1

機能安全、品質向上に欠かせない プログラム規約作成法

ー 新JIS規格を中心にー

(プロセス改善フォーラムセミナー　2012-06-26　資料の抄録版)

2

はじめに

- ✓プログラムコードの品質確保には、静解析ツールなど各種ツールも活用されますが、基本的な品質向上の視点や方針が確定していなければ、手あたりしだいのものとなり、情報過多の結果、かえって品質低下につながったりします。
- ✓本セミナーでは、プロジェクトの方針として確固としたプログラム規約を確立することによる品質向上策の王道を解説し、実装プロセスの基礎的・本質的な改善策を示します
- ✓具体的には、2011年に新しく制定された JIS X 0180「組込みソフトウェア向けコーディング規約の作成方法」の内容に沿った詳しい解説を含めて、全般的な解説を行います

3

目次

4

1. 安全で高品質なコードの必要性

- プロセス改善と高品質コード
- 機能安全等の要求
- セキュリティ、安全性等の要求
- 社会的な品質要求の高まり

本日のテーマの背景を理解する

5

プロセス改善と高品質コード

- ソフトウェアプロセス改善の課題の一環として、実装プロセスの改善がある
- 「プロセス改善」はごく管理的な話題として受け取られることもあるが、実際には技術的な課題を大きく含んでいる

- 本セミナーでは、かなり技術的な課題と言えるプログラム規約の作成の課題を見つめてみる
- 近年のトレンドとして、技術的な課題も、広めに見る必要がある
- 高品質なコードは、エンジニアの力量から生まれる面もあるが、プログラム規約といった組織的な取組から生まれる面も大きい

注: 本セミナーでは以下は同じ意味としてあつかう
- プログラム規約
- プログラミング規約
- コーディング規約

6

機能安全等の要求

- **ISO 26262 "自動車の機能安全"で求められているモデリング/コーディング・ガイドラインでカバーすべき項目は右の図の通り。**
  - レベルDが最も安全インテグリティ水準が高い
- **次の注記がある:**
  - 言語によりコーディングガイドラインは異なる
  - モデルベース開発ではコーディングガイドラインは異なることがある
  - 既存のコーディングガイドラインは特定の開発事項のみに対応することがある

「組み込み向けに適切な言語の選択」という要求もある:
- **曖昧性のない言語定義**
- **リアルタイムサポートと実行時エラー処理**
- **モジュール性、抽象化、構造化のサポート**

モデリング/コーディング・ガイドラインでカバーすべき項目	ASIL			
	A	B	C	D
低い複雑度の強制	◎	◎	◎	◎
言語の(安全)サブセット利用(*)	◎	◎	◎	◎
強い型付けの強制	◎	◎	◎	◎
防御的実装技術の利用	-	○	◎	◎
確立された設計原則の利用	○	○	○	◎
曖昧性のない図表現	○	◎	◎	◎
スタイルガイドの利用	○	◎	◎	◎
命名規則の利用	◎	◎	◎	◎

(*)の目的
- 異なる実装システムで別解釈となることの回避
- ミスにつながりやすいプログラム構成の回避
- 未対策実行時エラーを生むプログラム構成の回避

7

セキュリティ、安全性等の要求

- 国際規格でも、
  - ソフトウェアエンジニアリング担当の専門員会 SC7
  - ITセキュリティ担当の SC27
  - プログラム言語担当の SC22

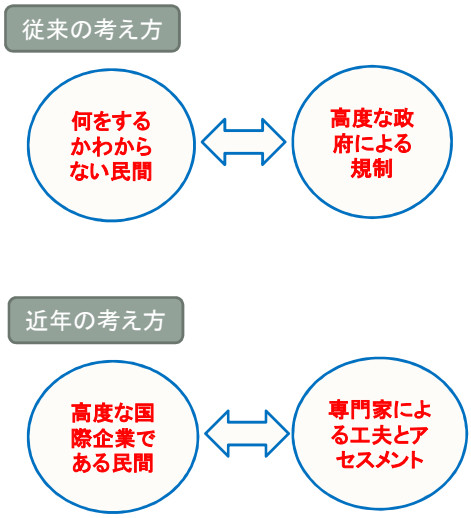
  の間でのコミュニケーションが十分ではないという現実がある

- セキュリティの課題は、運用や組織管理の課題でもあるが、開発における重要課題でもある
  - セキュアコーディングの課題は、一般的なコーディング規約の課題と統合的に考える方がよい
- いわゆる「機能安全」のみならず、ITシステムの安全性、信頼性の面からの顧客要求事項に十分注意する必要がある

8

社会的な品質要求の高まり

- 「機能安全」などのターゲット分野
  - 自動車等の輸送機
  - エレベータ等の制御
  - ロボット
- 社会インフラの安全
  - 通信、エネルギー、交通、水道などの社会インフラ
  - ITプロセスアセスメントからのアプローチ
  - 個別プロセスの技術向上からのアプローチ（要求、実装、テストなど）
  - 監査面からのアプローチ



9

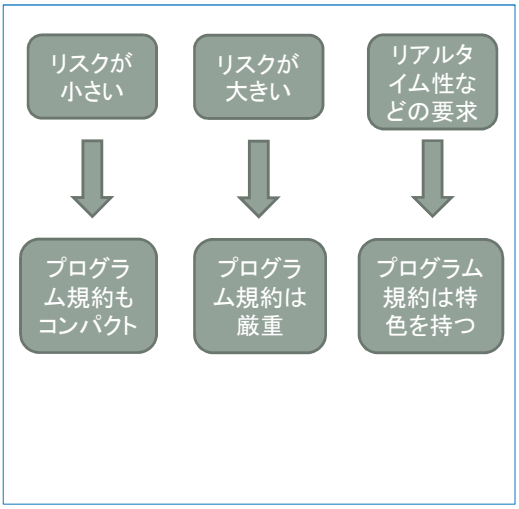
2. 品質要求の把握

- 目的からの分析
  - リスクの分析・高品質要求とは実際には何か
  - 実務的に必須な事項は何か

プログラミングの品質向上といっても、どのようなレベルと視点で対応するのかによって、コストやリソースについての考え方が大きく変わってくる

10

目的からの分析



- リスクの分析・高品質要求とは実際には何かの理解が重要
  - リスクの対象の認識
  - 誰にとってのどのようなリスクか
  - トラブル時の被害想定など
    - どの部分のトラブルをどの程度想定するのか
    - 誰にとってのどのような被害か
    - どのような利用者の広がりがあるか
  - 使用期間の問題、使用環境の変化の問題など

11

実務的に必須な事項は何か

- 利用者と社会のリスクの程度を押さえる（開発者にとっては、発注者とのリスク程度の合意が重要）
- プロジェクト自体のリスクを押さえる
  - 担当者スキルとチーム統制
  - 外注リスク
  - OSSやライブラリ利用方針、開発物再利用方針など
  - 開発成果物のライフサイクル義務（保守等）
  - 開発チームにとっては、受け入れテスト方針の内容と程度

- 契約書、仕様書、議事録、技術連絡書等
- プロジェクトリポジトリのあり方

多くの場合は、形式的なチェックが必要なわけではなく、条件に応じて柔軟な対応が可能ということの方が重要

12

3. プログラミング規約の作成、実施と保守

- 規約作成のなぜ・なに
- どのような手段があるか
- 静解析ツールの活用
  - 複雑度測定等を含む
- 規格の活用

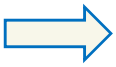
プログラミング規約の作成、実施、保守について、具体的アプローチにどのようなものがあるか見てみる

13

規約作成の「なぜ・なに」

- 今、プログラミング規約が必要な理由
  - コード再利用の広がり
  - 信頼性に対する要求の高度化
  - レビューが広く行われるようになった
  - 第三者からの調達の広がり（特に海外）
  - 静解析ツール利用の必須背景知識

- これまでも必要だったが：
  - チームや組織で、一定の品質水準の確保が必要
  - 一人の開発者でも、安定的な品質の確保が必要
    - ルール化により、考え抜かれたコーディングスタイルが得られる
  - 移植性（ポータビリティ）確保にもルールによる制約が有効



- プロジェクトレベルでの規約（ルール）化
- 開発組織レベルでの規約（ルール）化
- 産業分野、企業グループでの規約（ルール）化

- 信頼性ルール
- 保守性ルール
- 移植性ルール
- 効率性ルール
- その他のルール

14

どのようなアプローチ、手段があるか

- 開発チームで合意し文書化する
- 異なる出身の開発者の意識を合せる
  - 複数企業の共同作業の場合
  - オープンソースソフト開発などの場合
- 品質要求（機能安全等）に応えるためのプログラミング規約（ルール）

- 国際的な標準の活用
  - MISRA-C（C++）
    - 英国で開発された自動車向けソフトウェア安全基準のデファクト標準
    - MISRA, The Motor Industry Software Reliability Association,が開発したもので、1998年版と2004年版がある。2012(2011?)年版が近々発行される。
  - JIS規格（日本工業規格）X 0180
    - IPA/SECの開発した組み込み向けコーディング規約ガイドを基にJIS化したもの
  - C言語国際規格の安全性対応
- コードサンプル等の利用
  - 一組織内であれば、コードのサンプルやテンプレートなどの利用が実用的には用いられている
  - コードの自動生成環境がそれに役立つこともある

15

公的機関等のプログラミング規約の例

やや古いもの

- NASA SOFTWARE ENGINEERING LABORATORY SERIES SEL-94-003
  - C STYLE GUIDE (AUGUST 1994)
    http://ntrs.nasa.gov/archive/nasa/casi.ntrs.nasa.gov/19950022400_1995122400.pdf
  - C++ Coding standard and Style Guide
    http://ntrs.nasa.gov/archive/nasa/casi.ntrs.nasa.gov/20080039927_2008038882.pdf
- Secure Programming for Linux and UNIX HOWTO(最も知られているルール集)
  http://www.dwheeler.com/secure-programs/
  http:/www.linux.or.jp/JF/JFdocs/Secure-Programs-HOWTO/
- Secure Unix Programming Checklist (AusCERTのチェックリスト)
  http://auscert.org.au/render.html?it=1975&cid=1920

簡潔なルール(場合により、そのまま規約としてもよい)

- CERT Top 10 Secure Coding Practices
  https://www.securecoding.cert.org/confluence/display/seccode/Top+10+Secure+Coding+Practices
- ネット情報の例示(個人サイトのセキュリティプログラミングルール)
  http://www.gadgety.net/shin/tips/unix/programming.html

MISRAのガイドレパートリー

- **MISRA Autocode(モデリング言語とコード自動生成)**
- **MISRA C**
- **MISRA C++**
- **MISRA SA(安全性分析)**

16

静解析ツールの活用

- **製品例(ルールチェッカー/静解析ツールは多数ある)**
  - Klocwork Insight (商用、C/C++/C#/Java)
    http://www.klocwork.com/jp/
  - Coverity Static Analisys (商用、C/C++/C#/Java)
    http://www.coverity.com/html_ja/products/static-analysis.html
  - PGRelief C/C++ (商用、C/C++)(別途Java用もあり)
    http://jp.fujitsu.com/group/fst/services/pgr/
  - SQMlint (商用、MISRA-C用)
    http://japan.renesas.com/products/tools/coding_tools/misrac/misrac_rule_checker/index.jsp
  - QAC((商用、C) (別途C++用もあり)(MISRA-C, IPA/SEC対応も)
    http://www.toyo.co.jp/ss/qac/index.html
  - CodeSonar(商用、C/C++、基本はビルトインされたチェック機能だが、ルールチェックも可能)
    http://www.grammatech.com/products/codesonar/overview.html
  - Splint(フリー、C)
    http://www.splint.org/
  - RATS(フリー、C,/C++,/Perl,/PHP /Python)
    https://www.fortify.com/ssa-elements/threat-intelligence/rats.html
  - Uno(フリー、C、基本はビルトインされたチェック機能だが、ルールチェックも可能)
    http://spinroot.com/uno/
  - PMD(フリー、Java)
    http://pmd.sourceforge.net/
- 他のチェッカー:
  - 動解析ツールも静解析ツールと併用される
  - 静解析ツールの中には、プログラムの全体構造の可視化を目的とするタイプのものもある
  - セキュリティ脆弱性チェッカーは、別格として必要だが、ファジー的な入力データを生成するなど動的チェッカーも多い

- **ルールチェッカーの一般的な特質は、しばしば「厳しすぎる」ことである。Coverity等は、厳選されることを売りにしている**
- **意味論的解析**
  製品Polyspaceのように、抽象解釈の理論によりコードの振る舞いを静的に検証するツールもある(プログラミングルールとは距離が遠いが、製品としてはコーディング規約の適用可能をうたっている)
  http://www.mathworks.co.jp/products/polyspaceclientc/description4.html

KlockworkでのMISRA-C取扱い例
http://www.klocwork.com/products/documentation/current/MISRA-C_checker_reference/ja

下記に、チェックルールとMISRA-Cのルールの対応表の一部を引用紹介

17

複雑度測定

- データフローの複雑性
- 分岐判定の構造の複雑性
- 式や文の意図表現の複雑性
  （否定条件やその結合、否定の否定　など）
- ・ ループ停止条件等の複雑性

- 可読性（保守性）、再利用性の観点からのものが多い
- 複雑度には多くの視点があるので一律なルールで妥当に判定できるか検討が必要

18

規格類の活用： JIS規格の序文から

- **JIS規格 X 0180では、規格の目的を右のようにのべている**

- 「この規格は，業界，組織又はプロジェクトの特性に応じて組込みソフトウェア向けに開発されるソフトウェアのソースコードの品質をより良いものとすることを目的として，コーディングのときに用いることが望ましい規約を定める場合の，プログラム言語に共通した，規約作成方法の基本を規定したものである。また，開発されるソフトウェアのソースコードの作法を統一することによって，ソースコードの容易な理解，ソフトウェア作成の能率向上なども目的としている。」

19

規格類の活用: 入手方法

C++ 対応版もあり

- **JIS規格 X0180:2011**
  - **JISCホームページでの閲覧**
  - **JSA(日本規格協会)からの購入**
    - **製本版**
    - **PDF版**

- **IPA/SECからの出版物としての購入**
  - “【改訂版】組込みソフトウェア開発向けコーディング作法ガイド[C言語版]”
  - プロセス規格部分は補助的にしかなく、JISの附属書A-Dの元となったものを読みやすく記載
  - 附属書A-Dについて、記載内容に過不足はなく、有効に利用できる
    - http://sec.ipa.go.jp/publish/index.html#emb
    - http://www.seshop.com/product/detail/7950/

- **MISRA-Cの購入**
  - 英語版(2012年中に第3版を出版予定)
    - http://www.misra.org.uk/Publications/tabid/57/Default.aspx (Guidelines for the Use of the C Language in Critical Systems)
  - 日本語版
    - http://tech.jsae.or.jp/hanbai/list.aspx?category=522 (テクニカルペーパー TP-01002: 自動車用C言語利用のガイドライン(第2版))
  - 日本語解説本
    - 「組込み開発者におくMISRA - C:2004—C言語利用の高信頼化ガイド」日本規格協会刊、MISRA-C研究会編

20

参考(C言語自体の安全化その1):

- ISO/IEC TR 24731-1 (The "Safer C Library" i)
  - ISO/IEC TR 24731-1:2007 Information technology -- Programming languages, their environments and system software interfaces -- Extensions to the C library -- Part 1: Bounds-checking interfaces

- 主として場ファーオーバーフロー対策のセキュアコーディング対応文字列関数
- 実装例は多くなってきている(Visual C++ 2005以後など)
- ISO/IEC JTC1/SC22 (プログラミング言語)が制定

21

参考(C言語自体の安全化その2):

- ISO/IEC TR 24772
  - ISO/IEC TR 24772 Information technology -- Programming languages -- Guidance to avoiding vulnerabilities in programming languages through language selection and use

- 脆弱性の発生原因を列挙し、セキュリティガイドライン設定のためのガイドを提供している
- ISO/IEC JTC1/SC22(プログラミング言語)が制定

22

4. JIS規格の考え方

- 規約作成プロセス
- コーデイング規約とルール
- 規約の管理・改善プロセス
- JISの概要目次

JIS規格の基本的な考え方は重要であり、制定経過を踏まえてその「基本」を解説する
この規格は「規約作成プロセス」の規定として制定されている

23

規約作成プロセス

- この規格の取り扱う範囲
  - 次のものは，この規格の作法及びルールの対象外
    - a) ライブラリ関数
    - b) メトリクス（関数の行数・複雑度など）
    - c) コーディングミスに分類されると思われる記述誤り。

- 実は、全般に初歩的なルールが多いが、全部守るのはそれなりに困難な場合もある
- 附属書での具体的な例示や、理由づけの説明は傾聴すべき内容が結構多い

- 新規コーディング規約の作成(箇条5.3)

作成時期の決定

↓

作成方針の決定

方針の決定では，プロジェクトで重視したい品質特性，作法，及び次の視点などを考慮
a)プログラムを見やすくするコーディング
b)デバッグを考慮したコーディング
c)障害許容性を考慮したコーディング

↓

附属書等からルールの選択

↓

プロジェクト依存部分の定義

↓

作成したルールのレビュー

24

コーディング規約とルール

JIS X 0180箇条3から

- コーディング作法（coding convention）
  - ソースコードの品質を保つための慣習及び実装の考え方。個々のルールの基本概念を示す。この規格では誤解のおそれがない場合，単に作法ともいう
- コーディング規約（coding guideline）
  - 業界，組織又はプロジェクトにおいて，品質を保つために守ることが望ましいソースコードの書き方（ルール）を整理したもの
- コーディングルール（coding rule）
  - コーディング規約を構成する，具体的な一つ一つの決めごと。誤解のおそれがない場合，単にルールともいう

作法 （JISの基本視点、考え方）

25

規約の管理・改善プロセス（その1）

- **コーディング規約の改善（箇条5.4）**
  - 継続的に**安定的なコーディング規約**の運用
    - 可能な場合は，組織のもつソフトウェアプロセスの一環としてコーディング規約の活用を制度化していく
  - 同時に，特性の変化に対応して**適切なコーディング規約を改定・運用**することが必要
    - 定期的にレビューし，必要な改善を提起・計画・実装・周知
    - 改善を提起する根拠となる指標データを収集するのがよい

- **運用時のルール適用除外及びその手順(箇条5.5.2)**
  - プロジェクトで望ましい品質特性が異なる場合がある。
  - そのため、部分的にルールを**適用除外として認める**ことを手順化しておく必要がある。しかし、安易にルール適用除外を許してしまい，ルールが形がい（骸）化するのを防止する。
  - 適用除外を認める手順の例：
    a) 適用除外の理由書を作成する
    [理由書の項目例：“ルール番号”“発生箇所（ファイル名，行番号）”“ルール遵守の問題点”“ルール逸脱の影響など”]
    b) 有識者のレビューとレビュー結果の記載
    c) 工程の責任者の承認と記録

26

規約の管理・改善プロセス（その2）

- **教育**（箇条5.5.3）
  - コーディング規約は，開発要員に対して制度的に**教育し，要員がよく理解する**必要がある
  - 開発要員が自ら積極的にその必要性を理解し，実施し，また，改善を提起するものでなければならない

- 既存コーディング規約からの充実（箇条5.6）
  - **コーディング規約が既に存在する業界，組織又はプロジェクトでは，その規約を更に充実させるために利用できる**
  - **その際、次の点に留意する**
    a) 抜け漏れの防止
    既存のコーディング規約に抜けている観点やルールをうことができ，また，自プロジェクトが何に重点を置いているか再認識することができる
    b) ルール必要性の明確化
    理由も明示されず制定されていた規則の必要性を再認識するためのツールとして利用できる

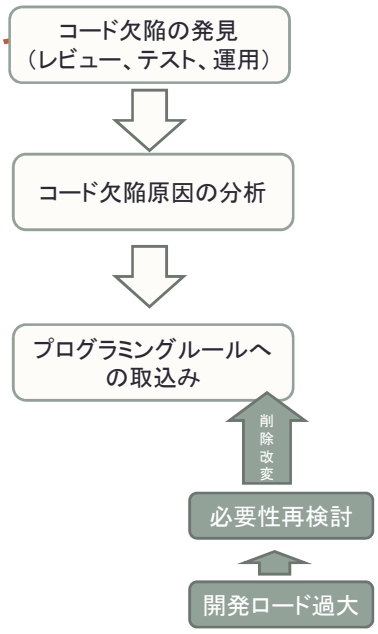
27

規約の管理・改善プロ

- プログラミング規約を用いたプロセスを実施した経験の蓄積は非常に重要

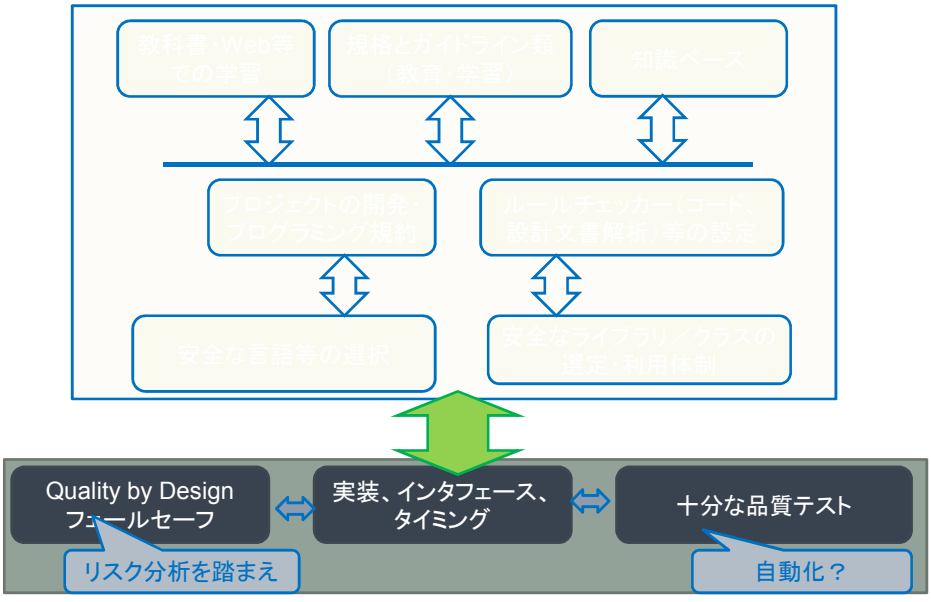
以前はIT利用者は特定者/専門家とみなせたが、今は利用者の行動が複雑で深い検討とフィードバックが必要

- 話す言語、文化的習慣の多様性
- 勝手な使い方
- 知識のない使い方
- 身体条件によるアクセスビリティ
- セキュリティ上の攻撃者
- 組込みシステム等でのハードウェア（機械部分）との故障の相互作用



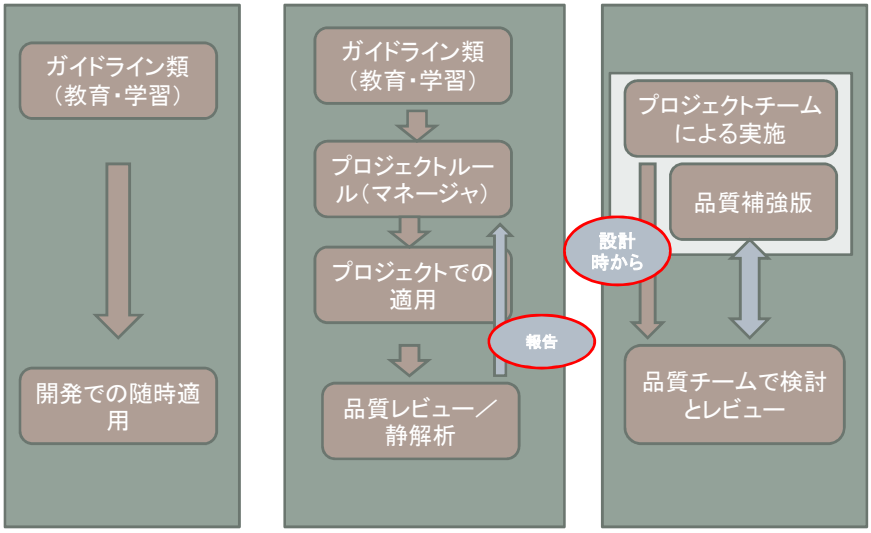
28

参考： 実施アプローチ（技術面）



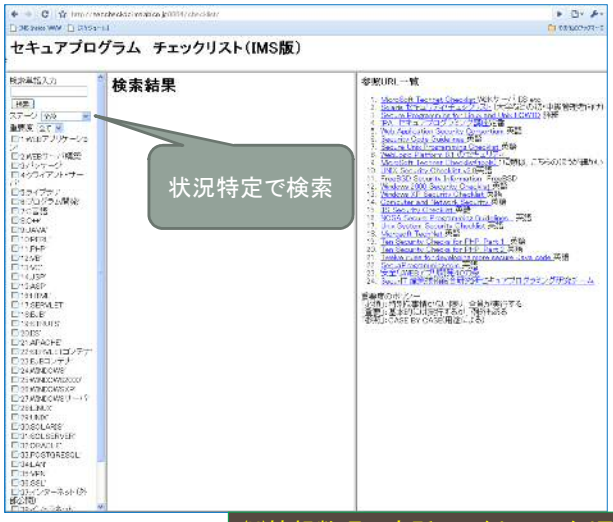
29

参考： 実施アプローチ例（組織面）



30

（参考）コーディングルールデータベース活用の例



㈱情報数理研究所での例:2005年頃

31

JISの概要目次

32

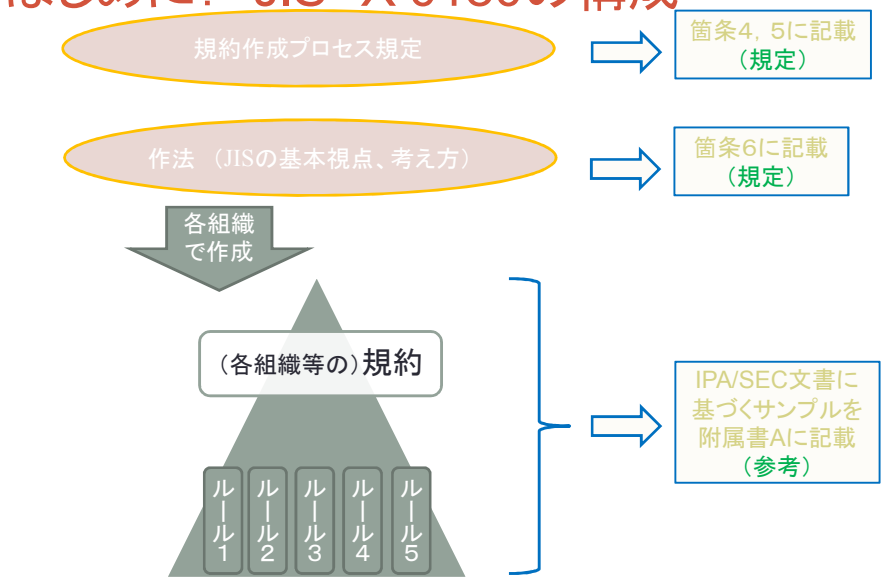
5. プログラミングルールの実際

- 信頼性作法(ルール)
- 保守性作法(ルール)
- 移植性作法(ルール)
- 効率性作法(ルール)
- セキュリティ確保作法(ルール)

JIS規格の内容詳細をこの講習の時間の許す限りで、点検してみる

33

はじめに: JIS X 0180の構成



34

信頼性作法（ルール）

- **領域の信頼性**
  (R1) 領域は初期化し，大きさに気を付けて使用する
- **データの安全性**
  (R2) データは範囲，大きさ，内部表現などに気を付けて使用する
- **障害許容性**
  (R3) 動作が保証された書き方にする

(R1.1) 領域は，初期化してから使用する
(R1.2) 初期化は過不足ないことが分かるように記述する
(R1.3) 領域へのポインタは，大きさ（指す範囲）に気を付けて使用する

(R2.1) 内部表現に依存しない比較を行う
(R2.2) 論理値などが区間として定義されている場合，その中の一点（代表的な実装値）と等しいかどうかで判定を行ってはならない
(R2.3) データ型をそろえた演算・比較を行う
(R2.4) 演算精度を考慮して記述する
(R2.5) 情報損失の危険のある演算は使用しない
(R2.6) 対象データが表現可能な型を使用する。
(R2.7) ポインタの型に気を付ける
(R2.8) 宣言，使用及び定義に矛盾がないことをコンパイラが検査できる書き方にする

(R3.1) 領域の大きさを意識した書き方にする
(R3.2) 実行時にエラーになる可能性のある演算に対しては，エラーケースをう（迂）回させる
(R3.3) 関数呼出しではインタフェースの制約を検査する
(R3.4) 再帰呼出しは行わない
(R3.5) 分岐の条件に気を付け，所定の条件以外が発生した場合の処理を記述する
(R3.6) 評価順序に気を付ける

35

作法(R1.1)の「ルール」への具体化例(附属書Aによる)

ルールの提示方法はこの程度の具体性があると誤解が少ない

(R1.1) 領域は，初期化してから使用する
具体的なルールは，次の二つが考えられる

(R1.1.1) 自動変数は宣言時に初期化するか，又は値を使用する直前に初期値を代入する。[採用指針:基本] [規約化:]

適合例	不適合例

void func() {
int var1 = 0;
/* 宣言時に初期化する */
int i;
var1++;
/* 使用する直前に初期値を代入 */
for (i = 0; i < 10; i++) {
…
}
}
	void func() {
	int var1;
	var1++;
	…
	}

注記 自動変数を初期化しないと，その値は不定となり，環境によって演算結果が異なる現象が発生する。初期化のタイミングは宣言時，又は値を使用する直前とする

(R1.1.2) const型変数は，宣言時に初期化する。[採用指針:基本] [規約化:]

適合例	不適合例
const int N = 10;	const int N;

注記 const型変数は後から代入ができないので，宣言時に初期化する。初期化しないと外部変数の場合は0，自動変数の場合は不定となるので，意図しない動作となる可能性がある。宣言時に未初期化でもコンパイルエラーにならないため，注意が必要である。
なお，C++ではconst型変数の未初期化はコンパイルエラーとなる

このようなルールの理由づけがあると応用が利く

36

作法(R3.2)の「ルール」への具体化例(附属書Aによる)

(R3.2)実行時にエラーになる可能性のある演算に対しては，エラーケースをう(迂)回させる
具体的なルールは，次の二つが考えられる

(R3.2.1)除算及び剰余算の右辺式は，0でないことを確認してから演算をする。[採用指針:] [規約化:]

適合例	不適合例

if (y != 0) {
ans = x/y;
}
	ans = x/y;

注記 明らかに0でない場合を除き，除算及び剰余算の右辺(右オペランド)が0でないことを確認してから演算する。そうしない場合，実行時に0除算のエラーが発生する可能性がある。

(R3.2.2)ポインタは，空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。[採用指針:] [規約化:]

適合例	不適合例

if (p != NULL) {
*p = 1;
}
	*p = 1;

注記 (なし)

保守性作法(ルール)

- 解析性
(M1) 他人が読むことを意識する
- 変更性
(M2) 修正誤りのないような書き方にする
- プログラムの簡潔性
(M3) プログラムはシンプルに書く
- コーディングスタイルの統一性
(M4) 統一した書き方にする
- 試験性
(M5) 試験しやすい書き方にする

(M1.1) 使用しない記述を残さない。
(M1.2) 紛らわしい書き方をしない。
(M1.3) 特殊な書き方はしない。
(M1.4) 演算の実行順序が分かりやすいように記述する。
(M1.5) 誤解を招きやすいある種の演算については，言語仕様で省略可能になっている場合でも，省略せずに明示的に記述する。
(M1.6) 領域は一つの利用目的に使用する。
(M1.7) 名前を再使用しない。
(M1.8) 勘違いしやすい言語仕様を使用しない。
(M1.9) 特殊な書き方は意図を明示する。
(M1.10) マジックナンバーを埋め込まない。
(M1.11) 領域の属性は明示する。
(M1.12) コンパイルされない文でも正しい記述を行う。

(M2.1) 構造化されたデータ及びブロックは，まとまりを明確化する。
(M2.2) アクセス範囲及び関連するデータは局所化する

(M3.1) 構造化プログラミングを行う。
(M3.2) 一つの文で一つの副作用とする。
(M3.3) 目的の違う式は，分離して記述する。
(M3.4) 複雑なポインタ演算は使用しない

保守性作法(ルール) 続き

(M4.1) コーディングスタイルを統一する
(M4.2) 注釈の書き方を統一する
(M4.3) 名前の付け方を統一する
(M4.4) ファイル内の記述内容及び記述順序を統一する
(M4.5) 宣言の書き方を統一する
(M4.6) 空ポインタの書き方を統一する
(M4.7) 前処理指令の書き方を統一する

(M5.1) 問題発生時の原因を調査しやすい書き方にする
(M5.2) 動的なメモリ割当ての使用に気を付ける

作法(M1.1)の「ルール」への具体化

(M1.1) 使用しない記述を残さない
具体的なルールは，次の二つが考えられる

(M1.1.1) 使用しない関数，変数，引数，ラベルなどは宣言(定義)しない。[採用指針:必要] [規約化:]

適合例	不適合例

void func(void) {
…
}
 	
	void func(int arg) {
	/* arg未使用
	*/
	…
	}

注記 使用しない関数，変数，引数，ラベルなどの宣言(定義)は，削除し忘れたのか，記述を誤っているかの判断が難しいため，保守性を損なう。

(M1.1.2) コードの一部を"コメントアウト"しない(MISRA 2.4)。[採用指針:必要] [規約化:]

適合例	不適合例
… #if 0 /* ～のため，無効化 */ a++; #endif …	… /* a++; */ …

注記 無効としたコード部を残すことは，コードを読みにくくするため，本来避けることが望ましい。
ただし，コード部の無効化が必要な場合は，コメントアウトせず，#if 0で囲むなど，無効化したコード部を明示するルールを決めておく。

移植性作法(ルール)

- 処理系からの独立性
  (P1) コンパイラに依存しない書き方にする
- コードの局所性
  (P2) 移植性に問題のあるコードは局所化する

(P1.1) 拡張機能及び処理系定義の機能は使用しない
(P1.2) 言語規格で定義されている文字及び拡張表記だけを使用する
(P1.3) データ型の表現，動作仕様の拡張機能，及び処理系依存部分を確認し，文書化する
(P1.4) ソースファイル取込みについて，処理系依存部分を確認し，依存しない書き方にする
(P1.5) コンパイル環境に依存しない書き方にする

(P2.1) 移植性に問題のあるコードは局所化する

41

作法(P1.1)の「ルール」への具体

(P1.1) 拡張機能及び処理系定義の機能は使用しない
具体的なルールは，次の三つが考えられる

(P1.1.1) 次のいずれかを選択する。[採用指針：] [規約化：選択，文書]

選択1	選択2
C90の仕様外の機能は使用しない。	C90の仕様外の機能を使用する場合は，《使用する機能とその使い方を文書化する。》

注記 C90では，// コメント，long long型など，最近のコンパイラでは提供されC99の規格で定義されている機能が許されていない。2)のルールを選択し，C99の規格で定義されている機能について利用を認めるというルールも現実的である。

(P1.1.2) 《使用する処理系定義の動作はすべて文書化する。》(MISRA 3.1)[採用指針：必要] [規約化：文書]

注記 C90では，ライブラリ部分を除くと41個の処理系定義項目がある。例えば，次は処理系定義であり，使用する場合には文書化の対象となる。
・浮動小数点の表現方法
・整数除算の剰余の符号の扱い
・インクルード指令のファイル検索順序
・#pragma
附属書Cも参照。

(P1.1.3) 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。》[採用指針：] [規約化：文書，規約]

注記 C言語規格では，他の言語で書かれたプログラムをC言語プログラムから利用するためのインタフェースを定めていない。すなわち，他言語で書かれたプログラムを利用する場合は，拡張機能を利用することとなり，移植性に欠ける。使用する場合は，移植の可能性の有無にかかわらず，コンパイラの仕様を文書化するとともに使用方法を規定する。

42

効率性作法(ルール)

- 資源及び／又は時間の効率性

(E1) 資源及び／又は時間の効率を考慮した書き方にする

(E1.1) 資源及び／又は時間の効率を考慮した書き方にする

作法(E1.1)の「ルール」への具体化例(附属書AIによる): 続き

(E1.1) 資源及び／又は時間の効率を考慮した書き方にする。
具体的なルールは，次の四つが考えられる

(E1.1.1) 関数形式マクロは，速度性能にかかわる部分に閉じて使用する。[採用指針:基本] [規約化:]

適合例	不適合例

extern void func1(int, int); /* 関数 */
#define func2(arg1, arg2) /* 関数形式マクロ */
func1(arg1, arg2);
for (i = 0; i < 10000; i++)
{
func2(arg1, arg2);
/* 速度性能が重要な処理 */
}
	#define func1(arg1, arg2)
	/* 関数形式マクロ */
	extern void func2(int, int);
	/* 関数 */
	func1(arg1, arg2);
	for (i = 0; i < 10000; i++) {
	func2(arg1, arg2);
	/* 速度性能が重要な処理 */
	}

注記1 関数形式マクロよりも関数の方が安全であり，なるべく関数を使用することが望ましい。しかし，関数は呼出しの処理と復帰の処理とで速度性能が劣化する場合がある。このため，速度性能を上げたい場合，関数形式マクロを使用する。ただし，関数形式マクロを多用すると，使用した場所にコードが展開されオブジェクトサイズが増加する可能性がある。

注記2 関連ルールとして(M5.1.3)を参照。

(E1.1.2) 繰返し処理内で，変化のない処理をしない。[採用指針:基本] [規約化:]

適合例	不適合例

var1 = func();
for (i = 0;
(i + var1) <MAX; i++)
{
…
}
	/* 関数funcは，同じ結果を返す */
	for (i = 0;
	(i + func()) < MAX; i++) {
	…
	}

注記 同じ結果が返される場合に，同じ処理を複数回実施すると非効率である。コンパイラの最適化に頼れることも多いが，例のように，コンパイラでは分からない場合は注意する。

作法(E1.1)の「ルール」への具体化例(附属書AIによる): 続き

(E1.1) 資源及び／又は時間の効率を考慮した書き方にする。
具体的なルールは，次の四つが考えられる

(E1.1.3) 関数の引数として構造体ではなく構造体ポインタを使用する。[採用指針:] [規約化:]

適合例	不適合例

typedef struct stag {
int mem1;
int mem2;
…
} STAG;
int func (const STAG *p) {
return p->mem1 + p->mem2;
}
	typedef struct stag {
	int mem1;
	int mem2;
	…
	} STAG;
	int func (STAG x) {
	return x.mem1 + x.mem2;
	}

注記 関数の引数として構造体を渡すと，関数呼出し時に構造体のデータをすべて実引数のための領域にコピーする処理が行われ，構造体のサイズが大きいと，速度性能を劣化させる原因となる。
参照しかしない構造体を渡す場合は，単に構造体ポインタにするだけでなく，const修飾を行う。

(E1.1.4) 《switch文とするかif文とするかは，可読性及び効率性を考えて選択方針を決定し，規定する。》[採用指針:] [規約化:規約]

注記1 switch文はif文より可読性に優れることが多い。また，最近のコンパイラは，switch文に対して，テーブルジャンプ，バイナリサーチなどの最適化したコードを出力することが多い。このことを考慮してルールを規定する。

注記2 関連ルール(M1.3.1)を参照。
ルールの例:
式の値(整数値)によって処理を分岐する場合，分岐の数が3 以上である場合，if文ではなくswitch文を使用する。ただし，プログラムの性能向上において，switch文の効率が問題となる場合には，この限りではない。

45

セキュリティ確保作法（ルール）

- (JIS X 0180にはこの項目はない; OWASP Top10などの作法・ルールを参照するとよい)
  https://www.owasp.org/index.php/Top_10_2010

- 信頼性のルールとセキュリティ対応のルールとは考え方がかなり違うので、必要に応じて両方考慮するのがよい
  - たとえば入力値チェックの方針、ライブラリ選択など
- セキュリティ固有の問題ではないが、例外処理や割り込み処理のルールを、設計事項としても、コーディングルールとしても明示しておくのがよい
- オブジェクト指向言語では、クラス継承やオペレータオーバーロードの方針などのルールが加わってくる

46

6. 開発ルール全般への展開・連携

- データチェック方針
- セキュリティ方針
- テスト方針
- 構成管理面／再利用面
- サプライチェイン問題

プログラミング規約はプロジェクトまたは開発組織全体での開発規約の一部であり、より広がりを持ったものの中で考える方がよい

47

はじめに

- 開発ルールへの展開
  - 開発・運用プロセス全般への気配りが必要
    - そのプロジェクトでどこまで厳密、または迅速に開発する必要があるのか
    - 要求事項の引き出し、要求変更管理、コミュニケーションなどの方針を決定し、プロジェクトの開発ルールとして明確化する
  - その際、次のような点に留意する
    - プロジェクト管理の具体的な方針
    - プロトタイピング作成といったプロセス設計上の課題
    - セキュリティ方針
    - 品質方針（レビューを含む）
    - テスト方針…
  - テスト駆動開発など、複数の開発フェーズやプロセスに関係する手法に留意する
  - 構成管理方針など全般を通じたフィードバックのありかたに留意する

48

参考：「組込みソフトウェアの開発力向上に向けた施策と提言」（平成16年）での実装局面での課題の広がりの図解

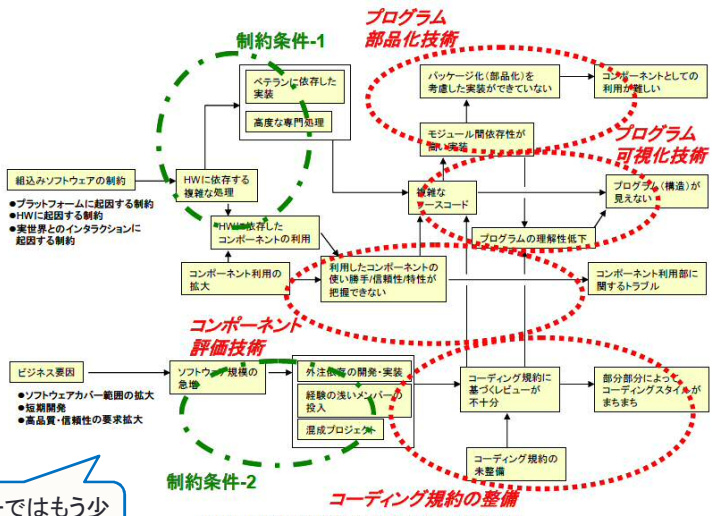


図 2. 14「実装技術」における課題と解決策

本セミナーではもう少し広めにも見たい

49

データチェック方針

- 安全面、セキュリティ面などから入力データのチェックを適切な厳密さで行うルールを設定する
  - 「電話番号入力はハイフンを除いて入力する」というような不合理な入力チェック規則は作らない

- データチェック方針と、入力データの検査コーディングルールとは連動する

50

セキュリティ方針

- 機能安全等の安全でのリスク分析に加え、セキュリティ面からのリスク分析も必要
  - それに対応したプログラミング方針、規約

- 通常問題となるセキュリティ課題に加えリリースに伴う次のような課題にも留意する
  - コードの著作権＋機密性保護技術
    - シンプルなものではテキストストリッパー、コメントストリパー、コードジャムラーなど
  - デバッグ用コードの削除
  - 必須以外のメンテナンスフックの削除
  - 不要なテストデータの除去
  - デフォルトおよびテスト用アカウント設定の削除
  - 不必要な付随サービスの停止
  - WindowsのUACのようなアクセス制御技術への厳密な整合
  - インストールパッケージの諸課題
  - 配布用コードへの電子署名
  - アップデートモジュール作成と適用管理システムの諸課題

51

テスト方針

- **テスト方針は、プロセス実施の開発ルールとしても明確にする**
  - 通常、実装者の責任となる単体テストの責任範囲の明確化
  - 記録のとりかた、変更管理との関連、リグレッションテストの方針など
- **リスク認識・分析に基づくテストデータの準備も開発上の大きな課題**
- **プログラミング規約に対応するテスト方針もある**
  - 規約で目指しているものの実現の検査
  - 入力データチェックなどの品質

- テストを重視したプロセス設計では、上流の設計プロセスや実装プロセスにテスト関連の作業が多数盛り込まれる
  - ライフサイクルを通じたコンセプトと実施手順上の整合性、作業効率性が求められる
- テストの範囲の確定
  - 正常値・異常値・境界値テストと組み合わせ
  - コードカバレジ
  - 実機テスト
  - ストレステスト
  - 性能、容量テスト
  - セキュリティテスト
  - 回帰テスト

52

構成管理面／再利用面

- 再利用、プロダクトライン開発等では、プログラム規約によるコードの標準化の重要性がさらに浮かび上がってくる

- 例： ある構成管理トレーニングコースのPRから
  - **複数のバリアントを含む有効なソースコード管理実践の実装**
  - **アプリの自動ビルド、パッケージ化、配布**
  - **ITガバナンスとコンプライアンスが実現できる管理策の確立**
  - **産業界の標準(IEEE,ISOなど)の利用**
  - **継続的に成長、改善できる構成管理の生成**

コードの標準化が重要なことが推察される

53

サプライチェイン問題

- 協力業者に委託するサプライチェーンも課題は多い
  - 技術力全般、開発手法
  - 品質管理能力（同じプログラム規約の摘要可否）
  - コミュニケーション力
  - 保守能力
- OSS利用問題
  - オープンソース利用の際には、その品質問題、セキュリティ問題等に関して開発管理上のチェックを重視する

- SAFECodeの文書
  - “Software IntegrityControls – An Assurance-Based Approach to Minimizing Risks in the Software Supply Chain” (June, 14, 2010)は、サプライチェイン問題を主題とした安全性検討