目　次

1 はじめに

本冊子の位置づけ

携帯電話、デジタルテレビ、自動車など、我々の身の回りには様々な情報機器が溢れています。こうした情報機器の多くには、いわゆるマイコンが搭載されており、その上で組込みソフトウェアが動作しています。このような情報機器の多くは、我々の日常生活の場で広く利用されるものがほとんどですが、もし、これらの機器が動作トラブルを起こしたらどうなるでしょうか？　恐らく、次の瞬間から、我々の生活の一部に支障をきたすことは明らかです。

我が国の工業製品は、これまでその品質が高いことを最大の特徴として世界を席巻してきました。しかし、情報技術の進化と普及に伴い、従来にない複雑な情報機器が出現し、それにつれて、製品の不具合が散見されるようになってきています。表1.1は、2004年にマスコミなどで報道された情報機器関連……恐らく何らかの形で組込みソフトウェアが原因となった製品不具合を示したものです。このように公表される製品不具合は全体の一部であり、実際にはこの数倍の製品トラブルが潜んでいるものと考えられます。

本冊子は、上記のような背景のもと、どのようにしたら、ソフトウェアの品質を安定したものにし、向上させることができるかについて、コーディング作業を中心にその取り組み方法などを紹介したものです。コーディングという作業は、最終的に製品上で動作するソフトウェア・プログラムを直接的に作る作業であり、ここでの不具合などの混入は製品品質にとって致命傷となります。ぜひ、ソフトウェア品質向上のための基本を理解して、よりよい製品を世の中に送り出していただければと思います。

本冊子の構成

本冊子では、組込みソフトウェア製品の品質を向上するための基本動作について、コーディング作業を中心に紹介します。また、ソフトウェア品質向上のための導入書として、ソフトウェア品質向上の考え方なども合わせて紹介していきます。

- ソフトウェア品質の考え方
- ソフトウェア品質向上の考え方
- ソースコード品質向上のための技術
- コーディング作法とその利用方法

公表日	製品名	不具合の内容
3月	携帯電話	メモリカードの画像を縮小加工後、電話着信や特定の操作で再起動する
4月	無線LANカード	無線LANカードを対応のADSLモデムに取り付けてもアクセスポイントとして使えない
5月	HDDビデオレコーダ	50Gバイト・ハードディスクの112Gバイト以降に録画／再生できない
6月	HDDビデオレコーダ	ハードディスクに録画した番組を分割して編集する際に操作できなくなる
8月	DVDレコーダ	DVDレコーダで電源がオン状態の時に録画予約メールを受信できない
8月	携帯電話	メールの題名入力後すばやく本文入力に移ろうとすると操作できなくなる
9月	携帯電話	定額制サービスの接続先として指定するサーバの設定を誤り、料金を誤請求した
10月	携帯電話	内蔵カメラで撮影した映像が青みがかるため、出荷日に販売を延期した
10月	デジタルカメラ	連続撮影中やオートパワーオフ状態でレンズを着脱後、デジタルカメラが操作不能になる
10月	携帯電話	電話を発信するとほぼ同時に着信があった場合、電話帳の内容が消失する

表1.1　組込みソフトウェア不具合の事例（2004年）
出展：日経コンピュータ、2004.12-27（No.616）「組込みソフトの巨大化に立ち向かう」より

本冊子の対象読者

この冊子では、以下のような読者層に対して、ソフトウェア品質向上のための基本動作や考え方を知っていただくことを目指しています。

経営者の方へ

組込みソフトウェアに求められる品質とはどのようなものか、その品質を安定させ､向上させるためにはどのような技術が必要となるかを認識し、組込みソフトウェア品質向上の意義と必要性を認識されることを期待しています。

組込みシステム開発のチームリーダの方へ
(ソフトウェア担当の方だけではなく、ハードウェア担当の方へも)

ここで紹介する組込みソフトウェア品質の概念や考え方や基本動作を理解していただき、それらをご自身が担当するプロジェクト内で実践していただくことを期待しています。

組込みシステムエンジニアの方へ

組込みソフトウェアのコーディングなどご自身が担当される業務の中で、常にソフトウェア品質を意識した作業を心がけ、品質の作りこみを実践されることを期待しています。

2 ソフトウェア品質の考え方

▶▶2-1 ソフトウェアバグと品質

ソフトウェアの基本的な品質……機能・性能

現代の組込み機器は、複雑で付加価値の高い機能をソフトウェアで実現するようになってきています。図2.1にあるように、組込み機器で実現する機能の増加とともに、ソフトウェアの規模が拡大していることからもこの傾向は裏付けられます。

このように、組込みソフトウェアは組込み機器の製品としての価値のかなりの部分を担っているといえます。

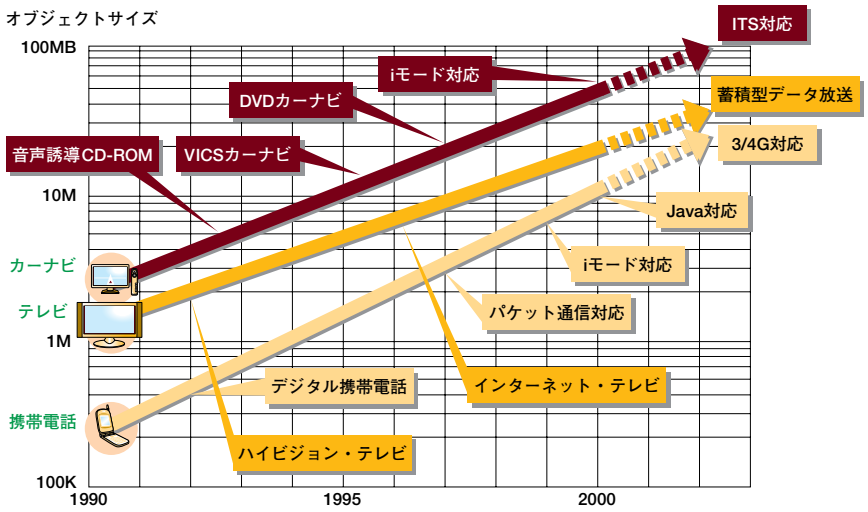


図2.1　オブジェクトサイズの推移　　出典：日経エレクトロニクス、2000 9-11（No.778）をベースに追加、修正。

皆さんが数ある候補の中からある製品を選ぶ時の理由にはどういったものがあるでしょう。価格やデザインも大きな選択理由ではありますが、どういった機能・性能があるか、操作性はどうかという部分が大きな比重を

占めているのではないでしょうか。そして一般の多くのユーザは、このような機能は完全に動作することを当然として考えられているはずです。

ソフトウェアバグの影響

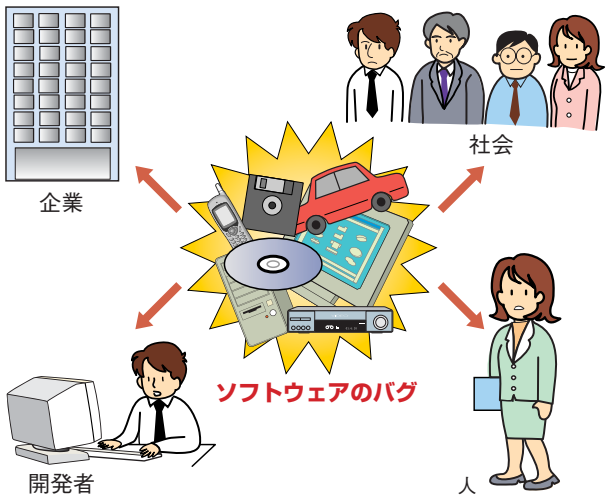
万一、そのような製品のソフトウェアにバグがあった場合どのようなことになるでしょう。

使用者側への影響

- 本来使えるはずの機能が使えないためユーザに不都合をもたらす。
- ユーザに物的損害や人的損害を引きおこす場合もある。

企業側への影響

- 製品の評価が下がる。
- 企業の信用が失墜する。
- 改修のためのコストが発生する。



開発部門への影響

- 現在行っている作業を中止して改修のための作業を行わなければならない。
- 開発部門の社内的評価が下がる。

このように、ソフトウェアのバグが及ぼす影響は多方面にわたり、かつ重大なものとなり得るのです。

▶▶2-2 ソフトウェア品質特性モデル

それでは、品質とは何でしょう。一般には、前項で取りあげたような、当然達成できている製品の完成度を漠然と指す場合が多いように感じられます。しかし、ソフトウェアの特性を厳密に見ていくと、いろいろな要素が絡み合って、総合的な品質が形成されていることがわかります。品質の概念とはバグだけでないのです。

ソフトウェア製品の品質について､その特性をいろいろな角度から分析､整理したものがソフトウェア品質特性モデルと呼ばれ、ISO/IEC9126（JIS X 0129-1）に定義されています。

このような特性を理解することで、ソフトウェア開発の各段階での出来具合を評価する時に、それぞれの視点から品質の悪い部分がないかチェックすることも可能となります。

品質特性は6つの大きな分類と、それぞれいくつかの副特性の定義とで構成されています。それを、図2.2と表2.1に示します。

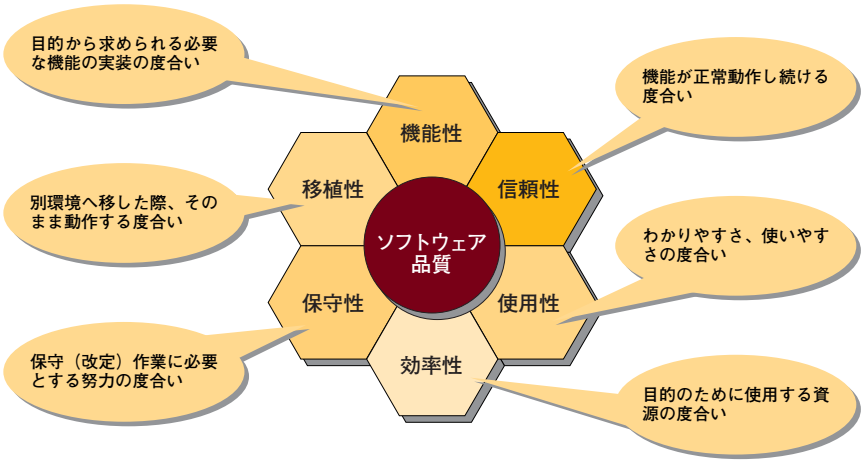


図2.2　ISO/IEC9126（JIS X 0129-1）ソフトウェア品質特性

品質特性（JIS X 0129-1）		品質副特性（JIS X 0129-1）	
信頼性	指定された条件下で利用するとき、指定された達成水準を維持するソフトウェア製品の能力	成熟性	ソフトウェアに潜在する障害の結果として生じる故障を回避するソフトウェア製品の能力
		障害許容性	ソフトウェアの障害部分を実行した場合、または仕様化されたインタフェース条件に違反が発生した場合に、指定された達成水準を維持するソフトウェア製品の能力
		回復性	故障時に指定された達成水準を再確立し、直接に影響を受けたデータを回復するソフトウェアの能力
		信頼性標準適合性	信頼性に関連する規格または規約を遵守するソフトウェア製品の能力
保守性	修正のしやすさに関するソフトウェア製品の能力	解析性	ソフトウェアにある欠陥の診断または故障の原因の追求、およびソフトウェアの修正箇所の識別を行うためのソフトウェア製品の能力
		変更性	指定された修正を行うことができるソフトウェア製品の能力
		安定性	ソフトウェアの修正による、予期せぬ影響を避けるソフトウェア製品の能力
		試験性	修正したソフトウェアの妥当性確認ができるソフトウェア製品の能力
		保守性標準適合性	保守性に関連する規格または規約を遵守するソフトウェア製品の能力
移植性	ある環境から他の環境に移すためのソフトウェア製品の能力	環境適応性	ソフトウェアにあらかじめ用意された以外の付加的な作法、または手段なしに、指定された異なる環境にソフトウェアを適応させるためのソフトウェア製品の能力

品質特性（JIS X 0129-1）		品質副特性（JIS X 0129-1）	
移植性	ある環境から他の環境に移すためのソフトウェア製品の能力	設置性	指定された環境に設置するためのソフトウェアの能力
		共存性	共通の資源を共有する共通の環境の中で、他の独立したソフトウェアと共存するためのソフトウェア製品の能力
		移植性標準適合性	移植性に関連する規格または規約を遵守するソフトウェア製品の能力
		置換性	同じ環境で、同じ目的のために、他の指定されたソフトウェア製品から置き換えて使用することができるソフトウェア製品の能力
効率性	明示的な条件の下で、使用する資源の量に対比して適切な性能を提供するソフトウェア製品の能力	時間効率性	明示的な条件の下で、ソフトウェアの機能を実行する際の、適切な応答時間、処理時間及び処理能力を提供するソフトウェア製品の能力
		資源効率性	明示的な条件の下で、ソフトウェア機能を実行する際の、資源の量及び資源の種類を適切に使用するソフトウェア製品の能力
		効率性標準適合性	効率性に関連する規格または規約を遵守するソフトウェア製品の能力
機能性	ソフトウェアが、指定された条件の下で利用されるときに、明示的および暗示的必要性に合致する機能を提供するソフトウェア製品の能力	合目的性	指定された作業及び利用者の具体目標に対して適切な機能の集合を提供するソフトウェア製品の能力
		正確性	必要とされる精度で、正しい結果もしくは正しい効果、または同意できる結果もしくは同意できる効果をもたらすソフトウェア製品の能力
		相互運用性	一つ以上の指定されたシステムと相互作用するソフトウェア製品の能力
		セキュリティ	許可されていない人またはシステムが、情報またはデータを読んだり、修正したりすることができないように、および許可された人またはシステムが、情報またはデータへのアクセスを拒否されないように、情報またはデータを保護するソフトウェア製品の能力（JIS X 0160:1996）
		機能性標準適合性	機能性に関連する規格、規約または法律上および類似の法規上の規則を遵守するソフトウェア製品の能力
使用性	指定された条件の下で利用するとき、理解、習得、利用でき、利用者にとって魅力的であるソフトウェア製品の能力	理解性	ソフトウェアが特定の作業に特定の利用条件で適用できるかどうか、およびどのように利用できるかを利用者が理解できるソフトウェア製品の能力
		習得性	ソフトウェアの適用を利用者が習得できるソフトウェア製品の能力
		運用性	利用者がソフトウェアの運用及び運用管理を行うことができるソフトウェア製品の能力
		魅力性	利用者にとって魅力的であるためのソフトウェア製品の能力
		使用性標準適合性	使用性に関連する規格、規約、スタイルガイドまたは規則を遵守するソフトウェア製品の能力

表2.1　ISO/IEC9126（JIS X 0129-1）ソフトウェア品質特性と副特性

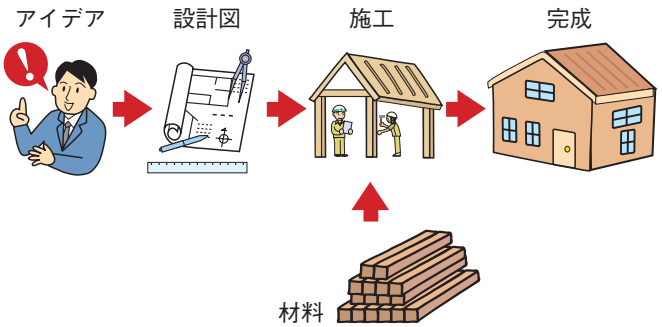
▶▶2-3 ソフトウェア品質低下の原因

品質が低下してしまう原因はどこにあるのでしょう。

ソフトウェアの開発も他の工業製品を作ることと似ている面が多くあります。ソフトウェア品質が低下してしまう原因を住宅建築の例と比べてみます。

住宅建築での品質低下の原因

まず初めに、どんな家にしようか考えます。次に設計図を書きます。次に部材を集め工事をします。工事が完了すれば家のできあがりです。



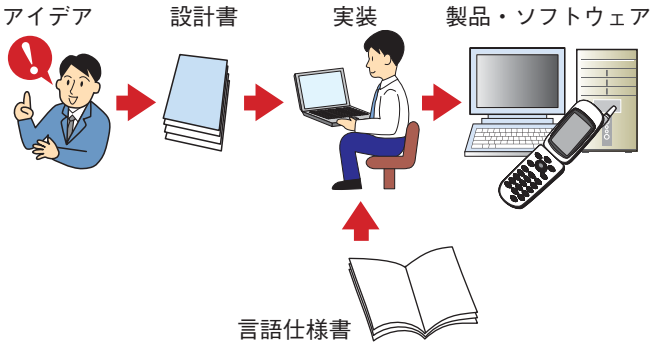
初めに想定した家のイメージと、完成した家の間にずれはないでしょうか。もし違いがあるとすれば求められる品質に達していないことになります。

- イメージから設計図を引く時に正確に要求を反映できていなかった。
- イメージから設計図を引く時に要求の一部が漏れてしまった。
- 設計図から部材を加工するときに間違えてしまった。
- 指定された材質以外の部材を集めてしまった。
- 設計図を間違って読んで工事してしまった。

などが不具合の原因でしょうか。さらに、そもそも結果的には使い勝手の悪いものができあがってしまうようなイメージをしてしまうと、その後の作業が100％間違いなく実施されても、できあがったものは使い勝手の悪いものとなってしまいます。傾斜のある土地に建ててしまい、悲惨なことになるなどというようなこともあり得ます。

ソフトウェア品質低下の原因

ソフトウェアも同じような流れで作業します。初めにどのようなソフトウェアにするか考え、設計書を書きます。次いで言語仕様に沿ってプログラミングしていきます。



さらに、ソフトウェアの動作はその開発の過程が家のように具体的に目に見えるものではないため、本当に期待したように動作するのかテストする必要もあります。

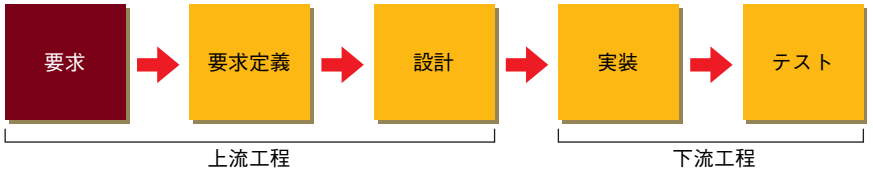


図2.3　ソフトウェア設計工程の例

ソフトウェア品質が低下する事例として

・上流では
①要求定義に抜けがある
②要求定義が曖昧
③設計の誤り
④設計の抜け

・下流では
①設計書の読み間違い
②コーディングの誤り
③テスト仕様の漏れ

などがあげられます。さらに、それぞれの作業に品質特性モデルで示した要素の側面があるのでさらに複雑になります。

着々と作業を行い、他は完璧なソフトウェアができてもたった1カ所のコーディングミスで大きな品質問題に発展してしまうこともあるのです。

管理的側面

実際の開発では、最終的にどのようなかたちにするか決まらないまま開発をスタートさせなければならない部分があったり、途中で変更が発生することがあるのは避けられないものです。このような事項を管理し、最終的な製品では漏れなく反映できる仕組みや、開発運営のやり方がまずい場合にも品質が低下する要因となります。

人的側面

ソフトウェアの開発は、いまだに人手に頼った部分の多い作業です。ソフトウェアの開発にはその場面、場面で必要な技術があり開発を担当する人はその技術を確実に使いこなすというスキルも必須となります。鋸が上手に使えない大工さんの作った家に住みたいと思うでしょうか。

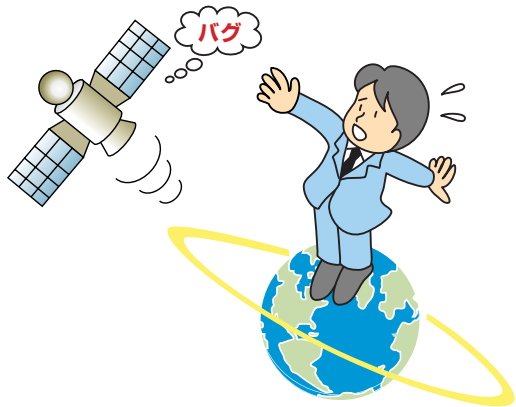
このようにソフトウェアの開発には、多くの要素が複雑にからみあっています。ソフトウェアの品質とはこういった要素すべての上に成り立っているものですから、どこかの要素に不都合があった場合は、その要素に関連する品質が低下する可能性が高くなるのです。

▶▶2-4 組込みソフトウェア特有の品質と取組み

組込み機器には、その特性から特有の品質もあり、そのための取組みも考慮しておく必要があります。

(1) 改修が困難

組込み機器は、一度製造元の手を離れるとソフトウェアの更新を行うことが困難なのが一般的です。また民生品では、不特定の多数に販売しているため、たとえ軽微な不具合でも改修しようとすると大掛かりな作業が必要となります。そのため、組込み機器は製品として出荷された時点で万全の品質を持っていることが要求されます。



(2) 安全性

例えば、自動車を初めとする交通機関、直接人の命に関わる医療機器等の場合、重大なバグがあり誤動作すると人命を損なうという最悪のことも起こりかねません。このような分野の製品は特に誤動作しない、停止しない等の品質が要求されます。

十分な設計やテストを実施したうえ、さらに想定されないケースでの異常発生を考慮に入れておかなければなりません。航空機等の話題でよく聞かれるフェールセーフ等がこれに当たります。ソフトウェアのフェールセーフとしてはウォッチドッグタイマを利用した異常検出からのプログラム再開などが良く知られています。

(3) ユーザビリティ

ユーザビリティとは、使用者からみた製品の使い勝手の良し悪しのことをいいます。組込み機器の場合、それぞれ独自の操作性を持つため、特に重要な品質となります。機能や性能は同じでも、使い勝手の良し悪しでその製品に対する評価は大きく異なってきます。ユーザビリティ向上策として、どういった操作性が望まれているか等を開発の上流で明確にするため要求工学の積極的な利用も始まっています。

(4) サービス

民生品の場合、その使用対象者は不特定の多数となります。通常は考えられないような使われ方をされたりする場合があり、開発時には思いもよらない不具合が見つかる場合もあります。また、特殊用途向けや産業機器は使用者が限定される代わりに、運用中の不具合は多大な影響を引きおこします。万一、不具合が発生した場合には、早急な対応が必要となります。

こういったサービスも製品の品質ということができます。ソフトウェアとしては、実際に製品が使われた際のいろいろな可能性を考慮しておく必要があり、万一、不具合が発生した時には、迅速な対応（原因究明・不具合改修）ができるよう工夫しておく必要があります。

(5) ネットワークへの対応・対応分野の拡大

セキュリティも重要な品質となりつつあります。

通信機器はもとより、現代の組込み機器はネットワークに接続することにより、新たなサービス形態を持ちつつあります。従来、限定的な範囲で使用されていた組込み機器が、ネットワーク化されていくことにより、セキュリティ面の対応も考慮しなければならなくなっています。これらは、もともとネットワークに繋がっていたPCでは広く知られており、それなりの対策もされています。組込み機器がネットワークに接続されることが当たり前となっていく状況の中、PCに比べての流通数の大きさや組込み機器であるがゆえの新たな問題も考えられるため、今後の課題としてしっかりと取り組んでいく必要があります。

7つの設計原理

今から20年以上も前、富士通の方が発表した論文の中で7つの設計原理というソフトウェア設計の規範が紹介されました。長年にわたり、高品質ソフトウェア開発のための行動規範として、この原理がエンジニアに用いられているそうです。シンプルな規範が実践的でもある好例だと思いますのでご紹介します。

単純原理
フィールドの二重定義のような高級（？）なテクニックは極力使わないようにしたり等々、単純にプログラミングするということ。あるいは、少々バカにされても、単純でとおすメンタリティを堅持したりすること。

一様原理
同じことは同じようにやることにこだわること。また、例外は設けないように固く決意すること。

対称原理
上があれば下、左があれば右、アクティブがあればパッシブというような対称性にこだわること。

階層原理
ものごとの主従関係、前後関係、本末関係などの階層関係を常に意識し、あるべき姿を求めるということ。

透過原理
フラグのON/OFFでプログラムの動きを制御したり、むやみに状態の数を増やしたりしないというようなこと。何事もスーッといくのが最高、その感じ。

明証原理
ロジックの明証性に敏感であること。少しでも不透明なロジックは証明しておくように努めること。そうしているうちに、面倒な証明などやっていられないので、自然と明証的で骨太なロジックのプログラムを書くようになる。

安全原理
必然性のないところや曖昧なところは、ちょっとルーズに、安全サイドで設計・作成しておくということ。

*日野克重（富士通）：『ソフトウェア障害の分析と予防』, ENGINEERS, Jan. 1985, PP6-10

3 ソフトウェア品質向上の考え方

▶▶3-1 ソフトウェア品質の向上

組込みソフトウェアに限らず、どのようなソフトウェアの開発でも、要求分析、設計、実装（コーディング）、テストといった作業を実施することで、目的のソフトウェアが作り上げられます。即ち、これらの作業一つ一つの積み重ねによって、ソフトウェアは作られ、その品質も決まってくるといっても過言ではありません。ソフトウェア開発の分野では、こうした一連の開発作業を開発プロセスと呼びます。 ソフトウェアの品質を向上させるためには、開発プロセス中の適切なタイミングで品質の作りこみや品質確認などを行う必要があります。

「品質作りこみ」とは

ソフトウェアなどを作る際に、どれだけ品質を意識し、開発するソフトウェアに高い品質を持たせるかという工夫をすることをいいます。

「品質確認」とは

作成したソフトウェアなどが、どの程度の品質を持っているか、あるいは品質面で不都合がないかを確認することをいいます。

▶▶3-2 品質作りこみ

品質作りこみのタイミング

ソフトウェア開発プロセスの中で、品質を作りこんでいくタイミングは大きく分けると2つあります。

①開発上流（要求や設計）段階での品質作りこみ

②開発下流（実装）段階での品質作りこみ

より品質の高いソフトウェアを作ろうとする場合、このどちらか一方に頼るのではなく、開発の上流、下流それぞれで品質作りこみを意識し、実践する必要があります。

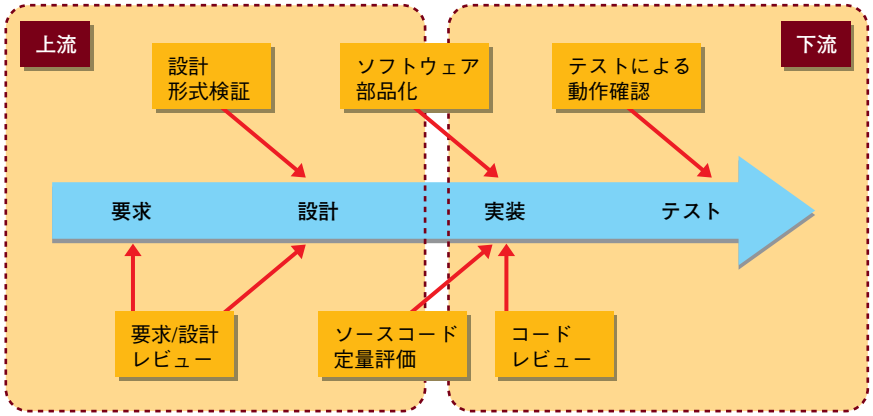


図3.1　品質作りこみのタイミング

開発上流での品質作りこみ

(a) 組込みソフトウェア開発における上流工程の特徴

ソフトウェアの不具合は、開発上流の要求や設計などの不備や誤りが原因になっていることが少なくありません。特に、組込みソフトウェアの場合、システムの動作は単純にソフトウェアの世界だけで考えたり決めたりすることは難しく、関連するハードウェアや組込みシステムが動作する環境などの影響を受けやすいといった特徴を持っています。このため、組込みソフトウェアでは往々にして要求や設計など開発上流段階での不備などが発生しやすいという特徴を持っています。

(b) 開発上流の課題点

ここでは、開発上流工程のもつ課題点をもう少し詳しく考えてみたいと思います。

- 曖昧な要求・不十分な要求

  どのようなソフトウェアを作るかという要求事項が曖昧だったり、不十分であったりすると、開発途中で要求事項などが変更したり、そのまま開発を進めると思わぬ不具合が発生することも少なくありません。

- 組込みソフトウェア特性の考慮の抜け

  組込みソフトウェアの場合、時間応答性や環境適合性など前述の非機能的な側面の要求（非機能要求）を十分に検討しておかないと、実環境での動作が保障されないといった不具合なども誘発しかねません。

- 貧弱なソフトウェアアーキテクチャ

  ソフトウェアアーキテクチャは、ソフトウェアの機能を実現するためのメカニズムを持ち、その振る舞いなどを実現するものです。ソフトウェアがきちんとしたアーキテクチャに基づいて作られていないと、ソフトウェアが再利用しにくかったり、ソフトウェアで実現すべきリアルタイム性、リアクティブ性など、組込みシステムにとって命とも言うべき特性が実現できなかったりと、ソフトウェア品質にとっても致命傷になりかねません。

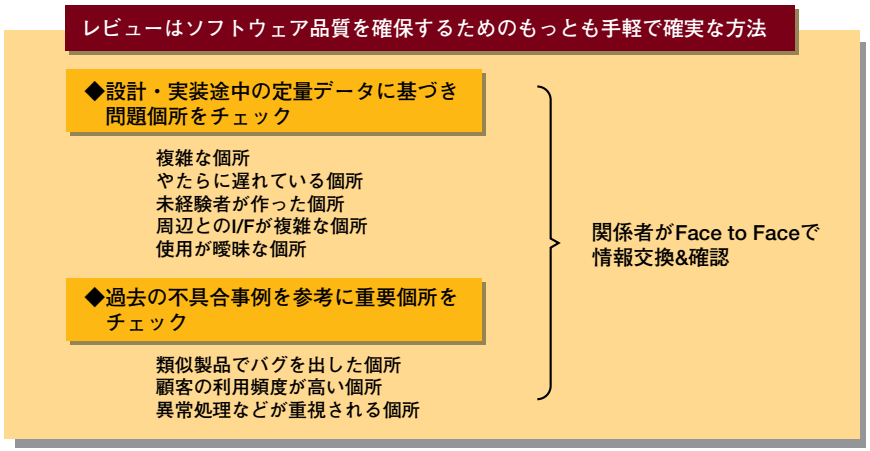
(c) 設計レビュー

前述のような品質面での課題を解決し、開発上流での品質作りこみを実現する手段としてレビューがあります。「レビュー」は、開発作業の中で作成した成果物（要求仕様書や設計書など）の内容をチェックし、その妥

当性を確認し、必要に応じて品質向上のためのきっかけとなる作業です。設計書に対するレビューは「設計レビュー」と呼びます。設計レビューの実施で特に重要な点は、

- 設計結果などは設計書などのドキュメントの形にして可視化しておく。
- 設計レビューでは、設計を担当した技術者以外に客観的にその内容を確認できる経験者などをレビューアとして参加させる。
- レビューにおける個々の指摘事項は、その対応策などを立案し確実に対応する。

といった工夫が必要になります。ソフトウェア品質については、出来るだけ開発の上流段階で確実なものにしておいたほうがコスト的に有利といわれています。その意味からも設計レビューは設計品質を確実にするための手段としてぜひ実施することをお勧めします。



開発下流（実装）での品質作りこみ

(a) 開発下流での品質作りこみの必要性

ソフトウェア開発の下流工程はコーディングなどによって、実際に動くソフトウェアを記述し（実装）、その動作確認をする（テスト）する作業が中心になります。中でもソースコードは、実際に動作するソフトウェアを実現する中心的な役割を持ちます。ソースコードの不具合は、製品システムの不具合に直結します。また、作成したソースコードが再利用しやすいかどうかなどは、その後のソフトウェアの生産性などに大きな影響を及ぼします。このように、ソフトウェアの実装段階で品質を作り込むことはきわめて重要です。

(b) ソースコードの品質

ソースコードの品質とはどのようなものでしょうか？ 一般的にバグがないことなどが真っ先に思い浮かびますが、必ずしもそれだけではありません。即ち、ソースコードの品質についても先に述べたソフトウェア品質特性と同様にいくつかの視点があります。例えば、

- ソースコードの保守のしやすさ（保守性）
- ソースコードの移植のしやすさ（移植性）
- ソースコードの誤りのなさ（信頼性）

などいくつかの視点を挙げることができます。

ソースコード品質の向上の方法

ソースコードの品質を向上させようとする場合には、まず「ソースコード作成」という作業を詳細に分解して考える必要があります。

ソースコード記述する際の工夫

ソースコードを書く際にいかに不具合を混入させないか、あるいは、保守性や移植性などをどの程度考慮してコードを書くか、あるいはどのように記述するかなどが重要になります。

記述したソースコードを確認する際の工夫

ソースコードは、プログラミング言語を利用して、コンピュータに実行させる処理を記述するものです。「言語」を利用して記述するということは、時として記述誤りなどが発生する場合があります。ソースコードの品質向上には、こうした記述誤りや保守性、移植性などについて適切に記述されているかどうかを確認し、品質面でのブラッシュアップを施す作業が欠かせません。こうしたソースコードの品質確認などについては、開発上流のところで記述した「レビュー」と同様に、ソースコードに対して行う「コードレビュー」などが有効です。

▶▶3-3 品質確認

品質確認の意義

ソフトウェアを実際の製品に搭載し、市場に提供する際には、不具合などが含まれないよう、十分に確認しておく必要があります。作成したソフトウェアが当初の仕様どおりに出来ているかどうかを確認する作業はテストと呼ばれます。ソフトウェアのテストは、開発の後半で実施されるため、ソフトウェア品質を向上させる最後の機会です。

テスト

通常、ソフトウェアのテストは単体テスト、結合テスト、システムテストの3段階で実施されます（図3.2）。

(a) 単体・結合テスト

組込みソフトウェアの場合、単体テストや結合テストは、システムを構成する部分を順番に動作確認しながら、組み上げていく作業がこれにあたります。組込みソフトウェアの場合、実際のソフトウェアは製品に組み込

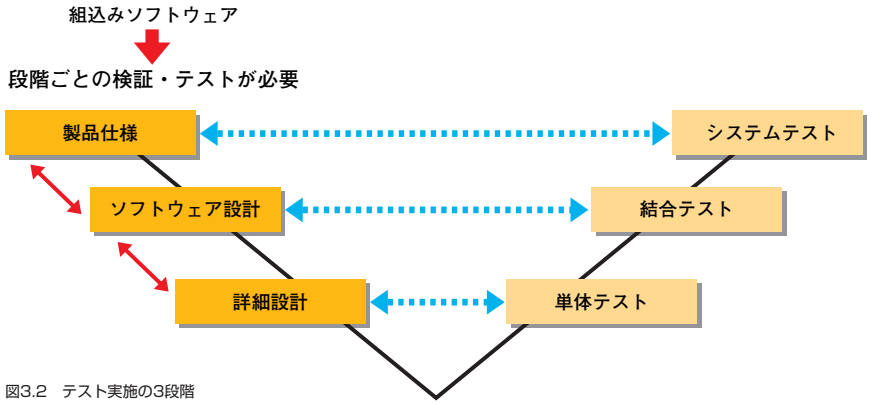


図3.2　テスト実施の3段階

まれたマイコンの上で動作するため、単体テストや結合テストは、開発環境上のシミュレータや、ICE＋試作基板などを利用して部分動作を実現して、テストを実施します。

(b) システムテスト

一方、システムテストでは、ハードウェアとソフトウェアを結合し、システムとしての動作の確認を行います。組込みソフトウェアの場合、組込みシステムが動作する周囲の環境などをテストの際に、実環境と同様に整えられるかどうかで、テストの精度や正確性が大きく異なってきます。

いずれにせよ、組込みソフトウェアのテストのポイントは、

①どのようなテスト項目やテストデータでテストするかを、システムの特性やユーザの利用シーンなども想定して設計する

②テスト実行するためのテスト環境については、実機ハードウェアの手配やソフトウェアの動作環境と同等の条件設定などテスト着手の前段階で様々な準備が必要となる

といった点であることを理解しておく必要があります。

仕様検証

ソフトウェアのテストは、品質を維持し向上させるための手段として極めて有力な手段です。ただし、テストは開発の最終段階で行うため、不具合などが見つかった場合、修正のコストはきわめて大きくなります。このため、テストよりも前の段階で品質に問題ないことを確実に確認しておくほうがコスト的に有利です。

仕様段階でその正しさを確認する手法を仕様検証と呼びます。組込みソフトウェアでは、ソフトウェアの動作を状態遷移モデルなどで記述し、それをモデル検査と呼ばれる手法などで検証する場合もあります。

コーディングの良形視

「良形視」とは認知心理学の用語で、「ものには一番見えやすい形」があるということです。

コーディングに良形視の発想を取り入れようということは、間違いやすさよりも人間の特性的にレビューでの見つけ難い書き方をしないようにするために、心理学的視点をうまく取り入れよう。ということです。例えば、if(a=b){　は多くの人がこれでバグを何度か出しているのではないでしょうか？　また、100行のコードがコメントアウトしてある中で1行だけ生きているコードがあると、普通の人はそれが生きているとは思いません。こういった錯視を防ぎ、レビューでバグを見つけやすいコーディングをしようというのが「良形視」の発想です。

エッシャーというオランダの芸術家がいます。エッシャーは正則分割や対称性、二次元と三次元の交差、遠近法や透視法を逆手に取った不可能な構造など、緻密に計算された幾何学的なイメージの不可思議な世界を精巧な版画技術によって作り出す芸術家です。エッシャーの版画は、人間が錯視を招く典型として、しばしば認知心理学の世界で用いられます。エッシャー自身は数学者でもあり、これらの絵は線形的に緻密に計算されたトリッキーな絵なのですが、コーディングの場面ではトリッキーな処理はバグやレビューでのチェック漏れを招き大敵です。コーディングの良形視とは、処理をシンプルにかつ自然に「やりたいことがストレートに」伝わるコードの記述です。

4 ソースコード品質向上のための技術

▶▶4-1 ソースコードの標準化

(a) ソースコードの標準化が必要な理由

ソースコードは、C言語などのプログラミング言語を利用して記述されます。プログラミング言語はいわゆる、言語のひとつであることに変わりありません。ということは、あるひとつのことを表現するにも、その言語を使う使い手によって様々な言い回し（表現）をとることができます。

例えば、「単純に1から10までの整数を合計する処理」ひとつとっても、① for文を利用した繰り返し処理で実現する、② while文を利用した繰り返し処理で実現する、③ if文とgoto文を利用して実現する……など、様々な方法を考えることができます。このうち、①や②の方法をとれば、ソースコードの構造は比較的簡潔なものとなりますが、③のようにgoto文などを利用すると論理構造はやや複雑になってしまいます。goto文で戻る先の指定を誤ったりすると、思わぬ不具合を誘発してしまいます。このような単純な例でも、ソースコードの書き方次第では、プログラムの複雑性などに影響することがわかるかと思います。

この事例でもわかるように、プログラミング言語という明確に文法が決められている言語を利用する場合でも、開発する技術者個々人の癖や経験によって、ソースコードの記述には様々なバリエーションが生まれてしまいます。企業活動の一環として開発されるソフトウェアが開発者個人の能力などによって、バリエーションが増えたり、品質にばらつきが生まれてしまうのはできれば避けたいものです。

(b) ソースコードの標準化のための方法

最近の組込みソフトウェアのほとんどは、複数人の技術者で開発されています。こうした場合、開発者が10人いれば10通りのソースコードの書き方が存在することになります。この状態を放置すると、前述したように、ソースコードの記述様式や品質のレベルはバラバラになってしまいます。こうした事態を避けるためには、関係者間で、ソースコードの書き方などのルールを決め、守っていくことが近道です。このようなソースコードの書き方に関するルールをコーディング規約と呼びます。コーディング規約については次章に詳細に説明します。

▶▶4-2 ソースコードの再利用

(a) ソフトウェアの再利用とは

ソフトウェアの品質を向上させるための最良の方法のひとつは、ソフトウェア資産を再利用することです。即ち、品質が予め保証されたソフトウェア資産（コンポーネント）を利用すれば、少なくともその部分についての品質は保証されます。ソフトウェア資産の再利用については様々な粒度が考えられます。小さいものではC言語の関数レベルの極めて小さい単位の再利用から、大きいものではシステムのあるまとまった機能を実現するサブシステムレベルの再利用まで様々です。

(b) ソースコード再利用の方法

ソフトウェア資産の再利用を考える場合、再利用する対象物も様々です。即ち、再利用の対象物としては、ソースコード、設計情報、ソフトウェア・アーキテクチャやテストケースとテストデータなどが対象として考えられます。この中でも最もよく再利用されるものがソースコードです。

ソースコードを再利用する場合には、

①再利用する範囲を特定

②再利用可能な形にソースコードを整形

して再利用する必要があります。通常、ソースコードの一部を単純に切り出した場合、その部分がさらに別の関数を呼び出していたり、その部分で利用する変数などの宣言や値の設定を別の部分で行ったりしていることが考えられます。このような場合、ソースコードの一部を単純に切り出しただけでは使い物になりません。このため、ソースコードの一部を再利用する際には、再利用する部分を明確にし、再利用部分とそれ以外の部分のインタフェースを確認し、整理してから利用する必要があります。このため、より効果的なソースコード再利用を目指すのであれば、予めどの部分を再利用するかといった戦略を考え、設計段階で出来るだけコンポーネントとして独立性の高い形で設計しておくことが求められます。

(c) ソースコード再利用の弊害

ソースコードを作成中に、その一部をコピー＆ペーストして、他の部分を作成することがあります。このようにして作られたソースコード断片をコードクローンといいます。

安易にコードクローンを利用すると、図に示すように、

①オリジナルのコードに不具合が発生した際に、コードクローンの部分まで修正を施す必要が発生する（不具合の拡散）

②コードクローンの増殖によって、コードサイズが増大し、コードのモジュラリティが低下する

などの弊害が発生し、結果としてソースコードレベルの保守性を低下させる可能性が増大します。このため、品質向上の視点からは極力コードクローンは作らないように心がける必要があります。

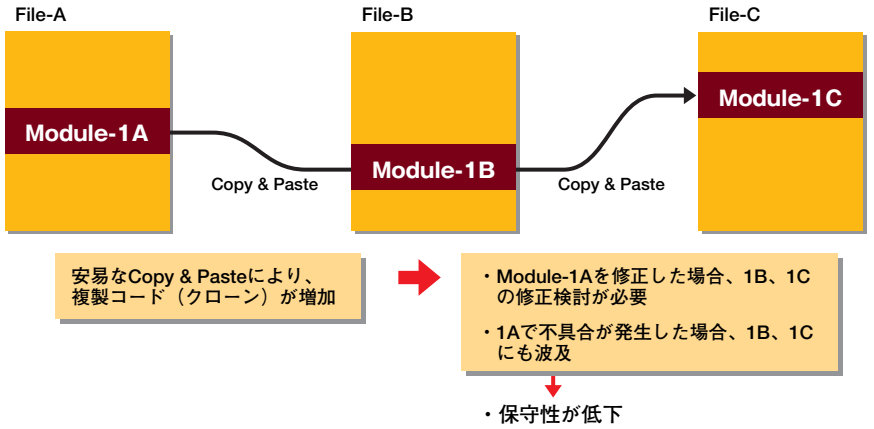


図4.1　クローンコードの弊害

(d) リファクタリング

一般に、ソースコードを作成し、コンパイル＆ビルドを行って動くようになると、とりあえず、コーディング作業は完了になります。しかし、動作可能なコードであっても、それは動作に支障がないだけであって、ソースコードの品質が十分であるかどうかはわかりません。場合によっては、ソースコード内部の構造がスパゲッティ構造になっていたり、あるいは、コードクローンなどを利用してやたらに冗長になって保守性や移植性が著しく低下している場合なども考えられます。

このようなソースコードの内部品質を向上させるために、ソフトウェアで実現する機能を変えることなく、その内部構造を洗練する作業をリファクタリングと呼びます。リファクタリングは、ソースコードの保守性や移植性などを向上させるために、ソースコードをいくつかの観点からチェックし、その内部構造を作り変えます。例えば、構造的にやたらに規模の大きなモジュールやクラスを分割して、多少規模の小さなモジュールやクラス、あるいは共通モジュールなどを利用して洗練します。

外部から見たときの振る舞いを保ちつつ、理解や修正が簡単になるように、ソフトウェアの内部を変化させること

リファクタリングの着眼点

1. 巨大なクラス
2. 長すぎるメソッド
3. 重複したコード
4. 不適切な関係
5. 変更の発散
6. 変更の分散
7. 多すぎる引数
8. スイッチ文

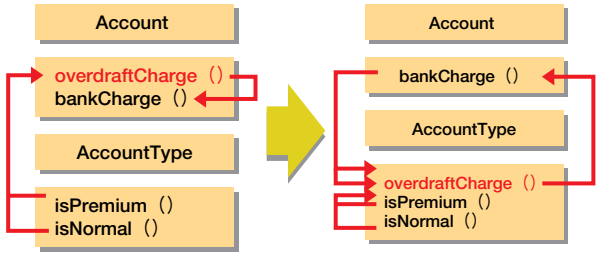


図4.2　リファクタリング（Refactoring）

4-3 ソースコードの定量評価とコードレビュー

(a) コードレビューとは

ソースコードの品質を向上させる手段のひとつとしてコードレビューがあります。コードレビューは、ソースコードの作成者やそれ以外の第三者が、作成したソースコードに目を通し確認する作業です。コードレビューでは、対象とするソースコードをどのような視点でチェックするかによって、その効果が大きく異なります。

視点1：ソースコードの保守のしやすさの視点
　　第三者が保守する際の理解しやすさなど
視点2：ソースコードの移植のしやすさの視点
　　マイコンの命令セットや周辺デバイスとの依存度合いなど
視点3：ソースコードの信頼性・安全性の視点
　　ソースコード内の論理的な誤り、意味的な誤りや安全性の視点からの確認

いずれにせよ、対象となるソフトウェアにどのような特性が必要とされるかによって、コードレビューの視点は変わってきます。

(b) ソースコードの定量評価

最近の組込みソフトウェアは高機能化が進み、これに伴いソースコードのサイズもきわめて大きくなってきています。こうした規模の大きなソースコードを、先頭から末尾までコードレビューでチェックするのは多大な労力を必要とします。このため、現実のコードレビューでは、ソースコードの隅から隅までチェックすることはせず、重要なところのみレビューする形がほとんどです。しかし、こうした方法ではソースコードのどの部分をレビューするかによって、完成するソフトウェアの品質はバラけてしまいます。このため、ソースコードの中のどの部分が問題か、予め当たりをつけてレビューする方式などが考えられます。

(c) 品質メトリクスの利用

規模の大きなソースコードのどのあたりに問題があるか、あるいは、ソースコード全体としての品質がどの程度であるかを客観的に評価する必要があります。特に、作成したソフトウェアが品質的にどの程度の水準であるかを把握することは、品質改善の点からも重要です。こうしたソフトウェア品質を客観的に評価するために、品質を定量的に評価する品質メトリクスを利用することができます。品質メトリクスには、ソフトウェアを構成する各部分ごとの論理的な複雑さや保守性など、いくつかの代表的な指標があります。ソースコードを評価するための品質メトリクスについては、対象とするソースコードを解析して、これらの指標値を計測するツールなどが公開されているものもあり、こうした品質メトリクス計測ツールなどを利用するのも品質向上の点からは有効です。

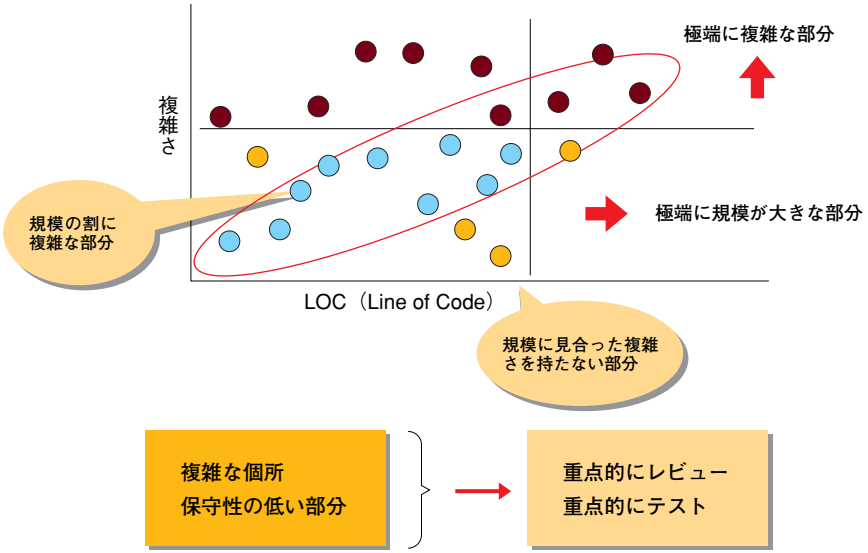


図4.3　品質メトリクス

コーディング作法（1）

「救いようのないスパゲッティ・コードだ！お手上げだ！」開発の現場で、誰かが作ったソースコードに手を加えようした時、時々聞かれる言葉ではないでしょうか？

でも、ちょっと待ってください。本当にそれはひどいスパゲッティ・コードなのでしょうか？

それは共通の決めごとがなく、ある人にとってはわかりやすいソースコードでも、ある人が読むとわかり難いソースコードだからではないでしょうか。

このような状況を克服するにはどうしたら良いでしょうか？　そのためには、コーディングの共通の作法を根付かせるべきではないでしょうか？

「自分がわかることと、皆がわかるということは全然違うこと」です。

ソースコードの一定の品質レベルを維持するためには、共通のコーディングの作法をもってプログラムを書くことが必要ではないでしょうか。

こうして共通の作法をもって書かれたコードであれば、他人の書いたソースコードを読むことの苦痛からも解放されるのではないでしょうか？

5　コーディング作法/規約とは

▶▶5-1 コーディング作法/規約とは

一般にいわれる、コーディング規約、コーディングルール、コーディングスタイル、コーディング作法、コーディング基準などのソースコードを取り巻くこれらの規約やルールは、これまで数多くのものが提案されてきました。これらの名前は沢山ありますが、ほとんどのものに関しては、ソースコードを記述する際にどのような点に注意すべきかを整理したノウハウ集のことを指しています（以下では総称してコーディング規約と記します）。コーディング規約のほとんどは、いままでに起こしたソースコードにまつわる問題の原因を参考に、「このような書き方をするとトラブルを引き起こす」という経験をルールや規約の形に整理したものなのです。また、コーディング規約では、個々の会社の経験に基づいて決められたルール以外にも、ドメインや製品などの特性に基づいて決められている場合もあります。このようなルールの制定経緯を考えると、これらのコーディング規約を守らなかった場合、ソースコード上の不具合を引き起こす可能性があると考えることが出来るでしょう。

▶▶5-2 利用可能な、コーディング作法/規約の紹介

本項では、現在世界中で数多く公開されているコーディング規約の紹介をします。世の中には、国際的に定められている規格のような位置付けのコーディング規約や、自動車のような特定の製品やドメインを対象にしたコーディング規約、あるいは、Javaなどの特定言語利用者を対象にしたコーディング規約など、様々なコーディング規約があります。これらの規約は、インターネット上に公開されているものや、購入可能なものもあり

ます。

一方、個々の会社あるいは部門毎のコーディング規約なども存在し、利用されている場合も少なくないのです。表5.1は、SECで行われている調査の途中経過をまとめたもので、現在、公開されているコーディング規約の代表的なものの特徴などを整理したものです。詳細は割愛しますが、いずれのコーディング規約も対象とする言語ごとに、その注意すべき点をルールの形で整理してあります。

また、MISRA-Cのように高信頼性を要求されるソフトウェアを主たる対象にしているものなど、対象とする分野や言語などを限定しているものも見られます。一方、コーディング規約を守る側の開発者からすると、規約が多すぎると、全てを守ることは非常に困難になり、また、少なすぎると、品質の確保につながらない場合があります。このようなことを考慮すると、公開してあるコーディング規約をそのまま全て利用することが製品の品質確保に適したコーディング規約とならない場合もあるかもしれません。このため、これらのコーディング規約を利用する際には、それぞれのコーディング規約がどのような特徴を持っているか、どのような開発を対象にしているかなどを理解して利用することが求められます。

基準名称	目的	開発年度	特色
GNU Coding Standards	クリーンで矛盾のないインストールが容易なGNUシステムを作成すること	2005/01	ドキュメントやソフトウェアのインストール、Makefileの規約まで論じられている
Linux kernel coding style	Linuxカーネルのソースコードの好ましい書き方	2004/02	Linux Kernelのソースに付属〈Linux kernelのソース〉/Documentation/CodingStyle
Java Code Conventions (SUN Microsystems, Inc.)	Java言語の標準的なコーディングスタイルの取り決め	1999/04	SUN Microsystems,Inc.が作成したコーディングスタイルの取り決め
Programming in C++, Rules and Recommendations (Ellemtel)	C++プログラミングの1つのスタイルを定義する	1992	AT&TのC++言語システムに基づいている
Recommended C Style and Coding Standards (Indian Hill)	移植性を高め、メンテナンスの手数を減らし、プログラムをわかりやすくする	1990/06	Cプログラムのために推奨された標準のコーディング
C STYLE GUIDE (SOFTWARE ENGINEERING LABORATORY SERIES SEL-94-003)	Cプログラムを書くため、ソフトウェア工学研究室（SEL）が推奨するスタイルを記述、ここで「良いコード」とは、組織化した、読みやすい、理解しやすい、維持しやすい、効率的なコードと定義される	1994/08	ソフトウェア工学原則が論議されて示され、コード例題がよい慣例を示すために提供される
MISRA-C	車載用ソフトウェアを対象としたC言語プログラミングガイドライン	1998	ソフトウェア開発プロセス全体についての開発ガイドラインがあり、この流れの上での、C言語でプログラムを作成する工程についての開発ガイドライン。127のコーディングルールを規定している
Safer-C（書籍）	安全関連ワークでのC言語の利用ガイダンス	1995	ソフトウェア品質について述べられている
Effective C++（書籍）等	C++、STLを正しく使う	1995～2001	オブジェクト指向デザイン、クラス・デザインにも言及
Coding Style (Embedded@IITD)	Department of Computer Science & Engineering,IIT Delhiというグループのコーディング規約	2004	ローカルな規約である
SPARK 95	完成度の高いアプリケーション・ソフトウェアを書くための高水準プログラミング言語	2005/01	Ada言語のサブセット。言語のみでなくツールセットを含む
Writing Robust Java Code (The Ambysoft Inc. Coding Standards for Java)	堅実なJavaコードを書く	2000/01	関連書「"The Elements of Java Style"Vermeulen, Ambler, Bumgarner, Metz, Misfeldt,Shur, & Thompson」では、より広い範囲のガイドラインを提示している
Mozilla コーディングスタイルガイド	Mozillaコードベースで使われる基本スタイルとパターン	2002/10	Mozillaコードベースで使われる基本スタイルとパターン
High Integrity C++ Coding Standard Manual（PRQA）	Programming Research Ltdの高品質C++ソフトウェアを開発するためのコーディング基準	2004/05	ルール集。ルールとガイドラインに分類される
Javaコーディング規約2004（電通国際情報サービス）	Java言語のコーディング規約を定義する	2004/09	『超図解 Javaルールブック』は関連図書
Javaコーティング標準（オブジェクト倶楽部）	Java言語のコーディング規約を定義する	2002/11	VB.NET、C++、C#に応用されている

*上記の図は、SEC調査の仮集計結果です。最終結果とは異なる場合があります。

表5.1　コーディング規約の集計（仮）

▶▶5-3 コーディング作法/規約の上手な利用方法

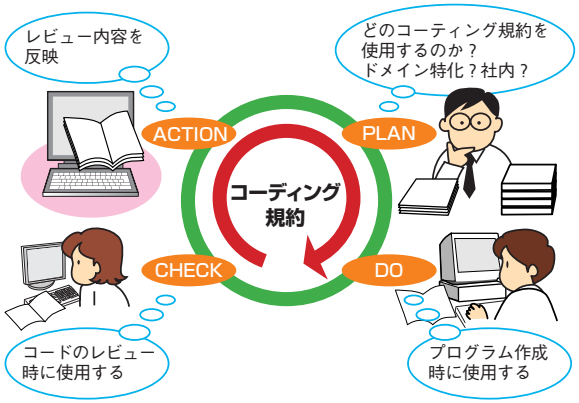
本項では、コーディング規約を実際の開発現場で利用する場合その利用方法として次の2つを紹介します。

1. ソースコード記述する際に利用

技術者がソースコードを書く際の参考として利用するやり方です。自分の所属する会社で定められたコーディング規約を守ることによって、ソースコード作成時の個人差を少なくすることが出来ます。また、その会社で定められたソフトウェア品質を確保し、ソースコードの信頼性や保守性などを向上させることが出来ます。これは、不具合を減らすという意味では非常に重要な利用の仕方と考えることが出来ます。

2. ソースコードレビュー時の基準として利用

小規模なソフトウェアだけでなく、大規模なソフトウェア開発では、一般にコードレビューやインスペクションを行う時にレビュー基準が必要になります。これは、レビューの際に、ソースコードのどこを見るべきか、どこをどのような観点でチェックすべきかといった視点を予め準備しておく必要があるからです。コーディング規約をこうしたソースコードレビューを行う際の基準として利用することも有効な利用方法のひとつだと考えられます。レビュー基準としてコーディング規約を利用することで、ソースコード品質の安定、向上を実現することも可能になります。



6 組込みソフトウェア用 C言語コーディング作法の概要

IPA／SECでは経済産業省組込みソフトウェア開発力強化推進委員会と連携して、企業やプロジェクトでコーディング規約を作成・運用する人に対して、コーディング規約作成の支援を目的としたコーディング作法ガイドを作成しています。このガイドの特徴は、コーディング規約を『品質を保つために守るべきコードの書き方』と考え、ルールの基本概念を作法としてまとめたことです。作法は「JIS X 0129-1 ソフトウェア製品の品質 第1部：品質モデル」に準拠した品質概念をもとに、作法概要、作法詳細に分類・階層化しています。さらに、それぞれの作法にC言語に対応したルールをその必要性とともに提示しています。この作法とルールにより、意義・必要性を理解できる、実用的な「コーディング規約」が容易に作成できることを目標としています。このガイドが想定している対象者と利用方法および期待効果は以下の通りです。

1. コーディング規約を作成する人

このガイドにより、容易に新規のコーディング規約が作成できます。または、既にあるコーディング規約の確認、整理ができます。

2. プログラマやプログラムレビューをする人

このガイドの作法・ルールを理解・修得することによって、信頼性の高い、保守しやすいコードの作成が無理なくできるようになります。

このガイドでは、作法、ルールを信頼性・保守性・移植性・効率性の4つの品質特性に関連づけて分類・整理しています。このガイドにおける作

法、ルールの意味は次のとおりです

- 作法　：ソースコードの品質を保つための慣習、実装の考え方であり、個々のルールの基本概念を示す。作法概要、作法詳細に階層化して示している。
- ルール：守らなければならない、具体的な一つ一つの決めごとであり、コーディング規約を構成する。このガイドではリファレンスとして示している。なお、ルールの集まりもルールと呼ぶことがある。

作法、ルールの多くは、複数の品質特性と関連しますが、最も関連の強い特性に分類しています。品質特性と関連づけることにより、各作法がどのような品質に強く影響するかを理解できるようにしています。

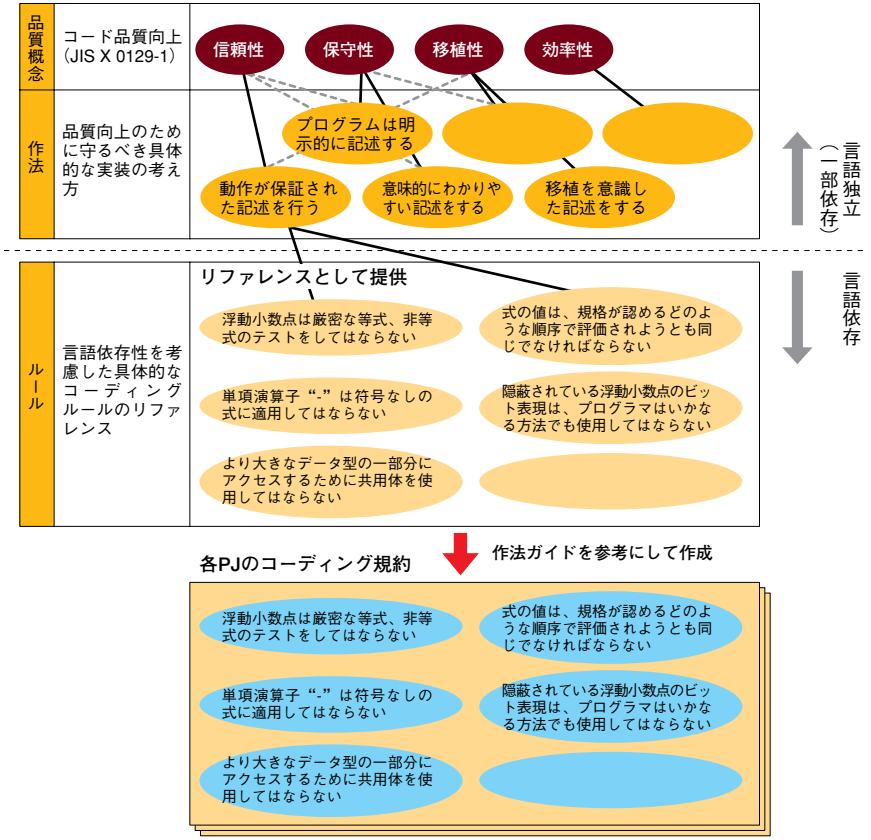


図　品質概念・作法・ルールの関係

コーディング作法（2）

「畳の縁や敷居を踏んではいけない」という行儀の作法がありますが、なぜ畳の縁や敷居を踏んではいけないのでしょうか？ これは、古く武士の習慣に由来しているそうです。忍者が武士をねらう時、床の下に忍びこみ、床から光が漏れる所、つまり畳と畳の間、あるいは畳と敷居のすき間に刃を上向きにして待ち構え、その光が消えたとき（その上を武士が踏み付けた瞬間）を狙って刃を突き刺したと言われています。

そのような命の落とし方は武士として恥ずかしいことだということから、畳の縁や敷居を踏んではいけないことが和室でのマナーとなったのだそうです。

SEC では行儀の作法同様に危険な言語仕様を避け、安全で保守性や移植性の高いソースコードを作成するためのノウハウを「コーディング作法ガイド」としてまとめています。このガイドでは、個々の作法の裏にある意味を解説しています。意味もわからず作法を守るのではなく、作法の意味を理解していくことがスキルアップのためにも重要です。

7 おわりに

我が国の工業製品は、その品質の高さを武器に国際間の競争を勝ち抜いてきました。それは、我々が開発している組込みソフトウェアの領域においても同様です。しかし、本冊子中でも述べたように、昨今の組込みソフトウェアを利用した機能増大などの結果、その品質をこれまでどおりに維持していくことは極めて難しくなりつつあります。

このような状況を打開するひとつの方策として、ソフトウェア工学に根ざした開発技術や品質向上技術の実践が考えられます。ソフトウェア品質向上のための技術にも、日ごろ我々の日常の開発業務中でのちょっとした工夫で対応できるレベルから、高価なツールや高いスキルの技術でなければ実践できない技術まで、様々なものが混在しています。その意味では、本冊子で紹介した内容は、組込みソフトウェアの品質向上のために、身近にかつ手軽に出来る工夫や個々の技術者の品質マインドなどに関係しています。ぜひ、組込みソフトウェア開発を手がける前の 15 分ほど、手にとっていただき、開発の実践の中での参考にしていただければと思います。

付 録 組込みソフトウェア用 C言語コーディング作法の例

作法	ルールNo.	ルール
信頼性	1.領域は初期化し大きさ、生存期間に気をつけて利用する	
領域は、初期化してから使用する。	1.1	自動変数は宣言時に初期化する。または使用する直前に初期値を代入する。
	1.2	const 型変数は、宣言時に初期化する。
初期化は過不足無いことがわかるように記述する。	1.3	要素数を指定した配列の初期化では、初期値の数は、指定した要素数と一致させる。
	1.4	列挙型（enum 型）のメンバの初期化は、定数を全く指定しない、全て指定する、または最初のメンバだけを指定する、のいずれかとする。【MISRA 9.3】
ポインタの演算を行う場合には、ポインタの指す範囲に気を付ける。	1.5	（1）ポインタへの整数の加減算（++、-- も含む）は使用せず、確保した領域への参照・代入は [ ] を用いる配列形式で行う。 （2）ポインタへの整数の加減算（++、-- も含む）は、ポインタが配列を指している場合だけとし、結果は、配列の範囲内を指していなければならない。
	1.6	ポインタ同士の減算は、同じ配列の要素を指すポインタにのみ使用する。【MISRA 17.2】
	1.7	ポインタ同士の比較は、同じ配列の要素、または同じ構造体のメンバを指すポインタにだけ使用する。【MISRA 17.3】
解放した領域をアクセスしない。	1.8	自動変数領域は関数終了後は使用しない。
	1.9	解放した領域にアクセスしない
信頼性	2.データは範囲、大きさ、内部表現に気を付けて利用する	
内部表現に依存しない比較を行う。	2.1	浮動小数点型変数は厳密な等式、非等式の比較はしない。【MISRA 13.3】
	2.2	浮動小数点型変数はループカウンタとして使用しない。【MISRA 13.4】
	2.3	構造体や共用体の比較に memcmp を使用しない。
真の値と等しいかどうかを調べてはならない。	2.4	真偽の結果を真として定義した値と等しいかどうかを調べてはいけない。判定する場合は、偽（0）と等しくないかどうかを調べる。
データ型を揃えた演算や比較を行う。	2.5	符号無し整数定数式は、結果の型で表現できる範囲内で記述する。【MISRA 12.11】
	2.6	条件演算子（?: 演算子）では、論理式は括弧で囲み、返り値は２つとも同じ型にする。
	2.7	ループカウンタとループ継続条件の比較に使用する変数は、同じ型にする。
型が異なる変数同士で演算する場合は、演算精度を考慮した記述にする。	2.8	演算の型と演算結果の代入先の型が異なる場合は、期待する演算精度の型へキャストしてから演算する。
	2.9	符号付きの式と符号無しの式の混在した算術演算、比較を行う場合は、期待する型に明示的にキャストする。
関数の、宣言と使用の整合性がとられていることを、コンパイラがチェックできる書き方にする。	2.10	関数呼出しと関数定義の前で関数のプロトタイプ宣言をする。
	2.11	引数を持たない関数は、引数型を void とする。【MISRA 16.5】

※組込みソフトウェア用 C言語コーディング作法ガイド0.8ドラフト版より

作法	ルールNo.	ルール
情報損失の危険のある演算は使用しない。使用する場合は問題のないことを明示する。	2.12	情報損失を起こす可能性のあるデータ型への代入（= 演算、関数呼び出しの実引数渡し、関数復帰）を行う場合は、問題がないことを確認し、問題がないことを明示するためにキャストを記述する。
	2.13	単項演算子 '-' は符号無しの式に使用しない。【MISRA 12.9】
	2.14	unsigned char 型 または unsigned short 型のデータをビット反転（~）したり、左シフト（<<）する場合、結果の型に明示的にキャストする。【MISRA 10.5】
	2.15	ビット列として使用するデータは、符号付き型ではなく、符号無し型で定義する。
対象データが表現可能な型を使用する。	2.16	ビットフィールドに使用する型は signed int と unsinged int だけとし、1 ビット幅のビットフィールドが必要な場合は singed int 型でなく、unsigned int 型を使用する。【MISRA 6.4, 6.5】
ポインタの型に気を付ける。	2.17	（1）ポインタ型は、他のポインタ型や整数型に変換してはならない。また、逆も行ってはならない。データへのポインタ型については、void* 型との変換は可とする。 （2）ポインタ型は、他のポインタ型に変換してはならない。ポインタ型のデータ幅以上の整数型との変換は可とする。また、データへのポインタ型については、void* 型との変換は可とする。 （3）ポインタ型は、ポインタ型のデータ幅以上の整数型との変換は可とする。関数へのポインタ型は、整数型以外の型との変換は不可とする。データへのポインタ型は、他のデータへのポインタ型との変換を可とする。
	2.18	const や volatile 修飾された領域を指すポインタに対し、const や volatile 修飾を取り除くキャストはしてはならない。【MISRA 11.5】
信頼性	3.異常値を考慮した書き方にする。	
領域の大きさを意識した書き方にする。	3.1	配列を順次にアクセスするループの継続条件には、配列の範囲内であるかの判定を入れる。
	3.2	（1）配列の extern 宣言の要素数は必ず指定する。 （2）要素数が省略された初期化付き配列定義に対応した配列の extern 宣言を除き配列の extern 宣言の要素数は必ず指定する。
	3.3	メモリ操作を行う関数を使用する場合、配列の範囲を超えてアクセスしない値を渡す。
実行時エラーになる可能性のある演算に対しては、エラーケースを迂回させるようにチェックする。	3.4	除算や剰余算の右辺式は、0 でないのを確認してから演算を行う。
	3.5	ポインタは、ナルポインタでないことを確認してからポインタの指す先を参照する。
関数が異常を返す可能性がある場合は、異常値をチェックする。	3.6	関数がエラー情報を返す場合、関数の復帰値を確認し、異常時の処理を行う。【MISRA 16.10】
関数のパラメタとして渡してはならない制限値がある場合は、制限値を渡さないように、チェックする。	3.7	関数に渡す引数に制限がある場合、関数呼び出しする前に、制限値でないことを確認してから関数呼び出しする。
多岐分岐処理において、所定の条件以外が発生した場合の処理を記述しておく。	3.8	if-else if 文は、最後に必ず else 節を置く。 通常、Else 条件が発生しないことがわかっている場合は、次のいずれかの記述とする。 （i）else 節には、例外発生時の処理を記述する。 （ii）else 節には、/* DO NOTHING */ というコメントを入れる。
	3.9	switch 文は、必ず default 節を置く。 通常、default 条件が発生しないことがわかっている場合は、次のいずれかの記述とする。 （i）default 節には、例外発生時の処理を記述する。 （ii）default 節には、/* DO NOTHING */ というコメントを入れる。

作法	ルールNo.	ルール
再帰呼び出しになるような関数呼び出しを行わない。	3.10	実行時の利用スタックサイズが予測できないため再帰呼び出し関数は使用しない。 【MISRA 16.2】
ループ制御の判定に気をつける。	3.11	ループカウンタの比較に等式、不等式は使用しない。(「<=、>=、<、>」を使用する)
保守性	4.他人が読むことを意識する	
意味のない式や演算や文を残さない。	4.1	存在する意味のある文だけを残す。
	4.2	使用しない関数、変数、引数、ラベルなどは宣言(定義)しない。
使用しない記述を残さない。	4.3	コード部は 'コメントアウト' してはならない。 【MISRA 2.4】
何もしない文が必要な場合は目立たせる。	4.4	意図的に何もしない文を記述しなければいけない場合はコメント、空になるマクロなどを利用し、目立たせる。
紛らわしい宣言をしない。	4.5	(1) 1 つの宣言文で宣言する変数は、1 つとする(複数宣言しない)。 (2) 同じような目的で使用する同じ型の自動変数は、1 つの宣言文で複数宣言してもよいが、初期化する変数と初期化をしない変数を混在させてはならない。
	4.6	適切な型を示す接尾語が使用できる定数記述には、接尾語をつけて記述する。long 型整数定数を示す接尾語は大文字の "L" のみ使用する。
	4.7	長い文字列リテラルを表現する場合には、文字列リテラル内で改行を使用せず、連続した文字列リテラルの連結を使用する
複雑なポインタ演算は使用しない。	4.8	2 段階を超えるポインタ指定は使用するべきではない。 【MISRA 17.5】
一般的には使用頻度が少なく、知らずに使用した場合に期待した結果にならない言語仕様は使用しない。	4.9	?? で始まる 3 文字以上の文字の並びは使用しない。
	4.10	(0 以外の) 8 進数定数を使用しない。 【MISRA 7.1】
暗黙の型宣言は使用しない。	4.11	関数や変数の定義や宣言では型を明示的に記述する。 【MISRA 8.2】
比較演算を省略しない。	4.12	式が真偽の値として取り扱われていない限り、ゼロとの比較を明示的にすべきである。 【MISRA 13.2】
領域は 1 つの利用目的に使用する。	4.13	目的毎に変数を用意する。
名前を再使用しない。	4.14	ある有効範囲と、その有効範囲を含むそれより外側の有効範囲で、変数や関数などに同じ名前を使ってはいけない。 【MISRA 5.2】
	4.15	標準ライブラリの関数名やマクロ名は再定義・再利用してはならない、また定義を解除してはならない。 【MISRA 20.1】
予約されている名前を使用しない。	4.16	下線で始まる名前(変数)は定義しない
1 つの副作用で、1 つの文とする。	4.17	(1) コンマ式は使用しない。 (2) コンマ式は for 文の初期化式や更新式でだけ、必要に応じて、使用する。
	4.18	1 つの文に、代入を複数記述しない。ただし、同じ値を複数の変数に代入する場合を除く。
	4.19	副作用が生じる場合は、条件演算子(?: 演算子)ではなく、if 文にする。
条件によって副作用の結果が異なる式を記述しない。	4.20	&& や || 演算子の右式は、副作用のない式を記述する。 【MISRA 12.4】
ループを制御する式と、その他の式は、分離して記述する。	4.21	for 文の 3 つの式には、ループ制御にかかわる処理だけを記述する。 【MISRA 13.5】

作法	ルールNo.	ルール
ループを制御する式と、その他の式は、分離して記述する。	4.22	for 文ループの中で for 文ループをカウントする変数はループの本文中で更新するべきではない。 【MISRA 13.6】
比較と代入は分けて書く。	4.23	真偽を求める式の中で代入演算子を使用しない。 【MISRA 13.1】
参照しかしない領域は明確にしておく。	4.24	参照しかしない領域は const であることを示す宣言を行う。
複数の実行単位から更新される領域は明確にしておく。	4.25	他の実行単位により更新される可能性のある変数には volatile を付ける。
アクセス範囲は局所化する。	4.26	1 つの関数内でのみ使用する変数は関数内で変数宣言する。
	4.27	同一ファイル内で定義された複数の関数からアクセスされる変数は、ファイルスコープで static 変数宣言する。
	4.28	同じファイルで定義した関数からのみ呼ばれる関数は、static 関数とする。
マクロ文の書き方を統一する。	4.29	マクロ定義は、定数、括弧で囲まれた式、型修飾子、記憶クラス指定子、do-while-0、及び 波括弧を使用した初期化にのみ使用する。 【MISRA 19.4】
行番号変更の機能は使用しない。	4.30	#line は、ツールによる自動生成以外では使用しない。
保守性	5.修正し間違えないような書き方にする	
ブロックは明確化し、省略しない。	5.1	if, else if, else, while, do, for, switch 文の本体はブロック化する。
構造化されたデータは、構造を明示する形で初期化する。	5.2	配列や構造体の初期化は、構造に従ってブロック化し、データを漏れなく記述する。ただし、全て 0 の場合は、{0} による初期化を行ってもよい。 【MISRA 9.2】
保守性	6.プログラムはシンプルに書く	
プログラムは対称的に記述する。	6.1	マスクのオン / オフ処理やファイルのオープン / クローズ処理など、対称性を保つ処理は、関数内で対称性が分かるように記述する。
構造化プログラミングを行う。	6.2	(1) ループを脱出させるための break 文は 1 つまでとする。 (2) ループを脱出させるための break 文を使用してもよい。
	6.3	(1) goto 文を使用してはならない。 (2) goto 文は、多重ループを抜ける場合とエラー処理に分岐する場合だけに使用する。
	6.4	continue 文を使用してはならない。 【MISRA 14.5】
	6.5	(1) switch 文の case 節は、break 文で必ず終了させる。 (2) 〈〈switch 文を break 文で終了させない場合は、プロジェクトで決定したコメントを入れる。例えば、/* FALL THROUGH */ 等〉〉
	6.6	関数は、1 つの return 文で終了させる。 【MISRA 14.7】
保守性	7.統一した書き方にする	
ファイル内の記述内容と記述順序を規定する。	7.1	〈〈ヘッダファイルに記述する内容(宣言、定義など)とその記述順序を規定する。〉〉
	7.2	〈〈ソースファイルに記述する内容(宣言、定義など)とその記述順序を規定する。〉〉
	7.3	外部変数は、複数箇所で定義しない。【MISRA 8.9】
	7.4	ヘッダファイルには、外部変数定義や関数定義を記述しない。 【MISRA 8.5】
	7.5	ヘッダファイルは重複取り込みに耐えうる作りとする。〈〈そのための記述方法を規定する。〉〉

作法	ルールNo.	ルール
コーディングスタイルを統一する。	7.6	〈〈波括弧（{）や字下げ、空白の入れ方などのスタイルに関する規約を規定する。〉〉
名前の付け方を統一する。	7.7	〈〈外部変数、内部変数などの命名規約を規定する。〉〉
コメントの書き方を統一する。	7.8	〈〈コピーライトの書き方、ファイルヘッダコメント、関数ヘッダコメント、行末コメント、ブロックコメントなどの書き方を統一させる規約を規定する。〉〉
宣言の書き方を統一する。	7.9	K&R 形式の関数宣言・定義を行わない。関数プロトタイプ宣言では、すべての引数に名前を付けるか、または全く名前を付けないかのどちらかに統一する。引数に名前を付ける場合は、関数定義時に使用した名前と同じにする。
	7.10	（1）構造体・共用体・配列の初期値式のリスト、および列挙子リストの最後の「}」の前に「,」を記述しない。 （2）構造体・共用体・配列の初期値式のリスト、および列挙子リストの最後の「}」の前に「,」を記述しない。ただし、配列の初期化の初期値リストの最後の「}」の前に「,」を書くことは許す。
宣言と定義は整合性をとる。	7.11	外部変数や関数（ファイル内でのみ使用する関数を除く）を使用したり定義する場合、宣言をヘッダファイルに記述し、そのヘッダファイルをインクルードする。
マクロ文の書き方を統一する。	7.12	#ifdef、#ifndef, #if に対応する #else や #endif にはコメントを入れ、同一ファイル内に記述し、対応関係を明確にする。
	7.13	#if で、マクロ名が定義済みかを調べる場合は、defined（マクロ名）により定義済みかを調べる。#if マクロ名 という書き方をしない。
	7.14	#if や #elif で使用する defined 演算子は、defined（マクロ名） または defined マクロ名 という書き方以外は書かない。 【MISRA 19.14】
	7.15	ブロック内で、#define、#undef を使用しない。 【MISRA 19.5】
	7.16	#undef は使用しない。 【MISRA 19.6】
	7.17	マクロを別の値で再定義しない。
保守性	8.試験しやすくなる書き方をする	
問題発生時の原因調査を実施しやすくするための工夫をする。	8.1	〈〈デバッグオプション設定時のデバッグのためのコーディング方法と、リリースモジュールにログを残すためのコーディング方法を規定する。〉〉
誤った使い方によりデバッグが困難なものは、その使用に気をつける。	8.2	1 つの #define 内では、複数の # や ## 演算子を使用しない。 【MISRA 19.12】
	8.3	関数形式のマクロよりも、関数にすべきである。 【MISRA 19.7】
動的なメモリ割り当ての使用には注意する。	8.4	（1）動的メモリは使用しない。 （2）〈〈動的メモリを使用する場合は、使用するメモリ量の上限、メモリ不足の場合の処理及びデバッグ方法等を規定する。〉〉
移植性	9.コンパイラに依存しない書き方にする	
標準化された規格に則った記述とし、拡張機能や処理系定義の機能は使用しない。	9.1	プログラムは C90 の規格に一致させる。 【MISRA 1.1】
	9.2	未定義や未規定の動作に依存した書き方をしない。 【MISRA 1.2】
	9.3	〈〈処理系定義の動作の利用方法はすべて文書化する。〉〉 【MISRA 3.1】
	9.4	〈〈使用する #pragma 指令を規定する。〉〉 【MISRA 3.4】
	9.5	〈〈他言語で書かれたプログラムを利用する場合、そのインタフェースを文書化し、使用方法を規定する。〉〉 【MISRA 1.3】

作法	ルールNo.	ルール
プログラムの記述には、言語規格で定義されている文字や拡張表記のみを使用する。	9.6	〈〈プログラムを記述する文字として C90 で規定している文字以外の文字を使用する場合、コンパイラがサポートしている文字を確認し使用する文字を規定する。〉〉
	9.7	言語規格で定義されている拡張表記（エスケープシーケンス）のみ使用する。 【MISRA 4.1】
データ型の表現および動作仕様の拡張機能及び処理系定義部分を確認し、確認したことを文書化し、使用する。	9.8	〈〈処理系の浮動小数点の実装（規格）を確認し、確認したことを文書化し、その規格に沿ったコーディング方法を規定する。〉〉
	9.9	単なる（符号指定のない）char 型は、文字の値の格納（処理）にだけ使用し、符号の有無（処理系定義）に依存する処理が必要な場合は、符号を明記した unsigned char または signed char を利用する。
	9.10	列挙（enum）型のメンバは、int 型で表現可能な値で定義する。
	9.11	処理系の整数除算の剰余の符号の扱いを確認し、確認したことを文書化し、使用する。 【MISRA 3.3】
	9.12	（1）ビットフィールドは使用しない。 （2）ビット位置が意識されたデータに対してはビットフィールドは使用しない。 （3）ビットフィールドを使用する場合は、処理系依存の動作を文書化する。
ソースファイル取り込みについて、処理系依存部分を確認し、確認したことを文書化し、使用する。	9.13	#include 指令は、〈ファイル名〉 か " ファイル名 " のいずれかの形式で記述する（マクロ展開後、この形式となる場合を含む） 【MISRA 19.3】
	9.14	〈〈#include 指令で取り込まれるソースファイルをコンパイラがどのように探すか（対応するファイル名の規則を含む）を確認し、確認したことを文書化し , それにあわせた書き方を規定する。〉〉
	9.15	文字 '、¥、"、/*、: は #include 指令のヘッダファイル指定に使用しない。
移植性	10.移植性に問題のあるコードは局所化する	
移植性に問題のあるコードは局所化する。	10.1	〈〈C 言語からアセンブリ言語のプログラムを呼び出す場合は、インラインアセンブリ言語のみが含まれる C 言語の関数として表現する、または、マクロで記述するなど、局所化する方法を規定する。〉〉
	10.2	（1）char,int,long,float および double という基本型は使用しない。代わりに typdef した型を使用する。〈〈プロジェクトで利用する typdef した型を規定する。〉〉 （2）char,int,long,float および double という基本型を、そのサイズに依存する形式で使用する場合、各基本型を typdedef した型を使用する。〈〈プロジェクトで利用する typdef 型を規定する。〉〉
	10.3	〈〈処理系が拡張しているキーワードを使用する場合、マクロを規定する。〉〉
効率性	11.資源や時間の効率を考慮した書き方にする	
資源や時間の効率を考慮した書き方にする。	11.1	〈〈swicth 文とするか if 文とするかは、可読性と速度性能を考えて選択方針を決定し、規定する。〉〉
	11.2	繰り返し処理内の条件が、繰り返し処理中に変化することがない場合、速度性能を上げるには、条件判定を、繰り返し処理前に行うようにする。
	11.3	マクロ関数はオブジェクトサイズを増加させる恐れがあるため、速度性能に関わる部分に閉じて使用する。

※ルール中の【MISRA XX.X】は、MISRA-C:2004からの引用

組込みソフトウェア開発における品質向上の勧め（コーディング編）

2005年5月20日　初版第１刷発行
2005年8月 1 日　初版第２刷発行

編　著　者　独立行政法人 情報処理推進機構
　　ソフトウェア・エンジニアリング・センター
発　行　人　速水浩二
発　行　所　株式会社翔泳社（http://www.seshop.com）
印刷・製本　日経印刷株式会社

本書へのご質問は、弊社 Web サイトの専用質問フォーム（http://www.seshop.com/book/qa/）をご利用ください。

ISBN4-7981-0950-9　Printed in Japan