技術論文

開発言語の変遷（アセンブラからJavaまで）

Trend of programming languages（Assembly language to Java）

大里　章[*]　戸木田　和彦[**]　朝比奈　衛[***]　梅谷　昌範[†]　池上　康[††]　田島　規義[†††]　吉田　規行[‡]

Akira Osato, Kazuhiko Tokita, Mamoru Asahina, Yoshinori Umeya, Yasushi Ikegami, Noriyoshi Tajima, Noriyuki Yoshida

ソフトウェア開発の草創期にはアセンブラ言語を主に開発が進められ、また、その他のハードウェアに依存する開発言語が使用された。

これらのプログラミング言語による開発では、ハードウェアの特性に合わせた様々なプログラミングの工夫がされた。

当社は創立以来、ソフトウェア開発を事業主体として成長してきたが、アセンブラ言語に始まって、C言語、Java等の高級言語にまで取り組み、その発展とともに当社のソフトウェア開発技術は発展してきた。

本稿では、特にソフトウェア開発の草創期における開発の状況とともに、当社の開発言語に関する技術の変遷について解説する。

Computer programming technique starts by using Assembly language, or other programming languages depend on hardware.

This report describes the MSS's technologies advanced with progress of programming languages.

1. はじめに

当社はソフトウェア開発を事業主体として成長してきた。半導体事業に始まり、プロセス計算機事業、通信衛星設備構築事業、ロケット誘導・解析事業、そして1970年代後半から衛星、防衛関連のソフトウェア製造を担ってきた。その間に、事業分野は9分野におよび業務範囲も、上流設計～保守まで行なう会社に成長してきた。

当社の歴史を振り返ると各場所で多種多様な言語仕様によるソフトウェア開発が行なわれてきた。1970年代後半のMSSの開発言語といえば、FORTRAN、アセンブラ、COBOL程度しかなく、設計プロセスもiCoPS（情報システム開発プロジェクト向け標準プロセス）のような統一プロセスは無くプロジェクトごとでプロセスは定義されていた。詳細設計レベルの設計書も、フローチャート、HIPO、NSチャートが記述の中心で、あとはプロジェクト毎にタイミングチャートや、データベース定義書等を個々のスタイルで作り、それらを基にプログラミングしていた。

プログラムリスティングも、開発者個人の趣味が表に現れるものであった。特に組込系であると、デバックは会社から実環境に出向いて、現地でデバックをするが、現在のようなノートPCが無かった時代、重たいリストを両手で持って現地に出向くとなると少しでも荷物は軽くしたい、特にアセンブラ言語では詳細設計書も欠かせないため、プログラムリストのコメント欄に詳細設計書のフローチャート部を記載するつわ者もいた。

このように、ソフトウェア開発は、十人十色の側面があり、過去の足跡を知る事も有意であることから、以下に過去の開発経験や開発言語を紹介する。

2. アセンブラ言語

2.1　ロケット航法・誘導モジュール開発への適用

現在主力大型ロケットとして運用されているH-IIAロケットの搭載ソフトウェア（航法・誘導モジュール）はC言語で記述しているが、1986～1992年に用いられたH-I ロケット、及びH-IIAの先代である H-IIロケット（1994～1999年）まではアセンブリ言語を使用していた。各ロケット用の搭載計算機の主要緒元は表1のとおりである。

航法・誘導モジュールの機能はロケットの位置、速度及び姿勢の航法情報を計算し、目標とする軌道に基づいて飛行方向を制御する誘導コマンド・レートとエンジンを停止させるまでの残り時間であるタイム・ツー・ゴーを出力することである。航法情報は射点を初期値とする積分で求めるため計算誤差が累積すること、また、エンジン停止時刻のわずかな変動が投入軌道の形状に大きく

*技術推進部　**つくば事業部 品質保証部　***鎌倉事業部 第一技術部　†同事業部 業務部　††同事業部 社会基盤システム部　†††同事業部 第五技術部　‡同事業部 第三技術部

表１　搭載計算機主要諸元 (1)

項　目	H－I	H－II	H－IIA
主要機能	・演算機能 ・ディジタル装置間インタフェース機能	・演算機能 ・ディジタル装置間インタフェース機能	・演算機能 ・ディジタル装置間インタフェース機能 ・機体インタフェース機能 ・計測通信制御機能
マイクロプロセッサ	ビットスライス型マイクロプロセッサ	ビットスライス型マイクロプロセッサ	32bitマイクロプロセッサV70
語長	16Bit	16Bit	32Bit
演算速度	0.26MIPS（H－I使用率計算）	0.34MIPS（H－II使用率計算）	2MIPS（ドライストーン）
演算方式	固定小数点	固定小数点	浮動小数点
OS	なし	なし	リアルタイムOS（RX616）
記憶容量	RAM：32Kbyte	RAM：64Kbyte	RAM：2Mbyte ROM：128Kbyte

影響することから、固定小数点演算の精度を十分に保つことが必要となる。搭載ソフトウェアの開発は航法・誘導アルゴリズム選定とソフトウェア製作から構成され、アルゴリズム選定にはFORTRANによるSSP（数学的シミュレーションプログラム：Scientific Simulation Program）を、ソフトウェア製作にはICS（慣性誘導計算機シミュレーションプログラム：Interpretive Computer Simulation Program）を使用した (2)。ここではソフトウェア製作における演算精度の確保を中心に述べる。

ICSの構造を図１に示す (3)。ICSは搭載計算機のアーキテクチュア（命令セット、語長、メモリ容量、レジスタ構成、割り込み処理、入・出力インタフェース等）に基づいて機械語命令の実行をホスト計算機上で模擬しており、製作したソフトウェアを実機と同等の環境で動作させることができる。搭載計算機は語長が16ビットで固定小数点での演算を行わなければならないため、丸め誤差に対しては計算機の内部表現（－1.0～1.0）と演算の対象となる物理量（最小値～最大値）とをSSPの実行結果から得られるデータの変動幅を用いて対応付けし、精度を落とさないプログラム設計を行った。また、演算時の桁落ちに対しては計算順序の工夫や実行時間を考慮しつつ倍精度計算を適用することで対処した。

ソフトウェアの検証は開ループ、閉ループの二段階で実施している。開ループ段階ではSSPを動作させるのと同じ入力（検知速度、検知角度、加速度）により演算を行わせて、SSPでの結果と比較することにより固定小数点演算の影響が問題のない範囲であることを評価する。ICSでは命令の実行過程をレジスタの内容も含めてビットレベルで確認できるため、オーバーフローの発生有無や演算精度の劣化部分を容易に特定することができる。また、閉ループ段階では搭載ソフトウェアが出力する誘導コマンド・レート、タイム・ツー・ゴーを軌道シミュレーションプログラムに反映して投入軌道を生成し、性能が計画どおりであることを評価する。航法・誘導モジュールのように繰返しを伴う固定小数点演算は、精度劣化を起こしている部分を特定することが難しいがICSの使用により自信をもって開発を進めることができた。

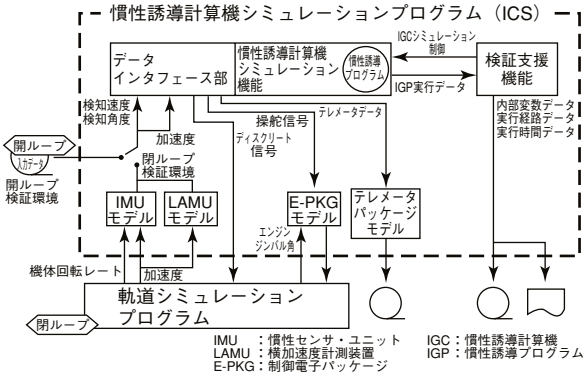

図１　ICS（慣性誘導計算機シミュレーションプログラム）

2.2　CMS-2M言語

2.2.1　CMS-2M言語の概要

CMS-2M言語は、コンピュータが実用機器として使われ始めた頃のプログラミング言語だが、今でも現役のシステムで稼働しており、まれに維持で使われることもある。

米海軍の指揮統制システムに使われる特定のコンピュータ用の言語として使用が義務付けられていたことから、わが国の艦船用に当該コンピュータを導入したのに伴い、この言語も採用したのが、日本での使用の始まりである。

米海軍としての標準のHOL（higher order language）と位置付けられた言語であるが、対象コンピュータごとに複数の方言があった。

後に米国ではAdaに引き継がれ、言語仕様としての進歩はなくなったが、日本では、一部のレガシーシステムで現在も使われている。

2.2.2　CMS-2M言語の経歴

CMS-2はCompiler Monitor System version 2に由来

する。CMS-2M はその一方言で16bit ISA（UYK-20）に対応するものである。これらは何れも、コンパイラとそれに対応する言語仕様が一体として認識されており、純粋に言語として独立はしているわけではない。

コンピュータが実用に供され始めたころ、米海軍の指揮統制システムにおいて利用可能なコンピュータは軍専用に開発された特定の機種に限定されていた。CMS-2はそこでのアプリケーション開発用の汎用言語とされて、使用が義務付けられていた。当時はコンピュータの利用もソフトウエアの開発も軍が民間より圧倒的に先行していた。

最初の版は、1960年代終わりにUNIVAC AN/USQ-20（32K 30bit Words,16 IO channels, 8 microsecond cycle time）をターゲットに、それまでのアセンブラCS-1に変わるべく開発された。

1970年代初めに、高性能マシン（Q-20よりは）UYK-7用に別バージョンのCMS-2（CMS-2L）が開発された。続いて、mini-computerとしてUYK-20が登場し、またしても別バージョンのCMS-2（CMS-2M）が誕生した。この時点でH/Wに依存しない標準言語を求める声はあったが、そうはならなかった。後にこれらのCMS-2を統一的に扱える開発環境が提供されるようになった。

1991年には法律により米軍のシステム用の言語を全面的にAdaに統一することとした。しかし、変更のリスク、コスト効率を考慮し既存システムについては、CMS-2の併用を可能とし強制的な全面変換とはしなかったため、最近までは生き残ったものと思われる。1997年にCMS-2からAdaへの変換ツールの評価報告が出されているが、どうも完全ではないようである。

2.2.3　導入

1970年代以降、当社は艦船用のソフトウェア開発を手がけることとなり、実質的には、国内では最も使い込んだ会社と言える。

1977年、国内では、既に汎用の大型、小型のコンピュータは実用の時代に入っていたとは言え、小型化、対環境性の点で、アメリカからSPERRY UNIBAC社製のUYK-20を購入して使う事になっていた。UYK-20は、米海軍が開発し、艦船に搭載している小型コンピュータで、軍用規格を全て満足しており、「二階から落としても壊れない」といわれていたほどの頑強な構造の製品であった。しかし当時はまだICメモリ等は実用に至っておらず、コアメモリーなるものを使っていて、32KWしかなく（後で増設されて64KW）、要求機能を実現する為には効率の良いメモリー使用の工夫が必要であった。まず汎用のOSは大き過ぎたので、コア機能のみのものを新規に開発し、アプリケーションで使用できるメモリーを確保し、その上でシステム設計、プログラム設計にも工夫を凝らした。

その後コンピュータの世代交代とともに開発言語もC++へと刷新したが、それまではいくつかの環境の進歩（ターゲットマシンの更新、アドレス空間の拡張、クロスコンパイル器材の充実）はあったもののCMS-2M言語による開発は続けられ、いまだに現役で活躍しているものもある。

2.2.4　CMS-2M言語によるソフトウェア開発

1977年に開発設備として最初にCMS-2Mコンパイルシステムを導入した時の話である。

コーデイング用紙、カード、紙テープなるものを実際に使用していたし、開発マシン（セルフのためUYK-20）にすら磁気ディスク装置は無い。大型の磁気テープ装置４台によりコンパイルする。動きを見ていると壮観に感じ、慣れてくるとコンパイラが今どのフェーズを実行しているかが分かった。

ソースコードはパンチカードと呼ばれる一枚に80カラム分の文字コードを穴あけできる紙のカードに打ち込む。ソースのエディットはカードの差し替えで行う。文字のパンチには専用の機械があるが、一文字でも打ち違えたら（backspace、deleteキーは無）そのカードは廃棄、一枚分打ち直しとなる。初回分は、コーデイング用紙に手書きし専門の業者に依頼するが、デバッグの過程でかなりの量を自分で打ち直すこととなった。

１台のテープ装置に、システムテープ（CMS-2Mコンパイラ）、２台のテープ装置はテンポラリストレージとして、最後の１台はオブジェクトモジュールの出力に使った。入力はカードリーダ（今のICカードリーダとは違い見るからにメカニカルなもので、動作音がとても勇ましい）からとなる。ソースコードをテープに収めることもできたが、その場合は、１台のテープ装置を入力用と出力用に使い分けることとなり、コンパイルの途中でテープの付け替え作業が必要であった。後に、待望のディスクシステムが付加され、ここら辺はずいぶんと楽になった。

もう１つの出力であるリスティングは、ラインプリンターにコンパイル中に直接印字するしかなかったので、コンパイル時間の大半が印字時間にさかれた。

ラインプリンター（これもすでに死語と思われる）とは、そもそも当時のプリンターは、活字、インクリボン、紙を重ねてハンマー（もしくは活字自体）で叩いて、その印影を紙に写すことによっていた。印字したい文字の活字を選んで打つわけだが、普通は１文字ずつプリント

していくのに対し、大きなドラムに、１文字分の位置に全種類の活字を貼り付けたものを１行分並べて高速で回転させ、１行分を活字ドラム１回転で印字するようにしたのが、ラインプリンターである。１行130文字で、130のハンマーが文字を叩き出す（実際はハンマーの数は半分で、左右にシフトしながらドラム２回転で１行だった）のは見ていても飽きないし、賑やかな音がするけれども、進みはとても遅かった。睡眠にはちょうど良いものだった。

2.2.5　CMS-2M言語の仕様

MIL-STDで示されている言語仕様によるプログラム構成の概要は以下の様なものである。

始めの10カラムは懐かしいカードIDである。カードの差し替えでEDITするのでこのIDはとても重要であった。続く70カラムがステートメントエリアである。1つのステートメントの終わりは $ で明示する。SYS-DD END-SYS-DD間にデータを定義する。SYS-PROC END-SYS-PROC間に実行文等を書く。下記の例で＊＊＊には例と同様のブロックが複数あることを示す。

■ソースコード例

```
OSAD00000 SSSS SYSTEM                                          $
SSDD00000 DDDD SYS-DD                                          $
SSDD00010         ***
SSDD00120         TABLE     TTT  V  3 5                       $…①
SSDD00130           FIELD   XX    A 16 S  7  0 15             $
SSDD00140           FIELD   YY    I  8 U     1 15             $
SSDD00150           FIELD   ZZ    A 16 S  14 2 15             $
SSDD00160         END-TABLE TTT                                $
SSDD00170         ***
SSDD00310 END-SYS-DD DDDD                                      $
SPR000000 PPPP    SYS-PROC                                     $
SPR000010       ***
PROC02000       PROCEDURE        RRR                           $
PROC02320             SET   TTT(0,XX)   TO   TTT(1,YY)+TTT(2,ZZ)  $…②
PROC02510       END-PROC         RRR                           $
PROC09920       ***
OSAD01200 END-SYS-PROC        PPPP                             $
OSAD01300 END-SYSTEM          SSSS                             $
```

上記例の②の部分が以下の様にマシンコードとアセンブラコードに展開される。

左から相対アドレス、続く４つはマシンコード（全て8進表示）、６番目は2W命令の場合２番目のコード（たいていは参照先の相対アドレス）、右側はコードのアセンブラ言語表現である。

■リスティング（オブジェクトコード）例

```
000517  01 3 14 00 000427  L     R12,TTT+8
000521  60 1 14 07          LARS  R12,+7
000522  00 3 15 00 000426  BL    R13,TTT+4
000524  61 0 15 07          LALS  R13,+7
000525  22 0 15 14          AR    R13,R12
000526  11 3 15 00 000422  S     R13,TTT
```

上記コードの内容は、①のテーブルTTTのフィールドXXにYYとZZの合計を格納するというもの。もう少し詳しく見るとTTTは3ワード5アイテム構成と定義されている。XXは16bitで7bit目に小数点、YYは8bitで整数、ZZは16bitで14bit目に小数点があると定義されている。②は0アイテム目のXXに1アイテム目のYYと2アイテム目のZZを加算して格納している。格納先のXXの小数点位置にあわせて、ZZをロード後右に7bitシフトし、バイトでロードした整数YYを左に7ビットシフトしたものと加算し、XXにストアしている。

上記の様にメモリ上のワード、ビットレベルの物理的な配置を強く意識した（意識しなければならない）操作を行える言語となっている。

2.2.6　CMS-2M言語の特徴

コンパイラとしては初期のものであり、上記のとおり、直接的にマシンコードが生成されるため、アセンブラで書くのと同等の性能と動作を品質良く（間違いなく）達成できるのが特徴であった。

現代の言語が概ねライブラリコールで組み上げるのに対して、CMS-2Mは原則直接コードに展開するのに加え、当時はOSもライブラリも利用しなかったので、高級アセンブラ的利用形態であった。

特に物理メモリのワード内の一部分に定義するビット単位の取り扱いが楽に実現できて便利であった。このことにあまり意義を感じない人も多いと思うが、当時は、何よりもメモリが貴重だったため、１ビットでも無駄にできない事情があった。ということもあったが、外部システムを制御するリアルタイムシステムとしては、入出力データがビット単位で定義されることが多く、これを取り扱うのにも便利であった。

2.2.7　CMS-2M言語の現在

コンピュータ及びソフトウエア工学の進歩とともに新たな言語や開発手法が次々と生まれ、本言語は遥か以前に淘汰され消えているはずだったが、この言語により開発された資産を引き継ぐ特殊な長寿命システムが、特殊故にリプレースもできずに稼働しているため、まだ僅かであるが生存している。これらのレガシーシステムの維持のみを目的としているため、言語仕様は更新されず昔のままであり、まさに生きた化石シーラカンスのような言語となっている。しかし、生存環境（ターゲットマシン）が固定されており、まもなく残存のレガシーシステムとともに寿命を終えるであろう。

2.3　アセンブラ言語の処理例

2.3.1　高速化や精度向上の例

以上に見られるように、ソフトウェア開発の草創期の基本と言えば、アセンブラ言語である。

ここでは、アセンブラ言語による処理例を紹介する。

アセンブラ言語は、コンピュータが処理する機械語（二進法で表現される命令とデータ）を記号化したものである。そのため、記述コードどおりに順次実行処理される。

アセンブラ言語がどんなものか、アセンブラ言語による処理例と高速化や精度向上で工夫した例を以下に示す。

例１：Ｘの値とＹの値を加えてＺにいれる処理（Z=X+Yの計算）
（Xの値はメモリのXX番地、Yの値はYY番地にあり、Zの値はZZ番地に格納されるものとする）

コーディング例１-１

```
L     R0,XX   .レジスタ0にXX番地の値Xを入れる
A     R0,YY   .レジスタ0にYY番地の値Yを加える
S     R0,ZZ   .レジスタ0の値をZZ番地に格納する
```

コーディング例１-２

```
L     R0,XX   .レジスタ0にXX番地の値Xを入れる
L     R1,YY   .レジスタ1にYY番地の値Yを入れる
AR    R0,R1   .レジスタ0とレジスタ１を加え、結果
                をレジスタ0に入れる
S     R0,ZZ   .レジスタ0の値をZZ番地に格納する
```

※レジスタ同士の演算の方が高速なので、場合によってはコーディング例１-２のようなやり方もする。

例２：Xを2で割ってYにいれる処理（Y=X/2）
（Xの値はメモリのXX番地にあり、Yの値はYY番地に格納されるものとする）

コーディング例２-１（通常）

```
L     R0,XX   .レジスタ0にXX番地の値Xを入れる
LARD  R0,16   .レジスタ0、1を2つ同時に右へ16ビッ
                ト算術シフトして2word長にする
DK    R0,2    .レジスタ0、１の値を2で割る
              .計算結果、レジスタ１に商、レジス
                タ０に余りが入る
S     R1,YY   .レジスタ1の値をYY番地に格納する
```

コーディング例２-２（高速化、精度向上）

```
L     R0,XX   .レジスタ0にXX番地の値Xを入れる
LL    R1,0    .レジスタ１に0を入れる
LARD  R0,1    .レジスタ0、1を2つ同時に右へ１ビッ
                ト算術シフトする
RR    R0      .レジスタ0、１をレジスタ0に四捨五
                入してまるめる
S     R0,YY   .レジスタ0の値をYY番地に格納する
```

※割り算や掛け算は処理時間がかかるため、シフト処理で対応できる場合はコーディング例２-２のようにシフトを利用する。

2.3.2　その他の工夫処理

その他、処理時間を考慮した工夫もいろいろあるが、例えば、ジャンプ命令なども時間がかかるため、ループ処理の多用を避けた。ループ回数が少ないなら、メモリをやや食うが単純に回数分を羅列したほうが処理速度が早く、後で見てわかり易いので、メンテナンス性も向上する。

計算方法にもいくつか工夫をするものがある。

一例として、近似した値の平均値をとる場合は、基準値を決め、基準値からの差分の和を取り、差分の平均値を求めた後、基準値に加えることで、オーバーフローを防いだり、使用ビット数を抑える等の工夫である。

$$AV=(X_1+X_2+X_3+\cdots+Xn)/n$$

$$\Rightarrow \quad AV=X_1+\{(X_2-X_1)+(X_3-X_1)+\cdots+(Xn-X_1)\}/n$$

（X_1を基準値とした場合）

※固定小数点演算では、X_1からXnまで単純に加えると、途中でオーバーフローする可能性がある。

要求事項とコンピュータ性能との兼ね合いでアセンブラ言語を使用した開発では、開発環境も含め、今から見ると考えられないくらい効率が悪かった。ただ、いろいろ苦労した分身に付いたものも多い。

処理方法を工夫すれば、コンピュータ性能が多少低くても、十分に長期に渡り機能するものを作ることができる。

昨今、コンピュータ性能の向上で、処理速度やメモリ容量にとらわれずにSW開発できるようになってきたが、SWの処理効率について少しでも興味をもってもらえると幸いである。

2.4　アセンブラ言語によるソフトウェア開発

現在Java等の世界で言うとEclipseといった統合開発環境（IDE）やStrutsなどの開発フレームワークが充実してきており、アセンブラでの開発のノウハウが今後生かす機会があるかどうか疑問な部分もあるが、アセンブラ言語を用いた開発時の状況を記述する。

ご存知のようにアセンブラ言語はハードウェア固有の言語となるため、ハードウェアの変更により命令セット（ISA）が変わり、都度開発が必要となる。

研究試作での開発は5年間を要することがほとんどであった。ここで紹介するのは、大量のデータをリアルタイムに処理するため、ハードウェアで構築された信号処理部の制御を行うことを目的としたソフトウェアである。信号処理部を含めたすべてのハードウェアが専用ハードウェアであり、制御方法を含めて特有の手順があったため、設計書の記述方法から検討を開始している。

2.4.1　設計について

検討した設計書は以下のような内容であった。

一般的にはフローチャート、PAD（Problem Analysis Diagram）図を使用していたが、信号処理部のハードウェアを制御するために必要な情報を記述するには適さない。そのため、以下のような記述方式にて設計を行っていた（ドキュメントは手書きであった）。

信号処理部の制御に必要となるデータ

- 入力データ側のメモリ制御
- 出力データ側のメモリ制御
- 信号処理内容に対応した番号（FFT、ディジタルフィルタ等の処理に対応した制御番号）

2.4.2　コーディングについて

コーディングはメインフレーム（タイムシェアリングシステム）にて実施しており、コーディング作業は現在とあまり変わっていない。コーディングとアセンブル、単体試験のフェーズは完全に切り分けられており、随時コーディング、コンパイル及び単体試験を実施し、問題をつぶしていくようなアプローチではなかった。そのため、フェーズ毎に必要な時間を確保していたため、時間はかかっていた。

2.4.3　試験について

メインフレームからバイナリモジュールを実機に読み込ませるための手順が面倒であり、実機にモジュールを反映するためにかなり時間がかかった。そのため、不具合発生時における原因追求及びパッチの実施はメモリに直接コードを記述して実施している。試験実施時に手動アセンブルは必須の知識であった。そのため、客先での不具合発生時でもその場で即対応が可能であり、そうした点がアセンブラ言語の利点であった（逆に言えば、アセンブルに時間がかかっていたことによる苦肉の策とも言える）。

信号処理部の試験は、試験用の入力データに対して正しいデータが出力されるかどうかを機能毎にひとつひとつ確認を行う。入力データを作成するためのツールがまだ整備されていなかったこともあり、同じデータを全メモリに書込むような単純なデータを作成するツールを作成し（ツールもアセンブラで作成）確認を行っていた（FFTなどは、プラス、マイナスのデータを交互に書込み、試験を実施している）。

全体の処理を通して全システムでの確認が必要となる。そのため、問題の切り分け（どの装置に問題があるのか、どの処理に問題があるのか）はかなり時間をとられ、その上、問題の切り分けのためには全装置のハードウェア、ソフトウェアの担当者が必要となる。全システムの試験時にはすべての要員が待機し、試験が実施された。

上記のように、開発すべてにおいて時間がかかっていたが、今と比較すると十分な開発期間の中で作業を進めることができ、かつハードウェアの特性も理解した上での開発であるため、今のようにハードウェアやCOTSに依存する想定外の不具合は少なかった（問題が発生した場合には、すべての要員が徹夜や、二交代を余儀なくされるため、「時間が足りない」という感覚は当時も今も変わらない）。

3.　高級言語の登場（FORTRAN、COBOL、C、Ada、C++、Java）

前節までに紹介したアセンブラ言語や機械語はその名のとおり、機械が理解する言語であるため、人間にとってはなかなか苦痛の言語であった。

その解消のために高級言語が登場したことは言うまでもない。

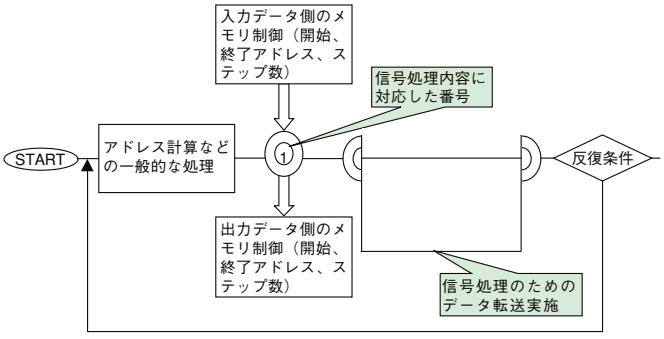

図2　信号処理部のハードウェア制御を記述するための設計図

いわゆる対話型又は会話型言語と呼ばれるもので、構造化言語又は手続型言語の代表格であるFORTRAN、COBOLに始まり、C言語、そして、オブジェクト指向型言語であるSmalltalk、Ada、C++、Javaに至る。

これらの言語の登場によりソフトウェア開発の効率と速度は飛躍的に向上した。

当然のことながら、ソフトウェアをこれらの言語で開発することがニーズとして、例えば、顧客の開発仕様に明示されるようにもなった。

当社もこれらのニーズに迅速に応えられるように、早くから高級言語を取り入れたことは言うまでもない。

特に、取っ付きやすさ、開発の容易さの点で、C言語は当社内でも多くの者が習得し、ソフトウェア開発に活かすようになった。

次いで、Webアプリケーション開発関連業務の増加とともに、機能的な容易さとGUIに優れたJavaは顧客ニーズの高まりもあって、Java専門の技術者の養成に社内でも特に力を入れるようになった。

これらの高級言語の歴史や特徴等、詳細については各種の解説書等が巷にあふれており、読者諸兄も熟知しているとおりであるので、ここではそれぞれの解説は割愛する。

4. Java言語

ここでは、Java言語で開発するための統合環境やデバッグツールに注目して問題解決を図るために有効なツールを紹介する。

4.1 歴史

Javaは1995年SunWorldExpoにて初めて一般に公表された。Javaの始まりは1990年にサンマイクロシステムが立ち上げたGreenプロジェクトで情報家電制御のために開発した言語で、最初はOakの呼称で呼ばれていた。しかし、Oakは既に商標登録されていたためにJavaと呼ばれるようになった。このOak開発の中心となった人物がJames Goslingであり、Javaの父とも言われている。その後、JavaはWebの世界に主眼を置くという方針変更を行い、現在に至っている。

4.2 Javaはコンパイラ＋インタプリタ

Javaは「Write Once Run Anywhere」といわれており、Java言語で書かれたプログラムはコンパイラによって中間コードといわれるCPUに依存しない命令コードにコンパイルされる。この中間コードはコンピュータで直接実行することはできない。各種CPUやOSに対応したインタプリタで実行することができる。従ってインタプリタさえ準備されていればどこでも実行できるようになっている。しかし、現実は他へ移植する場合、細かい部分で修正やデバッグが必要となる。

4.3 統合環境

Javaで開発を行うためには統合環境を準備したほうが効率的である。統合環境としては現在下記のようなものが上げられる。これら開発環境を利用すると、Java開発における問題解決を効率的に進めることが可能となる。

4.4 Javaによる開発の問題解決一例

Javaによる開発において、問題解決を効率的に行うための手段の一つにツールを有功活用することが必要である。Java開発だけではないが「パフォーマンスが出ない」、「メモリーリークが発生している可能性がある」など、プログラム開発に携わった方は、一度は経験されていると思われる。では問題を速やかに解決するためにデバッグする人はまず何を考えるか…。当たり前のことだが下記を思い浮かべるのではないかと思われる。

⑴ 問題解決のために、どうやって情報を集めればよいかを検討する。

⑵ 情報収集にはデバッグツールやプロファイリングツールで怪しい箇所を特定する。

4.5 メモリリークとは？

JavaVM（VM：Vertual Machine）は不要になったオブジェクトを、ガベージコレクション処理によって自動的に開放する。ガベージコレクション処理はオブジェクトが不要になったかどうかを、誰からも参照されていないことを判定してオブジェクトを開放している。しかし、ビジネスロジックから判断すると不要となったオブジェクトではあるが、参照が残っている場合がある。この時JavaVMは参照先のオブジェクトが不要になったことが判断できずにメモリが開放されないため、何度も同じ処理を実行しているとメモリリークが発生する場合がある。

4.6 プロファイラによる情報収集

このメモリリークを発見するには、何回かガベージコレクションが実行されているにもかかわらず、メモリ使

表２　Java開発統合環境

製品	提供
WebSphere Studio	IBM
Eclipse	Eclipse Foundation
NetBeans	Sun
JDeveloper	Oracle

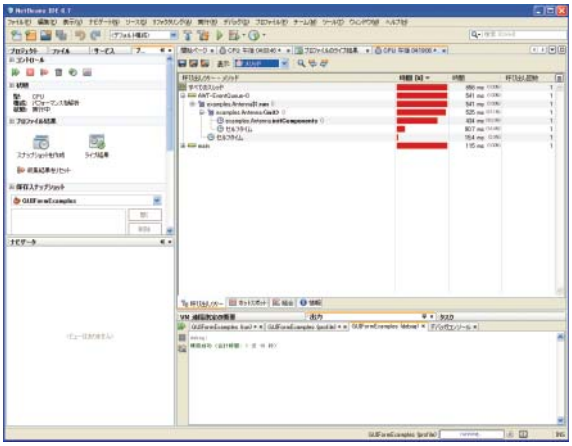

図３　NetBeansプロファイラ出力例

用量が増加し続けているオブジェクトを発見することで判断する。このメモリリークの絞込みを効率的に行うためにプロファイルツールを利用して、メモリが増加傾向にあるオブジェクトを発見できれば、メモリリークを引き起こしている可能性が高いコードを含むクラスを推測することができる（図３）。また、プロファイラでは、メソッド呼び出し回数や、合計処理時間等を計測できるので、パフォーマンスのボトルネックとなっている部分を判定してチューニングすることも可能である。

4.7　気をつけなければならない点

最適な解決方法を見つけたりするには経験やスキルが必要である、チームや組織内で前もって問題解決に取り組む環境を作り上げることが重要である。Javaを基盤としたWebシステム環境では、ある時突然応答を返さなくなったり、しばらくするとまた動き出したりといった事象がよく発生する。そのような問題への取り組みをリスクとして開発段階で計画しておくべきである。

5.　まとめ

以上から、開発環境や開発ツールの整備が進んだ近年と異なり、ソフトウェア開発の草創期は何をするにも不自由が付きまとう時代であったことが御理解いただけたと思う、さらに皆品質の良い製品作りを目指し、日夜努力していたことがうかがわれたかと思う。

冒頭にも記述したがソフトウェア開発は十人十色の側面があり、今回紹介から漏れたが当社にはAda言語の使い手も多数居り、解析技術を有する技術者等々、諸所の制約のため全てを取り上げられなかったが、MSSソフトウェア開発の草創期の状況を知っていただければ幸いである。

コラム

開発秘話　アセンブラ言語：「ロードモジュールは紙テープ！」

ミニコン室の床はピンク色の紙テープで足元も見えないほど埋め尽くされつつあった。なんと紙テープ！計算機の出力媒体が紙テープなんて、子供の頃に見たTV以来である。ROM書き込み用のロードモジュールファイルの媒体が紙テープであった。

1985年（昭和60年）頃はPCは未だ１グループに１台しかなく、しかもキャラクタ画面※1。ワープロソフトなどなく、仕様書などドキュメントはシートナンバと呼ばれる方眼紙に手書きで作成していた。組込S/Wの開発言語は殆どアセンブリ言語。インテルのi8085、MX-1（0）という16bitマイコンなどが組込みシステムの主なCPUであった頃・・・。

以前客先に納品した教育用システムのS/Wを改修する業務にアサインされた。改修内容も小規模。担当は自分一人。私はこのシステムのS／W開発は経験がなかったので、今回の改修では前任者より開発環境を教えてもらった。CPUはMX-R3という16bitマイコン。多くのシステムに使用されたMX-1（0）の前の型である。コーディング、アセンブルはミニコン室のミニコンM70を使用。クロス開発環境は製品ラインナップが未だ不十分で単体試験は実機のメンテナンスパネルを使用して実施。そしてROM切り用ロードモジュールファイルの出力媒体が、なんと紙テープであった。

ソース改修が一通り終わり、コードレビュー後、単体試験のために初めて紙テープ出力を試してみた。まるで大きなバームクーヘンのような新品の紙テープをミニコンにセットする。メモリに展開して64KBにも満たない小さなロードモジュールだが、出力してみると、あれよあれよと紙テープを吐き出す、吐き出す。30分近くも吐き出すと、小さな部屋は紙テープで足の踏み場の無いほど埋まった。これをまとめるには専用の巻き取り器で、手動で巻いていくのである。紙テープがからまないよう、切れないように最初はゆっくり巻いていったが、きりがない。5分たっても床の紙テープは少しも減らない。業をにやして少しずつ巻き取るスピードを速めてみる。「まだ大丈夫、まだまだ大丈夫」とスピードを上げていくと、突然互いにからんだ紙テープが目に入った。「しまった」と慌てて巻き取りを止めるより早く、「ブッチ」といやな音を立てて巻き取り器が止まった。恐る恐る見てみると紙テープが半分切れている。「ああどうしよう、テープでくっつくかな」と思案してみるが、ちぎれた箇所もあり、上手く修復しないと紙テープリーダが読んでくれそうにない。そう最初から出力し直し、巻きなおし。なんとか紙テープ出力が終わり、実機に読みこませてみる。プログラムロードは順調に行き、ようやく単体試験開始。メンテナンスパネルの操作は前に経験した他のシステムとも互換性があり、すぐに慣れた。と瞬く間にバグ発見。修正しないと単体試験が先に進めない。もう一度ミニコン室に戻り修正して紙テープ出力だ。

こうして鎌倉の夜は更けていった。

※１　文字しか表示できない画面のこと。

6. 参考文献

⑴ 鈴木、小林、林、橋本：H-ⅡAロケット用誘導制御計算機について、情報処理 43巻3号、2002年3月

⑵ 清水、島崎、杉、長野：搭載計算機シミュレータ概要、MSS技報 Vol.5、1991年10月

⑶ 戸木田：高信頼性ソフトウェアの検証技術 －H-Ⅱロケット誘導プログラムの開発と検証－、電子情報通信学会誌、1999年8月