Ⅴ プログラミング

到達目標

（1）テスト設計の手順を理解させる。

（2）テスト規格書を作成できるようにする。

（3）プログラミング言語の特徴を理解させる。

（4）妥当なコーディング作業を行えるようにする。

1　プログラミングの位置づけ

プログラミングは、ソフトウェア・ライフサイクルモデルの中で、プログラム設計の次の工程であるコーディング・デバッグに位置付けられる。プログラミングとは、システム開発において、外部設計、内部設計を通じて作りたかった目標の製品（システム）を実際に製作する工程である。

ここでは、最初にテスト設計について述べ、次にコーディング、デバッグ、最後にバージョン管理について述べる。

2　テスト設計

テストは、プログラマがコーディングしたプログラムに、不具合（バグ）がないかを試験するために行うものである。システム開発を行う側のテストは、3段階に分けて以下の順序で行う。

- モジュールテスト
- プログラムテスト
- システムテスト

そして、どのようなテストの項目を設定すればよいかを設計するのが、テスト設計である。テスト設計も、テストの段階に応じて、以下の3段階に分けて行う。

- モジュールテスト設計
- プログラムテスト設計
- システムテスト設計

ここでは各テスト設計について述べる。

（1）モジュールテスト設計

モジュールテストは、単体テストとも呼ばれ、各モジュールが単体で正しく機能しているかどうかを試験するものである。

モジュールテスト設計は、以下のような手順で行う。

イ　テスト項目数の設定

モジュールテストを何項目設定するかを決定する。テスト項目数をｎ、コーディングしたステップ数をｓ、言語別のアセンブラステップ換算係数をｔ、テスト密度をｄとすると、

$$n = s \times t \times d$$

という式で求める。

ここで、コーディングステップ数ｓは、高級言語で記述したときのコメント行を除く実質のステップ数とする。

また、高級言語でコーディングした場合、その１ステップがアセンブラの数ステップに相当する。実際には高級言語の命令ごとに、対応するアセンブラのステップ数は異なるが、ここでは単純化し、言語別のアセンブラステップ数への換算係数ｔを定める。

例えば、ＰＬ/１言語のような、アセンブラに近い記述のできる言語では、t=1.5 と設定する。また、Ｃ言語のような、ＰＬ/１言語より記述力の高い言語では、t=2 と設定する。

次に、テスト密度は、アセンブラ換算したコーディングステップ数に対して、どの程度の割合でテスト項目を設定するかを示すものである。

例えば、モジュールテストでは d=5(%)、プログラムテストでは d=3(%)、システムテストでは d=2(%)というように、テストの段階ごとにテスト密度を設定する。この場合、３段階のテストを合わせると、10%の割合でテストを実施することになる。

（例１）　Ｃ言語で文献表示システムを作成した。Ｃ言語でのコーディングステップ数 s=2000、Ｃ言語のアセンブラステップ換算係数 t=2、モジュールテスト密度 d=5(%)とすると、

$$n = s / t \times d$$
$$= 2000 / 2 \times 0.05$$
$$= 50$$

となり、モジュールテストの項目を 50 項目設定すればよいことになる。

（例２）　ＰＬ/１言語でモニタシステムを作成した。ＰＬ/１言語でのコーディングステップ数 s=1000、ＰＬ/１言語のアセンブラステップ換算係数 t=1.5、プログラムテスト密度 d=3(%)とすると、

$$n = s / t \times d$$
$$= 1000 / 1.5 \times 0.03$$
$$= 45$$

となり、プログラムテストの項目を４５項目設定すればよいことになる。

□ テスト方法の計画

モジュールテストは、各モジュールが単体で正しく機能しているかどうかを試験することが目的である。そのため、モジュールテスト用のドライバとスタブを作成し、動作の確認を行う。

ドライバは、テスト対象のモジュールを呼び出す目的で作成するプログラムである。ドライバから正しい引数を渡してテスト対象モジュールを呼び出したとき、正しい動作を行うか、あるいは誤った引数を渡して呼び出したとき、エラーに対応した動作を行うかなどを確認する。

ところで、テスト対象モジュールが階層構造の中で最も下位に位置するとき、すなわち、対象モジュールが他のモジュールを呼び出していないときは、ドライバからのテストは容易に行うことができる。

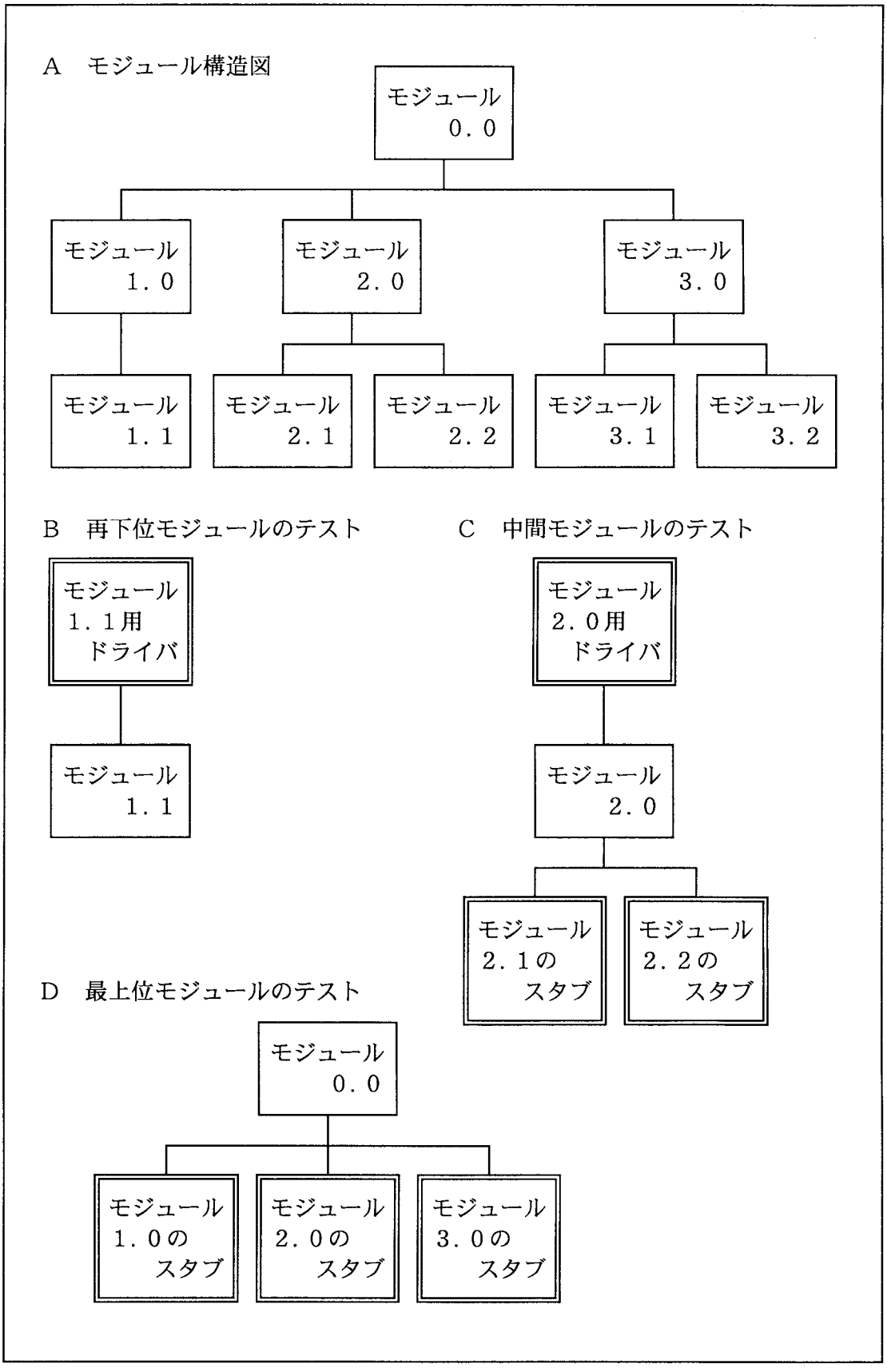
しかし、テスト対象モジュールが階層構造の中で最も下位に位置しないとき、すなわち、対象モジュールが他のモジュールを呼び出しているときは、ドライバだけでテストを行うことができない。

そのような場合はスタブを用意する。スタブは、テスト対象モジュールが内部で呼び出しているモジュールの代わりに、適当な応答を返すプログラムである。これによって、全てのモジュールを単体でテストすることができる。

図Ⅴ-１に、モジュール構造図の例と、その中の三つのモジュールのテスト方法を示す。

ところで、各モジュールごとにドライバとスタブを作成すると、それにかかる工数が膨大になる。そのため、最下位モジュールのモジュールテストをドライバを使って行った後は、そのモジュールは正しい動作をしているものと仮定し、中間モジュールのモジュールテストにおいて、スタブの代わりに下位のモジュールそのものを使用することによって、テスト工数を減らすということも行われる。ただし、この場合、不正な動作が検出されたときに、テスト対象モジュールがその原因なのか、下位のスタブ代わりのモジュールがその原因なのかを切り分けられないという可能性は残るので、スタブの代用は、注意しながら行う必要がある。

テスト方法の計画では、どこまでドライバおよびスタブを作成するかを決定する。



図V-1　ドライバとスタブ

ハ　テスト項目数のモジュールへの割り振り

最初に算出したテスト項目数を各モジュールに割り振る。このとき、重要性の高いモジュールがあらかじめ分かっていれば、そのモジュールに対するテスト項目数を増やしておく。特に、そのようなモジュールが存在しない場合は、各モジュールのステップ数に応じてテスト項目を割り振る。

また、テスト項目を管理するために、モジュールごとに分類記号を定める。例えば、モジュール番号２．１のモジュールの分類記号を決めるのならば、モジュールテストであることを示す「M」と、モジュール番号を短縮した「２１」を組み合わせた「Ｍ２１」とする。表Ｖ-１にテスト項目の割り振りの例を示す。

表Ｖ-１　テスト項目の割り振りの例

モジュール番号	モジュール名	分類記号	ステップ数	テスト項目数
０．０	ｘＭａｉｎ	Ｍ００	２２０	6
１．０	ｘＳｔａｒｔ	Ｍ１０	１９０	5
１．１	ｘＯｐｅｎ	Ｍ１１	２００	5
２．０	ｘＰｒｏｃ	Ｍ２０	３００	9
２．１	ｘＡｄｄ	Ｍ２１	１６０	4
２．２	ｘＳｕｂ	Ｍ２２	１７０	5
２．３	ｘＭｕｌ	Ｍ２３	２６０	6
３．０	ｘＥｎｄ	Ｍ３０	２００	5
３．１	ｘＣｌｏｓｅ	Ｍ３１	１５０	3
３．２	ｘＤｉｓｐ	Ｍ３２	１５０	4
合　　計			２０００	５０

ニ　モジュールごとのテスト規格書の作成

各モジュールごとに、テスト項目数分のテスト規格を作成する。そのとき、誰がテストを行ってもよいように、テスト方法とテスト結果確認事項を明確に規定する必要がある。

表Ｖ-２にモジュールテスト規格書の例を示す。

この例では、モジュール試験項目として４項目が設定されている。そのうち３項目は正常処理に対応し、１項目は異常処理に対応している。そのため、前者の分類はＮ１～Ｎ３（Ｎは Normal の略）、後者の分類はＥ１（Ｅは Error の略）という分類記号を決めてい

る。このテスト項目ごとの分類記号と、モジュールごとの分類記号を接続することによって、全てのテスト項目をユニークな分類記号で表すことができる。

Ｍ２１Ｎ１：モジュール番号２．１のモジュールの正常処理１番目のモジュールテスト
Ｍ２１Ｎ２：モジュール番号２．１のモジュールの正常処理２番目のモジュールテスト
Ｍ２１Ｎ３：モジュール番号２．１のモジュールの正常処理３番目のモジュールテスト
Ｍ２１Ｅ１：モジュール番号２．１のモジュールの異常処理１番目のモジュールテスト

表Ⅴ-２　モジュールテスト規格書の例

モジュール番号	２．１	モジュール名	ｘＡｄｄ	分類記号	Ｍ２１			
分類	テスト内容	テスト方法	テスト結果確認方法	テスト月日				
Ｎ１	正と正の数の加算	dProcドライバから、引数 iParm1=6 iParm2=9 で呼出す。	呼出し後、引数が iResult=15 iError=0 である。					
Ｎ２	負と正の数の加算	dProcドライバから、引数 iParm1=-2 iParm2=4 で呼出す。	呼出し後、引数が iResult=2 iError=0 である。					
Ｎ３	負と負の数の加算	dProcドライバから、引数 iParm1=-55 iParm2=-9 で呼出す。	呼出し後、引数が iResult=-64 iError=0 である。					
Ｅ１	オーバフローエラーの処理	dProcドライバから、引数 iParm1=32766 iParm2=4 で呼出す。	呼出し後、引数が iResult=0 iError=1 である。					

モジュールテスト規格書は、実際にモジュールテストを行うときに参照するものである。そのときは、テスト結果を書き込み、モジュールテスト成績書を作成することになる。

（2）プログラムテスト設計

プログラムテストは、モジュールテストにおいて単体で正しく機能していることが確認されているモジュールをそれぞれ結合して、プログラム全体で正しく機能しているかどうかを試験するものである。

プログラムテスト設計は、以下のような手順で行う。

イ　プログラムテスト項目数の設定

プログラムテストを何項目設定するかを決定する。テスト項目数をｎ、コーディングしたステップ数をｓ、言語別のアセンブラステップ換算係数をｔ、テスト密度をｄとすると、

$$n = s / t \times d$$

という式で求める。

ロ　結合方法の決定

プログラムテスト設計を行う際は、プログラムテストの際の結合をどのような手法で行うかを決定する必要がある。その手法とは、

- ボトムアップテスト
- トップダウンテスト
- ビッグバンテスト

である。それぞれの詳細については、Ⅵで述べる。

ハ　テスト項目数の内容への割り振り

最初に算出したプログラムテスト項目数をプログラムテストの内容に割り振る。これは結合方法にも依存する。

また、テスト項目を管理するために、内容ごとに分類記号を定める。例えば、初期処理の分類記号を決めるのならば、プログラムテストであることを示す「Ｐ」と、処理ごとに付けるユニークな番号を短縮した「０１」を組み合わせた「Ｐ０１」とする。

ニ　内容ごとのテスト規格書の作成

プログラムテストの内容ごとに、テスト項目数分のテスト規格を作成する。そのとき、誰がテストを行ってもよいように、テスト方法とテスト結果確認事項を明確に規定する必要があるのは、モジュールテストの場合と同様である。

また、プログラム規格書の中で、正常処理の分類はＮ１～Ｎ３（Ｎは Normal の略）、異常処理の分類はＥ１～Ｅ２（Ｅは Error の略）というような分類記号を決める。このテスト項目ごとの分類記号と、プログラムテストの内容ごとの分類記号を接続することによって、全てのテスト項目をユニークな分類記号で表すことができる。

Ｐ０１Ｎ１：プログラムテストの初期処理の正常処理１番目のテスト
Ｐ０１Ｅ２：プログラムテストの初期処理の異常処理２番目のテスト

プログラムテスト規格書は、実際にプログラムテストを行うときに参照するものである。そのときは、テスト結果を書き込み、プログラムテスト成績書を作成することになる。

（3）システムテスト設計

システムテストは、プログラムテストで正しく機能していることが確認されているプログラムに対して、システムの当初の目的や性能などの要求仕様を満足しているかを、総合的に検証するものである。

システムテスト設計は、以下のような手順で行う。

イ　システムテスト項目数の設定

システムテストを、何項目設定するかを決定する。テスト項目数をｎ、コーディングしたステップ数をｓ、言語別のアセンブラステップ換算係数をｔ、テスト密度をｄとすると、

$$n = s / t \times d$$

という式で求める。

ロ　テスト項目数の内容への割り振り

最初に算出したシステムテスト項目数を、システムテストの内容に割り振る。

また、テスト項目を管理するために、内容ごとに分類記号を定める。例えば、負荷テストの分類記号を決めるのならば、システムテストであることを示す「Ｓ」と、処理ごとに付けるユニークな番号を短縮した「０３」を組み合わせた「Ｓ０３」とする。

ハ　内容ごとのテスト規格書の作成

システムテストの内容ごとに、テスト項目数分のテスト規格を作成する。そのとき、誰がテストを行ってもよいように、テスト方法とテスト結果確認事項を明確に規定する必要があるのは、モジュールテストやプログラムテストの場合と同様である。

また、システム規格書の中で、０１～０９というような分類記号を決める。このテスト

項目ごとの分類記号と、システムテストの内容ごとの分類記号を接続することによって、全てのテスト項目をユニークな分類記号で表すことができる。

Ｓ０１０６：システムテストの中の負荷テストの６番目のテスト

システムテスト規格書は、実際にシステムテストを行うときに参照するものである。そのときは、テスト結果を書き込み、システム成績書を作成することになる。

3　各テスト設計の順序

テスト設計は、テストを実施する前に行うのは当然であるが、テスト設計とテストの順序は、一般に、明確には決まっていない。

しかし、構造化プログラミング設計手法を採用した場合には、以下のような順序で各工程を進めるべきである。

① 外部仕様書作成
② システムテスト設計
③ 内部仕様書作成
④ プログラムテスト設計
⑤ プログラム仕様書作成
⑥ モジュールテスト設計
⑦ コーディング・デバッグ
⑧ モジュールテスト実施
⑨ プログラムテスト実施
⑩ システムテスト実施

4　プログラミング言語

コーディングは、そこで使用する言語が何かによって、その方法も異なる部分が多い。そこで、最初にプログラミング言語を汎用プログラミング言語、専用プログラミング言語に大きく分類し、それぞれの言語の特徴を述べる。

（1）汎用プログラミング言語

汎用プログラミング言語とは、様々な目的で汎用的に利用されるプログラミング言語を示している。

この汎用プログラミング言語を、二つの視点によって分類する。

ｲ　機械語、アセンブラ言語、コンパイラ言語、インタプリタ言語、簡易言語

プログラミング言語の最も一般的な分類法は、言語を機械語、アセンブラ言語、コンパイラ言語、インタプリタ言語、簡易言語に分類する方法である。

(ｲ) 機械語

機械語は、コンピュータが直接実行することができる機械語コード（マシンコード）からなるプログラミング言語である。そして、コンピュータのＣＰＵのアークテクチャごとに機械語は、別の体系を持っており、他のコンピュータとの言語互換性は存在しない。例えば、ザイログ社のＺ８０というＣＰＵのリターン命令は１６進数のＣ９であるが、モトローラ社の６８０９というＣＰＵのリターン命令は１６進数の３９であるように、全く異なる命令体系になっている。

機械語は、機械には理解しやすいが、人間には理解しにくいため、生産効率が非常に悪い。

以前は、プログラムを高速に動かすために、職人芸的なプログラミングを機械語で行なうことも珍しくなかったが、最近では、プログラマである人間が直接機械語を使ってプログラミングすることはほとんどなくなった。

(ﾛ) アセンブラ言語

アセンブラ言語は、機械語の１６進数の代わりに記号を用いることによって、人間に分かりやすく表現できるようにした言語である。情報処理技術者試験で出題されるCASLなどは、このアセンブラ言語に相当する。

例えば、ザイログ社のＺ８０というＣＰＵでは、リターン命令を表現するのに、機械語のＣ９の代わりにＲＥＴと表現する。

アセンブラ言語で記述されたプログラムは、ＣＰＵごとに用意されたアセンブラという翻訳プログラムによって解析・翻訳され、機械語に変換されて実行される。

このように、人間がアセンブラ言語を使うことによって、機械語を直接使うよりも命令を理解しやすくなり、生産効率が向上する。ただし、アセンブラ言語の命令と機械語の命令は１対１に対応しており、コンピュータごとの言語互換性が存在しないのは、機械語の場合と同じである。

(ﾊ) コンパイラ言語

コンパイラ言語は、人間が理解しやすい命令体系をもつプログラミング言語である。FORTRAN言語、COBOL言語、Ｃ言語、PASCAL言語、C++言語などがこれに相当する。これらの言語で記述されたプログラムは、各言語ごとに用意されたコンパイラという翻訳プログラムによって解析・翻訳され、最終的には機械語に変換されて実行される。

アセンブラ言語とコンパイラ言語の違いは、アセンブラ言語の命令と機械語の命令が１対１に対応しているのに対して、コンパイラ言語の命令と機械語の命令が１対ｎに対応していることである。このため、人間が行うプログラミングの負荷は、その分軽くなることになる。

また、一般に、コンパイラ言語で記述した場合は、機械語やアセンブラ言語で記述した場合に比べ、無駄な機械語命令が多くなり、実行時の性能が遅くなるといった問題もあった。そのため、コンパイラには、最適化機能が備え付けられており、性能の向上を図る工夫がされている。

ところで、コンパイラ言語を使ってプログラミングを行う場合、すべての命令が正しくないとコンパイルエラーが発生し、プログラムを実行することはできないのが通常である。そのため、低速なコンパイラを使用している場合などは、次に述べるインタプリタ言語によるプログラミングに比べて、プログラミングやデバッグに時間がかかるという問題がある。

(ニ) インタプリタ言語

インタプリタ言語は、コンパイラ言語と同じく、人間が理解しやすい言語をもつプログラミング言語である。ＢＡＳＩＣ言語などがこれに相当する。

ＢＡＳＩＣ言語で記述されたプログラムは、インタプリタというプログラムによって命令ごとに解析・翻訳され、最終的には機械語に変換されて実行される。この場合、プログラム全体が終了するまで、命令単位で解析・翻訳・機械語変換・実行が行われるところが、コンパイラとの違いである。つまり、インタプリタ言語の場合、プログラム全体が完成していなくとも、できあがっている部分だけで動作の確認をすることができる。

しかし、実行速度を比べると、インタプリタ言語で記述したプログラムよりも、コンパイラ言語で記述したプログラムの方が、より高速である。それは、インタプリタ言語で記述したプログラムの実行のときは、命令ごとに翻訳と機械語変換を行わなければならないのに対して、コンパイラ言語で記述したプログラムの実行のときは、既にできあがった機械語を実行するだけだからである。

そこで、インタプリタとコンパイラの長所・短所を補完するため、最初はＢＡＳＩＣ言語で記述されたプログラムをＢＡＳＩＣインタプリタで実行しながらプログラミングやデバッグを行い、完全に動作するようになったら、同じＢＡＳＩＣ言語で記述されたプログラムをＢＡＳＩＣコンパイラで機械語に変換する（コンパイルする。）という方法が採用されることが多い。これは、ＢＡＳＩＣコンパイラでコンパイルしたプログラムの方が、ＢＡＳＩＣインタプリタで実行するよりも高速に動作するからである。

(ホ) 簡易言語

簡易言語は、コンパイラ言語やインタプリタ言語を使いこなすほどのプログラミング

知識をもっていない人でも、容易に使いこなすことができるプログラミング言語である。ＲＰＧ言語などがこれに相当するとされている。

ＲＰＧ言語で記述されたプログラムは、ジェネレータと呼ばれるプログラムによって、実行可能な形式に変換される。ジェネレータは、あらかじめ用意された部品プログラムを組み合わせ、それらに適切なパラメータを指定することによって、最終プログラムを自動的に作成するというものである。

ロ　低水準言語と高水準言語

プログラミング言語を機械（コンピュータ）と人間（プログラマ）とで分類した場合、機械が理解しやすい言語のことを低水準言語と呼び、逆に、機械よりも人間が理解しやすい言語のことを高水準言語と分類することができる。

機械語やCASL言語のようなアセンブラ言語は、低水準言語に分類される。

また、コンパイラ言語であるFORTRAN言語、COBOL言語、Ｃ言語、PASCAL言語、C++言語や、インタプリタ言語であるＢＡＳＩＣ言語、簡易言語であるＲＰＧ言語などは、高水準言語に分類される。

低水準言語は、コンピュータのＣＰＵのアークテクチャごとに言語が違うという特徴をもつ。そのため、あるコンピュータから別のコンピュータへ、低水準言語で記述されたプログラムを移植することは、一般には大変困難である。

一方、高水準言語は、一部にはコンピュータのアークテクチャに依存する部分を持っているものもあるが、低水準言語に比べ、言語ごとに標準化されているものが多い。例えば、FORTRAN言語、COBOL言語、Ｃ言語などは、国際的な標準化規格が制定されているため、あるコンピュータから別のコンピュータへプログラムを移植することは、比較的容易である。しかし、ＢＡＳＩＣ言語のように、コンピュータメーカごとに方言があり、互換性が失われてしまった高級言語もある。

ハ　主要なプログラミング言語

主要なプログラミング言語について、その概要を説明する。

(イ) ＦＯＲＴＲＡＮ言語

「FORmula TRANslator」を略した言語である。ＩＢＭによって1958年に開発された、最初の汎用プログラミング言語である。開発当時は、メモリなどのコンピュータ資源がきわめて貧弱であり、プログラミングは機械語またはアセンブラ言語で効率的に作るのが常識であった。ところが、FORTRANコンパイラが生成した機械語の効率がかなりよく、しかも高水準言語であったため、非常に多くのプログラムがFORTRAN言語で作成された。実用面で大成功を納めた、世界初のコンパイラ言語であるといえる。

ただし、言語仕様に曖昧な点があり、コンパイラの検出できないエラーが原因となり、

アメリカの宇宙開発であるアポロ計画を失敗に陥れたこともある。

最近では、これまでに蓄積された膨大なFORTRANライブラリを利用した、科学技術計算に使用されることが多い。

(ロ) ＣＯＢＯＬ言語

「COmmon Business Oriented Language」を略した言語である。CODASYL委員会によって1959年に発表された、ビジネス計算用プログラミング言語である。COBOL言語の特徴は、プログラムの記述が自然言語である英語に近い記述法であること、大量の入出力データを扱うことが得意であることなどである。これまで、ファイルシステムやデータベースシステムとCOBOL言語を組み合わせた、数多くのプログラムが作成されてきており、その資産は膨大である。

しかし、COBOL言語は、もともと構造化プログラミングに適していなかった。そこで、最近では、構造化COBOL言語へと進化し、構造化プログラミングも容易に行えるようになってきている。

(ハ) ＰＡＳＣＡＬ言語

ＰＡＳＣＡＬ言語は、スイスのN. Wirthが1960年代末に開発した構造化プログラミング言語である。

PASCAL言語は、ALGOL言語の考え方に基づき、読みやすい構文と処理系に負担をかけない構文解析ができるよう設計されている。また、型チェックが厳密であり、整った制御構造が容易に作成できるため、主に教育用プログラミング言語として高い評価を受けている。さらに、PASCAL言語で書かれたプログラムは、他のコンピュータへの移植性も高かったため、多くの種類のコンピュータ上でPASCAL言語コンパイラが動作している。

そして、PASCAL言語は、その後登場したＣ言語に大きな影響を与えることとなった。

(ニ) ＢＡＳＩＣ言語

「Beginner's All purpose Symbolic Instruction Code」を略した言語である。

元々は、1963年にアメリカのT. E. Kurtzらによって開発された初心者向けの会話型言語であった。その当時は、汎用コンピュータのＴＳＳ端末から、複数の人がＢＡＳＩＣ言語でプログラミングを行うという形態をとっていた。

その後、パソコンにＢＡＳＩＣインタプリタが付属されることになり、高水準言語として、数多くのプログラムがＢＡＳＩＣ言語で作成された。しかし、ＢＡＳＩＣを各メーカーが独自に拡張したため、他のコンピュータとのプログラムの移植性が失われてしまった。

最近、Windows用のプログラミングツールとして、Ｖｉｓｕａｌ ＢＡＳＩＣが注目され

ている。これは、構造化ＢＡＳＩＣ言語を採用しており、新たにＢＡＳＩＣ言語を使うユーザを数多く獲得している。

(ホ) ＲＰＧ言語

「Report Program Generator」を略した言語である。簡易言語の代表例とされる。

ＲＰＧ言語は、報告書作成に適した言語であり、データ処理を行うアルゴリズムを記述することなく、所定の用紙にパラメータを記入するだけで、プログラムが作成できてしまうというものである。

このため、プログラム作成の生産効率を大きく引き上げることができる。

(ヘ) Ｃ言語

Ｃ言語は、アメリカのベル研究所で、D. Ritchie が 1970 年に開発したプログラミング言語である。Ｃ言語は、ＵＮＩＸを記述した言語として有名である。この結果、きわめて効率よくシステムプログラムが記述できるプログラミング言語として、高い評価を受けることとなった。

Ｃ言語の特徴は、高水準言語のもつプログラミングの容易さと、低水準言語のもつ細かいところまで手の届く記述力の高さを、両方兼ね備えていることである。これが、従来の汎用プログラミング言語に不満を持つプログラマに受け入れられ、急速に普及することとなった。

また、Ｃ言語は移植性が高く、あるコンピュータから別のコンピュータへプログラムを移植することも比較的容易に行える。現在では、ほとんど全ての種類のコンピュータ上で、Ｃ言語コンパイラが動作している。

(ト) ＣＡＳＬ言語

ＣＡＳＬ言語は、情報処理技術者試験の出題に用いられているアセンブラ言語である。仮想的なコンピュータであるＣＯＭＥＴ上で動作する機械語を生成することができる。ＣＯＭＥＴは、１６ビットのワードマシンであり、ＧＲ０からＧＲ４までの５本のレジスタをもつ。

(チ) C++言語

C++言語は、アメリカのベル研究所で、B.Stroustrup が 1983 年に開発したプログラミング言語である。C++言語は、Ｃ言語から進化した言語という意味合いをもち、Ｃ言語を基礎にしながら、その上に、オブジェクト指向プログラミングの考え方を導入したものである。

構造化プログラミング手法では、実現が困難であった部品の流用による生産性の向上が、オブジェクト指向プログラミング手法によって容易にできるようになった。

（2）専用プログラミング言語

専用プログラミング言語とは、ある特定の問題に特化した専用のプログラミング言語である。この専用プログラミング言語は、以下のように大きく分類できる。

イ　シミュレーション言語

結果を知りたい問題に対して、実際に実物を試作し、それを動作させて結果を求めるのではなく、模擬的にコンピュータ上で動作させ結果を類推することをコンピュータシミュレーション（Computer　Simulation）という。この場合、莫大な量のデータを扱い、しかも試行錯誤を行いながら繰返し計算が必要となるため、従来はスーパーコンピュータのような高速で、多量のデータを処理できるコンピュータがなければ不可能であった。現在では、コンピュータの性能が急速に向上したため、従来よりも安価なコンピュータ上でシミュレーションを実行することができるようになっている。

特に、先端技術研究や製品開発の分野では、実際に実物を試作する費用と期間がかからないコンピュータシミュレーションは、大変魅力的であるといえる。

ところで、コンピュータシミュレーションに、汎用プログラミング言語であるFORTRAN言語やC言語を使用することも多いが、特定の問題解決には、専用のシミュレーション言語を使用した方が簡単であることも多い。

以下に、伝統のあるシミュレーション言語を三つ紹介する。

(イ) ＧＰＳＳ

GPSS（General Purpose Simulation System）は、データが非連続な値で存在している離散系と呼ばれる分野のシミュレーション言語である。GPSSでは、システムの構造をブロックダイアグラムという形で表現する。ブロックダイアグラムは、既存のブロックタイプの組み合わせで記述され、容易にシステムを記述することができる。GPSSを用いれば、待ち行列の解析を統計的な手法で行うことなどが容易になる。

(ロ) ＳＩＭＳＣＲＩＰＴ（シムスクリプト）

ＳＩＭＳＣＲＩＰＴ（SIMulation　SCRIPTer）は、GPSSと同じく、離散系と呼ばれる分野のシミュレーション言語である。ＳＩＭＳＣＲＩＰＴでは、英語に近い表現でシミュレーションを行うことができるという特徴を持つ。

(ハ) ＤＹＮＡＭＯ（ダイナモ）

DYNAMO（DYNAmic MOdels）は、データが連続な値で存在している連続系と呼ばれる分野のシミュレーション言語である。DYNAMOには、インダストリアルダイナミックスという考え方が使われている。これは、計量経済モデルでモデリングされた経済システムをシミュレートするために使用されることが多い。

ロ　人工知能言語

推論、学習、連想といった、人間は得意であるが、これまでコンピュータは苦手としていた機能をコンピュータに持たせようという試みを人工知能という。人工知能は、ＡＩ（Artificial　Intelligence）とも呼ばれ、日本でも、10 年間に渡り、通産省の第５世代コンピュータプロジェクトにおいて、ＡＩの研究が行われた。

以下に、人工知能記述言語として有名な二つの言語を紹介する。

(イ) ＰＲＯＬＯＧ

PROLOG（PROgramming in LOGic）は、記号論理を使って論理的な関係を記述する言語である。前述の第５世代コンピュータプロジェクトでも、PROLOGが人工知能言語として採用された。しかし、PROLOGを用いたＡＩの研究自体は、最近行き詰まりを見せており、現状打破（ブレークダウン）が必要な状況になっている。

(ロ) ＬＩＳＰ

ＬＩSP（LISt　Processing）は、PROLOGと並んで、人工知能言語を代表する言語である。リスト処理という手法を用いて、異なった種類のデータを処理することができる。また、実行中にプログラムを自ら修正できるという特徴を持つことが、人工知能言語と呼ばれる所以である。

ハ　プロセス制御用言語

生産工程で、データ誤差の制御や、機械の稼働時間測定などの目的で用いられる言語である。PROSPRO（PROcess Supervisory PROgram）や、ＢＩCEPS（Basic Industrial Control Enginnering Programming System）がその代表例である。

ニ　数値制御用言語

工業機械の動作を数値情報で制御することを数値制御（ＮＣ）という。数値制御用言語は、ＮＣのための専用言語である。ＡＰＴ（Automatically Programmed Tools）は、その代表例である。

最近は、パソコンによるＮＣが盛んに行われるようになってきた。その場合、制御するための数値制御用言語が、メーカーごとにまちまちであったため、現在、メーカ間で数値制御用言語の統一が行われている。

5　コーディング

コーディングを行う際には、コーディング規約を定め、システム開発を行うメンバーの中でそれを徹底させておかなければならない。ここでは、コーディング規約で規定すべきことにつ

いて述べる。

（1）ネスト

プログラムをコーディングする場合、見やすいプログラムを作ろうとする意識が必要である。特に、構造化プログラミング手法に基づいてコーディングする場合、ネスト（字下げ）を行うことによって、階層の深さが一目で分かり、プログラムの理解を助けてくれるという利点がある。

図Ⅴ-２に、Ｃ言語によるネストの例を示す。

```
012345678                                字下げ  階層
↓  ↓  ↓
#include <stdio.h>                        0      0
main()                                     0      0
    {                                      4      1
    int iCount;                            4      1
    iCount=1;                              4      1
    while (iCount<10)                      4      1
        {                                  8      2
        iCount=iCount+1;                   8      2
        printf("%d ",iCount);              8      2
        }                                  8      2
    }                                      4      1
```

図Ⅴ-２　Ｃ言語によるネストの例

（2）ネーミング規則

システム開発は、一般に複数のメンバーで行うことが多い。そのようなときに、プログラムのモジュール名をメンバーが自由に命名すると、他のメンバーの作ったモジュール名と重複する可能性がある。また、システムを幾つかのサブシステムに分けて開発する場合、あるモジュールがどのサブシステムに属するのかをモジュール名だけ見ればわかるようになっていれば便利である。

また、共通データ域の変数は、多くのモジュールが参照・更新するため、変数名も自由に命名できない。

そのような理由から、プログラムのモジュール名や共通データ域の変数名は、頭の何文字かに、規則的な記号を付けることが行われる。これをネーミング規則を決めるという。

例えば、大きなシステムを三つのサブシステムＡ、Ｂ、Ｃに分割した場合、サブシステム

Ａに属するモジュール名の頭には、必ずＡをつけ、サブシステムＢに属するモジュール名の頭には、必ずＢをつけるといったネーミング規則である。この場合、２文字から後は、自由な名前を付けることができる。

また、最近は、一つのモジュールの中で使われるローカルな変数にも、データ型を示す記号を頭の何文字かに付けることも行われている。これによって、変数間の型が一致しないなどの不具合をネーミング規則から判断することができる。

その代表例は、Windows のプログラムで使用されているハンガリー記法である。例えば、整数型の変数には、先頭に小文字の i を付けたり、0 で終わる文字列型の変数には、先頭に小文字の sz を付けたりする。これらの場合、その直後の文字は大文字で、さらにその後の文字は、大文字と小文字で自由に名前を付けることができる。

（3）コメント

プログラミング言語には、コメント行（注釈行）を記述する機能が存在する。この機能を使って、プログラムを理解しやすくすることができる。

コメントは、モジュールの先頭にまとめて記述するものもあれば、各実行文を理解しやすくするために、各実行文ごとに記述するものもある。

特に、モジュールの先頭にまとめて記述するコメントには、モジュール番号、モジュール名、呼出し手続き、入出力変数、モジュール概要などを記述しておけば、後でプログラムを読み返したときの理解力が高まる。

また、コピーライトもこの部分に表示しておくことにより、プログラム作成者の利益を保護することができる。

図Ⅴ-３に、Ｃ言語によるモジュール先頭のコメントの例を示す。

6　コンパイル／デバッグ

コーディングしたプログラムは、そのプログラミング言語のコンパイラでコンパイルチェックを行う。このとき、プログラムに文法的な誤りが発見されると、コンパイラエラーが表示される。そこで、コンパイルエラーに表示されるエラー発生行番号とエラーのメッセージを解析し、プログラムをデバッグする。デバッグは、プログラムの不具合（バグ）を検出し、修正していくという処理である。その後、再度、コンパイラでコンパイルチェックを繰り返す。

そして、最終的にコンパイルエラーが０個になったら、コンパイルチェックは終了である。

なお、コンパイラの中には、コンパイルチェックの程度を調節できるものがある。そのようなコンパイラの場合は、コンパイルチェックの程度を一番厳しくしておくことが望ましい。それによって、型変換の誤りなどチェックが甘い場合は、警告で終わるものをエラーとして検出してくれるため、コンパイルチェックの段階で不具合が紛れ込む可能性が低くなる。

デバッグには、以下の２種類の方法が存在する。

```
/*********************************************************/
/*                                                       */
/*  モジュール番号：２．１                               */
/*  モジュール名   ：xAdd                                */
/*  呼出し手続き   ：xAdd(iParm1, iParm2, iResult, iError); */
/*  入力変数       ：iParm1  INT  2byte  １番目の引数    */
/*                 ：iParm2  INT  2byte  ２番目の引数    */
/*  出力変数       ：iResult INT  2byte  結果出力        */
/*                 ：iError  INT  2byte  エラー出力      */
/*  モジュール概要：２つの引数の値を加え、その結果を     */
/*                  戻す。もし、オーバフローが発生し     */
/*                  た場合は、エラーに１を返す。         */
/*                                                       */
/*  Copyright (C) by Taro Kozo 1995,1996                 */
/*  All Rights Reserved.                                 */
/*********************************************************/
```

図Ｖ－３　Ｃ言語によるモジュール先頭のコメントの例

（１）マシンデバッグ

コンピュータの端末の前で、キーボードを叩きながら不具合を検出する方法である。コンピュータが高価であった頃は、コンピュータの端末を用いたデバッグに多くの費用がかかり、なるべくマシンデバッグは行わない努力がなされたが、安価なパソコンが普及した現在では、その努力は意味をなくしつつある。

デバッガは、マシンデバッグの効率を向上させるためのツールである。プログラムの実行を任意の行で一時停止させ、変数の値を調べたりすることができる。

（２）机上デバッグ

プログラムに誤りがないかを人力でチェックする方法である。プログラムリストをプリンタ用紙に出力し、目でロジックを追ってデバッグを行う。

プログラムを作成した本人が机上デバッグを行う場合は効率がよいが、本人以外が机上デバッグを行っても、理解のための余計な時間がかかり効率は良くない。

7　バージョン管理

コーディング後のデバッグを行う段階で、モジュールには数多くの修正が加わることになる。

しかし、もし誤った修正を行ったとき、修正前の状態に戻すことができれば影響は少ないが、戻せない場合には影響が大きい。

そこで、ソースプログラムのバージョン管理を行い、前のバージョンに戻せるようにすることによって、デバッグの効率を向上させることができる。

バージョン管理を行うときには、バージョン管理者を決めておくことが望ましい。そして、バージョン管理者は、開発担当者が開発した各部分を一つのプログラムとして集積し、それにバージョン番号を付加する。その際、開発担当者は、どの部分が正常に動作しないのかを明確にし、バージョン管理者に伝えておく必要がある。さもないと、集積したプログラムを使用する他の開発担当者は、発生する障害の原因の切り分けに無駄な時間を浪費してしまう可能性がある。

また、プログラムの集積を行うとき、以前のバージョンでは発生していなかった障害（バグ）が混入することがある。このようなケースは、劣化と呼ばれる。

劣化が発生した場合、集積後の開発作業が大きな影響を受ける。したがって、劣化を防止する手段を講じる必要がある。

具体的には、レグレッションテストを行うことによって、劣化を防止することができる。レグレッションテストは、既に前のバージョンでは動作していた機能が、新しく集積したバージョンでも正しく動作しているかを確認し、それによって劣化を発見する目的で実施するテストである。