情報処理学会論文誌 Vol.57 No.2 730–744 (Feb. 2016)

組込み向け進化型ソフトウェアの効率的な拡張性強化手法

佐々木 隆益[1,4] 吉岡 信和[2,3] 田原 康之[4] 大須賀 昭彦[4]

受付日 2015年5月25日, 採録日 2015年11月5日

概要：市場ニーズの変化速度に対応した製品開発プロセスとしてアジャイル開発プロセスのようなライトウェイトな開発プロセスが台頭してきている．アジャイルも含め，このような進化型ソフトウェアは，継続的にリファクタリングを施しながらの機能開発を行う．特に組込みシステムでは，関連するハードウェアの個別進化に対しても，システムとしての対応が必要であり，拡張性や可変性の設計がなされていないと，製品リリースが進むにつれ，メンテナンス性の悪化を引き起こしやすくなる．本稿では継続的に発生する将来の機能追加において，システムの拡張性を強化する情報の提供を目的とし，ソースコードから，自動的に拡張性の構造を抽出し，拡張性を強化すべき箇所を特定，強化方法を提示する手法を提案する．さらに継続的に機能追加された実際の製品コードに対し，提案手法を適用した結果，提案手法の有効性を確認することができた．

キーワード：組込みシステム，アジャイル，拡張性，レガシーコード

An Automatically Improvement of Extensibility Method for Software Evolution of Embedded System

Takanori Sasaki[1,4] Nobukazu Yoshioka[2,3] Yasuyuki Tahara[4] Akihiko Ohsuga[4]

Received: May 25, 2015, Accepted: November 5, 2015

Abstract: Light weight development processes like Agile have emerged in response to rapidly changing market requirements. However software evolution processes including Agile are inadequate for software in embedded systems, as software undergoes frequent refactoring, targeting only immediate requirements. As a result maintainability decreases because the system is not designed to respond to changes in the associated hardware. In this paper, we propose a method for improving of extensibility. We also propose a technique for detecting and suggesting extensible design pattern automatically. Our approach is based on analyses of the call graph and the inheritance structure of source code to identify a layer structure that is specific to embedded software. These techniques provide us with objective and quantitative information about extensibility. We applied the proposal method to an actual product's code continuously and could verify an improvement in system's extensibility.

Keywords: embedded system, agile, extensibility, legacy code

1. はじめに

近年，技術革新スピードが加速し，ネットワークの高速化，センサの小型化，コンピュータチップの高性能化にともない，モバイル，クラウド，AR 等，新しい概念の製品が登場してきている．これにより，ソフトウェアシステムはますます複雑化し，ソフトウェア開発に要するコストが増加する一方，市場のニーズは短期間で変化し，技術の陳腐

[1] キヤノン株式会社
Canon Incorporation, Ohta, Tokyo 146–8501, Japan
[2] 国立情報学研究所 GRACE センター
GRACE Center, National Institute of Informatics, Chiyoda, Tokyo 101–8430, Japan
[3] 総合研究大学院大学
SOKENDAI, Hayama, Kanagawa 240–0193, Japan
[4] 電気通信大学
University of Electro-Communications, Chofu, Tokyo 182–8585, Japan

化，急速な製品価値低下が起きている．そのため，綿密な計画に基づき，仕様，設計，実装，テストという開発工程により，製品を大量生産するという，従来から用いられているプロセスでは，早ければ開発中に，遅くとも製品後すぐにマーケットニーズと製品仕様との乖離が生じ，製品を出荷しても，製品計画段階で予想した販売数や価格帯が維持できず，開発に要したコストの回収さえ困難になっている．

そこで，ウォーターフォールに代表されるヘビーウェイトな開発プロセスではなく，アジャイルのようなライトウェイトな開発プロセスが台頭してきた [1]．このプロセスでは，将来を予測した機能や機構について，あらかじめ綿密な設計を行わず，まずはシンプルな機能，機構で開発を完了し，マーケットへの早期製品投入を実現する．そして，マーケットからのフィードバックを得た後，追加開発を行う．このプロセスにより，市場の要求と製品機能のギャップが低減されつつある．

このようなプロセスは進化型のソフトウェア開発プロセスに分類され，開発済みのソフトウェアに対し，リファクタリングを施しながら，次のリリースに必要な機能の追加開発を行う [1]．つまり継続的な進化の実現には，長期的なメンテナンス性の向上が不可欠である．

しかしながら，従来行われてきた，フィーチャ分析 [2] を基にした可変性設計手法 [2], [3] を用い，製品の可変点を確定してから，ロードマップに従い多様な要件に対応したソフトウェアシステムを構築していくプロダクトライン型開発の適用は難しい．なぜなら，実際の開発現場において，設計仕様の作成コストが大きく，また設計仕様の抽象度が高いため，実装と直接的に記述が一致しないことや，実装中の要件変更により，設計仕様と実装仕様が乖離してしまうため，機能追加時には，要件や設計仕様と，既存ソースコードとのトレーサビリティがとれなくなるからである．

また，アジャイル開発では，要求に対して直接的な成果物を作成するため，局所的な正解となるアーキテクチャを選択していく可能性があり，長期的な視点においては，歪んだアーキテクチャ構造になる可能性や，トータル開発コストの増大をまねく場合もある．これを防ぐため，アジャイル開発においても，機能追加が容易な拡張性の高い仕組みを設計することは，重要であるとされる．しかしながら，必要な機能の実装だけが優先されるなか，将来の拡張性を考慮して設計することは，設計者のスキルに依存した属人性の高いものになる．また，設計者が拡張性の高さを考慮していたとしても，継続的に発生するソフトウェア進化において，どのタイミング，どの要件で拡張性を強化すべきかの判断が難しい．

特に組込みシステムでは，関連するハードウェアの進化に対する柔軟な設計がなされていないと，製品リリースが進むにつれ，メンテナンス性の悪化を引き起こしやすくなる．なぜなら組込みソフトウェアと連携するハードウェアは動作制約を持っていることが多く，コストや納期の観点からソフトウェアによるアドホックな変更を誘発するからである．たとえば，特定の順序による初期化が必要なハードウェアや，フォーマットやプロトコルに限定があるハードウェア等である．

また，拡張性の問題等，一般的にリファクタリング対象個所を特定する手法としてメトリクスを用いた手法があり，後藤ら [4] は，メソッド長に関するコードメトリクスを用いた方法を提案している．Ramanath ら [5] は，代表的なメトリクスである CK メトリクスを用いた不具合箇所特定との関連について検証している．メトリクスを用いたアプローチでは，定義した問題を自動で定量化することはできるが，プログラムの意図も含めた解釈は難しく，プログラムの機能，開発者，言語等，様々な要因で値が変化してしまう．そのため，相対的な比較によって，問題の可能性を指摘するにとどまる．一方，Fowler [6] は，コードの記述により特定する方法や，“怠け者クラス”，“疑わしき一般化” 等のコードスメルを定義している．コードの記述については，特定コードの単純な変換であり，Eclipse 等の統合開発環境ツールにも実装されているものがある．しかしながら，これは，局所的にコードを改良するだけであり，構造全体の問題点を特定しているわけではない．また，コードスメルについては，概念として示したものとなり，具体的な特定方法は，実際のコードを開発者が解読し，開発者のスキルに依存して特定することになる．

我々は，ソースコード上から可変性構造を抽出する研究 [7] を，これまで実施してきた．本稿では，さらに進化型ソフトウェアの開発プロセスにおいて，長期的なメンテナンス性向上のために，属人性を排除した定量的な拡張性強化手法を提案する．

我々の手法は，概念的なコードスメル情報ではなく，特定の構造を持つソースコードの意味的な構造をふまえ，コードスメルをルールという形で定義し，自動検出を可能にしている．これにより，拡張性を改善すべき箇所の特定・ガイドを行うものであり，2 ステップで構成されている．（1）ソースコード内のコールグラフと継承構造から，拡張性構造を識別する，（2）拡張性を強化すべき箇所を特定し，拡張性の強化方法を提示する．さらに，この手法を一部ツール化し，実際の製品開発における複数回の機能改良コードに適用することで有効性の評価を行う．

以降，2 章では本稿が対象とする組込みシステムの課題について説明し，3 章で拡張性強化手法について提案する．そして，4 章で，実際の製品コードを用いた実験結果を示し，5 章で本提案手法の妥当性について評価する．6 章では関連研究を説明し，7 章で本稿をまとめる．

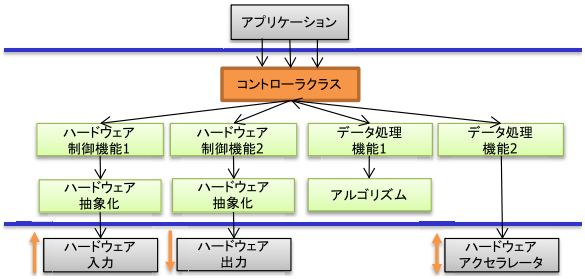

図 1 組込みシステムにおける基本的レイヤ構造
Fig. 1 Basic Layer Structure in Embedded System.

2. 組込みシステムソフトウェアの課題

2.1 本稿の対象ソフトウェア

本稿では，ハードウェアの仕様が確立していないような新規製品，市場のニーズが見えないような新規分野，ハードウェアの多様性および進化に追従しなくてはならないカスタムメイドな特徴を持ちつつ，それらの品質についても重要視される組込みシステムを対象としている．たとえば，ウェアラブルコンピュータ，ヒューマノイドロボット，インターネットに常時接続するネット家電，家やオフィスにおける機器間協調システム，先進安全機能の組込みが加速している自動車等であり，従来の製品単独の仕様では完結しない新しい組込みシステムといえる．

また，対象としている組込みシステムの構造は，**図 1** に示すレイヤ構造で表現可能なものとする．

図 1 は，概念的なレイヤ構造モデルであり，システムには，機能シーケンスの制御を担うコントローラクラスが 1 つ以上存在し，コントローラクラスから呼び出されている各クラス群が，各々システム機能の一部を担っている．各機能には，機能の呼び出し，機能の実装，ハードウェアの抽象化，アルゴリズムが含まれる．また機能には，ハードウェアの制御を主な役割とするハードウェア制御機能とデータ処理を主な役割とするデータ処理機能がある．システムが扱うデータは，入力用のハードウェアからコントローラクラスを経由し，アルゴリズムで処理後，再びコントローラクラスを経由し，出力用のハードウェアへ流れる．

図 1 で表現可能な組込みシステムの具体例としては，たとえばネットワークカメラである場合，規定されたフレームレートで，ハードウェアであるカメラから映像が入力され，映像解析処理を行った後，ネットワーク経由で，映像および解析情報を外部へ送信する．さらに外部からのコマンド入力を受け付け，回転デバイスに対し，カメラ向き等の変更出力を行うこともできる．このような回転デバイスや，カメラの仕様等は開発時期や機種によって異なる．また，たとえば，ロボットである場合，カメラからの画像や，距離センサからの距離情報を入力とし，周囲の障害物検知処理結果をふまえ，加速度センサや，ジャイロセンサにより動作制御を行いながら，各関節等の駆動出力を行う．ロボットでは，ハードウェア構成によって，関節数や，動作範囲，動作箇所も異なる．このように，組込みシステムでは，同じ種類のソフトウェアであっても，ハードウェア構成に依存する箇所が多いため，依存箇所を削減することで，ソフトウェアを変更不要，または容易に変更できるようにすることは非常に重要である．

なお，対象としている組込みシステムは，オブジェクト指向言語を用いて実装されたプログラムとする．

2.2 課題

対象としている組込みシステムでは，市場の変化が速いため，時間をかけ製品仕様を確定させ，手戻りなく製品化を行うウォーターフォール型の開発プロセスよりも，アジャイルのようなライトウェイトな方法をとることで，余分な機構のための実装を加えず，変化する要求に対応しながら，早期に市場投入実現を目指すことが多い．

上記のようなプロセスの場合，抽象度の低いコードを対象とするため，システムの品質や将来的なメンテナンス性が，開発者個人に依存してしまうという課題がある．本稿では，下記に示す 2 つの属人性による課題を対象とする．

課題 1：システムの拡張性に対する問題箇所の認識が異なる

現在のコードに，新しい機能を追加するだけでよいのか，もしくは，拡張性を強化した後，機能追加をした方がよいのかについては，開発者の経験に依存していた．これにより，必要以上に冗長な拡張性の仕組みのある箇所や，変更頻度が高い場所であるのに，変更しにくい構造になっていることがあった．

課題 2：拡張性の強化方法が異なる

拡張性をどのように強化するかについては，開発者の設計・実装スキルに依存していた．また，一般的なリファクタリング手法では，実装コード記述に対しての対処法が示されるだけであり，課題 1 が解決され，拡張性の問題箇所が分かったとしても，対処方法については，開発者ごとに異なるため，本当に強化されるのか，またどの程度の規模による変更が必要なのか把握が困難であった．

なお，本稿では，既存システムへ新しい機能を追加できることや，新しいデバイスに対応できるようなソフトウェアの特性を拡張性と定義する．さらに，拡張性の問題とは，だれが，どこで，どのデータ関係にアクセスするかという観点に対し，メンテナンスコストを増大させる要因であると定義する．また，その問題が生じるクラス構造を拡張性に関する問題個所として定義する．

3. 拡張性強化分析

3.1 提案手法概要

図 2 に示すように，提案手法は，拡張性に問題のあるコードに対し，自動的に拡張性を強化すべき箇所の特定と

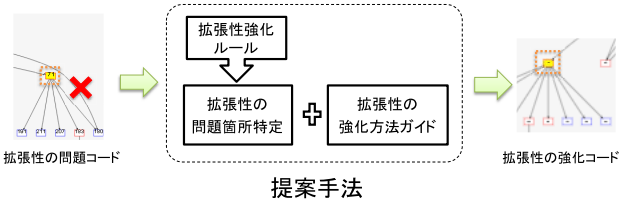

図 2 提案手法の概要
Fig. 2 Proposal overview.

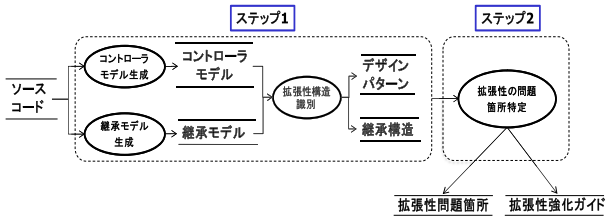

図 3 提案手法の構成図
Fig. 3 Architecture of proposed method.

強化方法のガイドを行う手法である．

提案手法は，図 1 に示した構造的特徴を持ったシステムを対象としているため，主にハードウェアの抽象化，処理アルゴリズムの抽象化，およびコントローラとの結合に対する拡張性の問題を特定する．

開発者は，自身の担当機能が拡張性を強化すべき箇所として特定された場合に，拡張性の強化ガイドに沿って拡張性の強化を図ることができる．

3.2 提案手法の構成

図 3 に示すように，提案手法は 2 ステップで構成されている．

まず，属人性を排除するため，入力はソースコードのみとした．拡張性の仕組みを定量的に把握するため，我々の従来研究手法 [7] を用い，クラスの呼び出し関係と継承関係という 2 つの異なる観点でソースコードを解析し，各モデル内に含まれるデザインパターン，継承構造の識別を行う（ステップ 1）．

次に，拡張性の問題構造のパターンを定義したルールとステップ 1 で識別した拡張性の構造を比較することにより拡張性の問題箇所を自動的に特定し，各ルールに対応した拡張性の強化ガイドを出力する．（ステップ 2）．

3.3 拡張性構造識別（ステップ 1）

3.3.1 機能の抽出

ソフトウェアの変更頻度や変更内容は機能ごとに異なるため，機能を抽出し，機能単位で拡張性を判定する必要がある．たとえば，ハードウェアの入出力機能であれば，使用しているセンサごとに進化速度が異なるため，当該機能については，頻繁に変更を要することになる．またアルゴリズムであれば，製品価値に直結するものほど市場の競争が激しく，処理内容の大きな変更を求められる場合がある．

機能を抽出するには，まず，ソースコードを解析し，コントローラモデル（Controller Model）と継承モデル（Inheritance Model）を生成する [7]．そして，コントローラモデルにおいて，コントローラクラスから呼び出されている各クラス以下を各機能として特定する．なお，コントローラクラスから呼び出されているクラスのうち，継承モデルで，同一の継承ツリーに含まれるクラスについては同一機能とする．

表 1 継承による拡張性構造
Table 1 Extension structures in terms of inheritance.

機能＼階層	第一階層	第二階層	第三階層以下
機能A	ー	SI	SI
機能B	MI	SI	MI
機能C	–	–	–

表 2 デザインパターンによる拡張性構造
Table 2 Extension structures in terms of design pattern.

機能＼階層	第一階層	第二階層	第三階層以下
機能A	–	–	–
機能B	CU	AF	ITF
機能C	–	–	–

3.3.2 継承による拡張性の構造識別

コントローラモデルの各階層に含まれるクラスに対し，機能ごとの継承構造タイプを自動的に抽出する．**表 1** に抽出例を示す．SI は単純継承構造，MI は多階層継承構造を示す [7]．なお，図 1 に示したコントローラクラスからの呼び出し距離が等しいクラス群を同一階層であるとし，コントローラクラスに近い階層を第 1 階層と呼ぶ．コントローラクラスから遠い階層ほど階層の数字が大きくなる．

図 1 におけるレイヤ構造を持つソフトウェアを対象とするため，上位層ほど，デバイスやアルゴリズムの制御に関する類似構造の存在を示し，下位層ほど，データタイプやハードウェアデバイスが複数存在することを示している．

3.3.3 デザインパターンによる拡張性の構造識別

コントローラモデルの各階層に含まれるクラスに対し，機能ごとのデザインパターンを自動的に抽出する．**表 2** に抽出例を示す．CU は，Mixed Creation and Use パターン，F は Factory パターン，AF は Abstract Factory パターン，P は Plugin Factory パターンを示している [7]．ITF は Inverse Template Method パターンとして，本稿で我々が定義したパターンであり，3.4.4 項に示すテンプレートメソッドパターンの逆パターンで用いられる構造である．

階層の表現については，表 1 と同様である．上位層ほど，デバイスやアルゴリズムの制御に関する隠蔽構造の存在を示し，下位層ほど，データタイプや，ハードウェアデバイスに関する隠蔽構造の存在を示している．

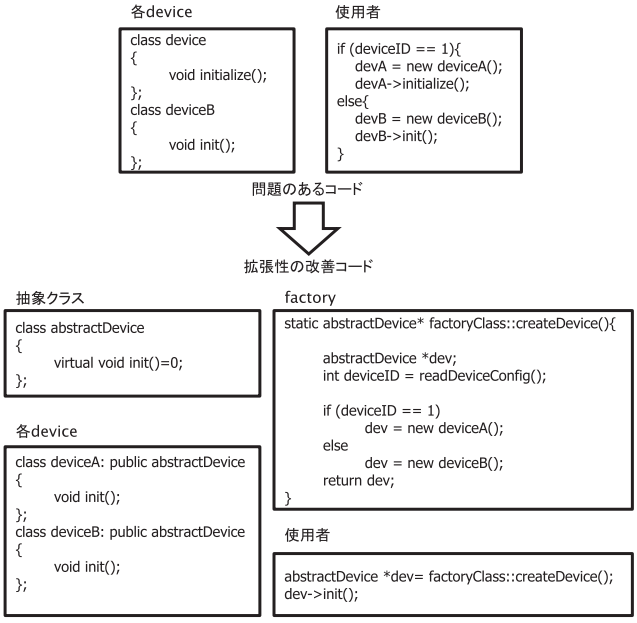

図 4 Factory 化によるソースコード改善例
Fig. 4 Example of improving code by using factory.

3.4 拡張性の問題箇所の特定および拡張性強化ガイド（ステップ 2）

組込みシステムにおいては，拡張性とパフォーマンス（リソース消費や実行速度）のバランスが重要であるため，パフォーマンスの仕様によっては，拡張性は犠牲になることがある．しかしながら，組込みシステムでは，PC アプリケーションやモバイルアプリケーションとは異なり，一般的に，ソフトウェアの更新が容易ではなく，また長期にわたって利用されることが多い．そのため，開発時点では存在しないデータフォーマットや，デバイスへの対応が容易であることが望ましい．

図 4 に拡張性の問題改善例を示す．

問題のあるソースコードは，デバイスごとに異なるインタフェースを持つ具象クラスを，使用者が生成条件を知ったうえで選択するコードとなっている．このようなコードでは，使用者はデバイスが変わるたびに制御側のコードも変更する必要性が生じ，また個々のデバイスのインタフェースに依存したコードになる．

改善したコードでは，デバイスの抽象クラスを作成し，各デバイスは抽象クラスを継承する構造にすることで，統一したインタフェースを提供することができる．また，Factory クラスにデバイスの生成条件を隠蔽することで，使用者は，デバイス制御における論理的なコード記述に特化することができる．このような，拡張性に関する開発現場の状況をふまえ，我々が定義した拡張性の強化ルールを**表 3** に示す．ステップ 1 で抽出した拡張性の構造識別結果に対し，表 3 に示した拡張性の強化ルールを適用することで，当該拡張性の強化ルールに反するような拡張性の問題箇所を，自動で特定することができる．

表 3 拡張性の強化ルール
Table 3 Rules for improving extensibility.

ID	判定ルール	対応強化ガイド
R1	クラスの生成処理を使用者から隠蔽する	G1
R2	隠蔽された具象クラスにアクセスしない	G2/ G3
R3	同一デバイスに関する生成操作は一箇所に隠蔽する	G4
R4	テンプレートメソッドの逆パターンを排除する	G5/ G6
R5	明示的なインタフェースを導入する	G7

表 4 拡張性の強化ガイド
Table 4 Guidelines for improving extensibility.

ID	拡張性強化方法
G1	Factory階層の挿入
G2	抽象インタフェースへの変更
G3	上位クラスの使用箇所を下位クラスへ移動
G4	使用箇所の集約およびAbstract Factoryの導入
G5	テンプレートメソッド化
G6	委譲関係への変更
G7	抽象インタフェースの導入

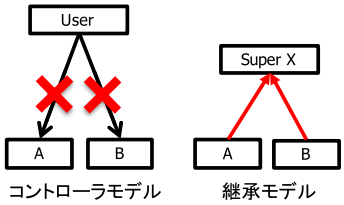

図 5 R1 に反するモデル内の構造例
Fig. 5 Example of R1.

表 4 に拡張性強化ガイドを示す．拡張性の強化ガイドは，拡張性の強化ルールごとに，拡張性の強化方法を関連付けたものである．開発者は強化ガイドに沿って，拡張性の問題箇所として特定されたコードの拡張性を強化することができる．

次項からは，各拡張性強化ルールによる拡張性改善効果，問題個所の特定方法および強化ガイドによる強化方法を強化ルールごとに示す．

3.4.1 クラスの生成処理を使用者から隠蔽する（R1）

図 5 に示した具象クラス A，B を使用するクラス User が，A，B の生成処理や，生成される具象クラスに直接依存しない構造とすることで，使用者側を変更せず新しい具象クラスの追加が可能になる．特に，使用者がコントローラクラスである場合は，巨大化，複雑化しやすいため，使用者にとって不必要な情報を分離することで，メンテナンス性の劣化を防ぐことができる．

当該ルールは，表 1 に示した継承による拡張性構造の第 1 階層に継承構造がある場合に特定される．また，表 2 に示したデザインパターンによる拡張性の構造において，CU と識別されたクラスの 1 階層下に表 1 で示した継承による拡張性構造がある場合（第 3 階層以下では各層で判断）も特定される．

拡張性ガイドとしては，上記どちらの場合においても，

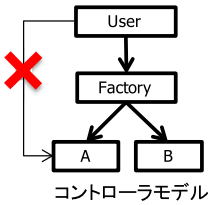

図 6 R2 に反するモデル内の構造例
Fig. 6 Example of R2.

“Factory 階層の挿入（G1）” が有効である．これは，図 5 の Super X を返す Factory クラスを作成し，作成した Factory を使用者側で呼び出すように変更を行うことで，コントローラモデルの第 1 階層もしくは，CU のクラス階層の下に Factory 階層を挿入する方法である．

3.4.2 隠蔽された具象クラスにアクセスしない（R2）

図 6 で示した Factory 層で隠蔽された具象クラス A，B を使用している Factory 層の上位レイヤにある User クラスから，具象クラス A，B への呼び出しを削除することで，失われていた隠蔽効果を取り戻すことができる．

当該ルールは，表 2 に示したデザインパターンによる拡張性の構造において，Inverse Template Method（ITF）パターン以外のデザインパターンが識別されたクラスにおける 1 階層下の継承構造を持つクラスが，前記，拡張性構造を持つクラスの上位階層に位置づけられているクラスから呼び出されている場合に特定される．

拡張性ガイドとしては，“抽象インタフェースへの変更（G2）”，または “上位クラスの使用箇所を下位クラスへ移動（G3）” が有効である．上位クラスで具象クラスを呼び出している箇所が，抽象インタフェースに含まれるべき具象クラスの機能であれば，当該機能を抽象インタフェースへ加え，上位階層では，Factory で取得する抽象インタフェースを利用するよう変更する方法である．一方，上位クラスで具象クラスの機能を利用している処理自体が具象クラスに依存する内容であれば，この処理自体を具象クラスの責務として移動し，使用側は処理結果だけを取得するように変更する方法である．

3.4.3 同一デバイスに関する生成操作は 1 カ所に隠蔽する（R3）

一般的にデバイスには，デバイス依存の設定や制御が存在する．システムに備わっているデバイスが DevA か DevB であるかを，これらデバイス依存処理部が個別に判断し，切り替えているモデルを図 7 に示す．このように，デバイスの選択処理が，システム内に分散した場合，デバイス追加時やデバイスの選択条件が変更されたときに，すべての上記箇所に対しても変更が必要となる．そのため，当該ルールでは，各デバイス固有の具象クラスに関する処理をシステム内の 1 カ所に集約・隠蔽することによって，デバイスの変更に対する耐性を向上させる．

当該ルールは，継承モデルで，複数の継承ツリー内の具象クラス名に共通性がある場合に特定される．

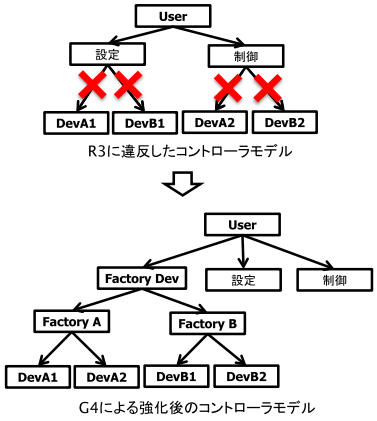

図 7 R3 に反するモデル内の構造と強化後の構造例
Fig. 7 Example of R3.

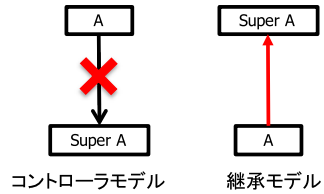

図 8 R4 に反するモデル内の構造例
Fig. 8 Example of R4.

拡張性ガイドとしては，“使用箇所の集約および Abstract Factory の導入（G4）” が有効である．デバイスごとの Factory と Factory を抽象化した Abstract Factory を作成し，各デバイス依存処理部が持っていたデバイスの選択処理を移動させる．そして，各デバイス依存処理部では，抽象化されたデバイスを使用するよう変更する方法である．

3.4.4 テンプレートメソッドの逆パターンを排除する（R4）

一般的に継承構造においては，親クラスと，その派生クラスとは置換可能である必要がある．そのため，親クラスでは，処理シーケンスを定義し，派生クラスにおいて，各処理の詳細を変えるようなテンプレートメソッドパターンが使われる．しかしながら，図 8 に示すように，派生クラス A で親クラスである Super A のメソッドを呼び出すことで，親クラスが想定していない振舞いを派生クラスが行うという実装も可能である．この場合，派生クラスが親クラスの実装に依存するため，親クラスの変更によりシステムが予期しない動作となる可能性がある．

当該ルールは，図 8 に示すように，継承モデルにおいて，クラス A は，親クラス Super A クラスの派生クラスであり，コントローラモデルにおいて，派生クラス A が親クラス Super A を呼び出している場合に特定される．この構造が Inverse Template Method（ITF）パターンであるため，このパターンが識別されると，当該ルールに違反したこととなる．

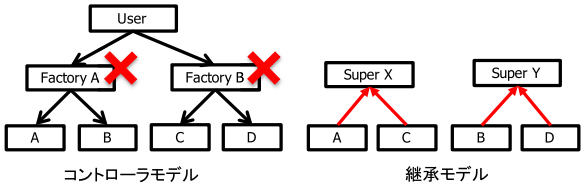

図 9 R5 に反するモデル内の構造例
Fig. 9 Example of R5.

なお，このパターンにおいては，親クラスの初期化だけを行っているものは含めない．

拡張性ガイドとしては，“テンプレートメソッド化（G5）”または “委譲関係への変更（G6）” が有効である．親クラスに純粋仮想関数を追加し，派生クラスにおいて，当該関数に派生クラスごとの実装を記述するというテンプレートメソッドを導入するか，親クラスと派生クラスとの継承関係ではなく，委譲の関係に変更する方法である．

3.4.5 明示的なインタフェースを導入する（R5）

暗黙的な共通のインタフェースに依存し呼び出される複数のクラスに対し，明示的なインタフェースの導入を行う．**図 9** に示すように，Factory A，B により，具象クラス A，B，C，D の生成を隠蔽している場合，Factory の呼び出しクラス User からは，共通インタフェースで Factory の呼び出し処理を行えるように，Factory の抽象クラスを定義することで，Factory の担う役割を明確にすることができる．また，共通処理を集約することも可能になる．

当該ルールは，表 2 に示したデザインパターンによる拡張性の構造において，CU と識別されたクラスの 1 階層下に表 1 に示した継承による拡張性の構造がない場合（第 3 階層以下では各層で判断）に特定される．

拡張性ガイドとしては，“抽象インタフェースの導入（G7）” が有効である．各クラス間の共通メソッドを持つ抽象クラスを作成し，当該抽象クラスを派生させた Abstract Factory パターンを導入する方法である．

3.5 解析ツール

本提案手法では，解析工程の一部について解析ツールを作成した．作成した解析ツールは，解析対象のソースコードを入力として，ステップ 1 で使用するコントローラモデル図と継承モデル図を出力する．他のデータについては，出力された情報を基に，手動ではあるが，機械的に算出することができる．

また，本ツールは，対象組込みシステムのドメインに依存した処理を持っていないため，オブジェクト指向言語であれば使用可能である．

4. 実験および結果

キヤノン株式会社の製品のうち，新規市場向けに開発を行っている製品の一部コード（C++）に対し，提案手法（ステップ 1～ステップ 2）を実施した事例について説明する．

表 5 対象コード一覧
Table 5 List of object code.

ID	内容
1	初期製品コード
2	ID1 のコードへの機能追加
3	ID2 のコードへの機能追加
4	ID3 のコードを提案手法により，拡張性改良

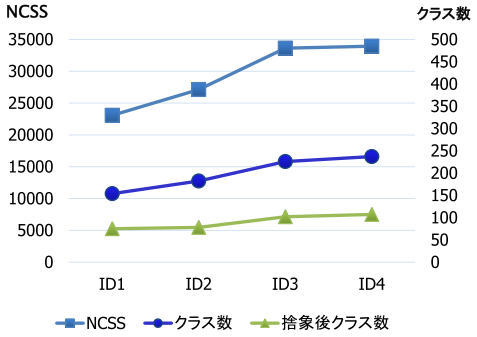

図 10 コード規模とクラス数の推移
Fig. 10 Progress of code size and number of classes.

4.1 対象コードの概略

表 5 に，当該実験で使用したソースコードの一覧を示す．各コードを ID1～4 で区別する．ID1～3 は，同一製品に対する継続的な 3 回のリリースにおいて開発されたコードであり，ID4 は，当該手法を用い ID3 のコードを改良したものである．

各 ID におけるコードのステートメント行数（NCSS），クラス数，および提案手法で求められたコントローラモデルで表現されるクラス数（捨象後クラス数）を**図 10** に示す．

機能改良が進むにつれコード行数が増加し，それにともないクラス数も増加している．しかしながら，提案手法を用いることで，42%～48%程度に捨象したクラス数を対象としてシステムを観察することが可能である．

4.2 拡張性構造識別（ステップ 1）

図 11 に各 ID のコードで抽出した機能数を示す．機能数の増加は，コントローラモデル内のコントローラクラスから呼び出されているクラス数の増加を示している．

図 12 に示したコントローラクラスの NCSS およびクラス結合度（CBO [8]）の推移を示す．ID1～ID3 へと開発が進むにつれ，コントローラクラスのコードが増加し，さらに結合クラス数も増加していることが分かる．

ID4 のコードでは，拡張性の強化箇所に対策を施すことで機能数が 13 へと減少した．ID3 と ID4 では機能的な違いはないが，ID3 では，コントローラクラスとの結合が増加したため，実際よりも機能が多く見えた．

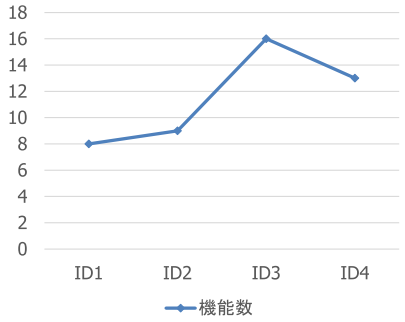

図 11 機能数の推移
Fig. 11 Change of number of features.

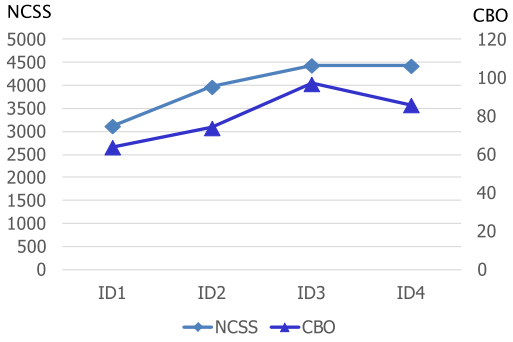

図 12 コントローラクラスにおける変更推移
Fig. 12 Change of data in controller class.

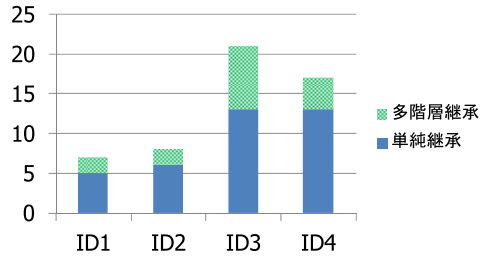

図 13 継承構造に関する拡張性構造の推移
Fig. 13 Change of extension structures in terms of inheritance.

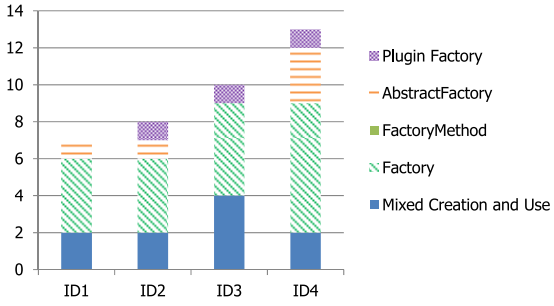

図 14 デザインパターンに関する拡張性構造の推移
Fig. 14 Change of extension structures in terms of design pattern.

図 13 は各 ID のコードに含まれる継承構造に関する拡張性構造の識別結果推移を示す．ID3 コードにおいて，継承構造の増加がみられる．

図 14 は各 ID のコードに含まれるデザインパターンに関する拡張性構造の識別結果推移を示す．ID3 コードにおいて，Mixed Creation and Use パターンの増加がみられる．

提案手法により，ID4 のコードでは，ID3 で増加した Mixed Creation and Use パターンの削減と，Factory パターンと Abstract Factory パターンの増加がみられる．

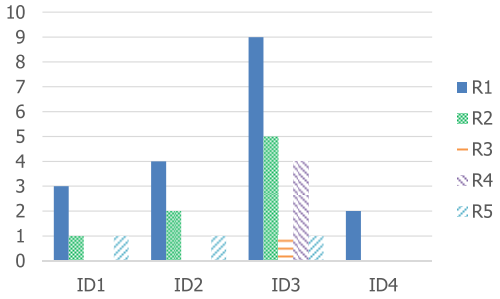

図 15 拡張性の問題箇所の特定結果推移
Fig. 15 Changes in results when extensibility reinforcement points are identified.

表 6 拡張性ルールごとの追加および修正クラス数
Table 6 Number of additional classes and modification classes.

ID		拡張性強化方法	追加クラス	修正クラス
R1	G1	Factory階層の挿入	6	1
R2	G2	抽象インタフェースへの変更	0	12
	G3	上位クラスの使用箇所を下位クラスへ移動		
R3	G4	使用箇所の集約およびAbstract Factoryの導入	4	2
R4	G5	テンプレートメソッド化	0	14
R5	G7	抽象インタフェースの導入	1	4
合計			11	33

4.3 拡張性の問題箇所特定および強化ガイド（ステップ 2）

表 3 に示した 5 つのルールに基づいた拡張性の問題箇所の特定結果推移を**図 15** に示す．“クラスの生成処理を使用者から隠蔽する（R1）” と “隠蔽された具象クラスにアクセスしない（R2）” については，ID1～ID3 にかけて単調増加している．機能追加にともない，R1 と R2 に関する拡張性低下が起こりやすいものと思われる．一方，“同一デバイスに関する生成操作は 1 カ所に隠蔽する（R3）” や，“テンプレートメソッドの逆パターンを排除する（R4）” は ID3 のみで特定されている．R3 と R4 は，機能追加要件に依存して拡張性低下が起こりやすいものと思われる．

ID3 の拡張性を強化した ID4 のコードでは，強化すべき箇所が減少している．

また，ID3 で，問題を検出したルールに対応した拡張性強化ガイドに基づき，ID3 のコードの拡張性を強化するに要した追加および修正クラス数を**表 6** に示す．

R1 と R3 に関する強化においては，新しい隠蔽構造の構築が主な強化項目であるため，修正クラス数に比べ，追加クラス数が多く，その他のルールでは，修正クラス数の方が多い．

また，ID3 のコードに対し，18 カ所の拡張性強化を行った（ID4 での残箇所を除いた図 15 に示す R1～R5 の合計値）．この 18 カ所の拡張性強化には，コントローラモデル

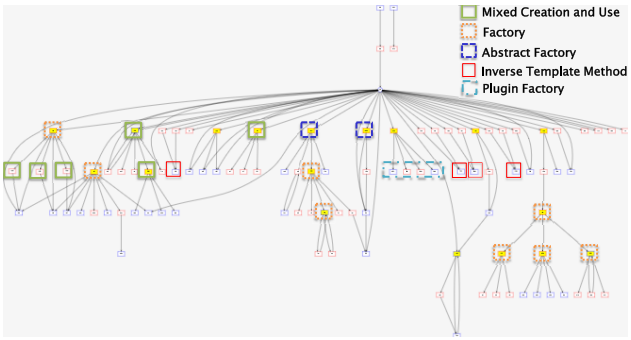

図 16 コントローラモデル（ID3）
Fig. 16 Controller model (ID3).

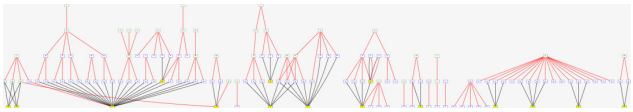

図 17 継承モデル（ID3）
Fig. 17 Inheritance model (ID3).

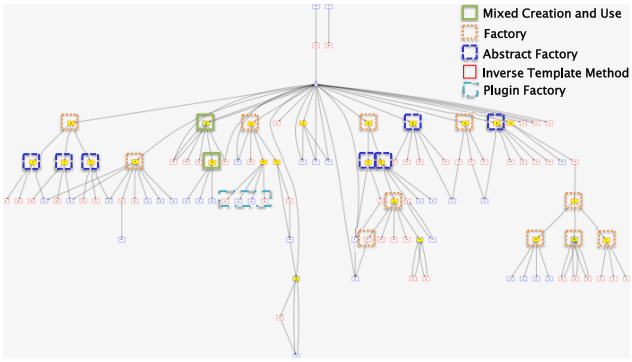

図 18 コントローラモデル（ID4）
Fig. 18 Controller model (ID4).

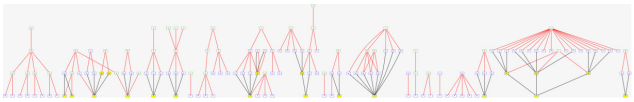

図 19 継承モデル（ID4）
Fig. 19 Inheritance model (ID4).

の半数程度に及ぶクラス数の追加または修正が必要となった．**図 16** および**図 17** に拡張性強化前である ID3 のコントローラモデルと継承モデルを示し，**図 18** および**図 19** に拡張性強化後である ID4 のコントローラモデルと継承モデルを示す [7]．4.3.1 項～4.3.5 項に，ID3 と ID4 における拡張性の問題箇所特定と拡張性ガイドについての具体例を述べる．

4.3.1 クラスの生成処理を使用者から隠蔽する（R1）

ID3 において，9 カ所の拡張性改善箇所が特定されている（図 15）．拡張性ガイドの G1 に沿って，9 カ所のうち，3 カ所については，Factory クラスを追加．4 カ所については，生成内容が組合せによるものであるため，1 つの抽象 Factory クラスを作成し，抽象 Factory クラスを継承した 2 つの具象 Factory クラスを作成することで，Abstract Factory 化を行った．残る 2 カ所については，後述するが，誤検知であった．

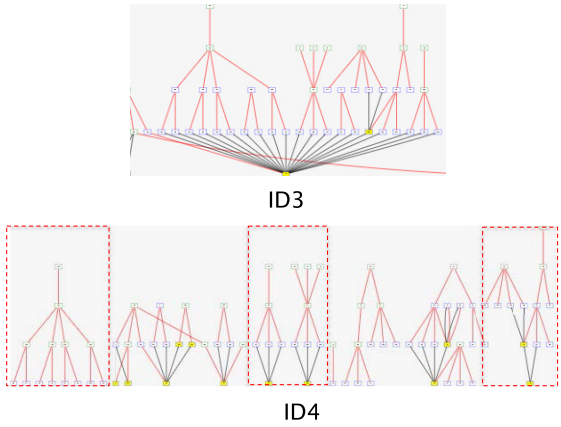

図 20 R1 の拡張性強化による継承モデルの変化
Fig. 20 Change of inheritance model by rule R1.

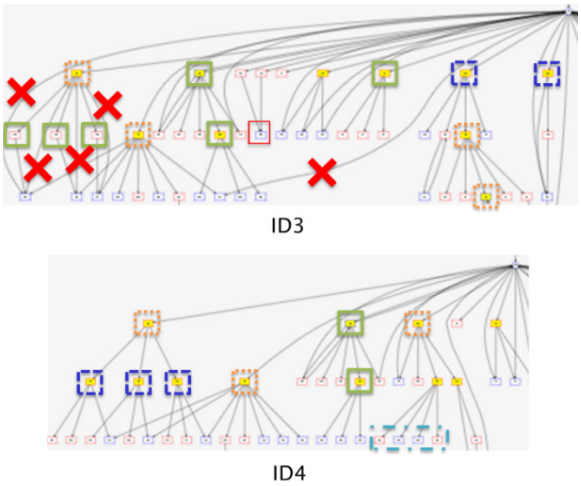

図 21 R2 の拡張性強化によるコントローラモデルの変化
Fig. 21 Change of controller model by rule R2.

図 20 に，拡張性強化により，変化した継承モデルを示す．ID3 においては，コントローラクラスから複数の継承ツリーが呼び出されていたが，ID4 では，継承ツリーごとに Factory 階層を設けることで，破線で示した部分に分割され，コントローラクラスへの集中がなくなっている．

4.3.2 隠蔽された具象クラスにアクセスしない（R2）

ID3 において，5 カ所の拡張性改善箇所が特定されている（図 15）．拡張性ガイドの G2 に沿って，上位クラスで具象クラスを利用していた箇所について，すべて抽象クラスの利用へ変更し，かつ抽象インタフェースで呼び出すために，具象クラスの複数メンバ関数を抽象化した．さらに，G3 に従い，抽象クラスや，具象クラスへ上位クラスで定義されていた処理を移動した．**図 21** に拡張性強化における，コントローラモデルの変化を示す．ID3 のコントローラモデル上に × 印で示した上位層のクラスからの呼び出し箇所が ID4 のコントローラモデル上では削除されている．

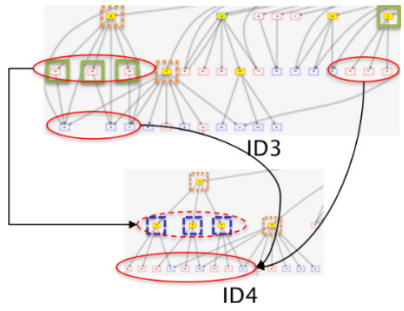

図 22 R3 の拡張性強化によるコントローラモデルの変化
Fig. 22 Change of controller model by the rule of R3.

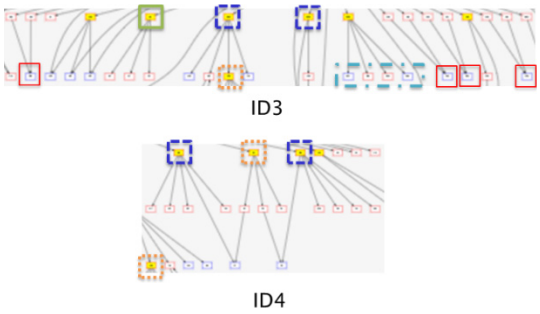

図 23 R4 の拡張性強化によるコントローラモデルの変化
Fig. 23 Change of controller model by the rule of R4.

4.3.3 同一デバイスに関する生成操作は 1 カ所に隠蔽する（R3）

ID3 において，1 カ所の拡張性改善箇所が特定されている（図 15）．拡張性ガイドの G4 に沿って，システム内に分散していた同一デバイスに関したクラス生成判断処理を 1 カ所に集約し，Abstract Factory を導入した．

図 22 に拡張性強化における，コントローラモデルの変化を示す．ID3 のコントローラモデルに存在した，デバイス種類に依存しクラスが切り替える 3 カ所の生成処理対象を楕円の実線で示す．ID4 のコントローラモデルにおいては，楕円の実線で示した 1 カ所に生成処理対象が集約している．また，上位クラスが Mixed Creation and Use パターンから，楕円の鎖線で示した Abstract Factory パターンに変化している．

4.3.4 テンプレートメソッドの逆パターンを排除する（R4）

ID3 において，4 カ所の拡張性改善箇所が特定されている（図 15）．拡張性ガイドの G5 に沿って，Inverse Template Method パターン（ITF）として識別されたクラスに純粋仮想関数を追加し，派生クラスから呼び出されていた親クラスのメンバ関数内で当該純粋仮想関数を呼び出す処理へと変更した．一方，派生クラスでは，親クラスの関数を呼び出すのではなく，純粋仮想関数の定義を記述するテンプレートメソッドパターンの形態へと変更した．

図 23 に拡張性強化における，コントローラモデルの変化を示す．

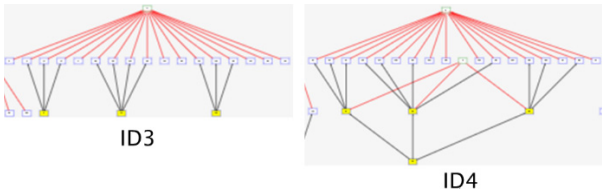

図 24 R5 の拡張性強化による継承モデルの変化
Fig. 24 Change of inheritance model by rule R5.

表 7 機能追加比較結果
Table 7 The comparison of changes by adding feature.

変更タイプ		シミュレーション例	対応ルール	比較項目	ID3	ID4	差異
デバイスの追加	類似デバイス追加（メーカ違い等）	機能11にクラスを追加	R2	追加クラス数	1	1	上位層へのデバイス隠蔽能力
				修正クラス数	4	2	
	新規デバイス追加（方式違い等）	機能7にクラスを追加	R1	追加クラス数	1	1	コントローラへの変更影響
				修正クラス数	1	1	
	設定方式が異なるデバイス追加	機能11にデバイス操作クラス、設定クラスを追加	R3	追加クラス数	3	4	具象デバイスに対する変更影響の局所化
				修正クラス数	3	2	
デバイスの多機能用途化		複数機能からの呼びだし追加	ー	追加クラス数	3	3	なし
				修正クラス数	2	2	

ID3 のコントローラモデル上には，細い実線で示した Inverse Template Method パターン（ITF）クラスが検出されているが，ID4 のコントローラモデルでは，Inverse Template Method パターン（ITF）が検出されない．

4.3.5 明示的なインタフェースを導入する（R5）

ID3 において，1 カ所の拡張性改善箇所が特定されている（図 15）．拡張性ガイドの G7 に沿って，3 つの既存 Factory クラスから共通インタフェースを抽出し，抽象 Factory クラスを作成した．さらに，既存 Factory クラスを抽象 Factory クラスの派生クラスとする Abstract Factory パターン化を実施した．

図 24 に拡張性強化における，継承モデルの変化を示す．ID3 の継承モデルでは，同一の継承ツリーに対して，3 つのクラスが呼び出しを行っているが，ID4 の継承モデルでは，ID3 の 3 つのクラスがさらに継承構造を持ち，他のクラスから呼び出されている構造になっている．

4.4 拡張性の強化前後における機能追加時の変更量比較

本節では，提案手法による拡張性の強化前後における機能追加時の変更量を比較する．

拡張性強化前後のコードに対し，同一の機能追加実施時における追加・修正クラス数を変更量とし，今後予想される当該製品における機能追加要求を機能追加内容とする．

拡張性強化前のコードを ID3 の製品コード，拡張性強化後のコードを ID4 のコードとし，今後予想される当該製品における機能追加要求については，開発者へのインタビューで 4 タイプに分類された機能追加要求を得た．**表 7** に機能追加要求ごとの比較結果を示し，表 7 の類似デバイス追加時における変更内容を**図 25** に示す．

ID3 では，デバイスの追加により，四角の実線で示された新しいクラスがコントローラモデル上に追加される．また，このデバイスは既存デバイスの類似デバイスであるた

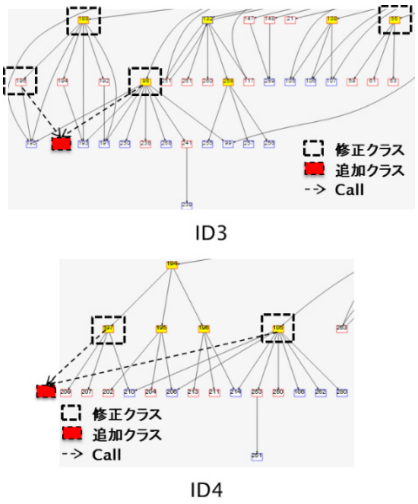

図 25 類似デバイス追加時の変更内容
Fig. 25 Change for adding similar device.

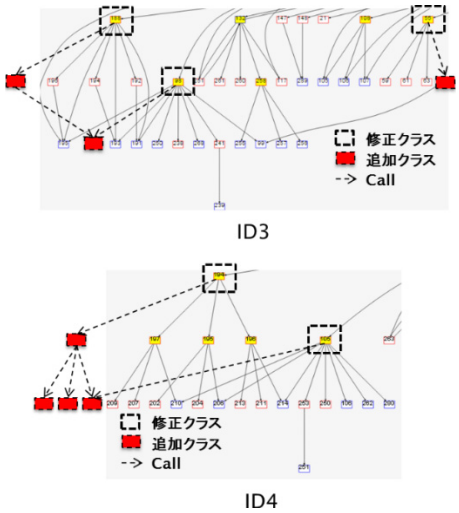

図 26 設定方式が異なるデバイス追加時の変更内容
Fig. 26 Change for adding device which has different setting way.

め，設定等については，流用できるものとしたが，ID3 では，上位クラスで既存の具象クラスを呼び出しているため，新しく追加するデバイスについても同様に呼び出しが必要となる．そのため，4 カ所の破線で示すクラスに対し修正が必要となる．一方，ID4 では，R2 の拡張性ルールに違反した箇所を修正しているため，ID3 とは異なり，Factory より上位層のクラスに対する修正が不要となる．

表 7 の設定方式が異なるデバイス追加時の変更内容を**図 26** に示す．ID3 では，図 25 に追加したクラスに加え，デバイスごとにデバイスの設定クラスと操作クラスがシステム内部で分散し追加される．さらに，追加されたインスタンスにおけるすべての上位クラスを修正する必要も生じる．一方，ID4 では，R3 の拡張性ルールに違反した箇所を修正しているため，デバイスに関するクラスすべてが 1 カ所に集約され追加される．しかしながら，ID4 では，ID3 とは異なり生成クラスを使用側へ隠蔽するための Factory クラスを追加する必要がある．そのため ID4 では，追加が必要となるクラス数は，ID3 よりも多くなるが，既存クラスに対する修正対象を少なくすることが可能であり，また，分散して記述する必要がない分，可読性にも優れている．

表 7 の新規デバイス追加における ID3 と ID4 の違いは，R1 の拡張性ルールに違反した箇所に対する，コントローラクラス以下への Factory 階層の導入である．このケースにおいて，追加・修正されるクラス数は同じとなるが，ID3 は複雑化しやすいコントローラクラスに変更が必要となり，ID4 ではクラスの生成に特化した Factory への変更となるため，変更の容易さ，変更ミスの予防に違いがある．

表 7 に示すデバイスの多機能用途化については，ID3 と ID4 に違いが現れなかった．よって本稿で定義した拡張性強化ルールでは，当該要件に対する拡張性の問題個所を検出することはできない．

5. 評価

提案手法の妥当性について評価を行った．評価観点としては，提案手法による課題の解決，拡張性強化の効果，ルールの十分性，適用範囲とした．

5.1 提案手法による課題解決能力

課題 1 に対しては，4.3 節に示したように，提案手法によって，開発者によらず拡張性の問題箇所を自動的，かつ定量的に特定することができる．しかしながら，拡張性の強化を実施した ID4 では 2 件の拡張性の問題箇所が残った．この 2 件についても，拡張性を強化すべきとして特定された箇所であったが，上位に対し隠蔽を要する箇所ではなく，各々のインスタンスをつねに要する箇所であった．クラスの呼び出し関係と継承関係に着目している本提案手法においては，可変性や拡張性なのか，類似のインスタンスなのかという判定が難しく，誤検知となるケースも存在する．

課題 2 に対しては，4.3 節に示したように，拡張性の問題箇所として特定された箇所のうち，誤検知であった 2 件を除き，すべてに対して拡張性強化ガイドに従い，拡張性を強化することができた．拡張性強化ルールとそれに対応した拡張性強化ガイドにより，開発者のスキルにかかわらず，同じ拡張性の仕組みを導入することができる．

5.2 提案手法による拡張性強化効果の有無

4.4 節に示したように，デバイス追加を目的とした変更に対しては，提案手法で，拡張性を強化しておくことで効果がある．一方，デバイスの多機能用途化を目的とした変更に対する効果はない．しかしながら，提案手法はルールベースであるため，新しい拡張性の問題パターンとし，拡

表 8 各観点の起こりうるケースと対応ルール
Table 8 Possible cases and rules.

観点	ケース	対応ルール
使用者	知るべきクラスのみにアクセス	拡張性問題なし
	アクセスしているクラスがない	拡張性構造なし
	隠蔽された具象クラスにアクセス	R2
	不必要に抽象クラスへアクセス	R4
場所	生成処理を一箇所で隠蔽	拡張性問題なし
	生成処理なし	拡張性構造なし
	生成処理を複数箇所で隠蔽	R3
	生成処理が隠蔽されないレイヤ	R1
データ	類似点・継承関係のある生成データ	拡張性問題なし
	類似点・継承関係のない生成データ	拡張性構造なし
	単一の生成データ	拡張性構造なし
	類似点はあるが，継承関係のない生成データ	R5

張性強化ルールを追加することで，対応は可能である．

また，4.4 節より得られた，提案手法による拡張性強化の効果を下記に示す．

- 追加・修正が必要となるクラス数の削減
- 複雑になりがちなコントローラクラスに対する変更を削減することによる，変更容易性向上や，可読性悪化の防止
- 既存クラスに対する変更を削減することによる修正時のミスによる不具合防止，テスト工数の削減

さらに，4.4 節は拡張性の問題箇所 1 カ所ごとについて比較した結果であるが，図 15 で示したように，実際のコードでは，多くの拡張性の問題箇所を含むため，さらに大きな効果があると考えられる．

現実においては，拡張性の問題箇所を正確に把握することができない状態で，機能追加が繰り返されていることが多く，拡張性を強化するためには大規模な改修をともなうことになり，実質的に変更不可能となってしまうと考えられる．このため，提案手法で継続的，自動的にコードから拡張性の問題箇所を示すことは，将来にわたって余計なメンテナンスコストの増加を防止するものであるといえる．

5.3 拡張性強化ルールの十分性

使用者（だれが），場所（どこで），データ（どのデータ関係）という拡張性の問題観点で，起こりうるケースを**表 8** に示す．

提案手法では，拡張性構造が存在しない場合と，拡張性構造上の問題が存在しない場合は，何も検出されない．しかしながら，拡張性構造の必要性は，製品要件に依存するため，双方ともに問題として扱う必要がないと考える．

上記以外の現状想定しているケースについては，すべてに対応したルールが存在している．このことから拡張性強化ルールは，対象としている拡張性の問題観点の範囲において網羅的であるといえる．

5.4 提案手法の適用範囲

提案手法は，図 1 のような構造を持ったソフトウェアについて適用することができる．図 1 の構造とは，拡張性を考慮したレイヤ設計がされているものであり，たとえば，アプリケーションレイヤ/コントローラレイヤ（制御レイヤ）/ハードウェアデバイスの抽象化レイヤに分割されているような構造である．

表 9 組込みシステム領域以外の比較対象 OSS
Table 9 Comparison OSS except embedded software area.

ID	内容
A	データベースソフトウェア
B	画像比較ソフトウェア
C	ムービープレーヤソフトウェア

表 10 各 OSS の計測データ比較
Table 10 Comparison of measurement data in each OSS.

ID	NCSS	クラス数	捨象後クラス数	捨象率
A	54755	593	110	81.5
B	47549	168	37	78.0
C	62675	212	60	71.7

図 27 コントローラモデル（ID A）
Fig. 27 Controller model (ID A).

表 9 に示す種類の OSS を用い，組込みシステム領域以外のソフトウェアに提案手法を適用した結果について比較する．

表 10 には，コードのステートメント行数（NCSS），クラス数，提案手法で生成したコントローラモデルに表現されるクラス数（捨象後クラス数），およびその捨象率を示す．

表 10 に示すように，比較対象のソフトウェアは，5 万行前後となる部分を抽出したものである．

また，ID A〜C のソフトウェアに対し，提案手法で生成したコントローラモデルを**図 27**，**図 28**，**図 29** に示す．

ID A（データベース）や ID B（画像比較ソフトウェア）では，リアルタイムの制御や，コントローラ層/デバイス層等のレイヤ分割が存在しないことにより，図 1 に示した組込みシステムのレイヤ構成を前提にした構成理解ができないため，適用が困難である．一方，ID C（ムービプレーヤ）では，図 1 に示した組込みシステムのレイヤ構成に相対的に似た形状を示している．内部処理として，組込みシステムにも存在するようなデータのリアルタイム再生部，各種データファイルごとの処理可変部，データごとの設定機能

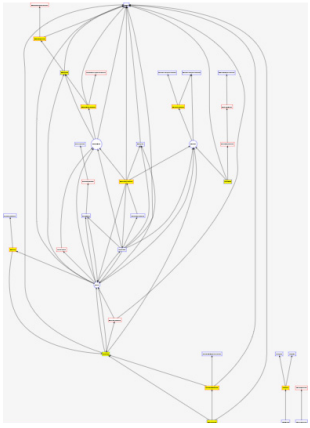

図 28 コントローラモデル（ID B）
Fig. 28 Controller model (ID B).

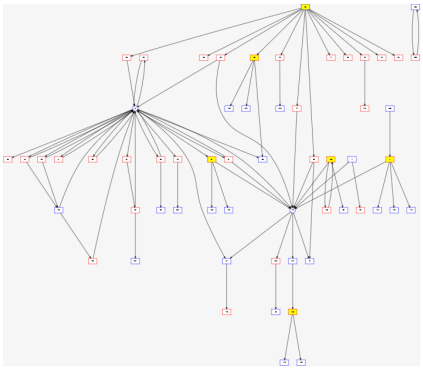

図 29 コントローラモデル（ID C）
Fig. 29 Controller model (ID C).

等があるためであるが，ハードウェア部分がないため，独立した 3 つの小さなシステムであるととらえてしまうことになる．また，本稿で対象とした組込みシステムでは，提案手法によるクラスの捨象率が 50%程度であったが，表 9 のような，図 1 に示した組込みシステムのレイヤ構成を持たないソフトウェアでは，捨象率が 70%を超える値となった．これは，拡張性を考慮したレイヤ構造がないことに加え，各種ハードウェアデバイスの切替え構造がないためであると考えられる．このような対象においては，提案手法が適用できないために，たとえば，隠蔽構造について評価することができず，機能追加時に，予期しない変更箇所も含め，多数の箇所への変更コスト増，変更の確認コスト増，さらには変更漏れによる不具合の発生も想定される．

5.5 提案手法の適用領域拡大

本提案では，特定の条件における組込みシステムに限定した開発領域について述べたが，他システムについても，提案手法の適用可能性を検討する必要がある．

また，IPA の調査報告書 [9] によると，組込みシステムにおいては，C 言語の使用率が 60.3%と最大ではあるが，C++/C#/Java 言語についても 29.7%の割合で使用されているため，本稿では，設計者のセマンティクスを読み取るために，クラス構造や，継承，委譲等の仕組みを持つオブジェクト指向言語 [10], [11] で記述されたコードを対象とした．さらに今後は，オブジェクト指向言語を用いた開発よりも，非オブジェクト指向言語を用いた開発の方が相対的に多いことから，非オブジェクト指向言語への拡張も検討する必要がある．たとえば，非オブジェクト指向言語から構造の意図を理解する仕組みとして，関数インタフェースレイヤによる内部処理の隠蔽，グローバル変数へのアクセスや，構造体の類似性，関数ポインタの切替えを利用することにより，オブジェクト指向言語を対象とした本稿での提案手法を拡張することが可能ではないかと考える．

6. 関連研究

Bansiya ら [10] はオブジェクト指向デザイン要素をデザインメトリクスや品質属性へマップした．また Hudli ら [11] は様々なオブジェクト指向メトリクスを評価した．このように，オブジェクト指向言語に対しては様々なデザインや，メトリクスによる評価手法が提案されている．本稿は，拡張性という観点でソースコードから設計を評価する手法である．

Walkinshaw ら [12] は，オブジェクト指向の言語で記述されたソースコードに対し，landmark method と barrier method を用い，コールグラフを分割することで，ユーザが指定した機能と関連する箇所を抽出するアプローチを提案している．また，このアプローチでは，状態爆発を起こさないように不必要なパスの削減も実施している．本稿では，コールグラフを分割するのではなく，可変点の構造に着目し，コール関係と継承構造を用いて，機能箇所の抽出を行っている．また同様に可変点に着目することで，不必要なクラスの削減を行っている．

Keepence ら [13] は，feature-oriented domain analysis で使われる the mandatory, alternative, and optional と closely resemble なデザインパターン定義を行っている．

本稿では，可変メカニズムの実装形態によって，変更の隠蔽程度が異なるという観点でデザインパターンを整理した．

Makkar ら [14] では，継承構造と再利用性について，継承ツリーの深さに着目して述べられている．継承ツリーが深くなりすぎると，再利用性が低下する．彼らは，その閾値は 3 レイヤ以内だと述べている．それゆえ，継承の深さと再利用性について定式化し，計算可能にした．本稿では，可変メカニズムと関連した箇所の継承ツリーに対して，再利用性が高いとされる 2 階層以内の構造に着目している．

Claudia ら [15] では，可変メカニズムの評価手法を提案している．述べられているように，可変メカニズムの評価観点として，基本的に実装方法，選択方法，切替えタイミングがある．本稿では，実装方法をソースコード上から自動的に検出している．

Babar [16]，Bengtsson ら [17]，Castaldi ら [18] は，プロダクトラインアーキテクチャの評価手法について述べており，著者らが述べているようにシナリオベースによる評価手法は確立している．本稿では，既存コードからのアプローチを提案している．

Hattori ら [19] は，実際の変更に対する影響を分析する手法の評価を行っている．本稿では，変更される前に，変更に対する機能ごとの影響を評価することができる．

Kaur ら [20] は，パッケージに着目し，コードサイズ，複雑度等のメトリクスを用いて算出する保守性指数により，保守性を数値化し，比較可能にしている．このような総合的な数値では，本稿で述べたような，拡張性構造等の具体的な実装構造を得ることはできない．

Munro [21] は，コードの不吉な匂いをモデル化し，コードメトリクスを用いて，不吉な匂いの自動特定を行っている．使用されるメトリクスは，コード行数，複雑度等のメトリクスであり，規定された基準値を，これらメトリクスが超えると当該コードの改善を促す．本稿では，拡張性に特化し，機能ごとに拡張性の構造を評価することで，拡張性の問題個所を自動特定し，強化方法を提示する．

7. おわりに

本稿では，継続的に進化する組込みシステムにおいて，拡張性強化手法を提案した．また本提案手法を複数回の機能追加が行われた実製品コードに適用し，拡張性を強化すべき箇所を自動的に抽出し，拡張性強化ガイドに沿って，拡張性を強化することができた．さらに，提案手法による拡張性の強化効果を検証し，効果があることを確認できた．本提案手法により，アジャイル開発プロセス等において，属人性を排した，効率的な拡張性強化を継続的に実施することが可能となる．

今後は，複数の組込みシステムへ本提案手法を適用した結果についての比較検証や，拡張性強化ルールの拡充を行う必要がある．

謝辞 本研究は，JSPS 科研費 24300005，26330081，26870201 の助成を受けたものである．ソフトウェア工学分野における国立情報学研究所と電気通信大学との連携に基づき，研究の機会と議論・研鑽の場を提供，およびご指導いただいた国立情報学研究所/東京大学本位田 真一教授をはじめ，活発な議論と貴重なご意見をいただいたトップエスイーの皆様に感謝いたします．また，製品のソースコードを提供いただいたキヤノン株式会社に感謝の意を表します．

参考文献

[1] Ghanam, Y., Andreychuk, D. and Maurer, F.: Reactive Variability Management Using Agile Software Development, *The international conference on Agile methods in software development*, Orlando, pp.27–34 (2010).

[2] Kang, K., Lee, J. and Donohoe, P.: Feature-Oriented Product Line Engineering, *IEEE Software*, Vol.19, No.4, pp.58–65 (2002).

[3] Rajasree, M.S., Janaki, R.D. and Jithendra, K.R.: Pattern Oriented Approach for the Design of Frameworks for Software Productlines, *Proc. International Workshop on Product Line Engineering: The Early Steps: Planning, Modeling, and Managing*, pp.65–70 (2002).

[4] 後藤 祥，吉田則裕，藤原賢二，崔 恩瀞，井上克郎：メソッド抽出リファクタリングが行われるメソッドの特徴調査，コンピュータソフトウェア，Vol.31, No.3, pp.318–324 (2014).

[5] Ramanath, S. and Krishnan, M.S.: Empirical analysis of ck metrics for object-oriented design complexity: Implications for software defects, *IEEE Trans. Softw. Eng.*, Vol.29, No.4, pp.297–310 (2003).

[6] Fowler, M.: *Refactoring: Improving The Design of Existing Code*, Addision-Wesley (2000). 児玉公信（訳）：リファクタリング：プログラミングの体質改善テクニック，ピアソンエデュケーション (2000).

[7] Sasaki, T., Yoshioka, N., Tahara, Y. and Ohsuga, A.: Evaluation of Flexibility to Changes Focusing on the Variable Structures in Legacy Software, *11th Joint Conference, JCKBSE 2014*, Volgograd, Russia, p.15 (2014).

[8] Sheldon, F.T., Jerath, K. and Chung, H.: Metrics for maintainability of class inheritance hierarchies, *Journal of Software Maintanance: Research and Practice*, Vol.14, No.3, pp.147–160 (2002).

[9] IPA 独立行政法人情報処理推進機構：2012 年度「ソフトウェア産業の実態把握に関する調査」調査報告書．

[10] Bansiya, J. and Davis, C.G.: A hierarchical model for object-oriented design quality assessment, *IEEE Trans. Softw. Eng.*, Vol.28, No.1, pp.4–7 (2002).

[11] Hudli, R.V., Hoskins, C.L. and Hudli, A.V.: Software Metrics for Object-oriented Designs, *Proc. IEEE International Conference on Computer Design: VLSI in Computers and Processors*, Cambridge, MA, pp.492–495 (1994).

[12] Walkinshaw, N., Roper, M. and Wood, M.: Feature Location and Extraction using Landmarks and Barriers, *Proc. International Conference on Software Maintenance*, pp.54–63 (2007).

[13] Keepence, B. and Mannion, M.: Using Patterns to Model Variability in Product Families, *IEEE Software*, Vol.16, No.4, pp.102–108 (1999).

[14] Makkar, G., Chhabra, J.K. and Challa, R.K.: Object oriented inheritance metric-reusability perspective, *International Conference on Computing, Electronics and Electrical Technologies*, Kumaracoil, pp.852–859 (2012).

[15] Claudia, F., Andreas, L. and Thomas, S.: Evaluating Variability Implementation Mechanisms, *Proc. International Workshop on Product Line Engineering: The Early Steps: Planning, Modeling, and Managing*, pp.59–64 (2002).

[16] Babar, M.A.: Evaluating Product Line Architectures – Methods and Techniques, *14th Asia-Pacific Software Engineering Conference*, p.13 (2007).

[17] Bengtsson, P., Lassing, N., Bosch, J. and van Vliet, H.: Analyzing software architecture for modifiability, *HK/R Research Report* (2000).

[18] Castaldi, M., Inverardi, P. and Afsharian, S.: A case study in performance, modifiability and extensibility analysis of a telecommunication system software architecture, *Proc. 10th IEEE Int'l Symp. on Modeling,*

Analysis, & Simulation of Computer & Telecommunications Systems, pp.281–290 (2002).
[19] Hattori, L., Guerrero, D., Figueiredo, J., Brunet, J. and Damasio, J.: On the precision and accuracy of impact analysis techniques, *Proc. 7th International Conference on Computer and Information Science*, Portland, Oregon, pp.513–518 (2008).
[20] Kaur, K. and Singh, H.: Determination of Maintainability Index for Object Oriented Systems, *ACM SIGSOFT Software Engineering Notes*, Vol.36, No.2, pp.1–6 (2011).
[21] Munro, M.J.: Product Metrics for Automatic Identification of "Bad Smell" Design Problems in Java Source-Code, *2005, 11th IEEE International Symposium on Software Metrics*, Como, Italy, pp.15–15 (2005).

佐々木 隆益 （学生会員）

1999 年広島大学大学院工学研究科博士前期課程修了．修士（工学）．同年三菱プレシジョン（株）入社．2008 年よりキヤノン（株）に勤務．2013 年より，国立電気通信大学大学院情報システム学研究科博士後期課程在学．ソフトウェア工学，フォーマルメソッド，クラウドコンピューティングの研究・開発に従事．

吉岡 信和 （正会員）

1998 年北陸先端科学技術大学院大学情報科学研究科博士後期課程修了．博士（情報科学）．同年（株）東芝入社．2002 年より国立情報学研究所に勤務，2007 年より国立情報学研究所 GRACE センター/総合研究大学院大学准教授．セキュリティ・プライバシソフトウェア工学，ソフトウェア工学，学術クラウドの研究・開発に従事．2015 年より IEEE CS Japan Chapter 役員．日本ソフトウェア科学会理事を歴任．電子情報通信学会，日本ソフトウェア科学会，人工知能学会，IEEE CS 各会員．

田原 康之 （正会員）

1966 年生．1991 年東京大学大学院理学系研究科数学専攻修士課程修了．同年（株）東芝入社．1993～1996 年情報処理振興事業協会に出向．1996～1997 年英国 City 大学客員研究員．1997～1998 年英国 Imperial College 客員研究員．2003 年国立情報学研究所着任．2008 年より電気通信大学准教授．博士（情報科学）（早稲田大学）．エージェント技術，およびソフトウェア工学等の研究に従事．日本ソフトウェア科学会会員．

大須賀 昭彦 （正会員）

1981 年上智大学理工学部数学科卒業．同年（株）東芝入社．同社研究開発センター，ソフトウェア技術センター等に所属．1985～1989 年（財）新世代コンピュータ技術開発機構（ICOT）出向．2007 年より，電気通信大学大学院情報システム学研究科教授．2012 年より，国立情報学研究所客員教授兼任．工学博士（早稲田大学）．主としてソフトウェアのためのフォーマルメソッド，エージェント技術の研究に従事．1986 年度情報処理学会論文賞，2013 年度人工知能学会研究会優秀賞受賞．IEEE Computer Society Japan Chapter Chair，人工知能学会理事，日本ソフトウェア科学会理事，同学会監事等を歴任．電子情報通信学会，人工知能学会，日本ソフトウェア科学会，電気学会，IEEE Computer Society 各会員．