Panasonic Technical Journal Vol. 56 No. 3 Oct. 2010

組込みソフトウェアにおける開発力強化の取り組み

Activities for Improvement of Embedded Software Development Ability

春名修介 Shusuke Haruna
松本範久 Norihisa Matsumoto
古山寿樹 Hisaki Furuyama
中川雅通 Masamichi Nakagawa

要　旨

家電組込みソフトウェアは，デジタル化・ネットワーク化により規模が拡大しているにもかかわらず，小規模開発時代のコード中心の開発スタイルが踏襲され，後工程依存の開発が行われており，品質・生産性を落とす要因ともなっている。このような状況を打破するためには，上流工程での完成度を向上させることが不可欠である。本稿では，実装力の向上および設計力の向上の観点から，当社におけるソフトウェア開発力強化の取り組みについて報告する。

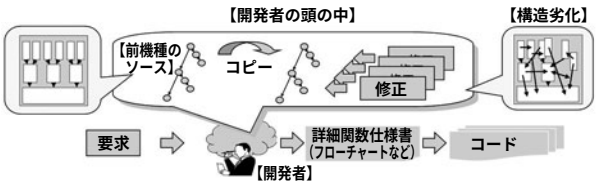
Abstract

Despite the growing scale due to digitization and networking, embedded software for consumer electronics has followed the code-centric development style used in the age of small-scale development. Therefore, downstream-centric development has been performed, which causes declining quality and productivity. In order to overcome this situation, it is essential to improve development capabilities that enhance the completeness of the upstream process. In this paper, we report on efforts to strengthen our software development from the viewpoint of improvement of software implementation ability and improvement of software design ability.

1. はじめに

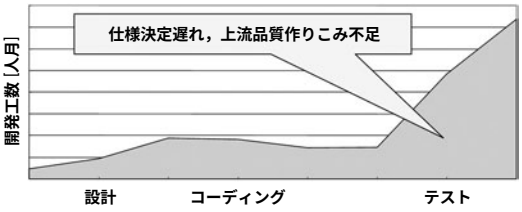
デジタル化・ネットワーク化が進む家電機器では，そのソフトウェア規模が拡大の一途をたどっている。そのような中，ソフトウェア工学手法を導入し，高品質・高生産性を実現することが期待されている。しかし，組込みソフトウェアでは，小規模であった頃の開発文化であるコード中心開発がまだ主流であることが多い。典型的な例に，**第1図**に示すような流用開発と呼ばれる開発形態がある[1)]。これは，以前の機種のソフトウェアを流用し，必要な部分をボトムアップ擦り合わせ的に修正または追加する開発形態である。このような開発形態では，全体を見ない追加・修正による構造の複雑化（構造劣化）や全体を把握できる人材が育ち難いなど，ソフトウェア工学手法の導入を阻むさまざまな問題が内在している。

第2図は，あるソフトウェアの工程ごとの開発工数を模式的に示した図である。仕様決定の遅れや設計段階での完成度不足から，後工程であるテスト工程依存の開発が行われている状況が見て取れる。ソフトウェアの開発力を強化するためには，テスト工程以前の工程での完成度を向上させることが不可欠である。



第1図　コード中心開発（流用開発）

Fig. 1 Code centric development (Clone & own development)



第2図　開発工程別工数

Fig. 2 Man-month of development phase

本稿では，当社において取り組んでいる開発力強化のための施策について述べる。

2. 取り組み施策

テスト工程以前の完成度向上施策として，実装力向上と設計力向上の2つの観点から取り組みを行っている。

2.1 実装力向上施策

コーディング段階で不具合を作り込まない，また可能な限り不具合を発見するために，コーディングスキルの向上およびコード解析ツールの利用を活動の柱としている。

〔1〕全社統一コーディング規約の策定

ソフトウェア規模の拡大と共に，参画するソフトウェ

ア開発者の数が飛躍的に増大する状況になった。それまでの組込みソフトウェア開発では常識であったC言語の安全な使い方が開発者全体に伝わっておらず，数々のコーディングにまつわる不具合がテスト工程で散見されるようになった。しかし，組込みソフトウェア開発におけるC言語の使い方は，属人的なスキルとして存在しているだけで，形式知としてはまとめられていない状況であった。このような状況を受け，2002年度より，以下のような社内の技術者からなる全社横断の検討グループを組織し，C言語コーディング規約の策定を開始した。

- 自社プロセッサ用C言語コンパイラ開発者
- 製品開発でのソフトウェア開発環境構築担当者および品質担当者
- 社内ソフトウェア研修担当者

策定に当たっては，C言語の文法や生成コードなどのコンパイラの視点だけではなく，実際に起こった不具合の事例を反映させ，開発者の立場に立った実際に使える規約とすることに注力した。また，数多くのルールを決めると開発者の負担が重くなり形骸化する恐れがあるため，厳選した95ルールを規定している。規約は事例を重視し，1規約1ページにまとめ，以下の項目から構成している。

- 実際に起こった問題例
- 正しい書き方
- 問題例がなぜ不具合を生むのかの解説

特に，開発者の納得を得るために解説の記載に力を注いでいる。策定されたコーディング規約は，開発のベテランにとっては，C言語の特性を再認識できるとともに，初心者にはノウハウを体得できる実践的な教科書として，社内研修にも活用されている。

C言語版に続き，C++言語コーディング規約やセキュアコーディング規約のような分野ごとのコーディング規約の策定も併せて行っている。

〔2〕静的解析ツールの利用普及

C言語コーディング規約の策定を受け，規約を真に開発の中で活用するためには，規約を遵守する方法を規定する必要がある。すべてを目視で確認することは不可能であり，静的解析ツールを有効に利用することが重要である。このような状況から，規約に規定されているルール群を以下の2つの観点に分類し，コーディング規約の遵守方法を社内標準として文書化した。

- ツールにより確認するルール
- レビューにより確認するルール

ツールによる確認では，市販の主要な静的解析ツールが出力する警告メッセージの意味とコードの修正の方法を解説している。レビューによる確認では，レビューの視点，勘どころをまとめている。通常，静的解析ツールは大量の警告メッセージを出力し，対処に苦慮する場合も多い。そのため，自社のコーディング規約のみチェックするフィルタを市販の主要な静的解析ツールごとに作成している。

このように静的解析ツールを利用することにより，テスト工程で発見される不具合の7 %～10 %がコーディング段階で検出可能となった。

また，筆者らも参加して策定されたIPA/SEC（Information-technology Promotion Agency / Software Engineering Center）発行のコーディング作法ガイド［C言語版］[2] のルールとの対応関係をまとめるなど，業界の動向を取り込んで，継続的なメンテナンスを行っている。

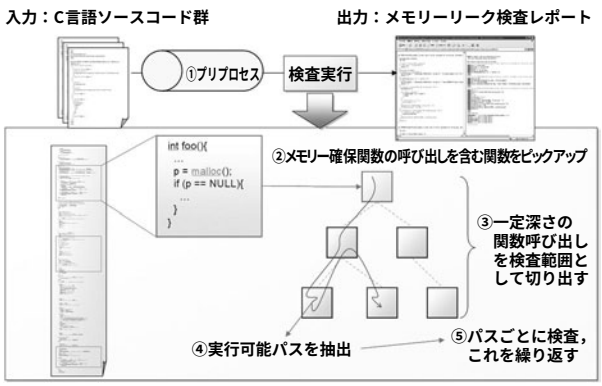
〔3〕パス検査を利用したソースコード解析

家電機器のソフトウェアの規模拡大と共に，未再現の不具合の出現確率が増加し，対策に多大な時間を要するようになっている。特に，動的なメモリーの確保とその開放漏れに起因し，システムダウンなどの深刻な症状を引き起こすメモリーリークと呼ばれる不具合が顕在化している。大規模なソフトウェアでは，非常に多くの実行パスが存在するため，このようなメモリーリークを引き起こす実行パスをテスト時点で発見することは困難になっている。

そこで，最近注目を集めているモデル検査技術を応用し，数百万ステップ規模の大規模ソースコードの中からメモリーの確保と開放にかかわる実行パスのみを抽出し，実装段階でメモリーリークを発見する検査システム（λ trace）の技術開発を行っている[3]。λ traceの動作概要を，**第3図**に示す。以下のような処理を行い，検査結果をHTML（HyperText Markup Language）形式で出力する。

①ソースコードをプリプロセス（gcc（GNU Compiler Collection）など）

②メモリー確保関数を呼び出す関数をピックアップ



第3図 メモリーリーク検出ツール（λ trace）

Fig. 3 Memory leak detection tool (λ trace)

③一定の深さまでで呼び出される関数を検査対象として抽出
④ソースコード検証のためのモデル検査器を利用し，ピックアップした関数を開始点として，実行可能なパスを抽出
⑤抽出されたパスについて検査を実行

モデル検査手法導入に際しては，モデル検査の知識がなくても使用することができることを主眼に置いた。そのため，新たに検査用のモデルを作成することなく，ソースコードに変換を施すことにより，検査用のモデルを自動的に作成することに注力している。

2.2 設計力向上施策

家電組込みソフトウェア開発では，第1章で述べた流用開発が行われている。このような開発形態を取らなければ短納期に対応することができないという側面もあるが，小規模であった頃の開発スタイルの名残と見ることもできる。このような開発スタイルでは，全体設計がおろそかになり構造劣化によるコードの複雑化が進行しやすく，再利用性の低下など一度開発したソフトウェアの活用性が低下し，生産性・品質の低下を引き起こす。ソフトウェアの活用性が低下していることは，経営的な視点で見ると，開発したソフトウェアの資産価値が低いことを意味する。筆者らは，ソフトウェアの資産価値を，**第4図**に示すようにコード指標と設計の可視化指標の2軸で評価している。

【コード指標】：内部を逐次読まないとわからないソースコードではなく，管理単位が明確で全体像がわかりやすく，再利用・テストが容易なソースコードであることを判定

【設計の可視化指標】：第三者が理解する観点で重要な全体を俯瞰（ふかん）するアーキテクチャ設計書の整備状況を判定。この2軸で資産価値は次のように分類できる。

Ⅰ資産価値が低い：コードの質が悪く，設計書が不十分
Ⅱ属人資産：コードの質は良いが，設計書が不十分
Ⅲ半組織資産：設計書は存在するが，コードの質が悪い
Ⅳ組織資産：コードの質が良く，設計書も整備されている

Ⅰは，非常に活用性が低く，早急な対策が必要である。Ⅱは，個人依存で活用ができている状況であり，属人性を排除し組織的な活用を目指す必要がある。Ⅳは，組織的に活用ができている状況であり，この状況を目指すために以下のような施策を推進している。

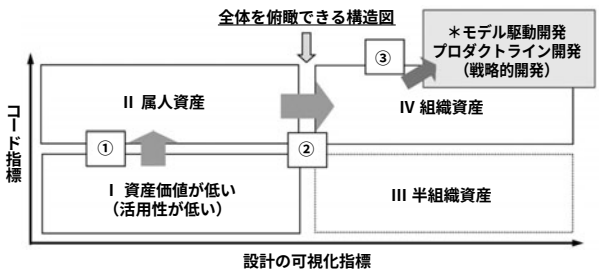
Ⅰ⇒Ⅱ ①資産価値向上のためのリファクタリング
Ⅱ⇒Ⅳ ②アーキテクチャ設計力の向上
Ⅳ⇒＊ ③アーキテクチャ設計からモデル駆動開発へ

これらの活動を円滑に進めるためには，全体を俯瞰した設計力が必要不可欠である。それを牽引（けんいん）する人材であるアーキテクトの育成も同時に進める必要がある。以下，それぞれの活動について説明する。

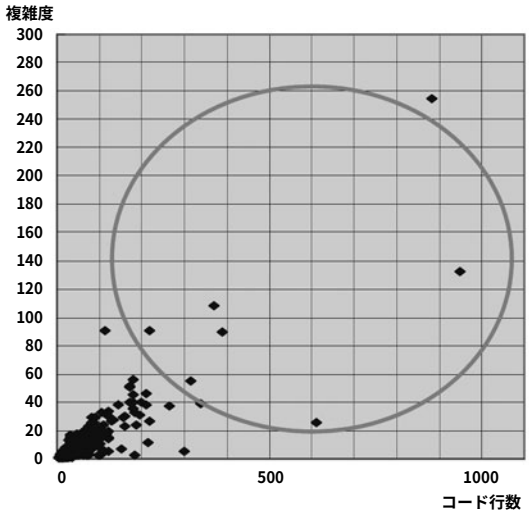
〔1〕資産価値向上のためのリファクタリング

独立性・可読性・テスト容易性の観点から，一般的な静的解析ツールにより計測可能な外部変数の数・関数行数・サイクロマティック複雑度・経路数などをコード指標として規定している。このコード指標の値を参考に，改善のためのリファクタリングを実施するかどうかの決定を行う。リファクタリングの実施については，開発パワーも必要なことから，経営判断が必要な場合が多い。コード指標は，経営判断を仰ぐ際の根拠としても用いられる。

コード指標の使い方の一例を，**第5図**に示す。**第5図**は，関数の行数と複雑度の散布図である。楕円で囲まれた領域にある関数は，複雑度・行数とも突出して大きい。ほとんどのソフトウェアでは，このような突出している関



第4図 ソフトウェアの資産価値
Fig. 4 Software asset quality



第5図 コード指標の例
Fig. 5 Example of code metrics

数の数は限られていることが経験上わかっているので，このような関数から以下の観点に着目しコード改善の検討を開始する。

- if文の深いネスト構造，巨大なループ，巨大なswitch-caseなどを解消するアルゴリズムの改善
- 関数，ファイルの分割
- コードクローンの共通関数化
- 外部変数の見直し　など

特に，外部変数の見直しを進めると構造レベルでの見直しが必要となる場合が多い。この場合は，以下の観点から設計レベルの検討を行う。

- 状態遷移に代表される非同期処理とハードウェア制御などの同期処理を分離
- 機能単位ごとに，関数・モジュールを集約　など

また，先行機種と後継機種の散布図の経年変化から，大きく複雑に変化している関数に着目してリファクタリング対象を絞り込むことも有効な方法である。

このようなリファクタリングの観点・ノウハウは，社内標準として文書化し，周知を図っている。

〔2〕アーキテクチャ設計力の向上

開発現場において，アーキテクチャ設計方法を普及させようとすると以下のような現実的な問題に直面する。

- アーキテクチャ設計について概念的な教科書が多く，具体的な実施方法がよくわからない
- 組込みソフトウェアの特性を反映した設計方法（例えば，トレードオフなど横断的な設計，応答性や低消費電力などの非機能要件に対する設計など）が属人的なスキルに依存し，明確にはなっていない
- 構造など，コードより上位視点での設計に慣れていない開発者が多い

このような状況を考慮し，アーキテクチャ設計とは具体的に何を行うかを周知することを中心に以下のような施策を実施している。

（1）アーキテクチャ設計手順の規定

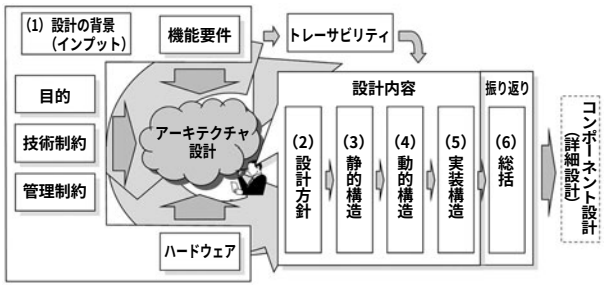
社内の有識者からなる少人数ワーキンググループを組織し，これまで属人的なスキルに依存して実施されてきたアーキテクチャ設計内容を他者にも伝えられるように，**第6図**に示すようにアーキテクチャ設計手順として可視化している。

- 設計の背景（インプット）

設計の入力となる重要な機能要件や各種制約事項を整理する。従来，このような情報は暗黙の了解となっていて，後から設計結果を見た場合，設計の意図が不明になる要因であり，特に重要視している。

- 設計方針

全体に関係する構造の基本的な考え方，制御方針，例外処理方針などを規定する。



第6図　アーキテクチャ設計手順

Fig. 6 Process of software architecture design

- 静的構造

システムをどのようなコンポーネントに分割したか，また，そのコンポーネント間の関係を構造図として記載する。

- 動的構造

静的構造をベースにタスク・スレッドなどの並行実行単位を明確にし，代表的な動作をシーケンス図などで規定する。組込みソフトウェアでは，静的構造と動的構造の両立が重要である。

- 実装構造

静的構造をどのようなファイル構成で実装するかを規定する。

- 総括

設計の入力に対して，アーキテクチャ設計が適切に行われているかを再整理する。この内容が設計書に書かれていれば，マネージャ層は，この部分だけ見れば設計の概要を把握できる。

- トレーサビリティ

複数視点の設計内容の整合を取る方法を規定する。

（2）アーキテクチャ設計書の記載項目・内容の規定

アーキテクチャ設計の最大の成果物は，アーキテクチャ設計書である。（1）で規定したアーキテクチャ設計手順に基づき，設計書の章構成，各章での記載内容，典型的な記載事例をまとめたアーキテクチャ設計のためのガイドラインを策定している。アーキテクチャ設計の中でも重要な静的構造の表記については，以下にあげる理由からUML（Unified Modeling Language）2.0で規定されているコンポジット構造図を用いて記述することにしている。

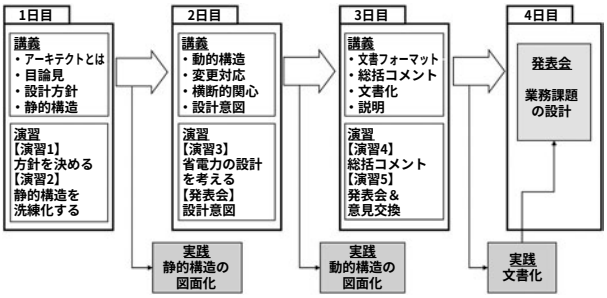
- クラス図より，直截的で理解しやすい
- 構造化設計の視点からも適用が容易
- 将来的には，モデル駆動開発への移行が容易

また，アーキテクチャ設計書に記載すべき内容のイメージが伝わるように，仮想的な製品開発を想定した中規模程度のソフトウェア開発のためのアーキテクチャ設計

書のサンプルも併せて社内標準として規定している。

（3）アーキテクチャ設計研修の実施

（1），（2）で得られたアーキテクチャ設計に関する具体的な実施内容を，開発者に伝えるための研修コンテンツを開発し，人材育成の一環として実施している。研修では，具体的な製品のソフトウェア開発におけるアーキテクチャ設計の一連の流れを体験する。**第7図**に，研修のカリキュラム構成を示す。講義＋演習に加えて，実際の担当業務でのアーキテクチャ設計書の作成が個人学習内容となる実践的な研修内容となっている。受講者からは“今まで漠然としていたアーキテクチャ設計の全体像を理解することができた”など高い評価を得ている。



第7図　アーキテクチャ設計研修カリキュラム

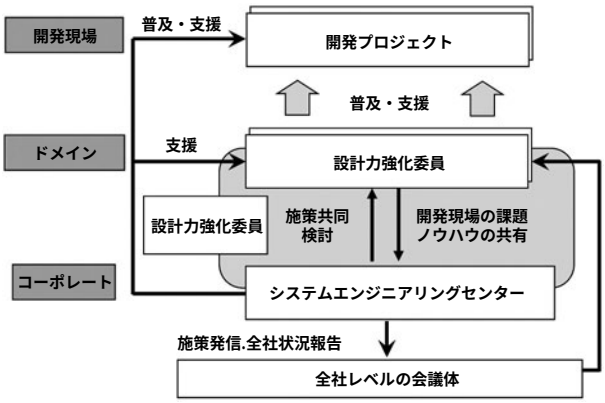
Fig. 7　Training curriculum of architecture design

〔3〕アーキテクチャ設計からモデル駆動開発へ

モデル駆動開発（MDD：Model-Driven Development）は，静的な構造設計と状態遷移という動的な振る舞いの設計を統一的に扱えるところに利点がある。しかし，設計中心の開発であり，トップダウン設計が強制される手法である。コード中心に開発を行っている部署には導入が困難である。一般的なMDDツールは，コンポジット構造図をモデル記述のベースにしているものが多く，**2.2〔2〕**で規定した設計方法を用いていると，容易にMDDへの移行を行うことができる。アーキテクチャ設計が浸透した部署からMDDの導入を進めており，適用事例も増えつつある[4]。また，**第4図**に示すように，アーキテクチャ設計が前提となるプロダクトライン開発に対しても取り組むことができる。

〔4〕全社普及施策

施策の円滑な普及のため，**第8図**のような体制を構築している。ソフトウェア開発力強化に関して，全社の方向性を決定する会議体の傘下にドメインの技術者から構成される設計力強化に関する委員会を組織している。当部署と共に開発現場の課題共有および施策の検討と開発現場への普及・支援を行っている。



第8図　開発手法普及のための体制

Fig. 8　Organization promoting engineering methods

3. まとめ

デジタル化・ネットワーク化によりソフトウェア規模が拡大している家電ソフトウェアの開発力強化の取り組みを実装力の向上施策，設計力向上施策の観点から述べた。施策普及のために最も注力していることは，教科書的な施策ではなく開発実態に即した施策を考え発信していくことである。開発現場に埋もれているノウハウを形式知化し全社に展開することも筆者らの活動の大きなミッションと考えている。

今後は，施策の更なる全社普及を行うと同時に，効率的なテスト設計のあり方，テスト設計とアーキテクチャ設計の融合などテスト視点から見た開発力強化の方策についても検討して行きたいと考えている。

参考文献

1）尾山壮一　他：組込み系ソフトウェア開発の課題分析と提言（社）電子情報技術産業協会 ソフトウェア事業委員会 平成19年度ソフトウェアに関する調査報告書Ⅱ pp.77-127 (2008.3).
2）（独）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編：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］（翔泳社）(2006.5).
3）青島武伸　他：メモリリーク検出システムλtrace　SEC Journal No.11, pp.6-15 (2007.9).
4）南光孝彦：モデル駆動型開発の導入による生産性向上の事例（社）電子情報技術産業協会 平成20年度ソフトウェアに関する調査報告書Ⅱ pp.101-103 (2009.3).

著者紹介

春名修介　　Shusuke Haruna
システムエンジニアリングセンター
System Engineering Center
博士（工学）

松本範久　　Norihisa Matsumoto
システムエンジニアリングセンター
System Engineering Center

古山寿樹　　Hisaki Furuyama
システムエンジニアリングセンター
System Engineering Center

中川雅通　　Masamichi Nakagawa
システムエンジニアリングセンター
System Engineering Center
博士（工学）