソフトウェアを設計・実装していくためには、今まで以上に論理的な思考訓練が必要と考えてこのような案を提起した。論理的思考は、より高度な設計手法として欠かせないformal methods (13) (14)にも必須の能力となっている。

埋め込み系のソフトウェアシステムの開発は、今まで以上にMDAを考慮しOCLを利用する場面(15) (16)の重要性が増してくると考える。

現在、応用課程の二年生（通常の大学四年）有志二人とゼミ形式で実験的に取り組んでいる。学生からの意見として、「学校の授業は実習が多いが、このような座学の授業があると良い」との感想が得られている。正規の授業ではないので時間の工面・調整が思うにまかせない状況のなかで取り組んでいる。しかし参加人数が少ないので組織的に明確なデータを取ることを意識し、来年も何らかの形で試行をし、より客観的なデータを取得するように心がけたい。

参考文献

(1) http://www.uml.org/
注：UML、MDA、OCL等の最新情報はここから得られる。

(2) C. Boehm & G. Jacopini, "Flow Diagrams, Turing Machines, and Languages with only Two Formation Rules", Communications of the ACM 9(5), 1966, 366-371. (http://www.cse.buffalo.edu/~rapaport/111F04/greatidea3.html)

(3) E. W. Dijkstra, "GOTO Statement Considered Harmful", Letter of the Editor, Communications of the ACM 11(3),1968, 147-148.

(4) http://homepages.cs.ncl.ac.uk/brian.randell/NATO/index.html

(5) S.Lipschutz ,"マグロウヒル大学演習離散数学 コンピュータサイエンスの基礎数学",オーム社, 1995

(6) http://www.mizar.org/

(7) http://markun.cs.shinshu-u.ac.jp/kiso/projects/proofchecker/mizar/index-j.html

(8) Rumbaugh, Jacobson, and Booch,"The Unified Modeling Language Reference Guide (1st ed.)", Prentice Hall, 1991

(9) Warmer and Kleppe, "The Object Constraint Language: Getting Your Models Ready for Mda", second edition, Addison-Wesley,2003

(10) 林,八杉, "情報系の数学入門",オーム社

(11) 小倉,高濱,"情報の論理数学入門", 近代科学社

(12) 小野, "情報科学における論理",日本評論社

(13) 荒木、張, "プログラム仕様記述論",オーム社

(14) 福良,"ソフトウェア技術者の道具としての「formal methods」",茨城職業能力開発短期大学校　紀要,1998,vol.11

(15) Model Integrated Computing for Embedded, Real-Time Systems Tuesday, 27 April 2004

(16) Annual Reprot, "Foundation of Hybrid and Embedded Systems and Software" NSF/ITR Project, Award Number : CCR - 00225610 University of California at Berkeley, Vanderbilt University, University of Memphis, May 31, 2004