学位論文

進化型組込みソフトウェアにおける
継続的なメンテナンス性向上手法に関する研究

A Method for Improving Maintainability
in Embedded Software Evolution

博士（工学）

電気通信大学大学院
情報システム学研究科

佐々木 隆益

Takanori Sasaki

2016 年 3 月

進化型組込みソフトウェアにおける
継続的なメンテナンス性向上手法に関する研究

A Method for Improving Maintainability in Embedded Software Evolution

博士論文審査委員会

審査委員主査　大須賀　昭彦

委員　広田　光一

委員　田中　健次

委員　田原　康之

委員　石川　冬樹

佐々木 隆益

Takanori Sasaki

2016 年 3 月

Abstract

A conventional planned development process or an adaptive development process is in adequate for "the embedded software evolution" developing continuously in response to rapidly changing market requirements. As a result, maintainability decreases and whole re-factoring is needed. In this study, I propose a method for improving maintainability in embedded software evolution. This method is consisted of three techniques, the extensibility visualization technique, the extensibility reinforcement technique and the extensibility evaluation technique. These techniques are corresponded to each requirement that I defined three necessary requirements in developing for "the embedded software evolution". We applied the proposed method to an actual product's code continuously and could verify an improvement in the target software's extensibility.

概要

近年，技術革新スピードが加速し，市場のニーズは短期間で変化し，技術の陳腐化，急速な製品価値低下が起きている．このような変化に対応し，継続的に開発を行うソフトウェアを“進化型組込みソフトウェア”と定義する．従来の計画的な開発手法や適応的な開発手法を進化型組込みソフトウェアに適用しても，開発が進むにつれ，メンテナンス性が低下することで，全体的なリファクタリングが必要になる．そこで，本研究では，進化型組込みソフトウェアにおいて，メンテナンス性を継続的に向上させる手法を提案する．本手法は，進化型組込みソフトウェアを開発する上で必要となる 3 つの定義した要件に対応する拡張性可視化手法，拡張性強化手法，拡張性評価手法で構成される．さらに，提案手法を実際の製品ソースコードに対し，継続的に適用することで，対象ソフトウェアのメンテナンス性が向上することを確認した．

目次

図目次

表目次

第1章

序論

1.1. 本研究の背景

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．さらに，IoT（Internet of Things）と呼ばれるあらゆるモノがインターネットに接続されていく時代を迎えつつある．これにより，ソフトウェア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，また，ハードウェアデバイスの急速な進化とコモディティ化も加わり，技術の陳腐化，急速な製品価値低下が起きている．そのため，対象となるソフトウェアに求められる全ての市場ニーズを洗い出し，綿密な計画にもとづき，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計画的な開発プロセスでは，早ければ開発中に，遅くとも製品化後すぐに市場ニーズと製品仕様との乖離が生じる．その結果，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を維持できず，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計画的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適応的開発プロセスが台頭してきた[1]．このプロセスでは，全ての市場ニーズを最初に洗い出すことをしないため，将来の予測を加味した機能や機構は設計されない．代わりに，直近の市場ニーズにもとづいた要件だけを対象として，シンプルな機能，機構を実装し，マーケットへの早期製品投入を実現する．その

後，マーケットからのフィードバックを得て，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．

また，このような適応的な開発プロセスは，従来の計画的な製品開発における開発フェーズとメンテナンスフェーズという概念に対しても変化をもたらしている．つまり，市場のニーズに対して継続的な対応を実施することは，段階的に開発とリリースを行う開発フェーズであるといえるが，市場リリース後に改修しているメンテナンスフェーズであるともいえる．

そこで，継続的な変化に対し，迅速に対応しなければならないようなソフトウェアを進化型のソフトウェアと呼ぶこととする．これは，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ことで，積極的に進化させていくソフトウェアである．

このような，進化型のソフトウェアに対して適応的な開発であるアジャイル開発を用いた場合，市場ニーズにあわせ，新しい機能を継続的に開発できるが，繰り返し開発の中で，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る．それ故，長期的な視点においては，メンテナンス性の悪化により，歪んだアーキテクチャ構造となり，大規模なリファクタリングを要する場合もある．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，とされるが必要な機能の実装と開発スピードが求められるプロセスにおいて，将来のメンテナンス性を考慮して設計することは難しい．

特に組込みソフトウェアでは，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．そのため，進化型組込みソフトウェアに対する長期的な観点による継続的なメンテナンス性の向上が特に重要である．

しかしながら，進化型組込みソフトウェアであっても，従来は局所的なコード上の問題を検出，修正するリファクタリングや，ソースコードのメトリクスを計測し，相対的な数値情報によるリファクタリングが開発者個別に行われるだけであり，長期的なメンテナンス性向上に寄与する体系的な手法は，現状見当たらない．

1.2. 本研究の目的

網羅的な要件を入力し，ソフトウェアを構築する従来の計画的開発手法から，適応的な開発手法へとシフトし，さらに市場ニーズにあわせ，ソフトウェアが継続的に進化する進化型のソフトウェア開発手法が重要となってきている．

進化型ソフトウェアでは，メンテナンス性が重要となるが，特に，進化型組込みソフトウェアでは，組込みソフトウェアの特徴も踏まえたメンテナンス性が重要となる．例えば，応答性等のリアルタイム性を損なわず，多種多様なハードウェアデバイスの制御を出来るだけ共通のソフトウェアで扱うことができ，多くの兄弟機種やシリーズ製品を共通のソフトウェアから構成できるようなメンテナンス性である．

しかしながら，アジャイル開発プロセスにおける短期的スコープによる開発や，開発者の経験やスキルに依存した開発によって，長期的には製品全体のメンテナンス性が悪化してしまう不安から，ウォーターフォール型の開発プロセスを採用する場面も多くみられる．

そこで，本研究では，進化型組込みソフトウェアにおいて，継続してメンテナンス性を向上させることができる手法の開発を目的とする．

一般的に，メンテナンス性の向上にとって，ドキュメント化や，開発者教育も有効であるが，実際の開発現場においては，適切なドキュメントの作成・維持は困難であり，また開発者教育においても，人的リソースの異動等により，開発チーム内で詳細をすべて理解している開発者を確保できる状況は少ない．そこで，本研究ではメンテナンス性に対し，ソースコードのみを入力とするアプローチをとる．

提案手法は，概念的なコードスメル情報やメトリクスによる相対的な数値情報ではなく，特定の構造を持つソースコードの意味的な構造を抽出し，抽出したメンテナンス性に関する構造情報を用いて，メンテナンス性の構造上の問題箇所を自動で検出することを可能にしている．また，メンテナンス性の問題の有無に関わらず，将来的な進化の方向性に合わせた柔軟なメンテナンス性の向上を可能にする．さらに，この手法を一部ツール化し，実際の進化型組込みソフトウェア製品に適用することで有効性の評価を行う．

1.3. 本研究の構成

本研究では，進化型組込みソフトウェアに対し，ソースコードを用いた拡張性の観点における継続的なメンテナンス性の向上手法について述べる．本研究の構成は以下の通りである．まず 2 章，3 章で，本研究の背景となる既存技術，用語の定義について説明する．2 章では，ソフトウェアの開発プロセスについて説明し，3 章でソフトウェアの品質および従来のソフトウェア分析手法に関する用語の定義について説明する．

続いて 4 章では，本研究で提案する手法の全体像を述べ，5 章，6 章，7 章で本研究における提案手法を構成する 3 手法について論じる．4 章では，進化型組込みソフトウェアについての定義を示し，進化型組込みソフトウェアにおける開発手法に求められる 3 要件を設定し，提案する手法の全体像と構成について論じる．5 章ではメンテナンス性の構造把握に関する要件に対応した拡張性可視化手法について論じる．拡張性可視化手法は，オブジェクト指向言語構造と組込みソフトウェアのレイヤ構造を利用したアルゴリズムを特徴とした手法である．具体的には，2 種類の可視化モデルを作成し，デザインパターンや継承構造

にもとづいて拡張性構造を抽出し，可視化モデル内で表示する．6 章ではメンテナンス性の問題検出および強化に関する要件に対応した拡張性の強化手法について論じる．拡張性の強化手法は，メンテナンス性に関する問題箇所の定義と問題箇所の検出アルゴリズムを特徴とした手法である．具体的には，メンテナンス性に関する問題構造のパターンを定義したルールと，可視化されたメンテナンス性の構造を比較することにより，問題箇所を自動的に特定し，各ルールに対応したメンテナンス性の強化ガイドを出力する．7 章では将来の進化にあわせた柔軟なリファクタリングに関する要件に対応した拡張性の評価手法について論じる．拡張性の評価手法は，進化の大きさの表現とメンテナンス性の定量化指標を特徴とした手法である．具体的には，メンテナンス性の問題有無に関わらず，拡張性の構造について相対的に定量化し，変更許容性と変更容易性を用い，進化に対する機能毎の拡張性を評価する．なお，6 章および 7 章では，5 章で算出した継承構造による拡張性構造とデザインパターンによる拡張性構造を入力として用いる．

最後に 8 章で関連研究を示し，9 章でまとめと今後の課題，展望について述べる．

第2章

ソフトウェア開発プロセス

2.1. はじめに

本章では，ソフトウェアの開発モデル，およびソフトウェアの派生開発における現状について述べる．

まず, 2.2 節で, ソフトウェア開発モデルについて述べる. ソフトウェア開発モデルには，現在でも大規模な開発を中心に採用されているウォーターフォールモデルに代表される計画的開発手法と，近年浸透しつつあるアジャイル開発に代表される適応的開発手法を中心に説明する．

2.3 節では，派生開発について，主にソフトウェアプロダクトラインの概念を説明し，提案手法の説明でも使用される用語の定義を行う．2.4 節では，ソフトウェアメンテナンスに関する一般的な定義について示す．2.5 節では，本研究において，重要な概念であるソフトウェア進化について述べる．

最後に 2.6 節で，本章で挙げたプロセスについてのまとめと問題点を述べる．

2.2. ソフトウェア開発モデル

1960 年代後半において，ソフトウェアが大規模化するにつれ，個人のスキルに依存した開発方法に限界が生じてきた．1970 年代に，ソフトウェアのライフサイクル・プロセスが提唱されて以降，ソフトウェアの開発工程について多くのモデルが考案されている

2.2.1. 計画的開発

計画を重視し，各ソフトウェア工程を順に進める手法である．開発途中における要件変更は，開発コストの増加につながるという考えのもと，開発初期にすべての要件を確定させることで，変更の防止を目的とした手法である．

ウォーターフォールモデル

要求分析，基本設計，詳細設計，実装，単体テスト，結合テスト，システムテストという工程を一つずつ，順次実施するプロセスである．このプロセスを水が上流から下流に向かって流れる様で表現したモデルである．当該モデルは，現在までに多くのプロジェクトで導入され，特に大規模開発において成功を収めている．また，当該モデルは，V 字モデルとも呼ばれ，開発の前半は，作成すべきソフトウェアを段階的に詳細化していくことで，ソフトウェアの作りこみを行い，後半は小さい範囲から段階的に範囲を拡大させてテストを行う．そのため，原則的に上流から下流への一方向で開発が進むことを想定しており，開発の最終段階で表面化した問題は大きな手戻りとなってしまう．実際，近年のように，技術革新速度が速い状況では，上流工程ですべての要件を定義し，設計を完全に完了させることは困難な場合が多い．そのため，一旦進んでしまった下流工程から上流工程に戻ってしまうことによる開発の遅延，開発コストの増加が問題となっている．

2.2.2. 適応的開発

段階的に開発を進めることで，要件との乖離や，手戻りを最小限に抑える手法である．また，計画的開発では，開発が進むにつれ，変更コストは増大するという前提であったが，適応的開発では，自動テスト等の対策により，変更コストを抑え，変更を受け入れることを目的とした手法である．

インクリメンタルモデル

システムが独立性の高い機能で構成されている場合，システム全体の要件定義後，独立した機能単位に，設計・実装・テストを行い，徐々にシステム全体の機能が追加されるモデルである．

スパイラルモデル

システムが独立性の高いサブシステムで構成されている場合，サブシステム単位に，要件定義・設計・実装・テストを繰り返し行うモデルである．各サブシステムは並行に実施される．

RUP

IBM 社のラショナルブランドが提唱したモデルである．システムの振舞いを複数のユースケースに分割し，ユースケース単位に設計・実装を行うことで，段階的にシステムを構築するモデルである．技術的なリスクの高いユースケースを初期に選択することで，開発の早期段階におけるアーキテクチャの確立を狙っている．当該モデルは，中・大規模開発用のモデルである．

プロトタイプモデル

開発初期段階で，試作品を製作し，ユーザ要件と成果物との相違を確認した後，本格的に製品の開発を行うモデルである．製品開発においては，作成した試作品をベースとして再利用する場合と，再利用せず最初から作り直す場合とがある．

フィーチャードリブン開発

1997 年にジェフ・デ・ルーカが開発で採用し，1999 年著書で紹介した[2]．全体モデル開発，フィーチャリストアップ，計画，設計，構築という 5 つのプロセスで構成される．

アジャイル開発

計画的開発で起こりうる開発途中における要求と成果物とのギャップを最小限に抑えるために，小さな反復により，開発を行う手法である．反復を繰返しながら，機能追加を行い，システム全体を構築する．小規模な開発チームによるプロジェクトに適している．

ウォーターフォールモデルは，数年という長期的な計画のもと実施されるものもあるが，アジャイル開発では，1 週間から 1 ヵ月という短い単位で動作可能なソフトウェアをリリースする．ドキュメントよりも，動作するソフトウェアが品質の確保にとって重要としている．アジャイル開発は，現実の状況に最も適応しながら開発を行う手法の総称であり，多くのプロセスモデルが提唱されている．

また，アジャイル開発は，短期間のサイクルごとに，計画，設計，実装，テストを行うが，カウボーイプログラミングという無秩序，無計画な開発と混同されることもある．しかしながら，アジャイル開発において，プロセスの規律を緩めてしまうと，カウボーイプログラミングに陥りやすいことも事実である．

エクストリームプログラミング

アジャイル開発の一つであり，難易度の高い開発や市場の要求が変化するような製品開発分野に適した開発手法である．事前計画よりも柔軟性を重視する．この手法は小さなプロジェクトに向いており，5 つの価値と 19 の具体的なプラクティスで構成されている．ペアプログラミングという特徴により広く知られている．

スクラム

アジャイル開発の一つであり，最も採用が多いモデルである．スクラムマスター等の役割，バックログという予定されたタスクの可視化，各タスクの見積もり方法，スプリントという短期間の反復期間終了後の振り返り，生産性の計測等，少人数のチームで進めるために必要となる方法論をまとめたプロセスである．これをチームメンバで協力し，チーム全体が成長しながら開発を進めていく．

2.3. 派生開発

2.3.1. 流用開発

ソフトウェアを全て新規に開発するのではなく，既に開発済みのソフトウェアを利用する開発方法である．製品ファミリを構成する上で，既製品コードをコピーしたコードツリーをベースに開発を行うケースや，開発中のソースコード内の記述を既製品から部分的にコピーするケース，ライブラリとして丸ごと利用するケース等が存在する．しかしながら，総じてこれらの開発では，アドホックに流用が行われるため，変更時等において，想定しないリスクを含むことがある．

2.3.2. ソフトウェアプロダクトライン開発（SPL）

製品ファミリを計画的に作成する開発方法である．製品を構成するソフトウェアをコア資産と，プロダクト資産に分割し管理する．コア資産とは，製品ファミリ共通で使用する唯一のコードであり，プロダクト資産とは，個別製品に対する依存部についてのコードである．つまり，製品は，コア資産に対し，いくつかの選択されたプロダクト資産を統合することで構成される．統合の仕方としては，ソースコードレベルもあるが，コンポーネント単位で組合せを行うためのコンポーネントの階層化と局所化を統合した階層的なコンポーネント可変性モデリングもある[3].

資産の組合せによって定義された製品ファミリを生成するための，ソフトウェアプロダクトラインにおける重要な概念を下記に示す.

可変点

ソフトウェアをコア資産とプロダクト資産に分割しうる箇所である．この可変点で，製品ごとに異なる機能仕様を組み込むことができる．なお，可変点で実装コードを入れ変える必要がある場合は，可変点までがコア資産となり，入れ替えるコード部品は製品ごとに管理するため，プロダクト資産となる.

可変メカニズム

ソフトウェアの機能を切替える機構である. 言語仕様によって異なる機構が存在する. C, C++, C#のような言語では条件コンパイルという機構により，ビルド時の設定で有効になるソースコードを切替えることができる．これにより，製品ごとに異なるコードを管理する必要がない．特に，組込みソフトウェアでは，ハードウェアリソースの制限が厳しいこともあり，コードの削減を目的として多く用いられる．しかしながら，条件コンパイルは，コードの可読性が低下することや，切替え条件として製品名等を用いた場合，製品が増えるたび，類似コードの増加が起きる．そのため，不必要な条件コンパイル文の削減や，製品単位ではなく機能単位での切替え導入や，機能の組合せによる製品の構成を自動化する仕組み等の考慮が必要となる.

他にも，ビルド時に，共通インタフェースを持った実装ファイルを入れ替える方法や，静的，動的にライブラリを入れ替える方法，オブジェクト指向の仕組みにより，切替え対象となるクラスインスタンスの生成をコントロールする方法等がある[4]．Fritsch ら[5]によると，このような可変メカニズムは，実装方法，選択方法，切替えタイミングでカタログ化できるとしている.

バリアント

可変点で切替えることが可能な機能である．バリアントには，コア資産のオプションとして可変点に追加されるものと，選択肢におけるいずれか一つが選択されるものとが存在する．すべての可変点で確定したバリアントの集合により，異なる機能，性能を持つ製品を構成することが可能となる.

2.4. ソフトウェアメンテナンス

IEEE によると，メンテナンスとは，不具合の修正，性能や他の属性の改善，変更された環境へ適応させるために，出荷後のソフトウェアシステムやコンポーネントを変更するプロセスであると定義されている．さらに，細分化された定義には，ハードウェアやソフトウェアにおける不具合の修正プロセスである是正保守，変更された環境において，プログラムを使用可能な状態に保つプロセスである適応保守，性能や保守性または他の属性に関する改良プロセスである完全化保守がある[6].

Ogheneovo[7]によると，ソフトウェアメンテナンスとは，ソフトウェアが長期間使用されなければならないとした場合，ソフトウェアメンテナンスと進化に対して，巨大なコスト，実装速度の低下，複雑度の増加，新しいバグを追加してしまうかもしれない．しかしながら，新技術に即した専門技術の要求，変化と改良は避けることができないものであると特徴づけられる.

また，IEEE によると，メンテナンス性とは，不具合の修正，性能や他の属性の改善，変更された環境へ適応させるために，出荷後のソフトウェアシステムやコンポーネントへの変更容易さとして定義されている．

2.5. ソフトウェア進化

ソフトウェア進化とは，メンテナンスプロセスと類似しているが，より積極的に変更を進めることで，既存ソフトウェアを劣化させないようにするプロセスである．このプロセスでは，既存ソフトウェアの特性を継承しながら，品質を保ち，ユーザ要求に応じた新規機能を追加していく．このソフトウェアの進化プロセスは，ライフサイクルを通したハードウェアとソフトウェアに関して行われる[7]．

また，ソフトウェアの変化を表現する言葉としては，“ソフトウェアの加齢”，“ソフトウェア保守”，“ソフトウェア進化”，“ソフトウェア発展”等がある．ソフトウェアの加齢とは老朽化を暗示する表現であるが，ハードウェアのように磨耗・損傷することはなく，老朽化の原因は，保守の反復による構造の劣化や環境変化への不適応である[8]．

ソフトウェア進化に対して，アジャイルチームが恒久的にアサインされることがあれば，ユーザはソフトウェアの継続的進化と進化の方向付けを得ることができる．しかしながら頻繁すぎる進化は，ユーザにとって混沌とした状態にさせてしまうかもしれない．それでも頻繁なる進化は，急激に変化するユーザの要求に対応できる重要な仕組みである[9]．

2.6. プロセスのまとめと問題点

本章では，ソフトウェア開発におけるプロセスモデルについて説明した．まず，開発モデルに触れ，次に，派生開発について示した．さらに，本研究において重要な概念として，メンテナンスの一般的な定義とソフトウェア進化の概念について述べた．

ソフトウェアの大規模化が進んだことで，計画的にソフトウェアを作成するモデルであるウォーターフォールモデルが登場し，現在に至るまで，数多くの開発プロジェクトで実施されている．しかしながら，ソフトウェアの技術革新スピードが加速したことで，計画ベースの開発が困難になり，要件の変更は起こらないという前提から，要件変更は常に起こりうるという考え方のもと，様々な適応型の開発モデルが誕生し，現在，特に欧米においてアジャイル開発が浸透しつつある．

また，ソフトウェアが大規模，複雑化したことにより，毎回すべてのソフトウェアを新規作成することは品質の確保が難しく，コスト的にも合わなくなってきた．そのため，一度作成したソフトウェアを再利用して，新しいソフトウェアを作成することで，高品質，低コストとなる派生開発が指向された．さらに派生開発をより効率的に行うため，製品系

列における共通部と可変部を厳密に定義し，それらの組合せを管理するソフトウェアプロダクトラインと呼ばれる開発手法が誕生し，多くの企業で試されている．

また，ソフトウェアはリリースするまでが，開発として重要なフェーズではなく，リリース後に継続して利用され続ける必要があり，ソフトウェアの動作環境や市場ニーズの変化に対して，ソフトウェアの価値を低下させないように積極的に変更を行っていくことが，近年特に重要になってきている．

このようなソフトウェア開発の進展において，現在では，アジャイル開発およびソフトウェアプロダクトライン開発が目指している短期間，低コスト，高品質なソフトウェアの作成方法を，ソフトウェアのライフサイクル全体にわたって利用することで，現実的・継続的に貢献できることが必要とされているが，容易ではない．

第3章

従来のソフトウェア分析方法論

3.1. はじめに

本章では，本研究における課題や，本提案手法を構成する分析技術の前提となる知識および従来のソフトウェア分析方法論等について説明する．

まず，3.2 節では，本研究において，課題としてとらえたソフトウェアのメンテナンス性に関し，さらに詳細なソフトウェアの品質特性についての定義を説明する．それらソフトウェアの性質を分析するための手法として，設計情報をもとにした分析手法を 3.3 節に示し，3.4 節では，ソフトウェアの実装情報をもとにした分析手法について述べる．3.5 節では，リファクタリングについて説明し，3.6 節で，本章で挙げた従来手法のまとめと問題点について述べる．

3.2. ソフトウェアの品質特性

ソフトウェアが持つ品質特性を定義し，それを定量化，可視化することにより，ソフトウェアの品質をコントロールすることができる．特に，本研究において議論される特性についての定義を下記に示す．

柔軟性

ISO/ IEC 25010[10]によると，製品利用時の品質特性の一つである利用状況網羅性をあ

らわす品質副特性として定義されている．利用状況網羅性とは，“明示された利用状況及び当初明確に識別されていた状況を超越した状況の両方において，有効性，効率性，リスク回避性及び満足性をともなって製品又はシステムが使用できる度合い”として定義されている．柔軟性は，“要求事項の中で初めに明示された状況を逸脱した状況において，有効性，効率性，リスク回避性及び満足性をともなって製品又はシステムが使用できる度合い”として定義されている．柔軟性を確保することは，前もって予測されていない周囲の状況，機会及び個人の好みを製品に考慮することを可能にすることができるものである．

拡張性

ISO/ IEC 25010[10]によると，製品品質特性の一つである移植性をあらわす品質副特性として適応性があり，拡張性はこの適応性に含まれると定義されている．適応性は“異なる又は進化していくハードウェア，ソフトウェア又は他の運用環境若しくは利用環境に，製品又はシステムが適応できる有効性及び効率性の度合い”として定義されている．また，“内部容量・能力（たとえば，スクリーン領域，表，業務処理量，報告書様式）の拡張性を含む”として定義されている．

可変性

ISO/ IEC 26550[11]によると，可変性は，“プロダクトラインにおけるメンバ間で変わりうる特徴である”として定義されている．

3.3. 設計情報をもとにした分析手法

ソフトウェア分析モデリングや，ソフトウェア設計工程で作成されたドキュメント，開発者個人が持っているドメイン知識等を入力として，ソフトウェアを分析する手法である．

3.3.1. フィーチャー分析

製品機能等に代表されるフィーチャーの違いに着目した要求分析方法である[12]．フィーチャーとは，製品ソフトウェアファミリにおける製品を区別するために使われる視認可能な特性である．フィーチャーには，必須フィーチャー（すべての製品ファミリに存在するもの），オプショナルフィーチャー（いくつかの製品に存在するもの），選択的フィーチャー（いくつかのフィーチャーグループから一つを選択するもの）という 3 つのタイプがある．さらに，フィーチャー間における包含や排他関係という制限もある．

ソフトウェアプロダクトライン（SPL）におけるフィーチャー分析では，静的なフィーチャーをモデル化する手法と動的なフィーチャーの一部がサポートされている．最近では，実行時の構造的可変性の変化が必要な特性を追加することができるランタイム可変機構が

提案されている．フィーチャーをオプションと選択によって切替えるように設計することで，製品ごとの可変性を生み出すことができる．

また SPL では，製品ファミリに存在する共通部と可変部を管理するために，フィーチャーモデルを利用することがある．フィーチャーモデルを用いることで，ドメインエンジニアリングプロセスやアプリケーションエンジニアリングにおける再利用可能なコア資産の開発を促進することができる．なお，フィーチャーモデルとは，製品ファミリ間における共通部と可変部を表現した階層的なダイアグラムであり，フィーチャダイアグラムの記述は，CBFM や featureRSEB，COVAMOF のような，当初のダイアグラムから拡張された提案も多く存在する[13]

3.3.2. アーキテクチャ評価

アーキテクチャに含まれる潜在リスクの識別や，品質要件をアーキテクチャが満たしているかを分析する．

SAAM

Scenario-Based Architecture Analysis Method の略称である．

最初にドキュメント化され，普及したアーキテクチャ解析手法であり，シナリオベースでアーキテクチャを解析する．比較的軽量なため，容易に実施可能である．当初は，変更容易性を評価するための手法として提案された．現在では機能性，移植性，拡張性等，様々な品質特性に対して，アーキテクチャが対応できる度合を解析する．また複数の候補アーキテクチャの比較を行うことができる．

ATAM

The Architecture Trade-Off Analysis Method の略称である．

システムに要求される品質特性に対応したシナリオを作成し，優先度の高いシナリオを実現するためのアーキテクチャ手法を明らかにすることで，シナリオ実現上のリスクやトレードオフを明らかにする．品質特性要求の観点からアーキテクチャ決定の帰結を評価する．

SBER

Scenario-Based Architecture Reengineering の略称である．

明示的にソフトウェアの品質特性を改善させる．品質特性を主にシナリオを用いて測定する手法である．要求を満たさない品質特性を改善するために，要求に一致するまで，アセスメントとデザイン変更をイテレーティブに繰り返す.

ALPSM

Architecture Level Prediction of Software Maintenance の略称である．

ソフトウェアアーキテクチャ設計段階で，ソフトウェアのメンテナンス性を予測するための手法である．要求定義，アーキテクチャ設計，ソフトウェアエンジニア専門知見，履

歴データを入力とし，保守にかかる平均工数の予測を作る．

SAEM

A Software Architecture Evaluation Model の略称である．

最終製品としてのソフトウェアアーキテクチャと同時に，設計プロセスでの中間製品としてのソフトウェアアーキテクチャを対象とする

ARID

Active Review for Intermediate Designs の略称である．

設計途中から利害関係者を巻き込み，アーキテクチャ設計のレビューを行うことで，設計アプローチの適合性検証および設計の共有化を行う．現在作成中の設計がアーキテクチャに適合するかを早期に評価する．詳細設計に進む前までに，設計アプローチの正しさを評価する．

CBAM

Cost Benefit Analysis Method の略称である．

投資対効果（ROI）を意識してアーキテクチャを評価する．ROI にもとづいてアーキテクチャ改善施策を選択し，計画を立てるための情報提供をする．

LAAAM

Lightweight Architecture Alternative Assessment Method の略称である．

開発対象システムにおける品質特性の価値をステークホルダ間で定量的に共有し，それをもとにアーキテクチャ設計の妥当性を評価する．システムにおいて考慮すべき品質特性や品質特性間の優先順位を可視化してステークホルダ間で共有する．それをもとに複数のアーキテクチャ設計案の優劣を客観的に比較する．

3.4. 実装情報をもとにした分析手法

実際に動作するソフトウェアのコードを入力とし，手法ごとに定義された指標を用いて分析する手法である．

3.4.1. メトリクス分析

メトリクス分析とは，ソフトウェアに対し，ある側面を定量化することによる分析である．メトリクスには，プロジェクトメトリクスとプロダクトメトリクスがある．プロジェクトメトリクスは，プロジェクトの進捗管理に使われるものであり，プロジェクトの活動を定量化する．プロダクトメトリクスはプロジェクトの成果物の品質管理に使われるものであり，ソフトウェアの品質を定量化する．

プロダクトメトリクスは，非オブジェクト指向言語とオブジェクト指向言語において異なる．主に C 言語のような非オブジェクト指向言語では，古典的なメトリクスであるサイ

クロマチック複雑度，サイズ，コメントの割合等が用いられていた[14] ．一方，オブジェクト指向言語においては，Chidamber ら[15]が提案し CK メトリクスと呼ばれる WMC（ Weighted Methods Per Class ），DIT（ Depth of Inheritance ），NOC（ Number of Children ），RFC（ Response For a Class ），CBO（ Coupling between object classes ），LCOM（ Lack of Cohesion in Methods ）の 6 つのメトリクスセットが有名である．また，Melo[16]は，カプセル化，継承，ポリモーフィズム，メッセージングに関する MHF（Method Hiding Factor ），AHF（ Attribute Hiding Factor ），MIF（ Method Inheritance Factor ），AIF（ Attribute Inheritance Factor ），POF（ Polymorphism Factor ），COF（ Coupling Factor）という MOOD メトリクスを提案している．後藤ら[17] 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．Martin[18]は，依存と抽象度合がよいと感じられるパターンのデザインであることを計測するメインシーケンスというメトリクスを提案している．

さらに，複数のメトリクス値をから算出されるメンテナンス指標も提案されている．ソフトウェアの品質は，多面的な側面を有するために，単一のメトリクスではなく，複合的なメトリクス値により，品質を測定しようとするアプローチである．

3.4.2. パターン分析

ソフトウェア開発における定石に名前をつけ，再利用しやすいよう汎用的にカタログ化したものをパターンと呼び，分析対象のソフトウェアが，どのようなパターンを用いているかについて分析する手法である．ソフトウェアを作成する時に，パターンを用いることで，リソース効率やパフォーマンスに優れ，メンテナンス性の高いソフトウェアを短期間に構築することができる．逆に，作成したソフトウェア内部に含まれるパターンを抽出することで，どのようなソフトウェア構成になっているのか，またどのような問題が考えられるのかを推測することが可能である．代表的なパターンを下記に示す．

アーキテクチャパターン

POSA（Pattern Oriented Software Architecture）で示されているように，ソフトウェア全体として，効率のよいアーキテクチャの例をカタログ化しているものである．

例えば，Layers パターンでは，アプリケーションを複数の抽象度が異なるレイヤに分けて，タスクごとの構造化を行うアーキテクチャパターンである．また，MVC（Model—View—Controller）パターンでは，ユーザインタフェースをともなうアプリケーションを 3 つのコンポーネントに分割し，ユーザインタフェースの出力部分（View）とソフトウェアシステムが対象とするデータやビジネスロジック（Model）を分離し，ユーザインタフェースの入力部分（Controller）からの更新を通知する．これにより，ユーザインタフェース部分を取り換えることが容易になるアーキテクチャである．

デザインパターン

アーキテクチャパターンよりも，粒度の小さいパターンであり，主にオブジェクト指向言語が持つ仕組みを利用したものが多い. 有名なデザインパターンとして, 表 1 に示す GoF（Gang of Four）のデザインパターンがある[19][20].

GoF のデザインパターンでは, 23 のパターンを定義しており, それら 23 のパターンは, 生成に関するパターン，構造に関するパターン，振舞いに関するパターンのいずれかに分類される.

表 1 GoF のデザインパターン

カテゴリ	パターン名
生成に関するパターン	Abstract Factory，Builder，Factory Method，Prototype，Singleton
構造に関するパターン	Adapter，Bridge，Composite，Decorator，Façade，Flyweight，Proxy
振舞いに関するパターン	Chain of Responsibility, Command, Interpreter, Iterator, Mediator, Memento，Observer，State，Strategy，Template Method，Visitor

たとえば，生成に関するパターン群には，次の 5 つのパターンが含まれている．Abstract Factory パターンとは, クラスを生成する Factory クラスの組合せが異なるような製品がある場合に，Factory クラスの組合せを生成するような Factory の抽象化パターンである．Builder パターンとは，パラメータを変更することで，処理を変えることができるようなクラスにおいて，個々の具現化したパラメータセットに対応したクラスを用意しておくことで，パラメータの詳細を隠蔽するようなパターンである．Factory Method パターンとは，クラスを生成する Factory クラスに，抽象的なクラスインスタンス生成関数を用意し，Factory クラスを継承したクラスにおいて, 特定のクラスインスタンスを生成するようオーバーライドするパターンである．これにより，個々の対象クラスごとに必要とするクラスのセットを設定することができるパターンである．Prototype パターンとは，パラメータが多い等，クラスのインスタンスを生成することが不便である場合に，一旦作成したクラスのクローンを生成することができるメソッドを用意しておくパターンである．Singleton パターンとは，クラスのインスタンスをシステム内で一つだけを生成し，共有することができるパターンである.

また，構造に関するパターンには，特定の機能を持っているクラスを利用するような委譲の関係となる構造を構築し，委譲したクラスの機能に追加の処理を行う部分だけを実装することで異なる振舞いを実現する Decorator パターン等が含まる．振舞いに関するパターンには，振舞い関数のひな形を作成しておき，オブジェクト指向言語における継承の仕組みを利用し，ひな形を作成したクラスの派生クラスで処理の実装を行うことによって，

振舞いを切替えることができるような Template Method パターン等が含まれる.

アンチパターン

過去に失敗した様々な問題に対し，パターン化したものであり，失敗に陥る原因や，その解決方法について文章としてまとめられている．アンチパターンは，GoF のデザインパターンに刺激され，1995 年に Koenig[21]が提唱した．アンチパターンはソフトウェアの設計やプログラミングのみならず，組織やプロセス等，様々な分野において利用されている考え方である.

Brown ら[22]は，ソフトウェア開発のアンチパターンとして，一つのオブジェクトに責務が集中してしまう肥満児パターンや，ソフトウェアアーキテクチャのアンチパターンとして，個別に設計された複数のシステムにより，相互運用性や再利用性を妨げてしまう全社的おんぼろ煙突化パターンを紹介している．また，分析過程における過剰な検討がプロジェクトの停滞に繋がる分析地獄等のプロジェクト管理に関するアンチパターンも紹介している.

Ujhelyi ら[23]は，default ケースのないスイッチ文，Null 文字の結合，例外処理内で使用されている instanceof によるタイプチェック，equal()による文字列リテラルの比較，equal()を使わない文字比較, 未使用パラメータ等の Java 文法レベルにおいて不具合につながるコードをアンチパターンとして検出している.

コードスメル

リファクタリングすべき，ソフトウェアの保守性を低下させる箇所の特徴をコードスメル（不吉な匂い）と呼び識別したものであり，コードスメルを検出した場合の対処方法も示されていることが多い．コードスメルは，実装に関する記述そのものである場合や，メトリクス値と関連するものや，抽象度が高い症状を示しているもの等，さまざまである．表 2 に, Fowler[24]が提案した 22 のコードスメルについて上記観点により分類した表を示す.

表 2 Fowler のコードスメル

カテゴリ	スメル名
コード記述	基本データ型への執着，スイッチ文，一時的属性
メトリクス	重複したコード，長すぎるメソッド，巨大なクラス，多すぎる引数，属性，操作の横恋慕，データクラス
起きている症状についての概念	怠け者クラス，疑わしき一般化，データの群れ，パラレル継承，変更の発散，変更の分散，メッセージの連鎖，仲介人，不適切な関係，クラスのインタフェース不一致，未熟なクラスライブラリ，相続拒否，コメント

コードの記述については，特定コードに対する単純な変換であり，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．一方，概念を定義しているものについては，実際のコードを開発者自身で解読し，見つけ出す必要がある．

3.4.3. 共通性・可変性分析

製品ごとのソースコードにおける共通部と可変部を抽出する手法について述べる．

共通部・可変部抽出

現実の製品開発においては，既にバリアントができた後，再利用戦略が明確になることが多く，共通部と可変部の正確な情報はコード上の可変点が明確にならないので提供されにくい．そこで，Duszynski ら [25] [26]は，類似したシステムの可変性を通して，正確な定量的情報を提供するスケーラブルなリバースエンジニアリングであるバリアント分析を提案している．

Xue [27]は，手動による共通可変分析と可変性モデリングに課題があり，トップダウンであるドメインアナリシスとボトムアップであるコードクローンの鋏方式を提案している．また，要件からの SPL 箇所抽出と，クローンによる可変箇所抽出をレガシーコードに対して行うモデルの差異検出，クローン検出情報の検索技術を統合した手法である．

可変点マイニング

コード内から可変点を抽出する手法である．可変点が分かることで，シリーズ製品における切替え箇所を知ることができる．また，当該可変点を管理できれば，メンテナンスコストの一つである，機種の切替え箇所に関する設計コストを削減することが可能である．

3.5. リファクタリング

リファクタリングとは，既存のソフトウェアにおける振舞いを変えることなく，ソフトウェアの内部構造を変更し，可読性や変更容易性等の品質特性を向上させる手法である．Mens ら[28]によると，リファクタリングには，表 3 に示すアクティビティがあり，それぞれのアクティビティには，異なったツール，技術，形式化がある．例えば，手順 1 の実装レベルにおけるリファクタリング箇所を識別する方法では，動的な振舞いに着目し，常に変化しない変数はリファクタリング対象として識別し，削除を行う手法や，クローンコードというコードスメルに着目し，クローンコードを削除する手法，メトリクスベースの手法がある．手順 4 におけるリファクタリングの適用では，リファクタリングすべき箇所について，あるデザインパターンを導入しようとした時に，関数の移動，クラスの追加，関

数の抽出，関数の引き上げ，関数名の変更等の確立された典型的な変更方法を用意しておき，順次適用する．このようなリファクタリングを行うためのツールが多く提案されている[29].

実際の開発現場においては，リファクタリングは重要な役割を持っているが，適切にリファクタリングを行える開発者は少なく，メンテナンス性の悪化が限界まで達した時点で行うため，困難な作業となることが多い.

表 3 リファクタリングアクティビティ

手順	リファクタリングアクティビティ
1	どこをリファクタリングすべきか識別する
2	どのリファクタリングを適用するか決める
3	リファクタリング前後で動作が同じになるよう保証する
4	リファクタリングを適用する
5	ソフトウェアの品質特性についてのリファクタリング効果を計測する
6	リファクタリングしたコードと他のコードとの一貫性を保つ

3.6. 従来手法のまとめと問題点

本章では，本研究における課題や，本提案手法を構成する分析技術の前提となる知識および従来のソフトウェア分析方法論等について説明した．まず，ソフトウェアの品質特性として，柔軟性，拡張性，可変性についての定義を示した．そして，それらを分析するための手法として，設計情報を用いた分析手法について触れた後に，実装情報をもとにした分析手法やリファクタリングについて示した.

国際標準化団体である ISO/IEC が定義した，柔軟性，拡張性，可変性の関係について本研究のコンテキストで表現すると，柔軟性は，想定外の要件に対応する能力であり，拡張性は，新しい要件に対して適応する能力であり，可変性は，柔軟性や拡張性によって，製品ファミリ間において，変化することができるようになった特性であるといえる.

フィーチャー分析やアーキテクチャ評価のような設計情報をもとにした分析手法では，ドキュメントやドメイン知識を持っている開発者が必要であり，また本研究の対象である進化型組込みソフトウェアのような，継続的に変更が発生する開発プロセスにおいて，実行することは難しいといえる.

一方，実装情報をもとにした各種分析手法についても多くの研究がなされているが，コードスメルや，パターン分析については，ソフトウェアの生産活動において，重要な知識を提供しているが，開発担当者のスキルや経験がないと有効活用が難しい．また，メトリクス分析については，ソフトウェアの分析ツールの発展により，簡単に算出でき，実際の開発現場においても，常時，自動計測しているプロジェクトも少なくない．しかしながら，

計測されたメトリクスは単なる数値であるが故，その数値を理解し，有効活用するには，やはり開発担当者のスキルや経験が必要不可欠となっている．

最後にリファクタリングについて触れているが，リファクタリングはどの開発手法においても必要となり，重要な役割を持っている．しかしながら，リファクタリング対象ソフトウェアの持つメンテナンス性が悪化しすぎると，作業量の大きなリファクタリングとなってしまうため，適用することが困難となる．また，リファクタリングを行うにあたっては，開発者に高いレベルの経験とスキルが要求される．

第4章

継続的なメンテナンス性向上手法

4.1. はじめに

本章では，本研究で提案する手法の全体像について述べる．
まず，4.2 節で，本研究の対象である進化型組込みソフトウェアについて定義する．4.3 節では，進化型組込みソフトウェアに対して，従来の開発手法を適用した場合の課題について示す．4.4 節では，進化型組込みソフトウェアに対して必要とされるメンテナンス性について定義し，メンテナンス性が劣化した場合の例を説明する．4.5 節では，進化型組込みソフトウェアが満たすべき要件について示し，4.6 節では，本研究で提案する手法の全体像について説明する．4.7 節で本章のまとめを示す．

4.2. 進化型組込みソフトウェア

市場リリース後のソフトウェアは，ハードウェアと異なり，機械的な経年劣化は起こらないが，ソフトウェアのリリース直後から，市場におけるハードウェアの進化やユーザニーズの変化が進むことで，ソフトウェアの陳腐化が進行する．それにより，機能変更や追加等の要件変更が発生し，それを満たすためにソフトウェアを変更する必要性が生じる．変更が適切に行われないとソフトウェア構造が劣化し始め，ソフトウェア構造の劣化が大きくなると，要件変更時に必要となるメンテナンスにかかるコストが増大する．メンテナンスコストが増大しすぎると，新規ソフトウェア構築コストの方が，当該ソフトウェアに

対するメンテナンスコストよりも安価となりうる．その場合，当該ソフトウェアが放棄されるが，実績のない新規ソフトウェアに潜む不具合リスクも生じる．このような状態に陥らないためにも，市場のニーズに対し，継続的な変更を適切に実施していくことが必要となる．

このような開発は進化型のソフトウェア開発に分類され，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い，ソフトウェアを積極的に進化させていくのである．

本研究では，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ソフトウェアを対象としている．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ソフトウェアである．このような組込みソフトウェアを進化型組込みソフトウェアと呼ぶ．

4.3. 進化型組込みソフトウェアの課題

既存の開発手法を用いた進化型組込みソフトウェア開発の課題について述べる．

4.3.1. 計画的な開発手法における課題

図 1 に，計画的な開発手法を進化型組込みソフトウェアに適用することの課題を示す．ウォーターフォールに代表されるような計画的な開発手法では，事前に全ての市場ニーズを抽出し，段階的に詳細化しながら開発を行う．しかしながら，現在のような複雑なソフトウェアに対する全ての市場ニーズを洗い出すことは難しい．

それ故，開発の途中で市場ニーズの変化が起きた場合は，大きな手戻りコストが発生することになる．一方，開発手戻りによるコスト増加を嫌い，開発途中での変更を受け入れない場合は，開発中にも関わらず，製品価値が低下し始めてしまう．さらに，次機種の開発においては，市場ニーズと製品仕様との大きな乖離を埋めるため，結局，構造全体に及ぶリファクタリングが必要になる．

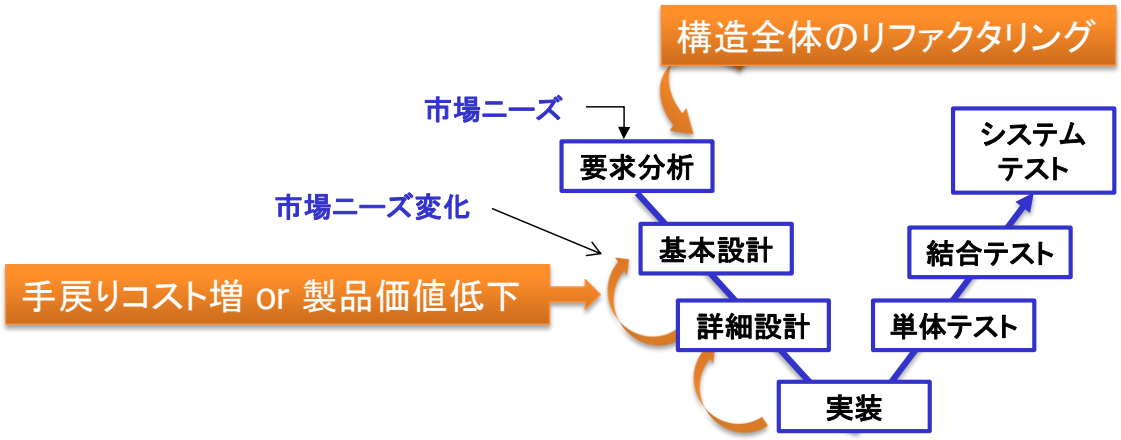


図 1　計画的な開発手法における課題

4.3.2. 適応的な開発手法における課題

図 2 に，適応的な開発手法を進化型組込みソフトウェアに適用することの課題を示す．アジャイル開発に代表されるような適応的な開発手法では，変化は必ず発生するものとして常に受け入れるため，目先の市場ニーズのみを抽出し，シンプルな機能を迅速に開発した後，新たな市場ニーズに対応する開発を繰り返す．適応的な開発手法では，繰り返しの度に局所的なリファクタリングを施しながら，市場ニーズとあった機能を実装することができる．しかしながら，リリース後の製品に対して，局所的な正解となるアーキテクチャの選択を長期間にわたり行う可能性がある．それ故，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り，メンテナンス性が悪化することにより，大規模なリファクタリングを要する場合もある．これを防ぐため，アジャイル開発においても，メンテナンス性は重要であるとされるが，必要な機能の実装だけが優先されるなか，将来のメンテナンス性を考慮して設計することは難しい．

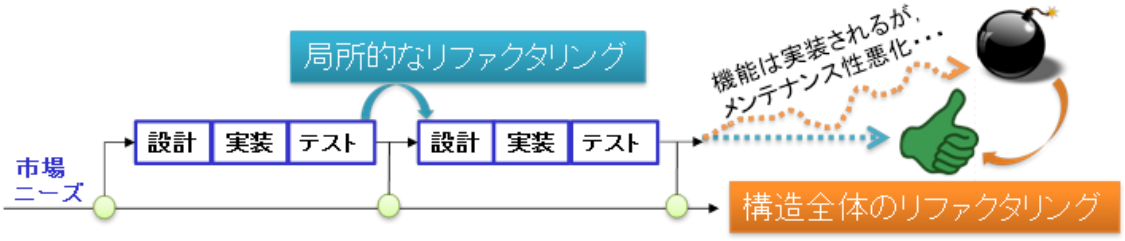


図 2　適応的な開発手法における課題

4.4. 進化型組込みソフトウェアのメンテナンス性

4.4.1. メンテナンス性における主要な観点

ほとんどの組込みソフトウェアでは，何かしらのハードウェアデバイスをソフトウェアで制御する必要がある．そのため，ハードウェアデバイスが要求する応答性等のリアルタイム性能が重要である．また，ハードウェアデバイスの違いによる兄弟機種を開発するようなシリーズ製品開発を行うことが多い．

このような特徴を含む進化型組込みソフトウェアでは，応答性等のリアルタイム性を損なわず，多種多様なハードウェアデバイスの制御を出来るだけ共通のソフトウェアで扱え，多くの兄弟機種やシリーズ製品を共通のソフトウェアから構成できるようなメンテナンス性が重要となる．このようなメンテナンス性には，下記に述べる拡張性の概念が当てはまる．

拡張性とは，3.2 節で示した通り，適応性に含まれる性質であり，特に組込みソフトウェアにおける拡張性には，図 3 に示したように，横軸である時間軸方向への拡張性と，縦軸である機能・性能方向への拡張性がある．時間軸方向の拡張性は，時間の経過とともに新しくなる関連技術への追従や，市場ニーズの変化によって生じ，機能・性能方向への拡張性は，ハードウェアデバイスの性能・機能にあわせ，価格帯ごとの購買層をターゲットとした製品戦略として生じる．さらに，時間軸方向の拡張性は，時間の経過方向へのみであるが，機能・性能方向については，高機能・高性能であるハイエンド製品から開発し，徐々に機能制限することで，ローエンド製品を作り出す場合と，ローエンド製品から機能追加によってハイエンド製品を作り出す場合がある．また，同時にすべての価格帯におけるラインナップを揃えることもあるが，一般的には，時期をずらして製品ラインナップを構築する場合が多いため，時間方向の拡張性と混在した進化となることが多い．

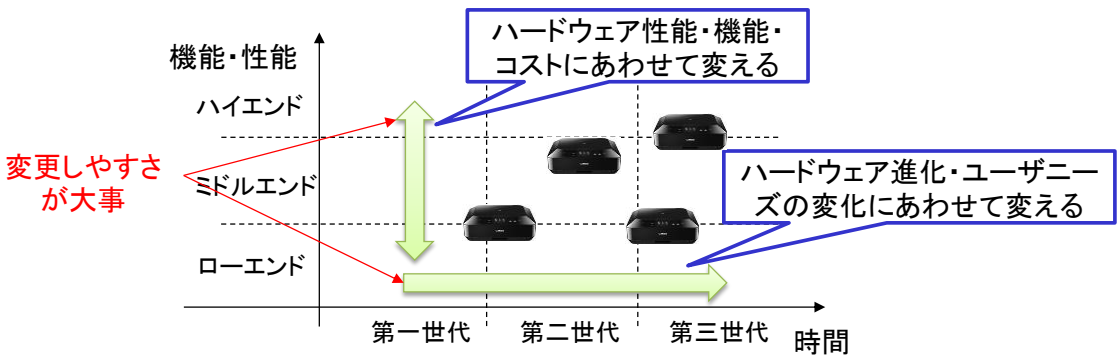


図 3　進化型組込みソフトウェアにおける拡張性

期待される拡張性としては，拡張するために必要となる変更規模をなるべく小さくできることや，変更作業の手間が少ない方が望ましい．たとえば，手を入れずに変更できる構造や，容易に変更ができる仕組みを持っていること等である．具体的には，組み込まれているハードウェア個々の進化に非依存で動作可能であることや，新しいハードウェアを追加した場合に，ソフトウェアの追加が容易であることや，ユーザの要求が変わり，新しい機能を実装する必要が生じた場合にも，同様に追加が容易であること等である．

4.4.2. メンテナンス性の悪化した例

進化型組込みソフトウェアにおけるメンテナンス性の悪化した例を図 4 に示す．開発初期版では，実線で示したデバイス A（DevA）とデバイス B（DevB）を製品によって切り替えて制御する機能 1 が実装されていたが，その後，段階的に機能 2，機能 3，機能 4，機能 5 と追加されるような例である．

まず，既存のデバイスと類似なデバイス C（DevC）が機能 1 に追加される場合，既存実装に追加する形で User クラスから呼び出される DevC 用の機能 1 を追加してしまう．このように User クラスは，後の機能追加によって，肥大化，複雑化していくことになる．

次に，機能 2 を追加した開発者は，デバイス D（DevD）とデバイス E（DevE）を製品によって切り替えて制御する機能 2 を User クラスからは隠蔽する設計として実装した．しかしながら，後に機能 3 を追加した開発者は，DevE を操作可能な機能 2 を利用することにしてしまうことで，隠蔽されているはずの DevE 用の機能 2 を変更することができなくなってしまう．

さらに，機能 5 を追加した開発者は，DevE を直接アクセスしていることで，DevE の仕様が変更した場合，ソフトウェア内の複数箇所に同様の変更を加える必要が生じてしまう．一方，機能 4 を追加した開発者は，機能のあらゆる段階での変更に対応する仕組みを用意していたが，変更が発生しない場合，用意した冗長な仕組みにより，仕組みのない場合と比較し，性能を無駄に低下させることになる．

このように，ソフトウェアの機能的な実現を目的として開発することで，市場のニーズには，迅速に応えることができるかもしれないが，ソフトウェアのメンテナンス性についての戦略がなく，個々の開発者における経験とスキルに頼って継続的な開発を行った場合，将来的なデバイスの進化や，アルゴリズムの進化に対して，大きなリファクタリングを行う必要性が生じることになる．

特に組込みソフトウェアでは，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.

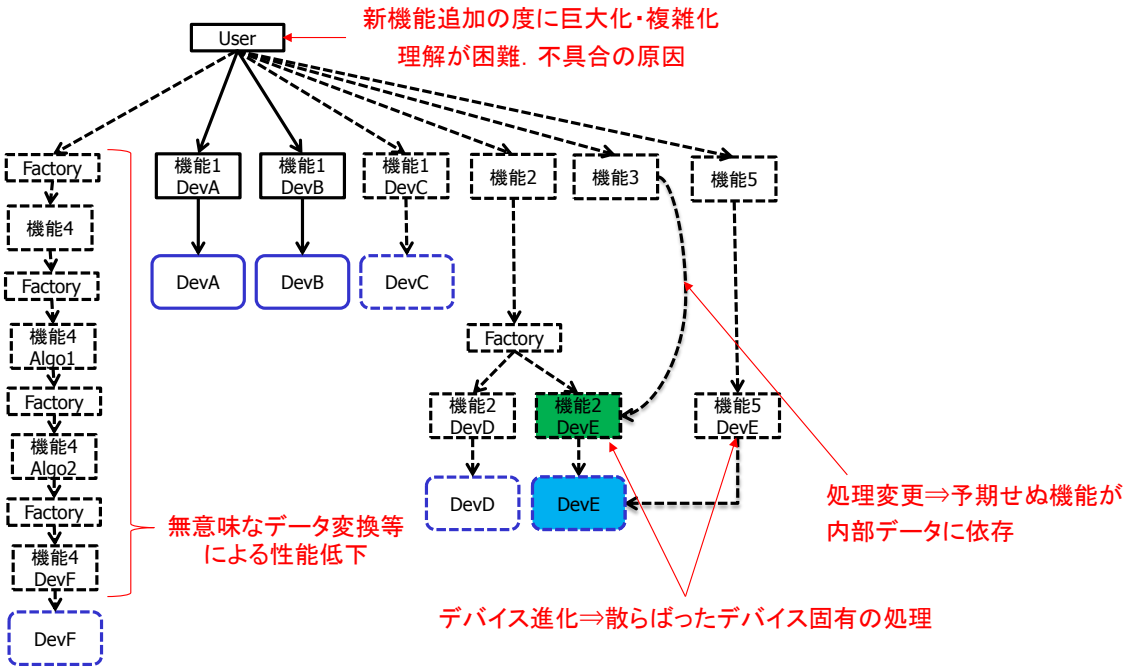


図 4　メンテナンス性の悪化した例

4.5. 進化型組込みソフトウェアの開発手法が満たすべき要件

進化型組込みソフトウェアにおいて，継続的な進化に対し，迅速に対応し続けるには，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, Fowler[24]の提案に代表されるような従来のコードスメルによるリファクタリングでは，メンテナンス性の問題があることに気づくことが難しい．なぜなら，コード記述レベルのメンテナンス性低下箇所は，ツールでも自動検出できるが，設計レベルの場合，概念として示唆されているに過ぎないため，開発者が実際のコードを解読し，発見する必要がある.

また，メトリクスベースでのリファクタリングにおいては，定義した問題を自動で定量化することはできるが，その値はプログラムの機能や，開発者，言語等による様々な要因で変化する．そのため，プログラムの意図も含めた解釈は難しく，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.

そこで，進化型組込みソフトウェアを開発する上で，必要となる要件を定義する.

要件 1：メンテナンス性に関する実装構造を把握できなくてはならない

特に進化型組込みソフトウェアでは，継続的な進化に対し迅速に対応することが必要であるため，開発現場において，作成コストの大きい設計ドキュメントは十分に整備されないことも多い．また，実装中の要件変更により，設計と実装が乖離してしまうことがあり，設計ドキュメントの信頼性が失われる場合もある．ソフトウェアの構造がわからないと，潜在的な不具合の表面化，性能劣化を恐れ，メンテナンス性を根本的に向上させるような変更が難しくなる．それ故，実装構造をベースとした情報の取得が必須である．

なお，取得すべきメンテナンス性の情報としては，組込みソフトウェアの特徴である多種多様なハードウェアデバイスに依存した制御の切り替え箇所や，シリーズ製品を切り替えている箇所に関する情報である．

また，継続的な進化に対し迅速に対応するためには，ソフトウェアのメンテナンス性に関する全体的構造を容易に把握できることが必要である．

要件 2：メンテナンス性の構造的な問題を自動で検出し，正しく対策できなければならない

進化型組込みソフトウェアでは，継続的な進化に対して，機能追加を繰り返すと，メンテナンス性が低い場合，次第に変更コストが大きくなる．また，ソフトウェアの構造が複雑になり，機能追加時の変更箇所も多くなってくると，新しい機能と同時に不具合を混入させてしまうことにもなりうる．

そこで，進化のたびに，局所的なリファクタリングを行うことにより，メンテナンス性を向上させる必要がある．しかしながら，概念的なコードスメルの知識だけで，実際のコードから，設計レベルの構造に関する問題を見つけることは難しい．さらに，メンテナンス性を向上させたつもりが，部分的な対応だったため，実際に進化が発生した場合には利用できないという場合もある．

それ故，ハードウェアデバイスやアルゴリズムの共通・可変部切り替えを妨げる箇所や，機能追加のたびに大きな変更コストが発生する箇所について，自動で検出し，開発者のスキルに依らず，ソフトウェアの問題箇所を改善できることが必要である．

要件 3：将来の進化にあった柔軟なリファクタリングができなければならない

組込みソフトウェアでは，搭載メモリ量に制限のある製品が多いため，メモリ使用量の

増加につながるコードを組込みたくないという要求がある．また，ハードウェア制御等のリアルタイム性を確保するため，性能劣化につながるようなコードを組込みたくないという要求もある．しかしながら，メンテナンス性を向上させるためのコードは，一般的に冗長なコードとなる．一方，進化型組込みソフトウェアでは，継続的な進化を実現するために，将来の進化を見越してメンテナンス性を高めるコードをなるべく多く実装したい等の相反した要求が存在する．

それ故，例えば，性能という面が重要な製品である場合は，性能と将来の進化に対するメンテナンス性とのバランスをとれるような，柔軟なリファクタリングを実施できることが必要である．

4.6. 提案手法の全体像

本研究では，進化型組込みソフトウェアにおけるメンテナンス性を向上させる手法を提案する．本章では，提案手法の全体像について示す．

4.6.1. 提案手法によるアプローチ

進化型組込みソフトウェアでは，市場のニーズや，ハードウェアの進化等に対応するため，要件変更や新規要件追加による機能変更を継続的に行うことになる．従来の適応的な開発プロセスでは，メンテナンス性を向上させるためのリファクタリングにおいて，図 5（a）に示すような，開発者の経験から局所的なメンテナンス性を向上させるリファクタリングが行われてきた．このような局所最適による機能追加やメンテナンス性の向上を継続することで，徐々に組込みソフトウェア全体として歪んだアーキテクチャとなってしまう．

そこで，提案手法では，図 5（b）に示すような，進化型組込みソフトウェアに必要となるメンテナンス性の向上を体系的に実現可能な後述する 3 つの手法をプロセスに組込むことによって，客観的な分析結果を用いた構造全体を対象とするリファクタリングを行う．このような客観的分析結果に基づいたリファクタリングを継続的に行うことで，迅速な機能追加を行いつつ，開発者の経験やスキルによらず，長期的なメンテナンス性も向上させることができる．

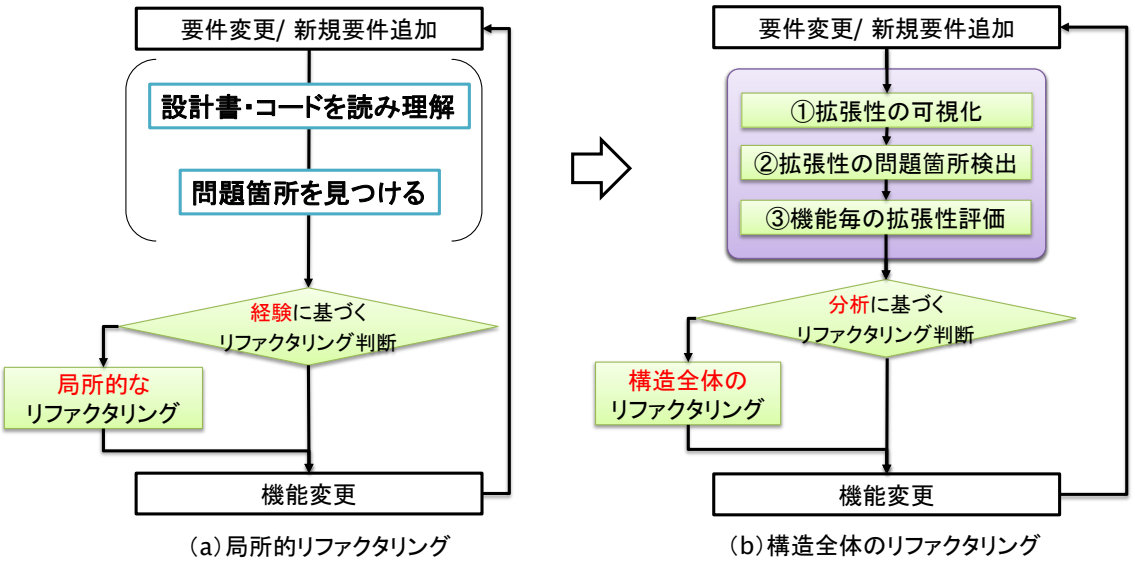


図 5　提案手法を用いた進化型組込みソフトウェア開発プロセス

4.6.2. 提案手法の構成

提案手法は，4.5 節で示した手法が満たすべき要件に対応し，図 6 に示すような 3 つの手法で構成されている．（1）可変点における拡張性構造を抽出し，可視化を行う拡張性可視化手法，（2）抽出した拡張性の問題箇所や，問題箇所の対処方法を提示する拡張性強化手法，(3)変更に対する 2 つの指標で機能単位の拡張性を評価する拡張性評価手法である．

拡張性可視化手法は，“要件 1：メンテナンス性に関する実装構造を把握できなくてはならない”に対応している．また，オブジェクト指向言語構造と組込みソフトウェアのレイヤ構造を利用したアルゴリズムを特徴とした手法であり，2 種類の可視化モデルを作成し，デザインパターンや継承構造にもとづいた拡張性構造を抽出し，表示する．

拡張性強化手法は，“要件 2：メンテナンス性の構造的な問題を自動で検出し，正しく対策できなければならない”に対応している．また，メンテナンス性に関する問題箇所の定義と問題箇所の検出アルゴリズムを特徴とした手法であり，メンテナンス性に関する問題構造のパターンを定義したルールと可視化されたメンテナンス性の構造を比較することにより，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．

拡張性評価手法は，“要件 3：将来の進化にあった柔軟なリファクタリングができなければならない”に対応している．また，進化の大きさの表現とメンテナンス性の定量化指標を特徴とした手法であり，メンテナンス性に関する問題の有無に関わらず，拡張性の構造について相対的に定量化し，変更許容性と変更容易性を用い，進化に対する機能毎の拡張性を評価する．

提案手法は，概念的なコードスメル情報やメトリクスによる相対的な数値情報ではなく，特定の構造を持つソースコードの意味的な構造を抽出し，抽出したメンテナンス性に関す

る構造情報を用いて，メンテナンス性に関する構造上の問題箇所を自動で検出することを可能にしている．また，メンテナンス性の問題の有無に関わらず，将来的な進化の方向性に合わせた柔軟なメンテナンス性の向上を可能にする．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組込みソフトウェアの進化過程において，適用することで有効性の評価を行う．

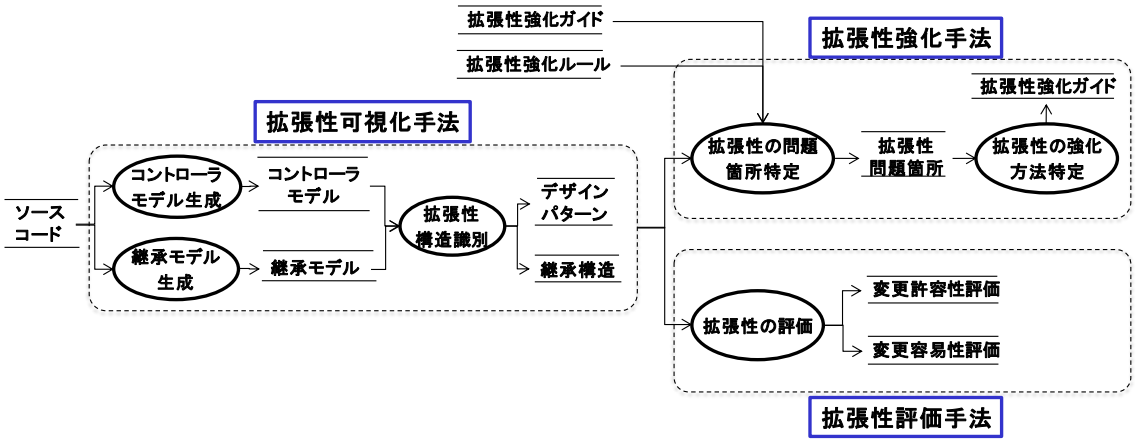


図 6　提案手法の全体像

4.7. まとめ

本章では，進化型組込みソフトウェアについての定義を行い，従来手法における課題を明らかにした上で，進化型組込みソフトウェアに求められる要件を設定し，提案手法の全体像を示した．

市場ニーズの変化やハードウェアデバイスの進化に対し，リファクタリングを施しながら継続的な変更を繰り返し開発の中で行うことで，ソフトウェアを積極的に進化させていく必要があり，従来の製品単独の仕様では完結しない新しい組込みソフトウェアを進化型組込みソフトウェアと定義した．

このような進化型組込みソフトウェアに対し，従来の計画的開発手法や，適応的開発手法を用いた場合の課題として，市場ニーズの変化やハードウェアデバイスの進化に対応して進化することが難しいことや，長期的にはメンテナンス性の低下により，歪んだアーキテクチャになることを示した．また，組込みソフトウェアにおけるメンテナンス性として，容易にシリーズ製品化できるような拡張性が重要であることを示し，メンテナンス性が低下する原因およびその症状について説明した．

そこで，進化型組込みソフトウェアに求められている次の 3 つの要件を設定し，その要件を満たす提案手法の全体像について説明した．要件 1：メンテナンス性に関する実装構造を把握できなくてはならない，要件 2：メンテナンス性の構造的な問題を自動で検出し，正

しく対策できなければならない，要件 3 : 将来の進化にあった柔軟なリファクタリングができなければならない.

提案する手法は，上記 3 つの要件に対応した（1）拡張性可視化手法,（2）拡張性強化手法,（3）拡張性評価手法で構成されており，この結果を用いて客観的なリファクタリングを行うことで，進化型組込みソフトウェアのメンテナンス性を継続的に向上させる手法である．各手法の詳細については，次章以降で説明する.

第5章

拡張性可視化手法

5.1. はじめに

本章では，4.5 節で挙げた進化型組込みソフトウェアが満たすべき要件の一つに対応する手法について説明する．まず，5.2 節では，提案手法が満たすべき要件に対する課題について述べる．5.3 節では，要件の一つに対応する拡張性可視化手法の概要を説明する．5.4 節では，可視化手法のベースとなる可視化モデルについて説明する．5.5 節では，機能の切替えを行う可変点や可変メカニズムについて説明する．5.6 節では，可視化モデル内の要素について説明する．5.7 節では，可変構造を特定し，5.8 節では，求められたモデルから抽出する拡張性の構造を定義する．5.9 節では，可変メカニズムのパターン分類方法を説明し，5.10 節では，抽出された拡張性構造を組込みソフトウェアにおけるレイヤ構造にマッピングする方法について説明する．

5.11 節では，提案手法を実際の進化型組込みソフトウェア製品に適用した実験内容について示す．5.12 節では，組込みソフトウェア以外のドメインに対し，提案手法を適用した実験について述べる．5.13 節では，可視化手法を実装したツールについて説明する．実験結果で示した図については，本提案手法の一部として作成されたツールの出力である．5.14 節で考察をし，5.15 節で本章のまとめを示す．

5.2. 拡張性可視化手法の課題

拡張性可視化手法が，“要件 1：メンテナンス性に関する実装構造を把握できなくてはならない“を満たすための課題について述べる．

(1) 全ての実装構造について役割を正確に抽出することは難しい

設計情報や，開発者のドメイン知識を利用できる手法であれば，メンテナンス性に対して，どのような仕組みを導入しているのか容易に知ることができる．しかしながら，実装コードでは，多くの実装形態があるため，全ての構造が持つ役割を理解し，メンテナンス性への関与を判断し，抽出することは困難である．そのため，提案手法では，メンテナンス性に関係する構造を可視化し，抽出できることが必要である．

(2) コード規模拡大に伴い，構造の把握が困難になる

ソフトウェアの解析ツールは，近年，その性能やユーザビリティが向上している．しかし，同時にソフトウェアの規模も増大していることから，大規模なソースコードを解析する場合，構成要素が爆発的に検出され，コンピュータのメモリ不足に陥ることがある．また，全ての解析を完了したとしても，膨大な解析結果から適切な情報を抽出することや，検出された膨大な構成要素が表示されたダイアグラムを読み取ることは困難である．そのため，提案手法では，実装から生成したデータのうち，メンテナンス性に関係しないデータを捨象できることが必要である．

5.3. 拡張性可視化手法の概要

5.3.1. 構成

図 7 に，本章で説明する提案手法の概要を示す．ソースコードのみを入力とし，拡張性の仕組みを定量的に把握するため，ソースコードのクラス構造を継承とコントロールフローという異なる観点のモデルで表現する．当該モデルでは，拡張性の分析を行うために不必要なクラスを捨象することで，最小限度のクラスによる可視化を行うことができる．さらに，可視化した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い，機能と拡張性構造に関するマトリクスを算出する．

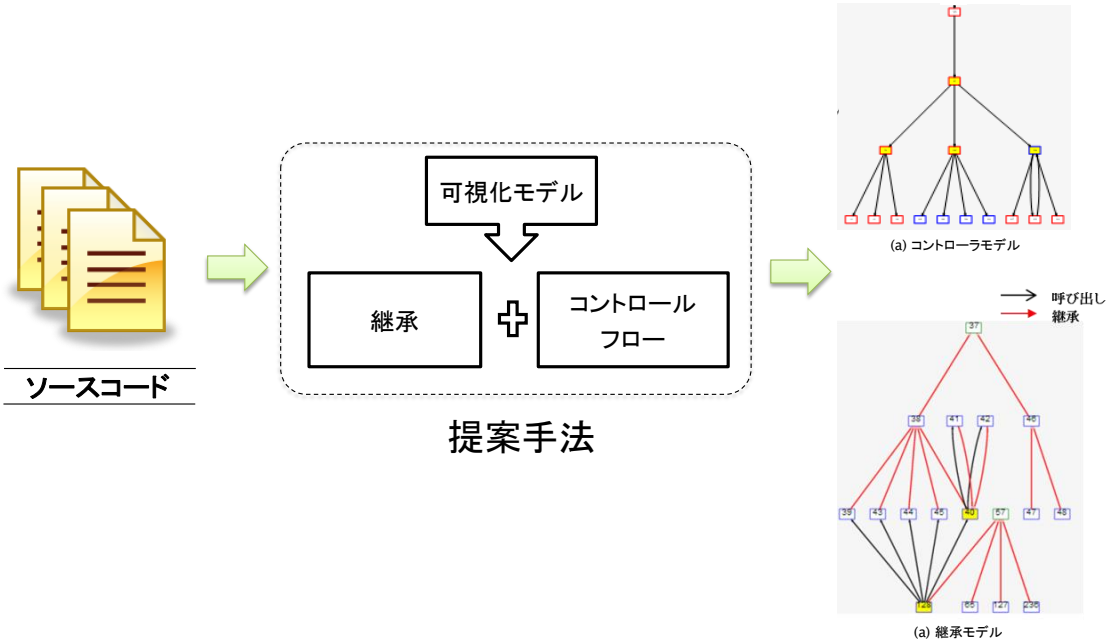


図 7　拡張性可視化手法の概要

5.3.2. 特徴

メンテナンス性に関係する構造を特定するために，下記 2 つの情報を利用した．

(1) オブジェクト指向言語の言語構造に含まれる特徴

継承関係，デザインパターンといった構造的な意味をもつオブジェクト指向言語を対象とし，可変性に関係する拡張性の構造を各々の観点で抽出・可視化を行うためのモデルを定義した．

(2) 組込みソフトウェアに特有のレイヤ構造に含まれる特徴

組込みソフトウェアでは，ハードウェアデバイスをリアルタイム制御しやすいレイヤ構造を持っていることが多い．そこで，組込みソフトウェアの概念的なレイヤ構造モデルを図 8 のように定義し，可視化した実装構造について当該モデルを用い整理する．

組込みソフトウェアには，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．

たとえば，ネットワークカメラの場合，ハードウェアデバイスであるカメラから規定されたフレームレートで映像が入力され，データ処理機能として，映像解析処理を行った後，ネットワークデバイス経由で，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，ロボットの場合，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，組込みソフトウェアでは，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．

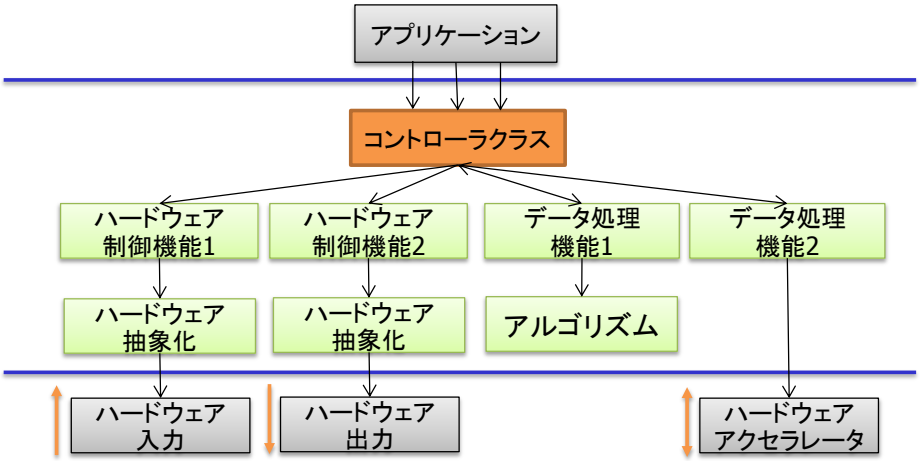


図 8　組込みソフトウェアにおける基本的レイヤ構造

5.3.3. 期待される効果

メンテナンス性が悪化したことで，歪んだアーキテクチャになった図 4 で示した例に対して，拡張性可視化手法を用いることで可視化される内容を図 9 に示す．拡張性可視化手法を適用すると，進化型組込みソフトウェアに含まれる下記構造を抽出・可視化することができる．

- 機能を切り替える可変メカニズム
- 可変メカニズムによって切り替えられる対象であるバリアント
- 機能のシーケンス制御を担うコントローラ
- 進化型組込みソフトウェアが持つ機能

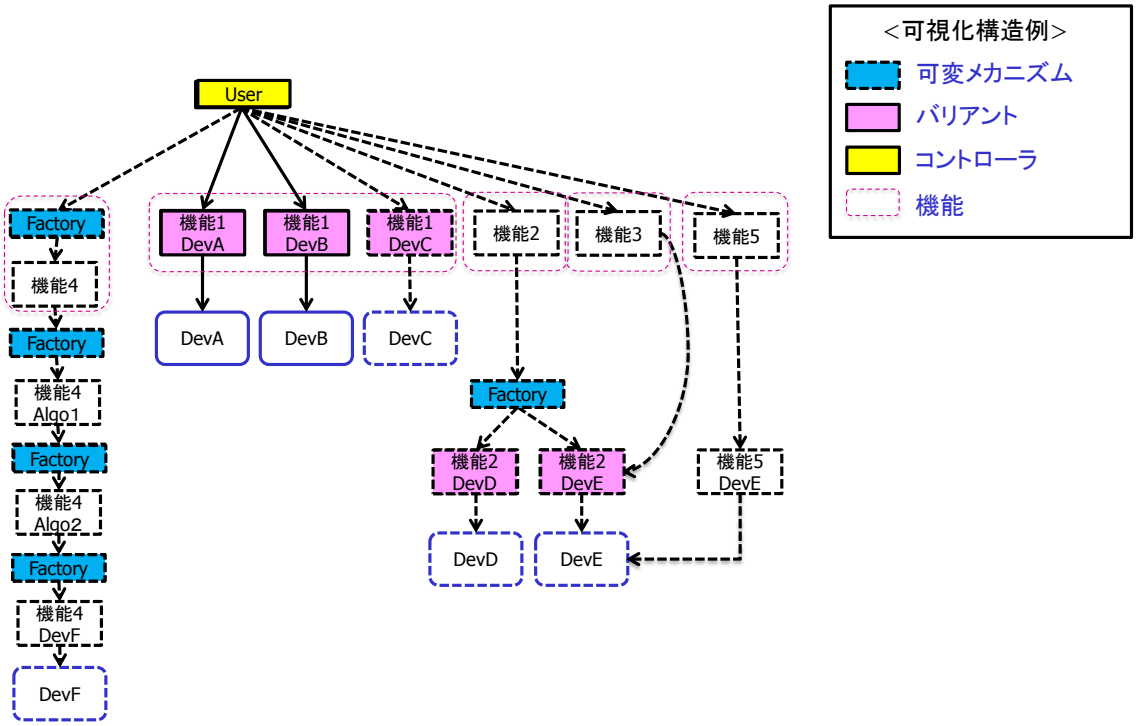


図 9　拡張性可視化手法による可視化内容

5.4. 可視化モデル

クラス間の呼び出し関係であるコントロールフローに着目したソースコードの可視化モデルとクラス間の継承関係に着目したソースコードの可視化モデルを定義する．これら 2 つの可視化モデルを，本研究では，コントローラモデルと，継承モデルと呼ぶ．なお，これらモデルは，後述する本提案手法を実装したツールにより生成することができる．

コントローラモデル

コントローラモデルの例を図 10 に示す．コントローラモデルは，拡張性，可変性に関し設計上考慮すべきクラスの切替え制御構造について観察することを目的とする．そこで，クラス間の呼び出し関係に着目し，まず一定の条件を満たすクラスを抽出する．次に，抽出されたクラスに関連するクラスをさらに抽出し分類する．そして，分類したクラスのうち，拡張性，可変性の分析上，影響が少ないと考えられるカテゴリについては，捨象し，残りのクラスを矢印線により接続することでツリー上に表現する．

リバースモデリングによってすべてのクラスが生成される一般的なクラス図と比べ，コントローラモデルでは，当該分析において不必要なクラスを全て捨象することができる．

クラス間の呼び出し関係は黒色の矢印線で示し，クラスは四角の BOX で示す．

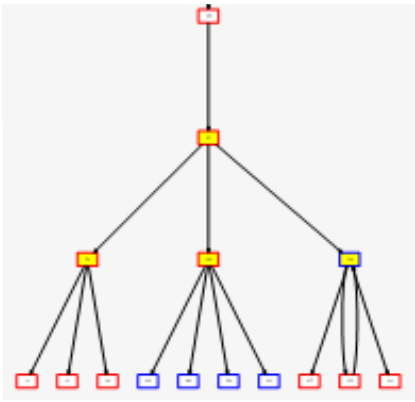


図 10　コントローラモデル

継承モデル

継承モデルの一例を図 11 に示す．継承モデルは，オブジェクト指向言語において重要な構造である継承構造と，継承されたクラスのインスタンス化について観察することを目的とする．継承モデルでは，Base クラス，または派生クラスを抽出する．そのため，継承構造を持たないクラスは捨象される．また，複数の派生クラスを呼び出すクラスについても継承モデルの中で表現する．

クラス間の継承関係は赤色の矢印線で示し，派生クラスの呼び出しは，黒の矢印線で示す．クラスは四角の BOX で示し，矢印で接続することでツリー上に表現する．

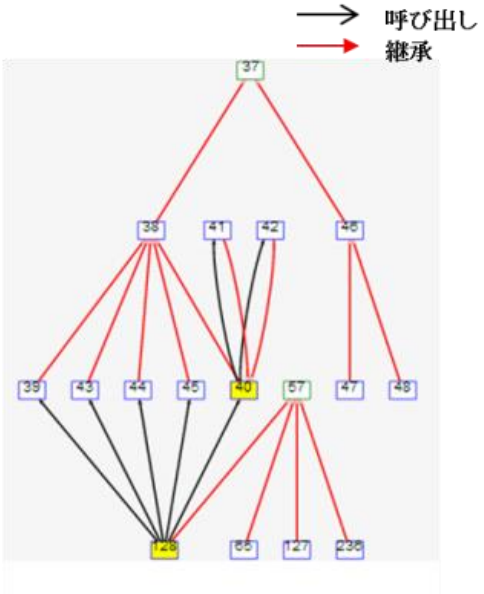


図 11　継承モデル

5.5. 可変構造

ソフトウェアの拡張性は，図 3 に示したように，時間方向と機能・性能方向の 2 種類がある．これらの拡張性において，何らかの変更を可能にする仕組みを可変構造として定義する．可変構造には，可変点と可変メカニズムがある．

可変点

類似した複数のクラスを切替えることで, 機能の選択を実現している箇所を可変点と呼び, 図 12 に示す.

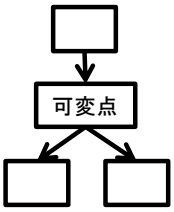


図 12　コントローラモデル内の可変点表現

可変メカニズム

クラスを切替える仕組みのことを可変メカニズムと呼ぶ．可変メカニズムは，後述するデザインパターンの構造を持つ.

5.6. モデル要素

5.6.1. コントローラモデル

コントローラモデルには，図 13 に示す要素がある.

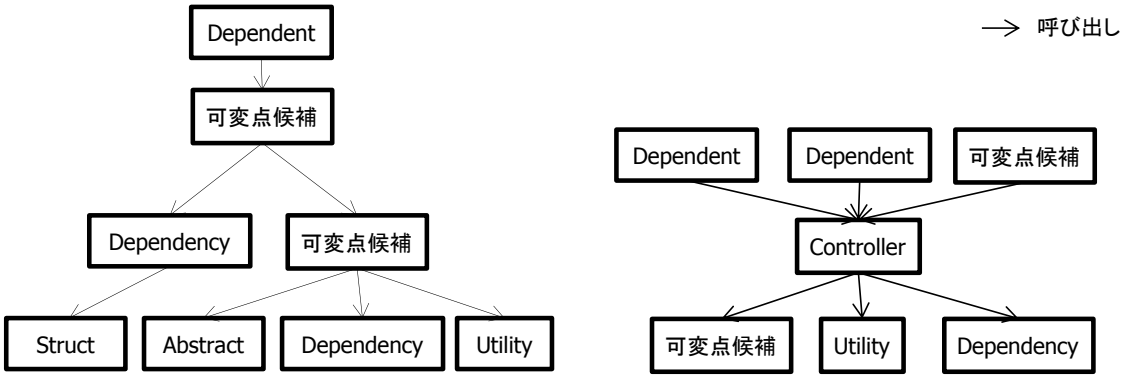
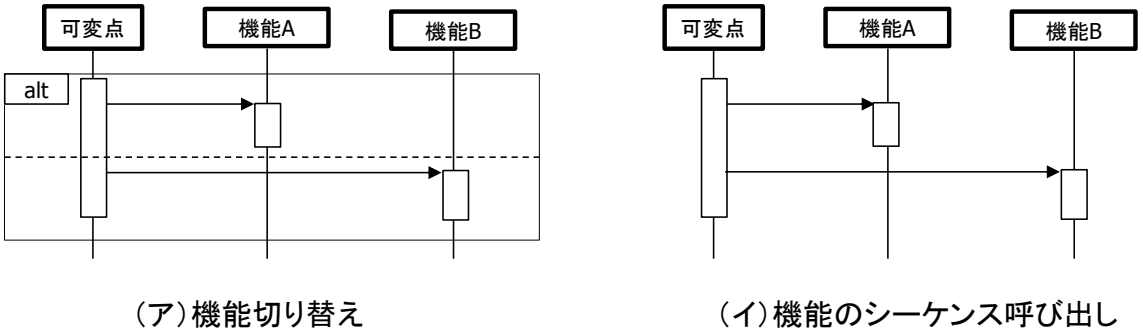


図 13　コントローラモデル要素

可変点候補クラス

クラスとして機能が実装されている場合，機能の切替えはクラスの切替えとして表現されるため，可変点では複数のクラスに対する，呼び出し関係が必須である．そこで，コントローラモデル内の一つのクラスから呼び出され，かつ 2 つ以上のクラスを呼び出しているクラスを抽出する．これらのクラスは，可変点である可能性があるが，コントローラモデルだけでは確定することができないため，候補クラスと呼ぶこととする．なお，可変点候補クラスは下記の 2 種類に識別される．

- 可変メカニズムを持ち，クラス生成を行うクラス（図 14：ア）
- シーケンス的な機能呼び出しを行うクラス（図 14：イ）



（ア）機能切り替え　　（イ）機能のシーケンス呼び出し

図 14　可変点候補クラスの種類によるシーケンス図

Dependency クラス

抽出された可変点候補クラスから呼び出されているクラスを Dependency クラスと定義する．ただし，Dependency クラスに分類されたクラスにおいて，可変点候補クラスの条件を満たす場合は，可変点候補クラスに分類される．Dependency クラスは，呼び出されるだけのクラスとなる．つまり，機能が実現すべき実処理を保有しているクラスである．

Dependent クラス

Dependency クラスとは逆で，抽出された可変点候補クラスを呼び出しているクラスを Dependent クラスと定義する．Dependent クラスも Dependency クラス同様，可変点候補クラスがネストしている場合，可変点候補クラスになりうる．Dependent クラスは，呼び出すだけのクラスとなる．一つのコントローラツリーにおいて，1 つ以上存在し，当該ソフトウェアのシーケンスと外部とのインタフェースを担うクラスである．

Controller クラス

このクラスが呼び出すクラス数，およびこのクラスを呼び出すクラス数が共に規定の閾値を超えているクラスを Controller クラスと定義する．このクラスは，対象ソフトウェア

のシーケンス制御や，データの管理を担っている箇所である．また，複数の異なるシーケンスがこのクラスを介して結合する場合もある．そのため，このクラスの検出が多いほど，対象ソフトウェアの複雑度が高いことを意味し，複数の異なる目的を持つ制御を含んでいることを示している．

Utility クラス

Dependency クラスの一つであるが，複数のクラスから呼び出されているだけの場合，Utility クラスと定義する．Utility クラスは複数の機能に対して，共通の処理を提供しているクラスである．この場合，拡張性という観点における分析では，この Utility クラスは不必要であり，また当該クラスが描画された場合，モデルの可読性を低下させるため，捨象することによって，コントローラモデルには表示されない．

Abstract クラス

実装をともなわないクラスであり，純粋仮想関数のみで構成されている抽象クラスを Abstract クラスと定義する．継承の観点における分析では，必要とするクラスであるが，コントロールフローの観点における分析では不必要であるため，捨象することによって，コントローラモデルには表示されない．

Struct クラス

オブジェクト指向言語におけるクラスではない構造体を Struct クラスと定義する．このクラスは，データを表現しているだけであり，機能の切替えという観点では不必要なため，捨象することによって，コントローラモデルには表示されない．

5.6.2. 継承モデル

継承モデルの要素を図 15 に示す．

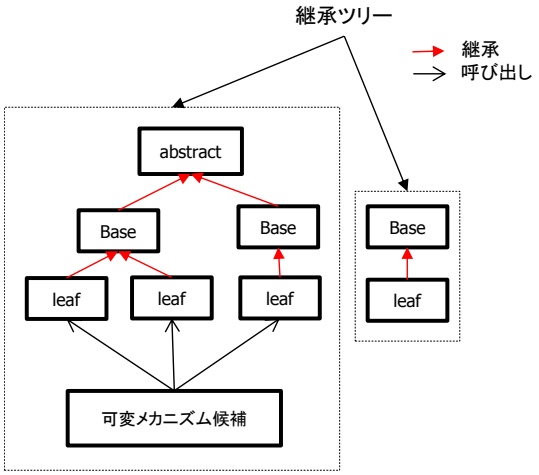


図 15　継承モデル要素

継承ツリー

抽象クラス，または具象クラスから派生したクラス群の固まりである．継承モデル内には，通常複数の継承ツリーが含まれている．派生クラスが多い継承ツリーは，平らな形として表現され，継承階層が深い継承ツリーは縦に細長く表示される．

Base クラス

継承元クラスである．継承元クラスには，純粋仮想関数のみが定義されている抽象クラス（インタフェースクラス），少なくとも一つは純粋仮想関数を持っている抽象クラス，純粋仮想関数を持っていない具象クラスがある．

Leaf クラス

Base クラスから派生しているクラスであり，継承ツリーの末端に位置するクラスである．このクラスは，すべて具象クラスである．Base クラスのない具象クラスは捨象されるため，当該モデル上に表示されない．

可変メカニズム候補クラス

一つのクラスから複数の leaf クラスを呼び出しているクラスである．このクラスは継承構造とは関係なく，継承モデル内の Leaf クラスへのアクセス関係から算出される．呼び出す Leaf クラスと継承ツリーの関係により，下記の種類に分けることができる．

- 一つの継承ツリー内の，同一 Base クラスから派生した複数の Leaf クラスを呼び出すクラス
- 一つの継承ツリー内の，異なる Base クラスから派生した複数の Leaf クラスを呼び出すクラス
- 異なる継承ツリーにおける Leaf クラスを呼び出すクラスのうち，同一継承ツリー内で複数の Leaf クラスを呼び出すクラス．
- 異なる継承ツリーにおける Leaf クラスを呼び出すクラスのうち，同一継承ツリー内では一つの Leaf クラスだけを呼び出すクラス．この場合は，Leaf クラス間の関係性がないため，機能の切替えを意図していないとし，捨象する．
- 複数の可変メカニズム候補クラスが同じ Leaf クラス群を呼び出す場合は，Leaf クラスがコントローラモデルで示すところの Utility クラスに該当する場合であるため，捨象する．

上記のうち，捨象対象ではないクラスについては，機能を切替える可変メカニズムである可能性があるが，継承モデルだけでは確定することができないため，可変点同様，候補クラスと呼ぶこととする．

5.7. 可変点および可変メカニズムの特定

図 16 に可変点および可変メカニズムクラスを特定する仕組みを示す．これらは，コントローラモデルと継承モデルで抽出された候補クラスを利用することで特定することができる．青点線で示したクラス A は，コントローラモデルでは可変点候補，継承モデルでは可変メカニズムクラス候補として抽出されるクラスである．

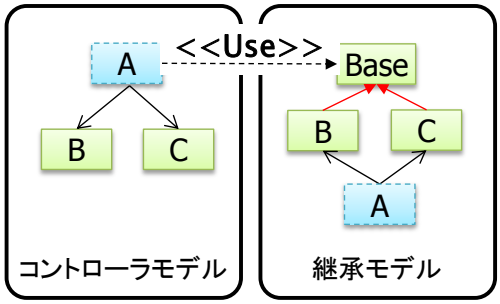


図 16　可変メカニズムの特定例

可変点候補および可変メカニズムクラス候補が同一のクラスであり，鎖線矢印で示すように，可変点候補であるクラス A が，切替え対象クラスの継承元である Base クラスを使用している場合，このクラス A を当該ソフトウェアにおける可変点かつ可変メカニズムクラスの一つとして特定する．可変点では，クラスの切替えを行う際，切替えたクラスによらず同じ扱い方を実現すると考えられるため，Base クラスにアクセスしていることを可変点特定の条件とする．なお，双方のモデルに含まれる候補クラスについて，クラス名が一致しない場合は，機能の切替えではなく，複数の機能をシーケンス的に呼び出している場合であるとして，対象から除外する．ここで，除外される可変点候補クラスは，図 14（イ）で示したパターンである．

5.8. 拡張性構造

5.8.1. 継承による拡張性構造

図 17 に，継承モデルから抽出する 2 種類の継承構造を示す．可変点同様，クラスとして機能を追加する場合に発生しうる変更量は，既存の継承構造に依存するため，継承構造を分析することは重要である．また一般的に継承階層が深くなるとメンテナンスコストが悪化するため[31]，本提案手法では 2 階層以内の継承構造を対象とする．なお，図 17 では，インタフェースとして使用する Base クラスを abstract クラスと記載する．

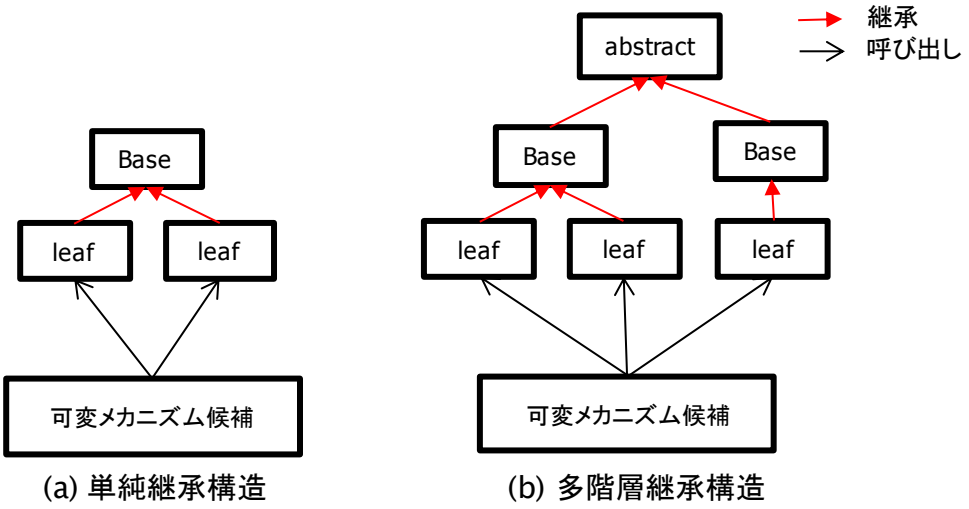


図 17　継承による拡張性構造

表 4 に継承による拡張性構造の抽出例を示す．コントローラモデルにおける階層と機能とのマトリクスで表現する． SI は単純継承構造，MI は多階層継承構造を示す．なお，コントローラクラスからの呼び出し深さが等しくなるクラス群を同一階層であるとし，最も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 8 に示したレイヤ構造を持つソフトウェアを対象としているため，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意味し，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意味している．

表 4 継承による拡張性構造の抽出例

階層 / 機能	第一階層	第二階層	第三階層以下
機能A	—	SI	SI
機能B	MI	SI	MI
機能C	—	—	—

単純継承構造

図 17 (a) に示す構造は，最も単純な継承構造であり，二つ以上の leaf クラスと Base クラスからなる継承構造である. Base クラスをインタフェースとして, leaf クラスを切替え，生成する場合に使用される．

多階層継承構造

図 17 (b)に示す構造は，異なる継承ツリー同士が一つ上位階層にある abstract クラスまたは Base クラスを介して接続している継承構造である．凝集度を高めながら，異なる複数の継承ツリーに所属するクラスによって機能を切替える場合に使用される．

5.8.2. デザインパターンによる拡張性構造

デザインパターンによる拡張性構造では，機能を切替える仕組みである可変メカニズムが抽出される．抽出される可変メカニズムを下記に示す 6 つのデザインパターンとして定義する．ここでは，特定の機能をクラスとして実装しており，クラスを切り替えることで，機能切り替えを行うということを前提としている．そのため対象とするデザインパターンは，クラス切り替えにより機能切り替えを実現可能なクラス生成に関するデザインパターンとする．

- GOF のデザインパターン[20]で定義されている，オブジェクトの生成に関係する 5 パターン（Factory Method, Abstract Factory, Builder, Prototype, Singleton）から，オブジェクトの使用者と生成者の分離という観点で選択した Factory Method パターンと Abstract Factory パターン
- 基本的な Factory パターン
- 動的な機能追加を実現する Plugin Factory パターン
- 使用と生成が分離されていないパターン（ここでは，Mixed Creation and Use と呼ぶ.）
- Template Method の参照関係が反転しているパターン（ここでは, Inverse Template Method と呼ぶ.）.

上記，可変メカニズムのパターン例を図 18 に示す．なお，User と示されたクラスは，クラスのインスタンスであるオブジェクトの使用者であり，Factory と示されたクラスは，オブジェクトの生成者である. また, A と B は同じ Base クラスから派生したクラスであり，切替え対象機能が実装されているクラスである．

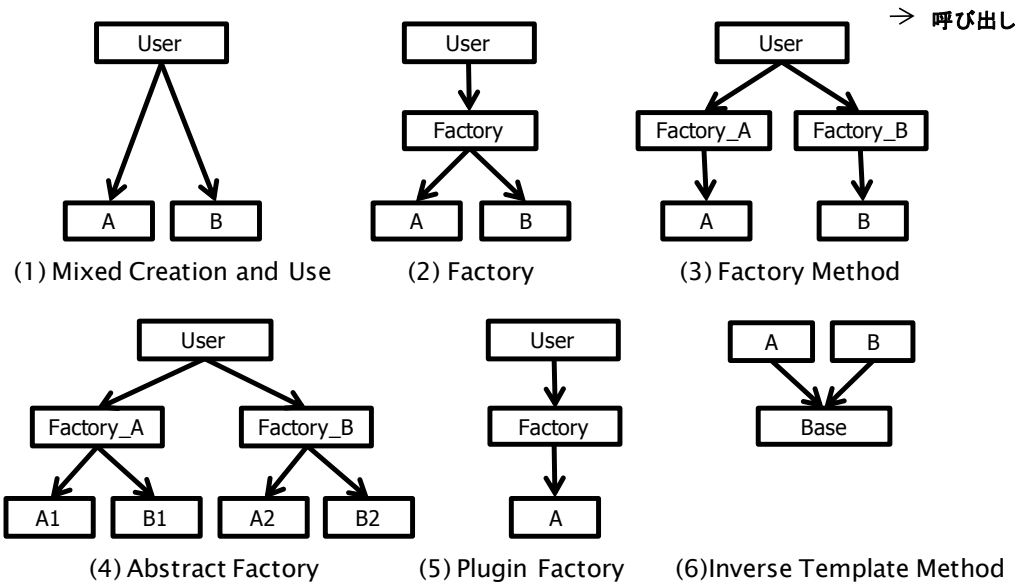


図 18　コントローラモデルにおける可変メカニズム例

表 5 にデザインパターンによる拡張性構造の抽出例を示す．表 4 と同じく，コントローラモデルにおける階層と機能とのマトリクスで表現する．CU は，Mixed Creation and Use パターン，F は Factory パターン，FM は，Factory Method パターン，AF は Abstract Factory パターン，P は Plugin Factory パターン，ITF は Inverse Template Method パターンを示す．階層の表現についても表 4 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意味し，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意味している．

表 5 デザインパターンによる拡張性構造の抽出例

階層 / 機能	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	―	―	―

Mixed Creation and Use パターン

部品を使用した処理アルゴリズムが記述されているクラス内で，部品となるオブジェクトの生成処理も行っているパターンである．この場合，部品の変更による影響が常に当該クラスに影響する．また複雑になりがちな処理アルゴリズム記述クラス内に，部品の生成処理も含まれることで可読性の低下も起きやすい．このパターンの例を図 18 (1)に示す．

図 18 (1)では，オブジェクトの使用者である User クラスが可変点であり，生成者も兼ね

ているため，Factory クラスが存在しない．

Factory パターン

部品を使用した処理アルゴリズムが記述されているクラス内では，部品として使用するオブジェクトの生成処理を行わず，生成処理を担当する別のクラス内に部品の生成処理を隠蔽し，使用するクラスからは抽象クラスを介してアクセスするパターンである．これにより，使用と生成を分離することができる．しかしながら，生成すべき部品の種類が増加すると，生成処理を担当するクラスが巨大化，複雑化する．このパターンの例を図 18 (2)に示す．図 18 (2)では，Factory クラスが可変点である．

Factory Method パターン

Factory パターン同様，使用と生成を分離することができ，さらに部品ごとに生成処理を分離することができるため，生成処理の複雑化を回避することができる．しかしながら，生成部品の種類が増えると，クラス数が増加し，ソフトウェア全体の可読性が低下する．このパターンの例を図 18 (3)に示す．

図 18 (3)では，User クラスが可変点であり，Factory_A および Factory_B は単一のクラスのみを生成するため，可変点として認識されない．

Abstract Factory パターン

部品の組合せを使用側から隠蔽するパターンである．部品生成において，複数の異なる組合せが存在する場合，組合せパターンの追加が容易である．しかしながら，組合せ対象部品が追加された場合，組合せを生成するすべてのクラスにおいて変更が入るため，部品の追加・削除が多い箇所では変更時の影響範囲が広くなる．このパターンの例を図 18 (4)に示す．

図 18 (4)では，User クラスが可変点である．また，A1 と A2 は同じ Base クラスを持ち，B1 と B2 も同じ Base クラスを持つが，A 系と B 系は異なる継承構造である場合，Factory_A および Factory_B は，可変点として認識されない．一方，A 系と B 系が同じ継承構造である場合，可変点として認識される．

Plugin Factory パターン

部品として使用するクラスの生成処理をコード上に直接記述せず，動的に部品を生成する仕組みである．継承モデル上には，生成される部品が継承構造として表現されるが，コントローラモデルでは，抽象クラスを介した使用側による部品へのアクセスが表現されるだけである．生成条件処理の記述がないため，変更に対する影響も小さい．しかしながら，動的ローディングをともなうため，切替えによるパフォーマンス低下が無視できない箇所では使用することが難しい．このパターンの例を図 18 (5)に示す．

Inverse Template Method パターン

テンプレートメソッドパターンの逆パターンとなる構造であり，当該パターンは，間違った使用方法である可変メカニズムを検出するものである．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を使うことで，機能の切替えが行われる．しかしながら，派生クラスが，親クラスで定義されたメソッドを呼び出し，さらに独自処理を追加することにより，親クラスの想定していない振舞い，および機能の切替えを表現する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ソフトウェアが予期しない動作となる可能性がある．このパターンの例を図 18 (6)に示す.

5.9. 可変メカニズムパターンの判別

可変メカニズムパターンは，コントローラモデルと継承モデルのセットを用い判別される．図 19 に Plugin Factory パターン，Inverse Template Method パターンを除いた可変メカニズムの判別フローを示す.

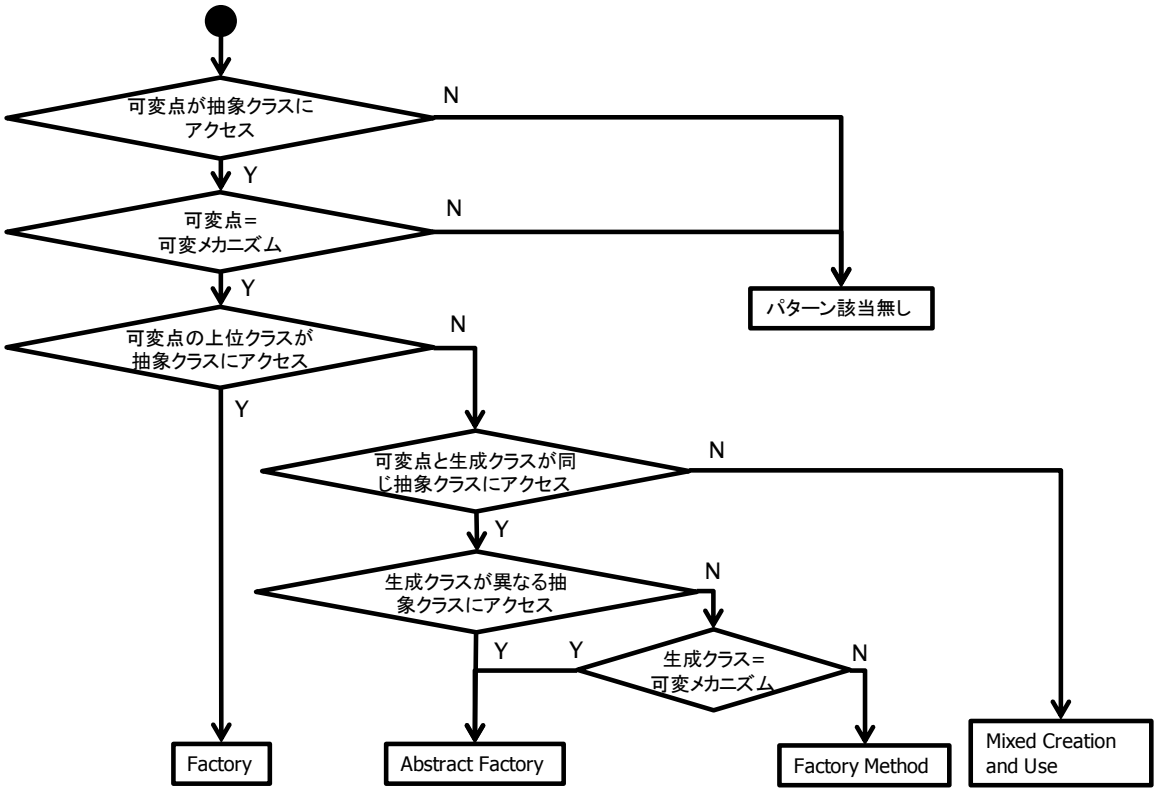


図 19　可変メカニズム判別フロー

Factory パターンでは，図 18 (2)に示す通り，Factory が生成したオブジェクト A または B を User クラスが使用するため，User クラスもオブジェクト A または B の継承元である抽象クラスにアクセスする．

Factory Method パターンでは，図 18 (3)に示す通り，可変点である User クラスが生成した Factory_A および Factory_B が，オブジェクト A または B を生成するため，当該 Factory は，オブジェクト A または B の継承元である抽象クラスにアクセスする．さらに User クラスも，生成されたオブジェクト A または B を使用するため，オブジェクト A または B の継承元である抽象クラスにアクセスする．

Abstract Factory パターンでは，図 18 (4)に示す通り，Factory_A および Factory_B が生成するクラスの継承構造によって，2 種類の判定を行う．各々異なる継承構造のクラスを生成する場合は，アクセスする抽象クラスの数を判定に用いる．一方，同じ継承構造のクラスを生成する場合は，Factory_A および Factory_B が可変点であるかを用いる．

Plugin Factory パターンは，ファイルから部品クラスを読みだす等の動的ローディングメカニズム箇所をコード上からキーワードにより抽出する．

Inverse Template Method パターンは，継承モデル上で継承構造関係になっているクラスが，コントローラモデル上での呼び出し関係において，図 18 (6)に示す通りの関係になっている場合に抽出する．

5.10. 可変メカニズムマッピング

検出された可変メカニズムについて，図 8 のようなレイヤ構造へのマッピング，および影響する機能へのマッピングの例を図 20 に示す．青点線で示したクラス A，D，G は可変点である．

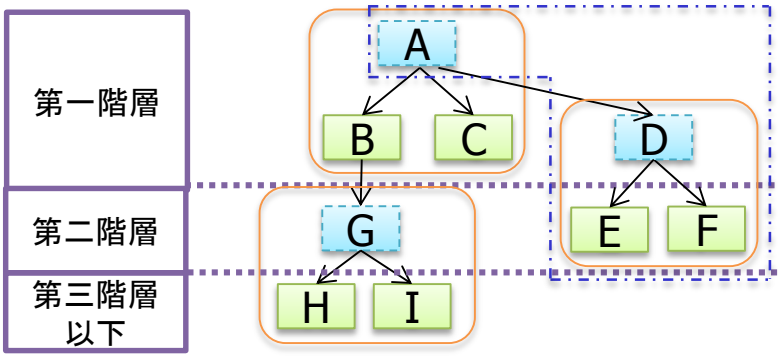


図 20　可変メカニズムのマッピング例

(1) レイヤマッピング

レイヤマッピングでは，抽出された各可変メカニズムがどのレイヤに導入されているか

を特定する．可変メカニズムを使用するクラスを導入箇所とすると，導入箇所は，図 18 の User クラスに該当する．

図 20 において，すべての可変メカニズムが Mixed Creation and Use パターンの場合，導入箇所はクラス A，D，G となり，橙枠で示した枠内に 1 つずつ可変メカニズムが導入されていることとなる．一方，クラス D の可変メカニズムが Factory パターンであった場合は，Mixed Creation and Use パターンの例とは異なり，A，G が導入箇所となる．

(2) 機能系統マッピング

機能系統とは，単一の機能の一部を担うクラスの集合であり，コントローラモデルにおいて深さ方向に伸びる枝の一つで表現されるものである．また，図 8 のようなレイヤ構造においては，コントローラクラスから呼び出されているクラスが各々機能を指しているものとする．

図 20 において，可変メカニズムが導入されている機能系統はクラス B，C，D から各々呼び出されるクラス群となる．ただし，コントローラクラスに可変メカニズムが導入されており，第 1 階層にある複数のクラスを切替えている場合は，切替え対象となるクラスを含む複数の機能系統を同一の機能系統であるとみなす．たとえば図 20 において，クラス A に可変メカニズムが導入されており，クラス B と C を切り替えている場合は， B および C を同一機能系統とする．よってクラス B および C の機能系統には，合計 2 つの可変メカニズムがカウントされる．

5.11. 進化型組込みソフトウェアへの適用

進化過程である 1 種類の組込みソフトウェアに対して，提案手法を適用した．対象とした組込みソフトウェアは，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を用いた．なお，対象コードは代表的なオブジェクト指向言語である C++で記述されている．

5.11.1. コードの概略

ソフトウェア進化過程における対象コードの一覧を表 6 に示す．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．

表 6 ソフトウェア進化過程における対象コード一覧

ID	進化内容
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

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 21 に示す．

使用したコードは，2 万行～4 万行弱のコード規模であり，製品コードの一部である．本提案手法においては，図 8 に示したようにアプリケーション部とハードウェア入出力部に挟まれた部分を対象としている．

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．なお，提案手法で扱うクラス数は，約半数程度に捨象されている．

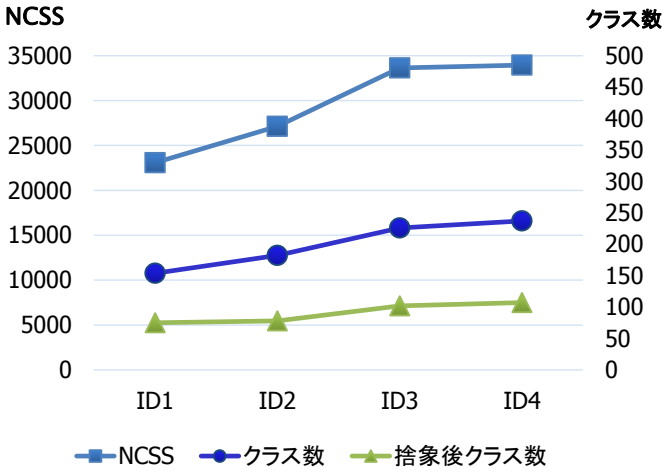


図 21　コード規模とクラス数の推移

5.11.2.　構造可視化

作成したツールを用い，各 ID のコントローラモデルおよび継承モデルを生成した．生成された各モデルを図 22～図 29 に示す．図 22 および図 23 には，対象とする組込みソフトウェアの初期製品（ID1）におけるコントローラモデルと継承モデルを示す．図 24 および図 25 には，ID1 に対し，機能追加を実施したコントローラモデルと継承モデルを示す（ID2）．図 26 および図 27 には，さらに ID2 に対し，機能追加を行ったコントローラモデルと継承モデルを示す（ID3）．図 28 および図 29 には，ID3 に対し，提案手法による拡

張性強化を行ったコントローラモデルと継承モデルを示す（ID4）．なお，ID1 および ID2 のコントローラモデル上には，抽出された拡張性構造であるデザインパターンの情報も重ねて表示してある．

コントローラモデルおよび継承モデルを進化過程の順に観察することで，どのように変化していくかを容易に理解することができる．たとえば，コントローラモデルの ID2 において，新しいハードウェアデバイスに対応するための進化が発生したことで，コントローラモデルの ID3 では，点線で示した領域が複雑化していることがわかる．さらに，このことは，継承モデルにも表れており，継承構造に変化が起きたことを知ることができる．また，コントローラクラスに処理が集中し複雑化していることが，コントローラモデル ID1 から ID3 を比較することでわかる．なお，ID1 から ID3 における継承モデルでは，多くの継承構造が一つのクラスから呼び出されるようになっていることが見てとれる．

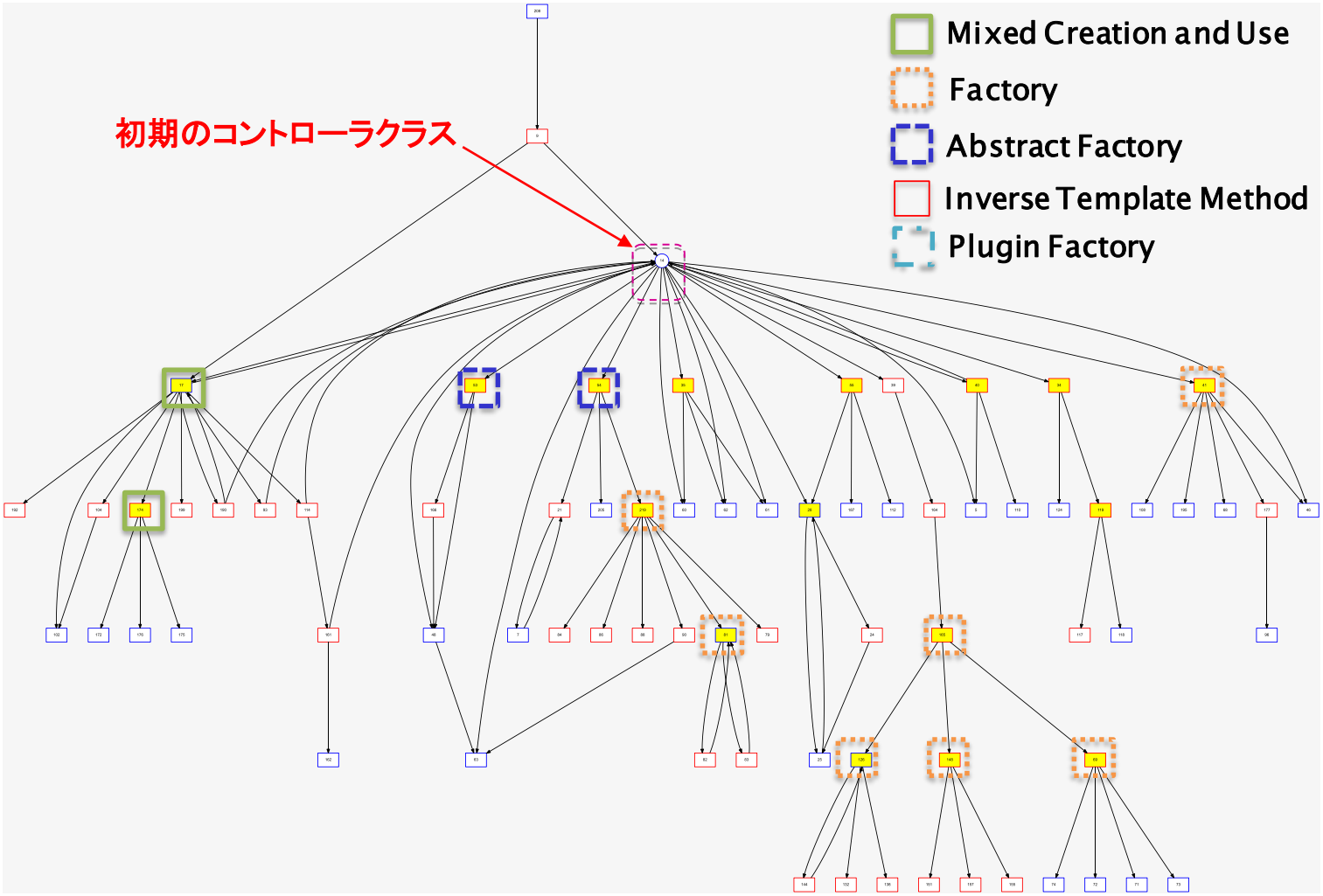


図 22　コントローラモデル 図（ID1）

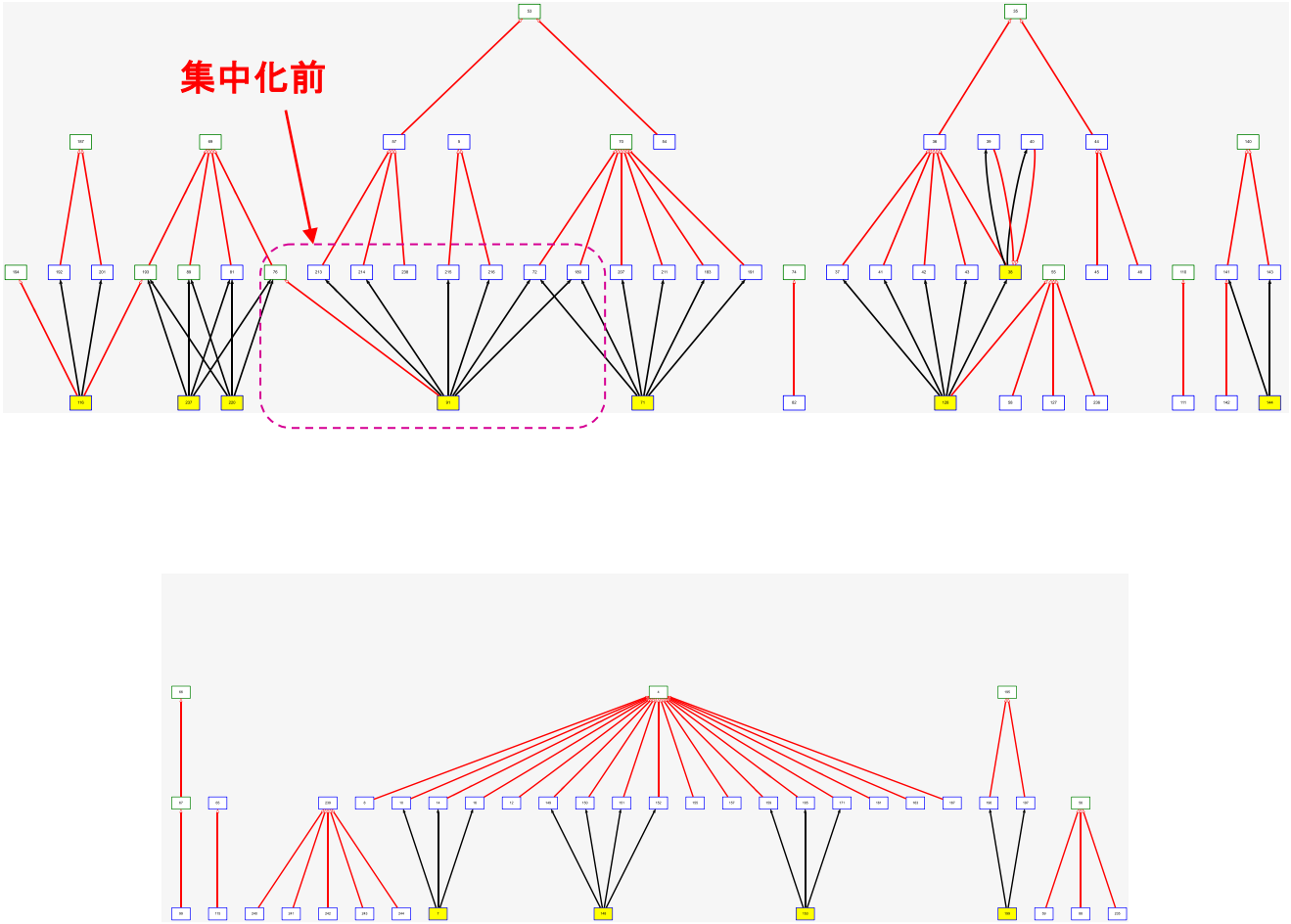


図 23　継承モデル 図（ID1）

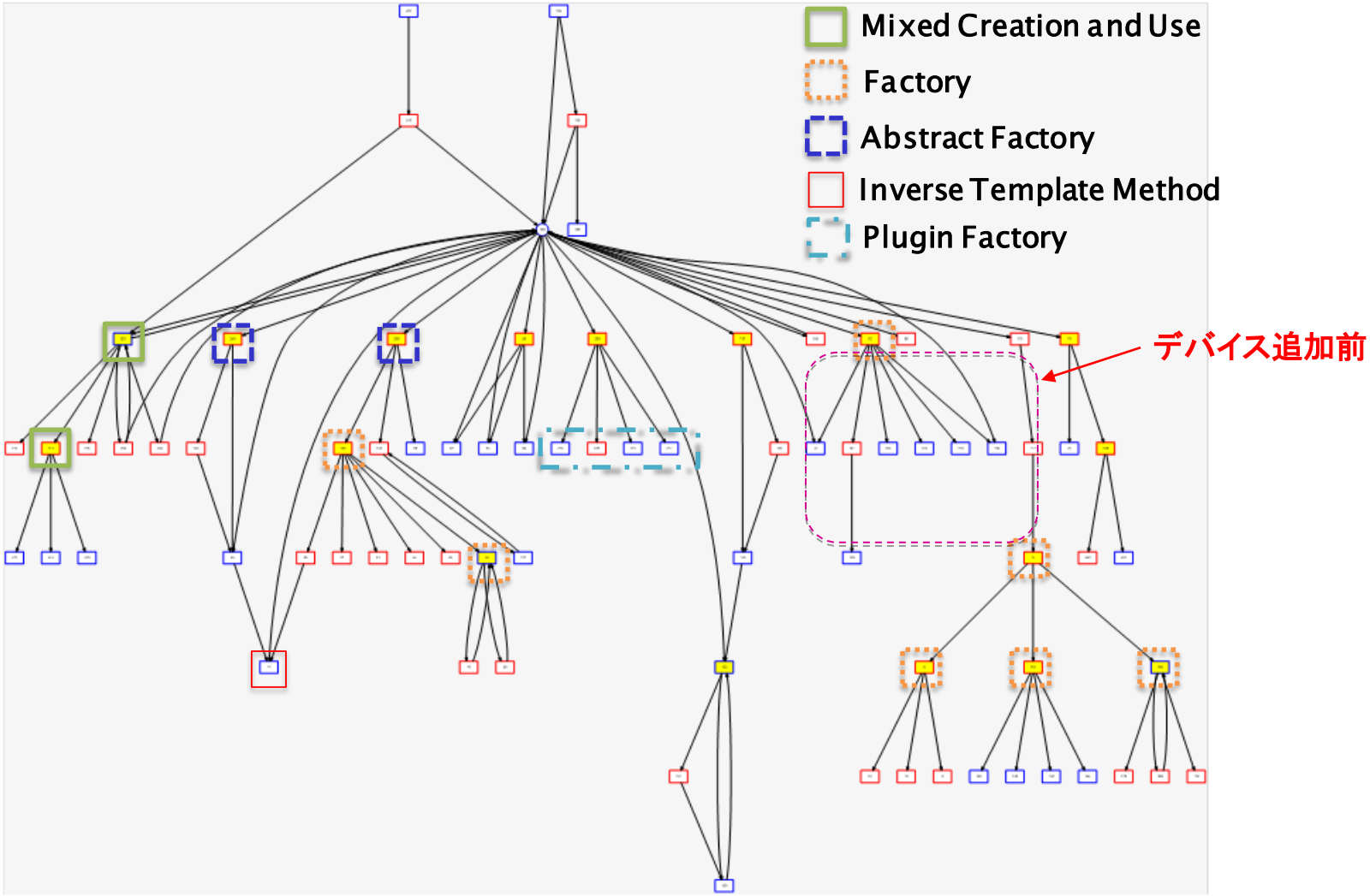


図 24　コントローラモデル 図（ID2）

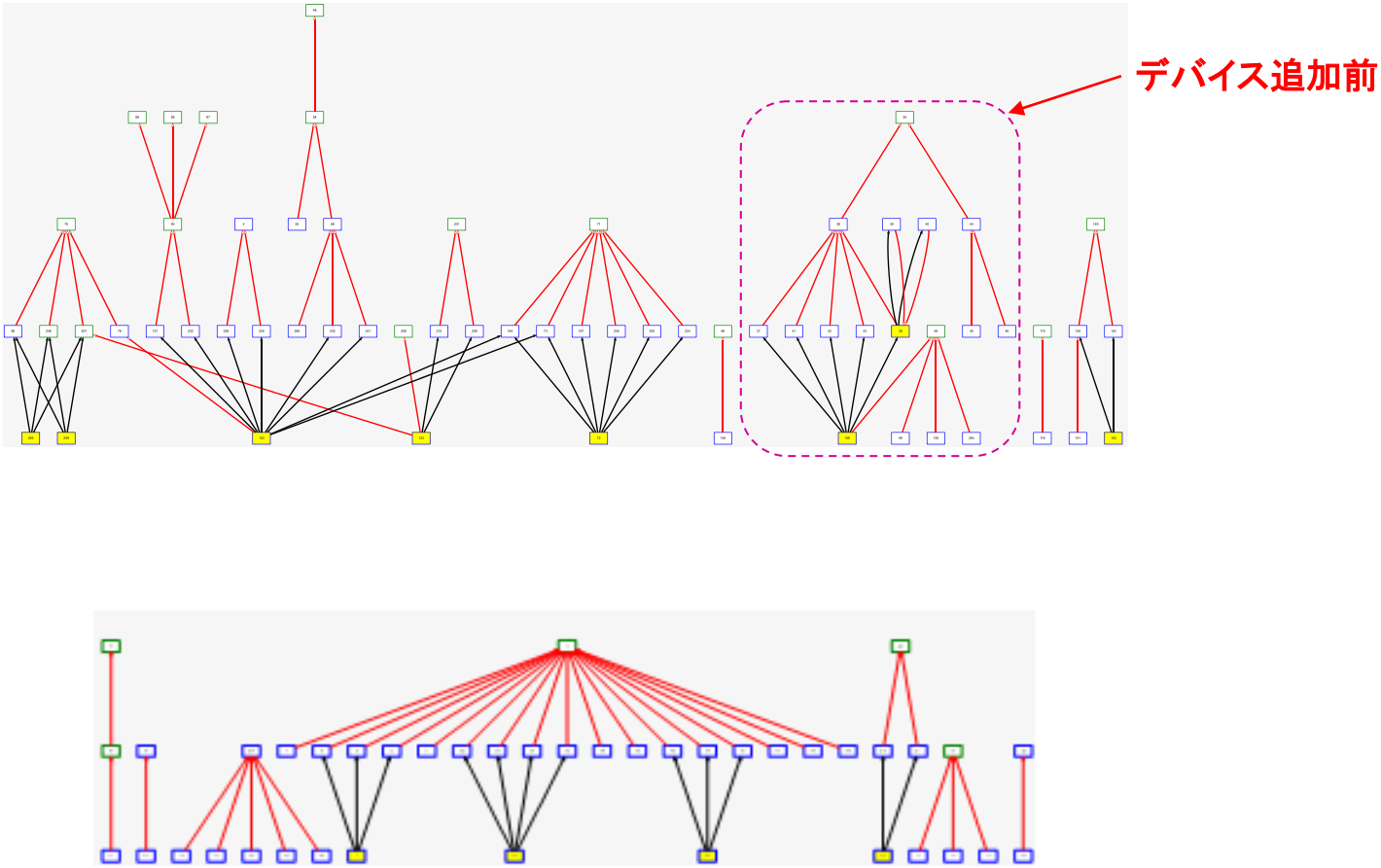


図 25　継承モデル 図（ID2）

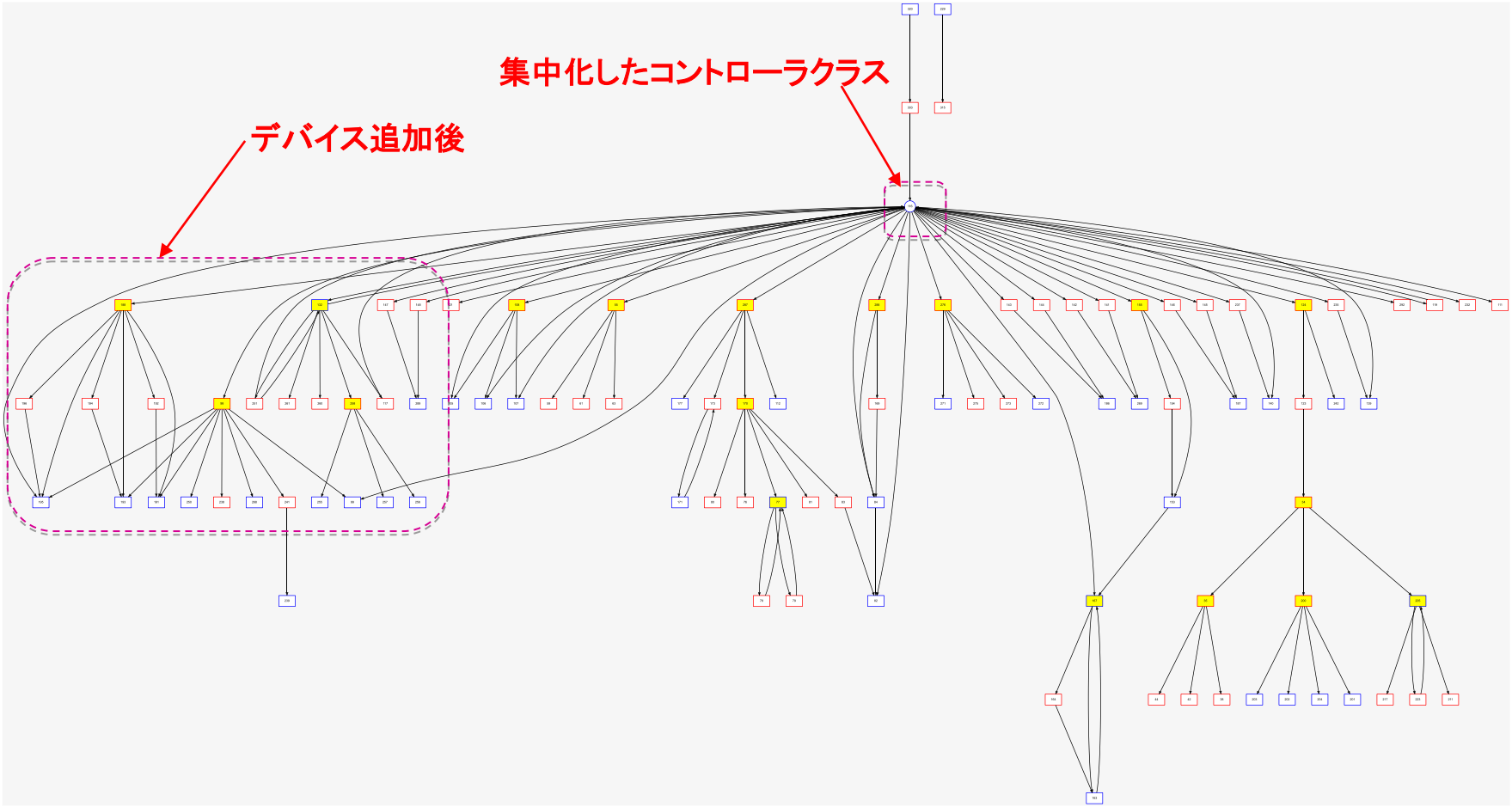
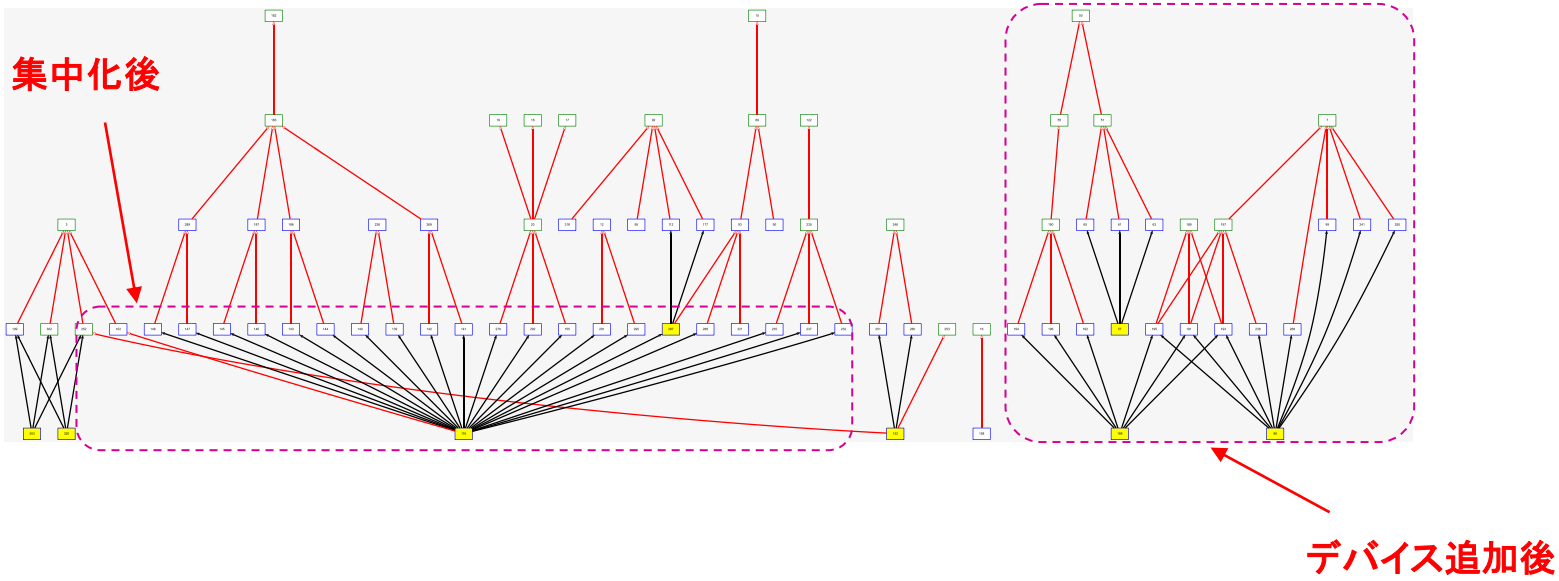


図 26　コントローラモデル 図（ID3）



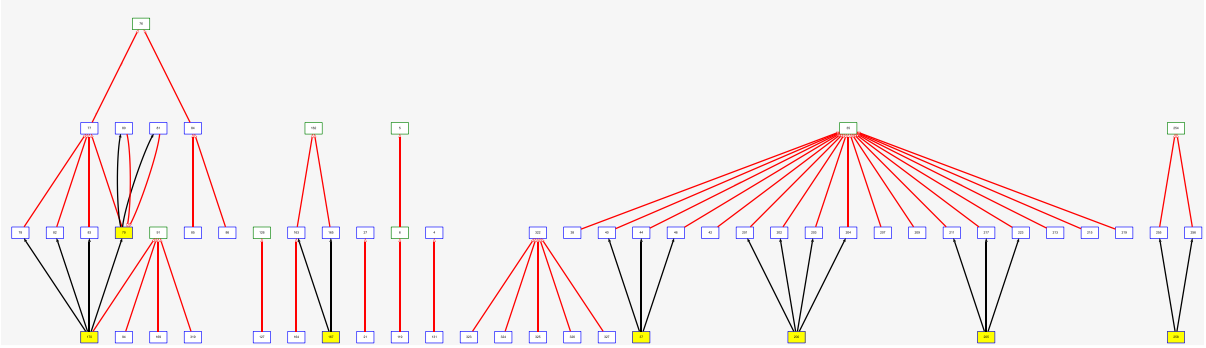


図 27　継承モデル 図（ID3）

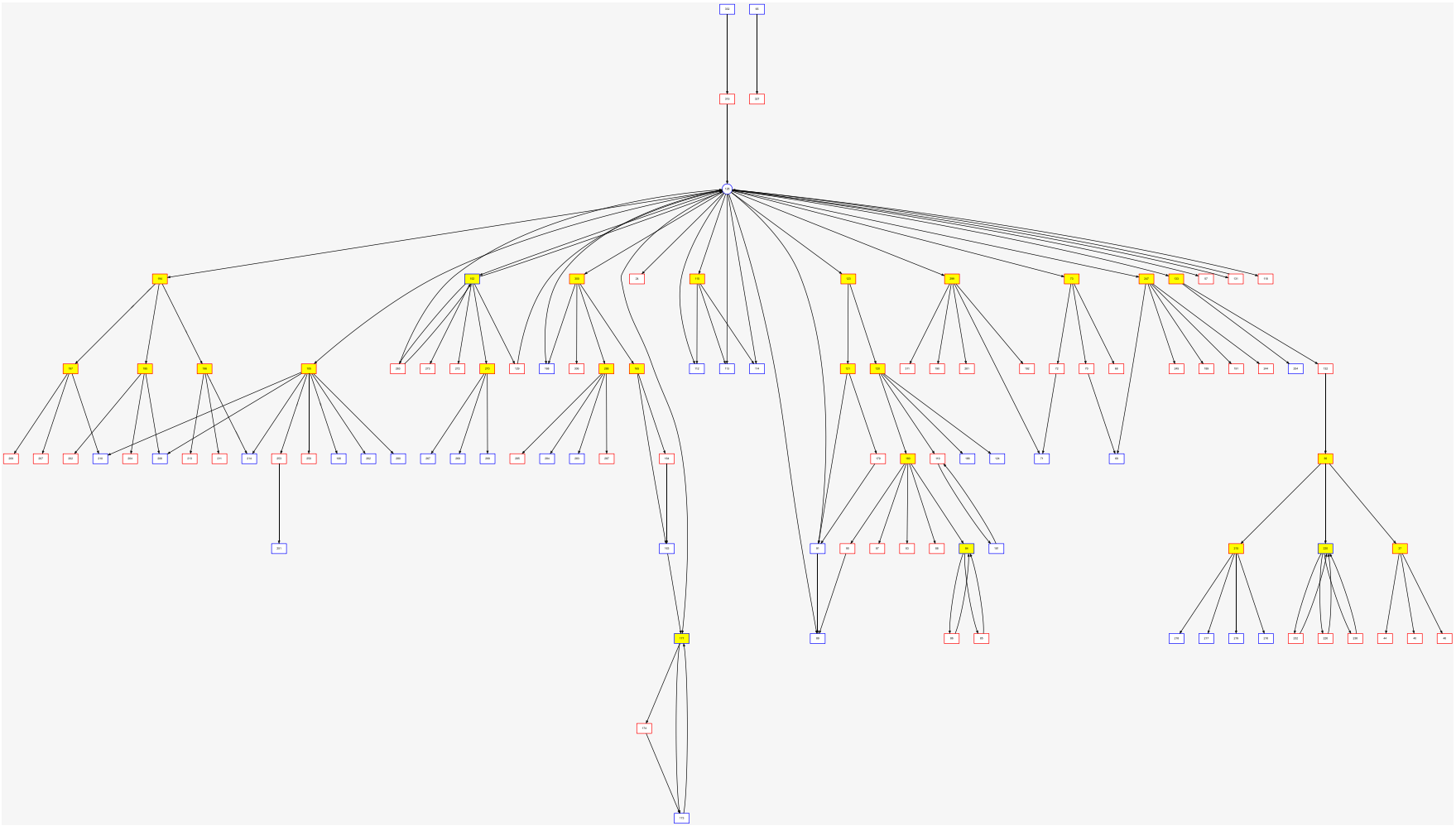
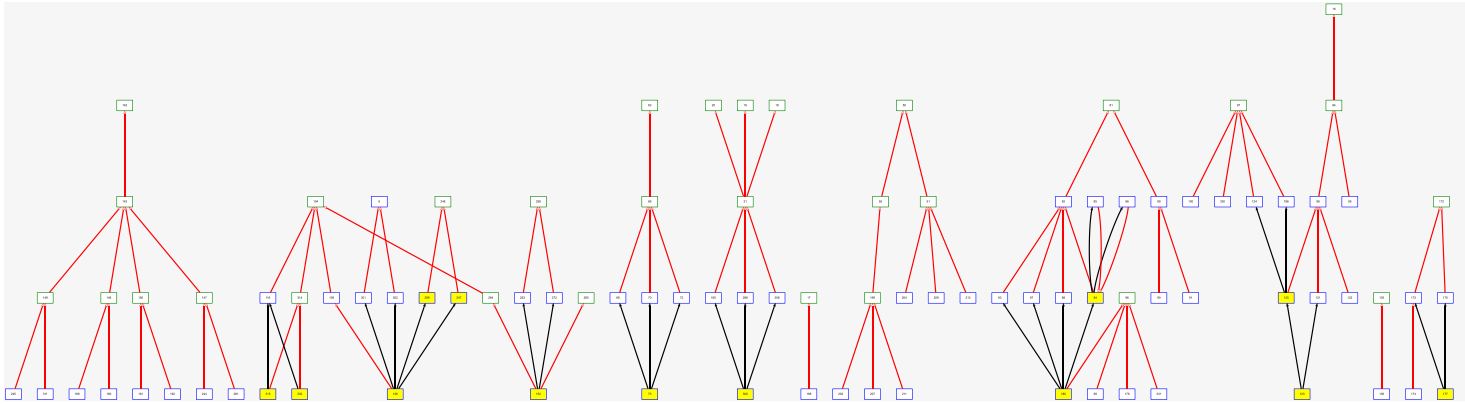


図 28　コントローラモデル 図（ID4）



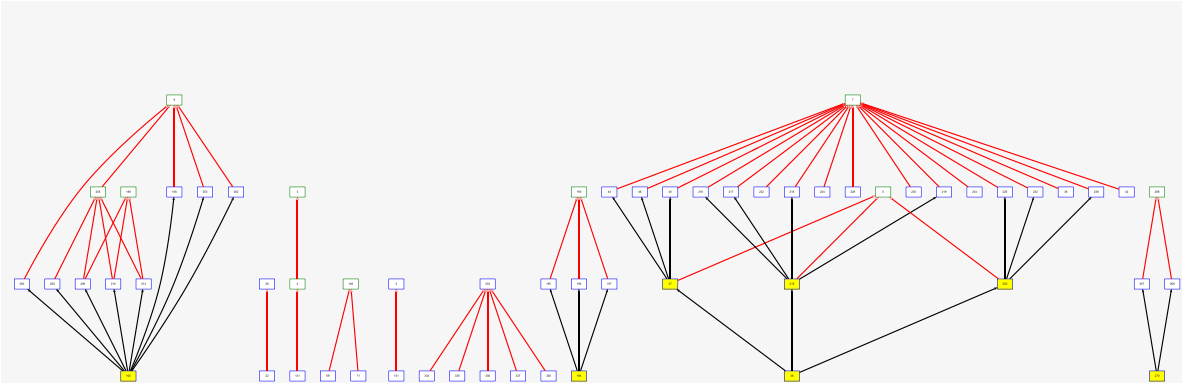


図 29　継承モデル 図（ID4）

5.11.3. 拡張性構造識別

ID1～4 について抽出された機能数を図 30 に示す．機能数とは，5.10 節で述べた機能系統の数である．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意味している．表 6 に示した組込みソフトウェアにおけるコントローラクラスの NCSS およびクラス結合度（CBO[15]）の推移を図 31 に示す．

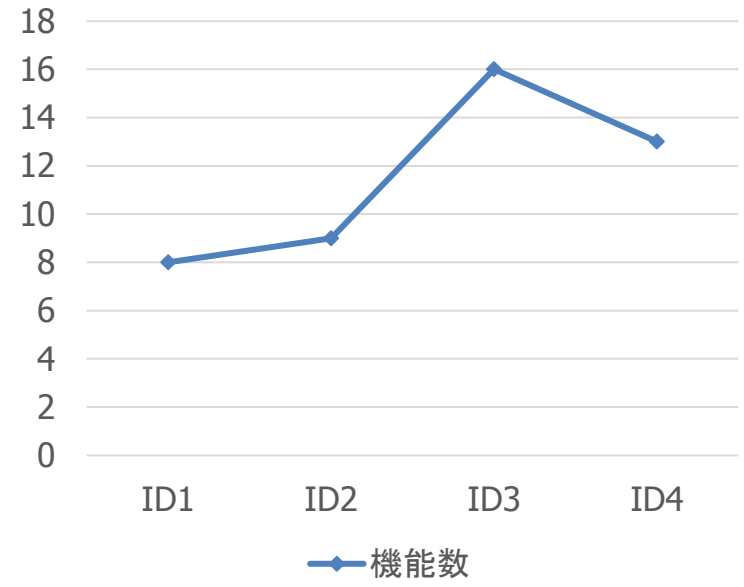


図 30　機能数の推移

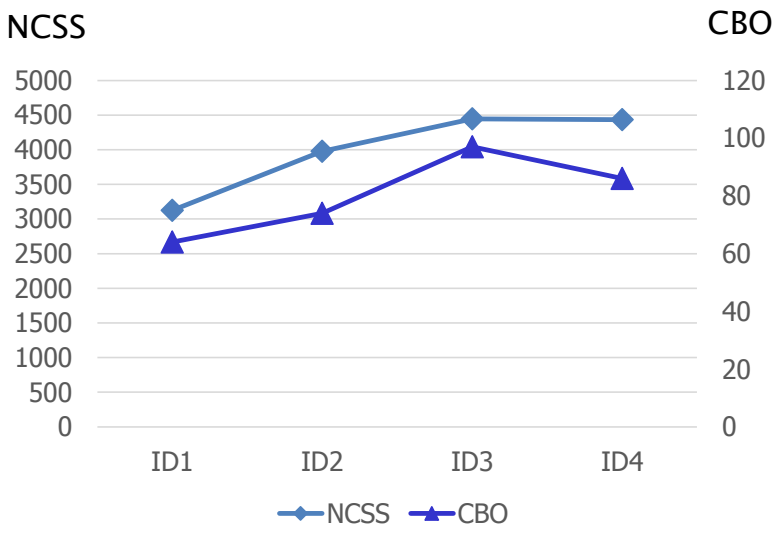


図 31　コントローラクラスにおける変更推移

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，提案手法による拡張性構造の強化により，16 から 13 へと機能数の減少が見られる．これは，ID3 と ID4 において機能的な違いはないが，メンテナンス性の悪化によりコントローラクラスとの結合が増加し，機能数が実際より多く見えていた部分の解消によるものである．

ID1～ID4 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図 32 に示す．

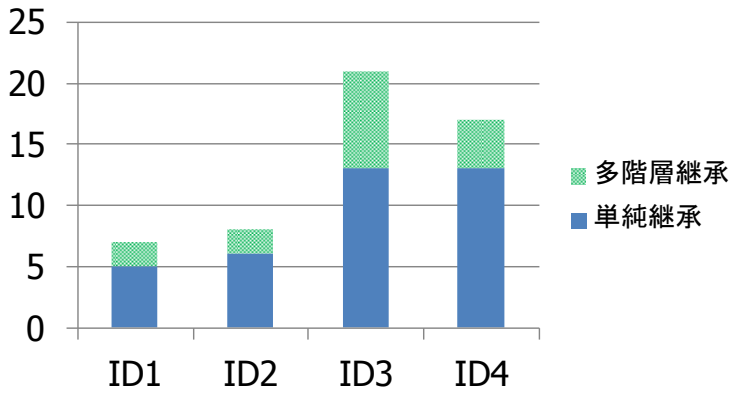


図 32　継承構造に関する拡張性構造の推移

ID3 のコードにおいて，継承構造の増加がみられる．

ID1～ID4 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図 33 に示す．

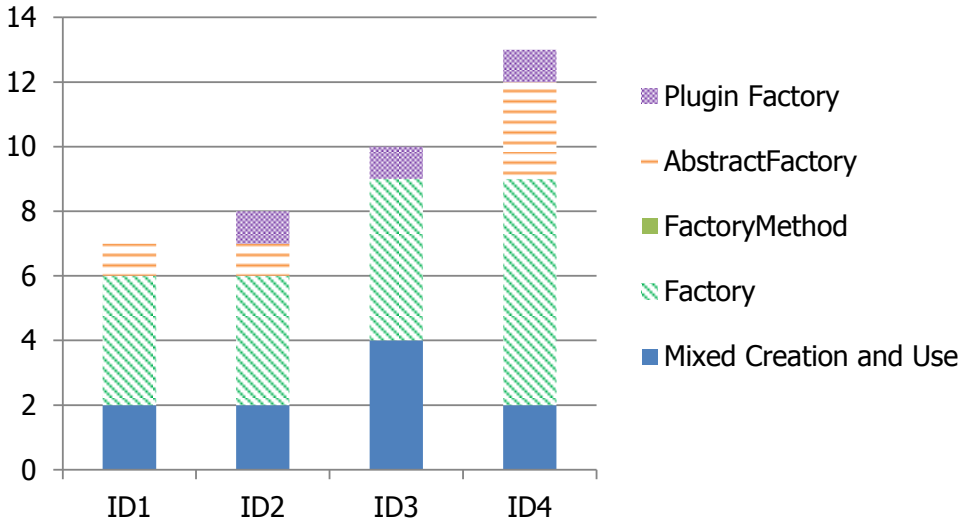


図 33　デザインパターンに関する拡張性構造の推移

ID3 のコードにおいて，Mixed Creation and Use パターンの増加がみられる．また，ID4

のコードでは，提案手法によって，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．

5.11.4. 提案した可視化手法におけるクラス捨象率

提案した可視化手法におけるクラス捨象率を比較した．比較対象は，提案手法によるコントローラモデルと既存ツール出力によるクラス図と手動で作成した概念モデル図である．また，可視化には ID1 のコートを用いた．図 34 に作成した概念モデルを示す．

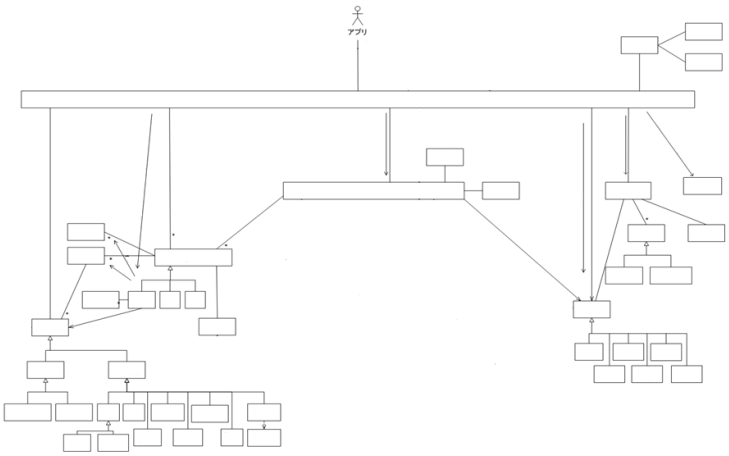


図 34　手動で作成した ID 1 における概念モデル

比較対象である概念モデルは，表 6 に示した ID1 から ID2 への変更を行う際に，変更すべき箇所の特定や変更方法の検討を目的として作成したものである．したがって，提案手法による可視化の目的と一致するため，比較対象の要件を満たしている．

また，概念モデルの作成にあたっては，対象ソフトウェアのドメイン知識を持たないソフトウェアの分析者 1 名（10 年以上のソフトウェア経験者）が対象ソフトウェアの開発者 2 名（5 年以上のソフトウェア経験者 1 名，10 年以上のソフトウェア経験者 1 名）にヒアリングを行いながら, ID1 に対するコードの構造および振舞いを把握するために必要と思われるクラスを人手により，抽出することで実施した．

表 7 クラス捨象率比較

	クラス図	提案手法（コントローラモデル）	概念モデル
クラス数	154	75	44
捨象率	0%	51.3%	71.4%

クラス捨象率の比較結果を表 7 に示す．提案した可視化手法は，手動で作成した概念モ

デルに比べ捨象率は低いが，一般的なクラス図を用いた手法に比べ，解読すべきモデル数が半数程度に削減されていることが分かる．

5.11.5. 提案した可視化手法におけるクラス抽出精度

可視化手法におけるクラスの抽出精度を比較した．比較対象は，コントローラモデルと概念モデルである．また，可視化には前項同様 ID1 のコード用いた．当該比較では，コントローラモデルと概念モデルで出力されたクラスの論理和である 88 クラスを対象とし，抽出精度は，出力されなかったクラスの内訳により求める．表 8 に出力クラス数と未出力クラス数の比較を示す．また，各モデルにおける未出力クラス数の内訳を図 35 および図 36 に示す．

表 8 提案手法と概念モデルの未出力クラス比較

	提案手法（コントローラモデル）	概念モデル
出力クラス数	75	44
未出力クラス数	13	44

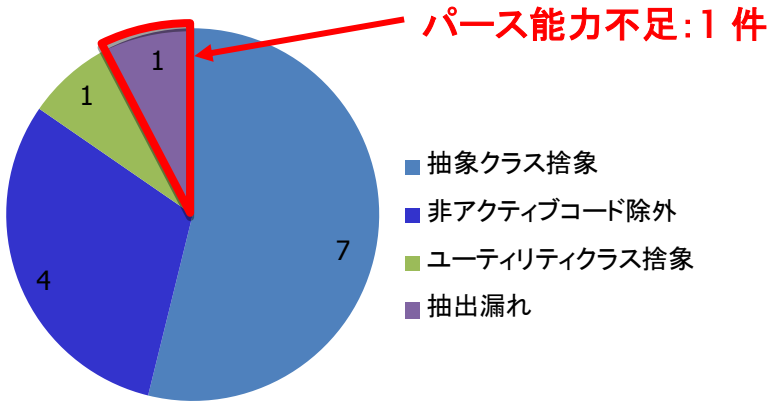


図 35 提案手法での未出力クラス内訳

提案手法で未出力とされた 13 件のクラスのうち，1 件の抽出漏れを除くと，提案手法内部で捨象したクラス，および#if 0 等のプリプロセッサにより非アクティブコード化されているため，意図的に除外したクラスであった．なお，1 件の抽出漏れが起きた原因は，該当箇所がスマートポインタを使ったコード記述であり，この記述は，作成したツールの解析対象外であった．これは提案手法の限界ではなく，提案手法を実行するためのツールにおけるプログラム言語のパース能力に依存するものである．

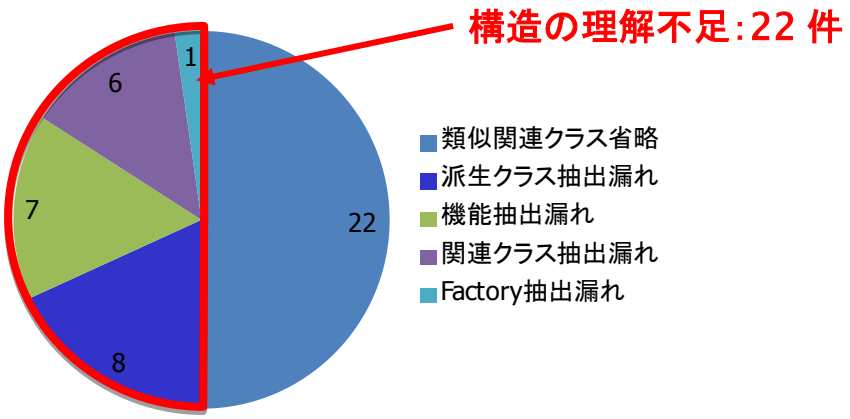


図 36　概念モデルでの未出力クラス内訳

概念モデルで未検出とされた 44 件のクラスのうち，50%のクラスは，出力されたクラスと同じ Base クラスから派生した類似構造を持つクラスであった．これらは，人手による概念モデルの作成工数を削減するため，意図的に省略したものと思われる．しかしながら，残り半数のクラスについては，出力されたクラスとは構造が異なっており，ソフトウェアの正確な構造理解には必要なクラスであるため，抽出漏れとした．なぜなら，今回の分析者は，ドメイン知識をもっていないため，ソースコードを短期間で完全に解読することは，現実的に不可能である．また，既存ツールにより自動生成されるクラス図等を参考にしたとしても，クラス図から，クラスの役割を把握・取捨選択することは困難であるため，重要なクラスの見落としが起きていた．一方，ドメイン知識を持った開発者にとっても，ソフトウェアの理解に必要なクラスを，複雑で大規模なコードから漏れなく取捨選択することは容易ではない．

5.12.　他ソフトウェアドメインへの適用

組込みソフトウェア以外のソフトウェアドメインに対して提案手法が適用可能か検証することを目的とし，3 種類のソフトウェアに対して，提案手法を適用した．対象ソフトウェアは，オープンソースソフトウェア（OSS）として公開されており，代表的なオブジェクト指向言語である C++で記述されたソースコードである．

5.12.1.　コードの概略

表 9 には，組込みソフトウェア以外のドメインにおける 3 種類のソフトウェアを示す．なお，比較対象とした組込みソフトウェアは表 6 に示した ID1 のコードである．

表 9 異なるソフトウェアドメインに対する適用対象コード一覧

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

図 37 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含まれるクラス数（捨象後のクラス数）を示す．

ID A は，他の対象と比較して，コード行数に対するクラス数が 3 倍程度多く，捨象されたクラス数の割合も他の対象より多い．つまり，ID A は，小さなクラスが多く，またそれらのクラスの大多数は，分岐するようなクラス呼び出し関係を持たず，呼び出しに対する処理が 1 対 1 になるような単純な構造が多いと推測できる．

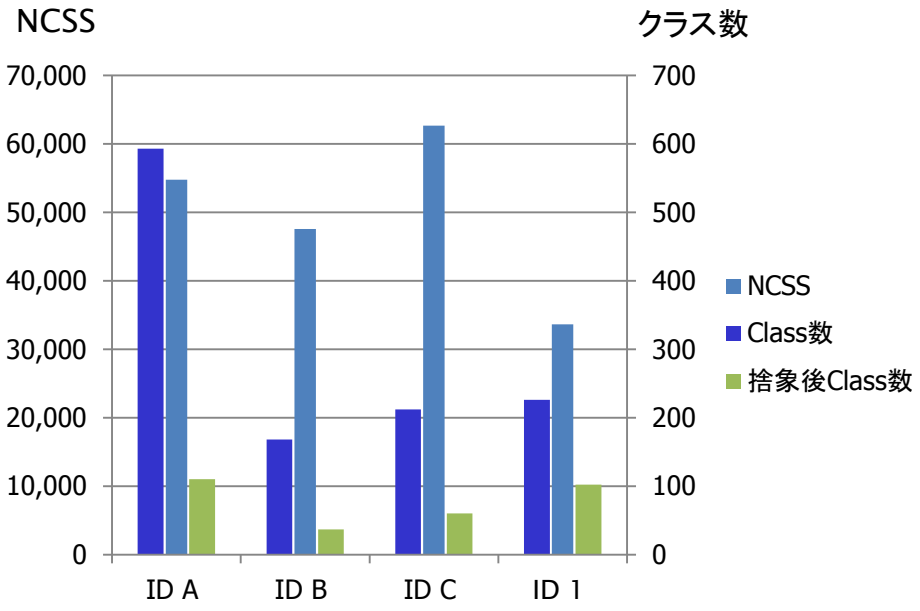


図 37　対象コードの計測データ比較

5.12.2.　構造可視化

- **ID A**（データベースソフトウェア）

ID A のソフトウェアにおけるコントローラモデルと継承モデルを図 38，図 39 に示す．ID A では，ソフトウェアのシーケンス制御部をあらわす Controller クラスが 3 つ存在する．また，リアルタイムの制御や，コントローラ層/ハードウェアデバイス層等のレイヤ分割が

存在しないことにより，図 8 に示した組込みソフトウェアのレイヤ構成にならない．そのため，構成理解が難しく，適用が困難である．

継承モデルでは，わずか 7 クラスが可変メカニズム候補となっている．また，サブクラスを多く持つ継承ツリーが 1 つ存在する．

- **ID B**（画像比較ソフトウェア）

ID B のソフトウェアにおけるコントローラモデルと継承モデルを図 40，図 41 に示す．ID B も，3 つの Controller クラスを持つ．これらは，画像比較ソフトウェアにおける各々の機能に該当するものである．ID B も，ID A 同様に，リアルタイムの制御や，コントローラ層/ハードウェアデバイス層等のレイヤ分割が存在しないことにより，図 8 に示した組込みソフトウェアのレイヤ構成にならず，構成理解が難しい．

さらに，IDB の継承モデルでは，可変メカニズム候補クラスが存在しない．

- **ID C**（ムービープレーヤソフトウェア）

ID C のソフトウェアにおけるコントローラモデルと継承モデルを図 42，図 43 に示す．

ID C では，ID A，ID B とは異なり，図 8 に示した組込みソフトウェアのレイヤ構成に似た形状を示している．内部処理として，組込みソフトウェア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等があるためである．しかしながら，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り，Controller クラスが 2 つになっている．

継承モデルでは, わずか 3 クラスの可変メカニズム候補クラスが存在するだけであるが, コントローラモデルとマッピングすることで，可変メカニズムとなるクラスである．

- **ID 1**（組込みソフトウェア）

ID 1 のソフトウェアにおけるコントローラモデルと継承モデルについては，既に図 22，図 23 で示しているが，コントローラモデルに含まれるクラスの半数以上にあたる 34 のクラスに可変点候補が存在し，継承モデルでは，12 個の可変メカニズム候補がある．

図 38　コントローラモデル 図（IDA）

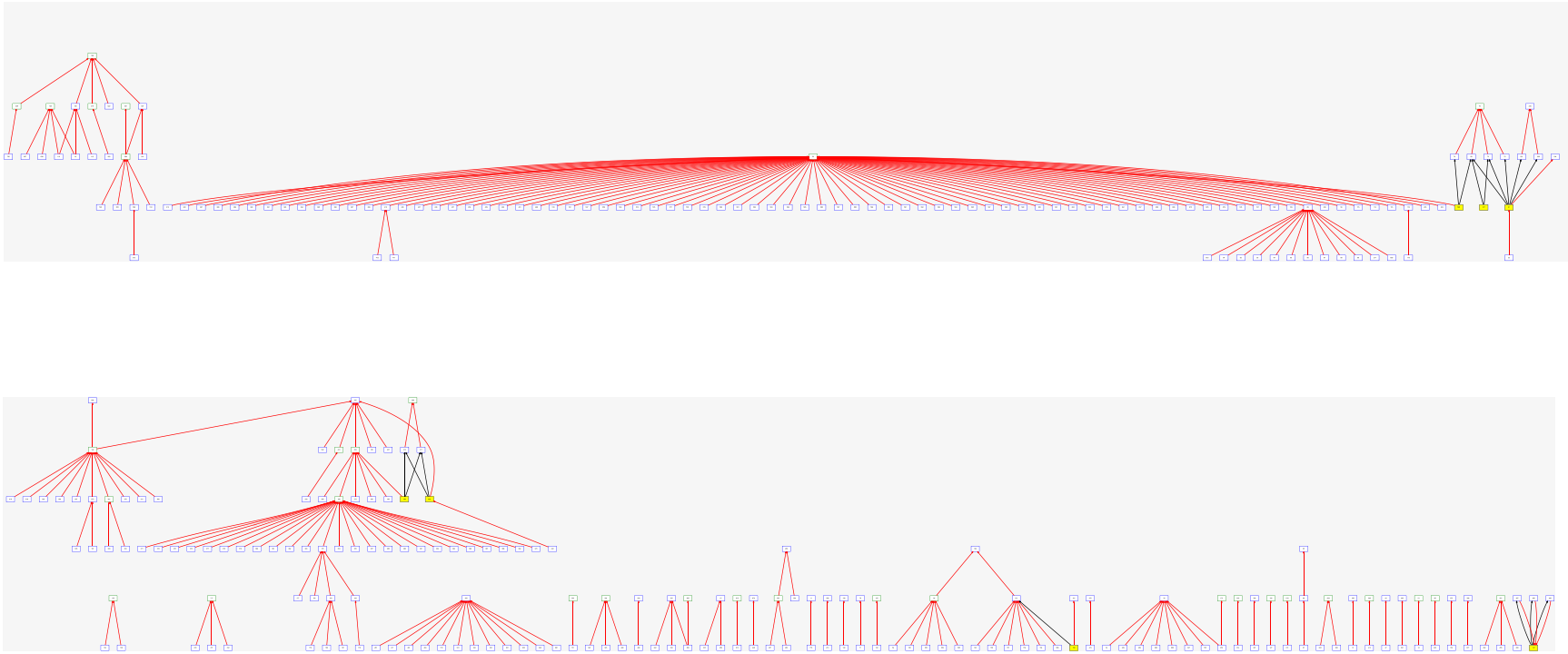


図 39　継承モデル 図（IDA）

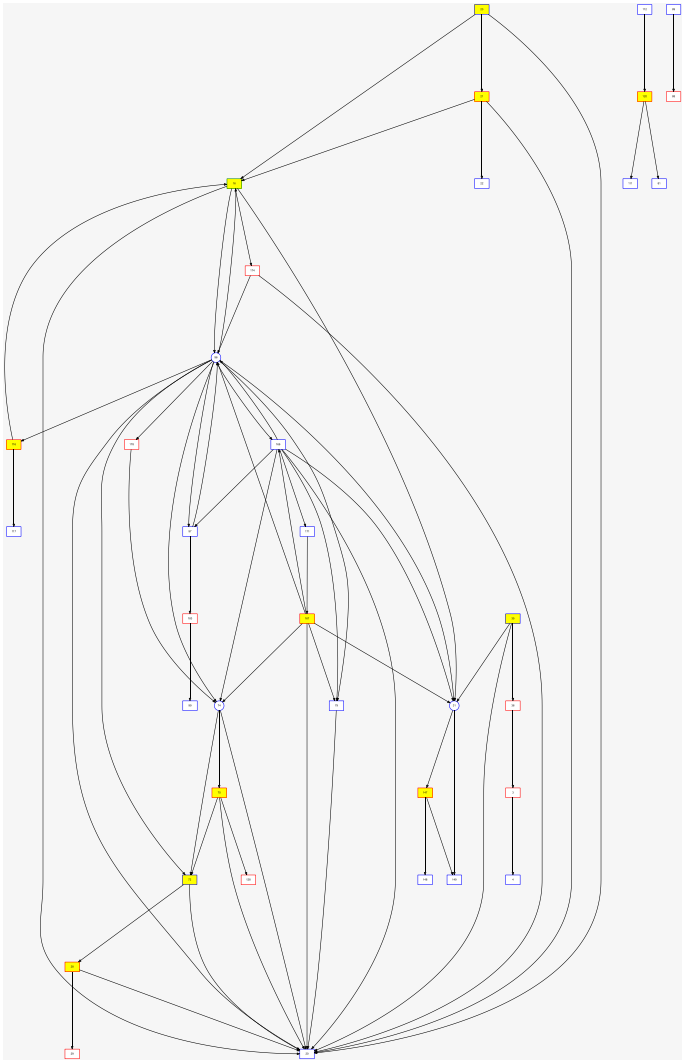


図 40　コントローラモデル 図（IDB）

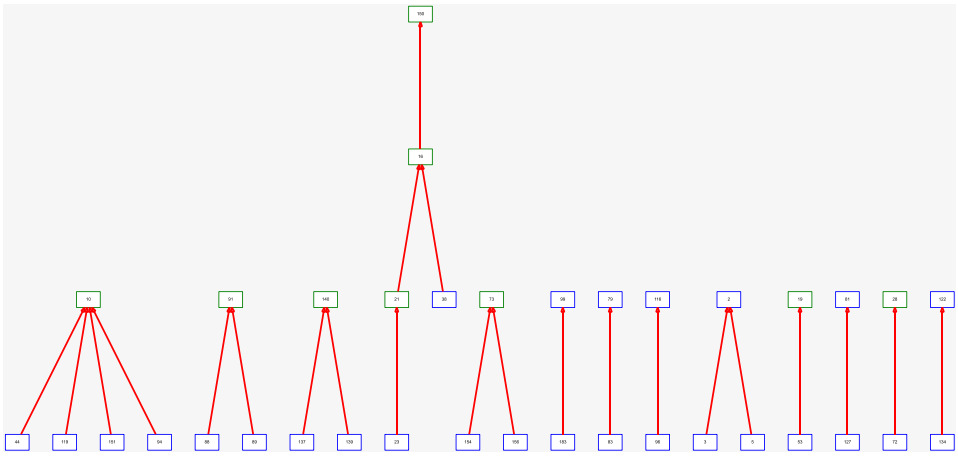


図 41　継承モデル 図（IDB）

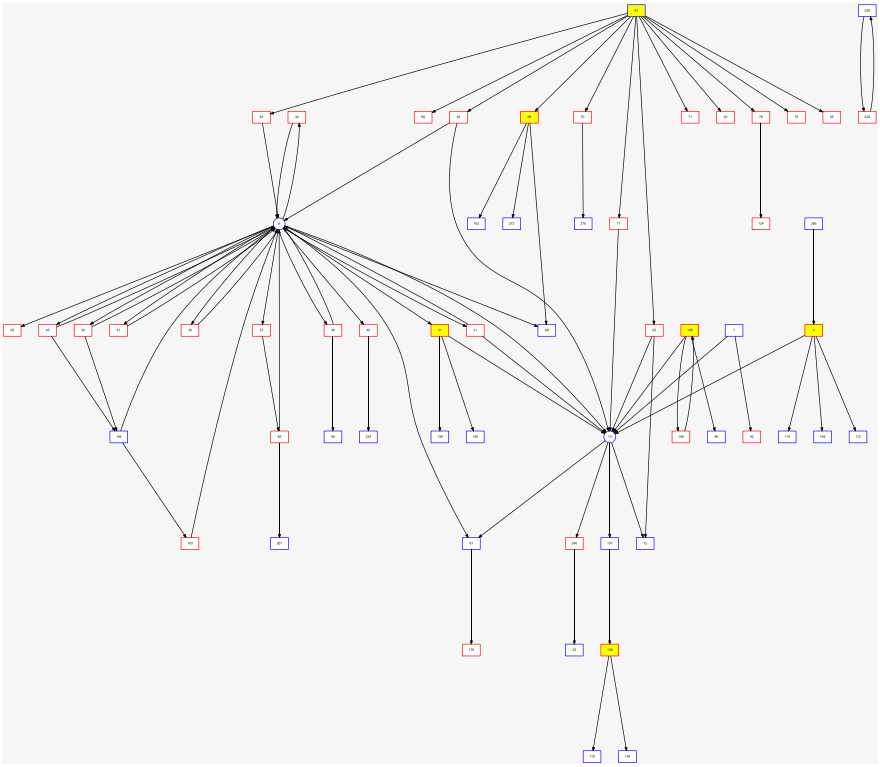
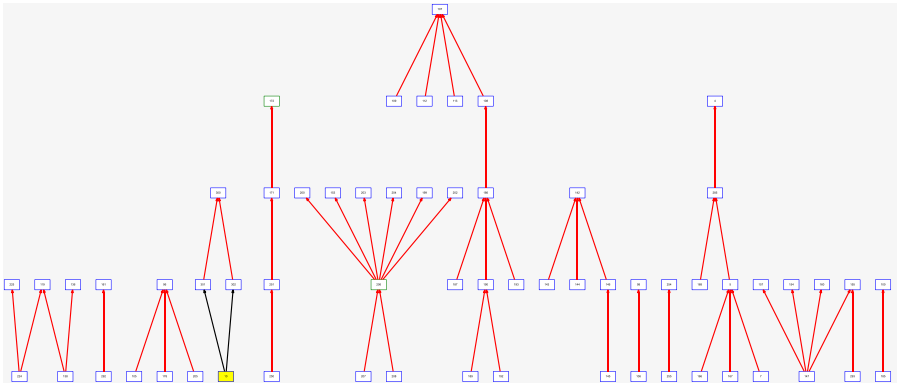


図 42　コントローラモデル 図（IDC）



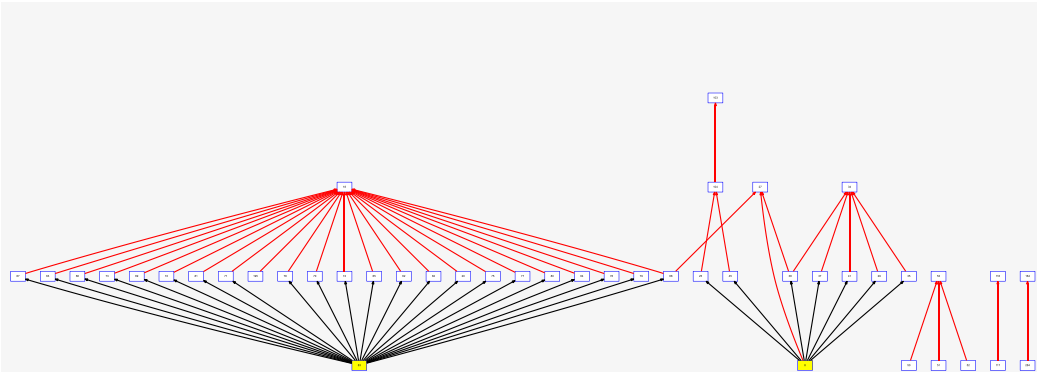


図 43　継承モデル 図（IDC）

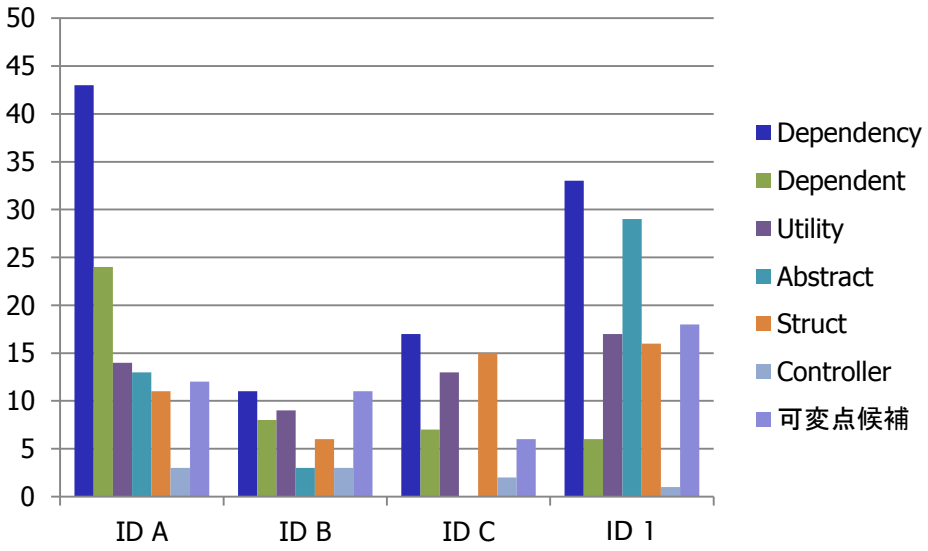


図 44　検出されたコントローラモデル要素の比較

コントローラモデルにおける構成要素の比較を図 44 に示す.

ID A は，他と比較して，Dependency クラスや，Dependent クラスが多い．これは，データーベースにアクセスするインタフェースを多く用意しており，インタフェースから呼び出される処理が多いためである．ID C は，Utility クラスが多い．これは，共通処理が多いことを示している．ID 1 は，Abstract クラス，可変点候補クラスが多い．これは，継承関係を用いて，機能切替えを行う仕組みを多く保有していることを示している．

5.13.　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，コントローラモデル図と継承モデル図を出力する．さらに，本ツールでは，CK メトリクスの値をコントローラモデル図および継承モデル図へ重ね合わせた出力が可能である．たとえば WMC メトリクスでは，値が高くなるほど，各クラスの表示色を緑～赤まで変化させることで WMC の悪化程度を表現することができる（図 71 参照）.

また，本ツールは，組込みソフトウェアのドメインに依存した処理を行っていないため，オブジェクト指向言語であれば使用可能である.

5.14.　考察

可視化手法についての評価を行った．評価観点としては，可視化手法に求められる要件が満たされているかである．したがって，5.2 節で定義した 2 つの課題を解決していること

を評価した.

5.14.1. 進化過程における可視化の有効性

5.2 節の（1）に示した課題については，メンテナンス性に関係する構造を可視化し，抽出できているかを評価した．提案手法により，コントローラモデルと継承モデルという異なる観点で，可視化を行うことで，5.11.2 項に示したように，コントローラモデル・継承モデル上から構造的な変化を観察することができた．他のクラスを制御しているクラスや多くのクラスに機能を提供しているクラス等，個々のクラスを役割ごとに抽出することができるため，抽出されたクラスの要素数を観察することで，5.11.3 項で示したように，ソフトウェアの特徴を推測することができた．また，拡張性構造を抽出し，定量的にコントローラクラスの劣化を計測したことにより，対象となる組込みソフトウェアがどのような拡張性の構造を有しており，どのように変化したかを定量化できた．これらにより，提案手法は，継承またはデザインパターンで表現される拡張性構造を定量的に可視化できる手法であるといえる.

5.14.2. コード規模増大に対する優位性

5.2 節の（2）に示した課題については，実装から生成したデータを捨象することができているかを評価した．5.11.4 項に示したとおり，提案した可視化手法は，一般的なクラス図を用いた手法よりも 50%程度に捨象することができた．また，5.11.5 項に示したように，概念モデルのようなドメイン知識に依存し手作業で作成するモデルでは，捨象率は高いが，抽出すべきクラスの漏れが発生する可能性があり，精度的に問題があることがわかった．故に，提案手法は，手作業による概念モデルや，ツールによる自動生成された一般的なクラス図に比べ，捨象率，精度の面において，コード規模増大に対し実用的で優位性のある手法といえる.

5.14.3. 提案手法の適用範囲

5.11 節に示したように，組込みソフトウェアにおける進化において，提案手法を使用した場合，機能数の増減や，拡張性構造を通じてどのような目的の変更を行ったのかを推測することができた.

しかしながら，5.12 節で比較した，組込みソフトウェア以外のドメインにおけるソフトウェアにおいては，提案した可視化手法を用いて拡張性の構造を識別することが困難であった．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．

5.15. まとめ

本章では，進化型組込みソフトウェアが満たすべき要件の一つに対応する手法について説明した．ソースコードをコントロールフローと継承関係という 2 つの異なる観点でモデル化することによって，メンテナンス性に関する構造の可視化を行った．さらに組込みソフトウェア，および組込みソフトウェア以外のドメインへの適用実験により，提案手法の有効性を評価した．

本章で提案した拡張性可視化手法が満たすべき要件は“要件 1：メンテナンス性に関する実装構造を把握できなくてはならない”であり，これを満たすためには，“全ての実装構造について役割を正確に抽出することは難しい“という課題と”コード規模拡大に伴い，構造の把握が困難になる“という課題がある．そこで，提案手法では，対象とするソフトウェアを 2 種類のモデルで可視化し，さらに，生成されるコントローラモデルでは，機能を切替える仕組みである可変メカニズムを 6 つのデザインパターンとして抽出し，継承モデルでは，2 種類の継承構造を抽出する．また，提案手法では，これら抽出した構造を，機能とレイヤ構造の表として可視化することができる．

提案手法を進化型組込みソフトウェアへ適用した実験では，拡張性構造を可視化し抽出することにより，対象となる組込みソフトウェアの拡張性構造や，拡張性構造の変化を定量化できた．このように，提案手法は，継承またはデザインパターンで表現される拡張性構造を定量的に可視化できる手法であり，手作業による概念モデルや，ツールによる自動生成された一般的なクラス図に比べ，捨象率，精度の面において，コード規模増大に対し実用的で優位性のある手法であった．

一方，組込みソフトウェア以外のドメインにおけるソフトウェアにおいては，本提案手法による可視化では，拡張性の構造を識別することは困難であった．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．

第6章

拡張性強化手法

6.1. はじめに

前章では，提案手法における全体構成のうち，可視化手法について説明した．本章では，可視化結果を分析することで，拡張性を強化する手法について説明する．また，本章は，4.5 節で挙げた要件の一つに対応する手法についての説明である．まず，6.2 節では，提案手法が満たすべき要件に対する課題について述べる．6.3 節では，要件の一つに対応する拡張性の強化手法について概要を説明する．本手法は，前章で説明した拡張性可視化手法の結果を入力として実施する．6.4 節では，拡張性の強化ルールを定義する．拡張性の強化ルールを定義することで，拡張性の問題を抽出することが可能になる．6.5 節では，拡張性の強化ガイドについて説明する．ユーザは，拡張性の強化ガイドを用いることで，拡張性の問題に対処することができる．

6.6 節では，提案手法を実際の進化型組込みソフトウェア製品に適用した実験内容について示す．6.7 節で考察をし，6.8 節で本章のまとめを示す．

6.2. 拡張性強化手法の課題

拡張性強化手法が“要件 2：メンテナンス性の構造的な問題を自動で検出し，正しく対策できなければならない“を満たすための課題について述べる．

(1) 問題箇所の発見が難しい

例えば，goto 文や，マジックナンバー等のようなコード実装レベルの問題であれば，検出・判断は容易であるが，拡張性の問題は，設計レベルであるため，開発者の設計経験が不足していると，問題があること自体を認識することが難しい．そのため，提案手法では，自動的に，問題箇所を検出できることが必要である．

(2) 適切に拡張性を強化できない

同様に，コード実装レベルであれば，関数の置き換え程度で済むため，対策は容易であるが，設計レベルの問題であると，一般的なリファクタリング手法においても，実装コード記述に対しての対処法が示されるだけであるため，開発者の設計スキルが不足していると，よりよい設計の仕組みを思いつかず，進化に対し，中途半端な修正になってしまう．そのため，提案手法による拡張性強化では，開発者によらないメンテナンス性の向上が求められる．

6.3. 拡張性強化手法の概要

6.3.1. 構成

提案手法は，第 5 章で述べた可視化手法の出力を入力とし，図 45 に示すような拡張性の問題構造パターンを定義した拡張性強化ルールと可視化され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．これにより，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．

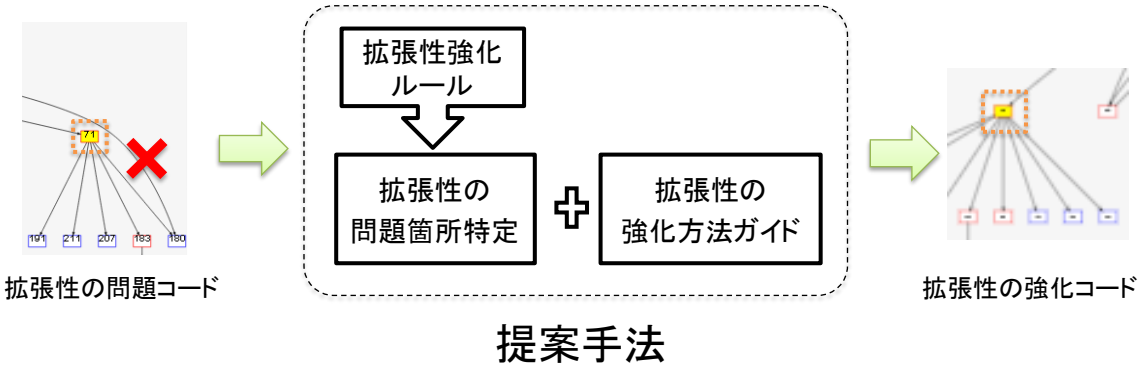


図 45　拡張性強化手法の概要

また，提案手法は，図 8 に示した構造的な特徴を持ったソフトウェア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，および機能シーケンスを制御するようなコントローラクラスとの結合に対する拡張性の問題を特定する．

6.3.2. 特徴

提案手法においては，メンテナンス性の問題個所についての定義と検出方法が特徴的である．

(1) メンテナンス性に関する問題箇所の定義

進化型組込みソフトウェアにおけるメンテナンス性として，4.4.1 項に示したように，複数の機種を容易に作成できるような拡張性の構造を持っていることが重要である．そのため，クラスとして機能が実装されている場合，メンテナンス性の問題とは，新しいクラスの生成が困難になっている箇所である．

そこで，使用者（だれが），場所（どこで），データ（どのデータ関係）という観点における新しいクラスの生成について理想的な状態は，「1 カ所で隠蔽された継承関係のある複数のデータを，ハードウェアデバイスの詳細を知らない使用者が利用できる」ことであると定義する．この状態に対して，起こりうるバリエーションを網羅的に抽出した結果を表 10 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ない場合を除き，後述する拡張性の強化ルールとして設定する．

表 10 各観点における起こりうるケースと拡張性の問題

観点	ケース	拡張性の問題
使用者	知るべきクラスのみにアクセス	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	あり
	抽象クラスへの不必要なアクセス	あり
場所	生成処理を 1 カ所で隠蔽	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	あり
	生成処理が隠蔽されないレイヤ	あり
データ	継承関係のある複数データを生成	なし
	類似点がなく，継承関係のない複数データを生成	拡張性構造なし
	類似点があり，継承関係のない複数データを生成	あり
	単一のデータを生成	なし

(2) 拡張性に関する問題箇所の特定方法

第 5 章で抽出した拡張性の構造に対し，主に，レイヤ構造上の位置関係を利用して拡張性の問題箇所を特定する．図 46 に示した例は，継承に関する拡張性構造とデザインパターンに関する拡張性構造において，SI（単純継承構造）と CU（Mixed Creation and Use パターン）の位置関係が，ある状態になった場合，後述する R1 のルールに違反しているとして，自動的に特定することができる．

機能＼階層	第1階層	第2階層	第3階層以下
機能A	—	SI	SI
機能B	MI	SI	MI
機能C	—	—	—

機能＼階層	第1階層	第2階層	第3階層以下
機能A	—	—	—
機能B	CU	AF	ITF
機能C	—	—	—

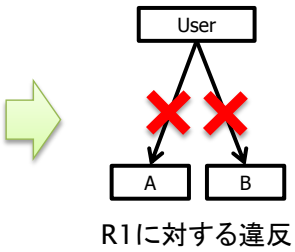


図 46　問題個所特定方法

6.3.3. 期待される効果

メンテナンス性が悪化したことで，歪んだアーキテクチャになった図 4 で示した例に対して，拡張性強化手法を用いることによって検出される問題を図 47 に示す．

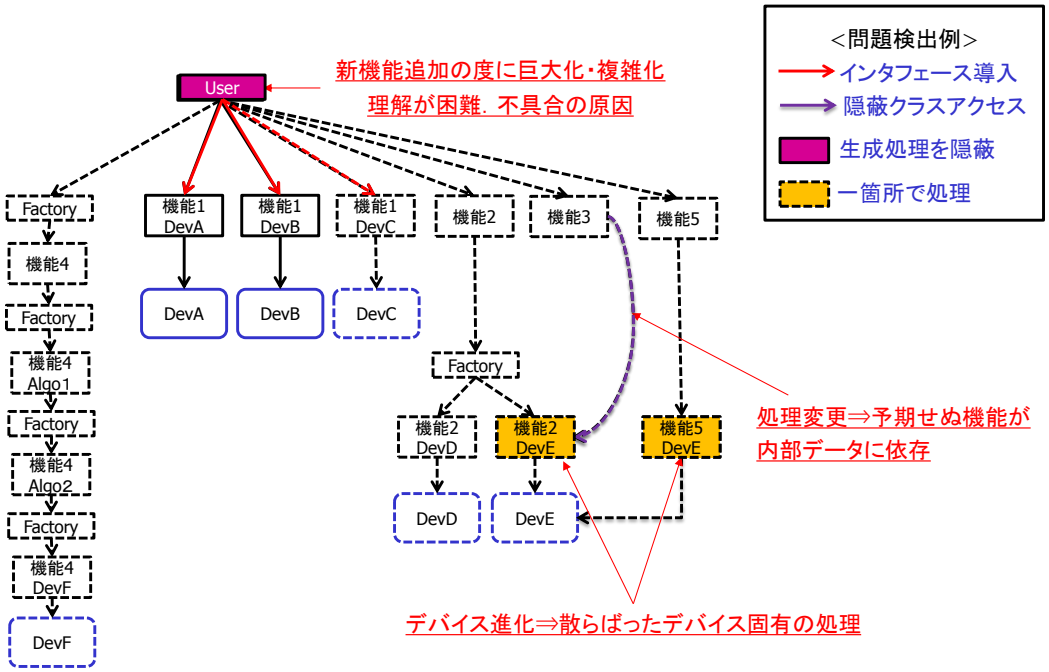


図 47　拡張性強化手法により検出される問題

拡張性強化手法を適用すると，歪んだアーキテクチャによって生じる問題箇所を特定し，拡張性を強化することができる．検出することができる問題は，後述する R1～R5 の 5 つである．

6.3.4. 拡張性の問題改善例

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．

図 48 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，各デバイスのインタフェースに依存したコードになる．

改善されたコードでは，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注力することができる．

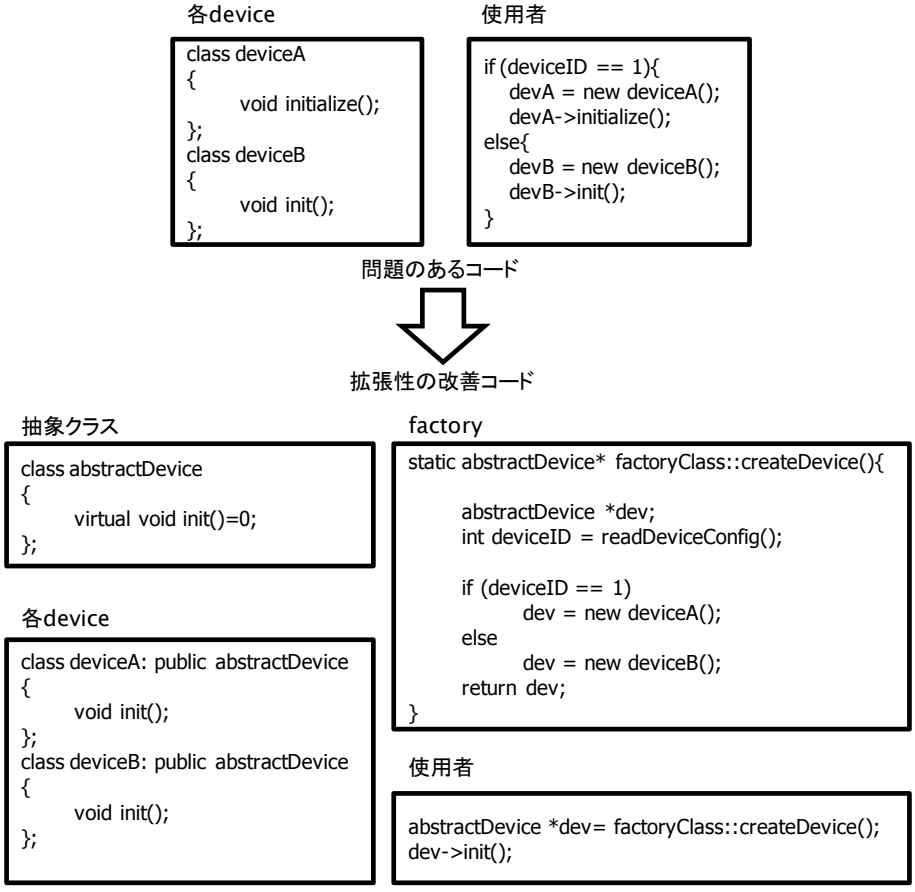


図 48　Factory 化によるソースコード改善例

6.4. 拡張性の強化ルール

表 10 に示した起こりうるケースのうち, 拡張性に問題のあるケースについて拡張性の強化ルールとして定義した内容を表 11 に示す．第 5 章で算出した可視化結果に対し，表 11 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．定義したルールは，継承を利用したクラス設計により, 機能をインスタンス化することを前提としているため, 委譲を用いて，機能拡張させる設計や，クラス外で機能の拡張を行っている設計は対象外となる．また，異なる責務である機能を，既存のクラス内に関数レベルで混在させてしまったことにより肥大化したクラスが問題となるような例においても，可視化手法にて観察できないために対象外となる．しかしながら，定義したルールが想定している，新しいデバイスや機能に対して，責務を隠蔽したクラスを生成するような設計は，一般的によく行われるものである．

表 11 拡張性の強化ルール

ID	拡張性強化ルール	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

クラスの生成処理を使用者から隠蔽する（R1）

図 49 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，メンテナンス性の悪化を防ぐことができる．

当該ルールは，表 4 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 5 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 4 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，“Factory 階層の挿入（G1）“が有効である．

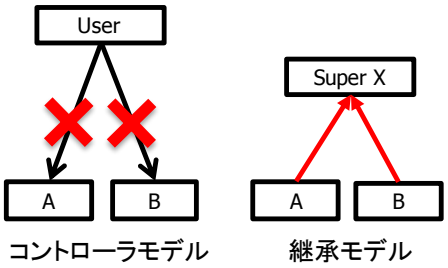


図 49　R1 に反するモデル内の構造例

隠蔽された具象クラスにアクセスしない（R2）

図 50 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 5 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．

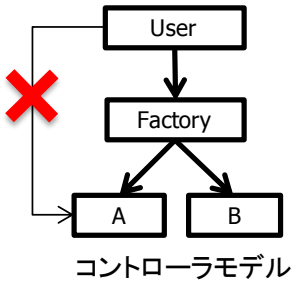


図 50　R2 に反するモデル内の構造例

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，切替えているモデルを図 51 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時に，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．

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．

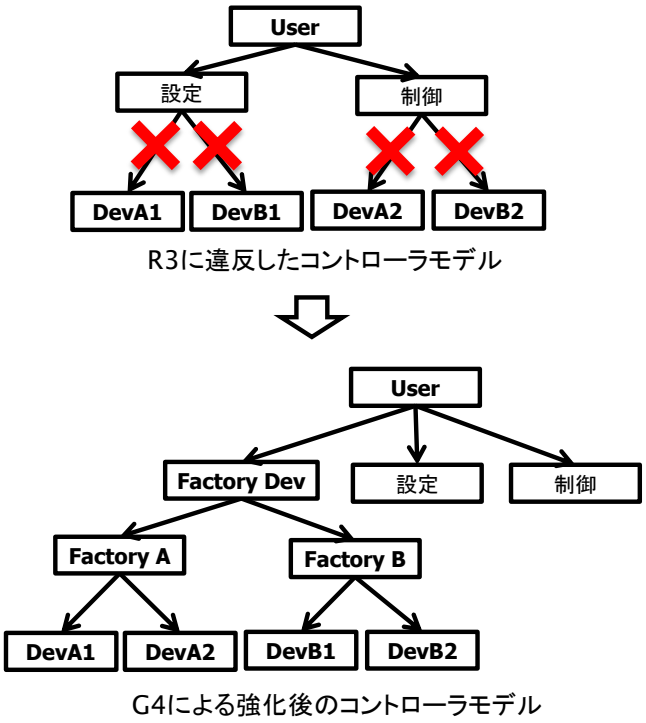


図 51　R3 に反するモデル内の構造と強化後の構造例

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 52 に示すように，派生クラス A が親クラスである 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ソフトウェアが予期しない動作となる可能性がある．

当該ルールは，図 52 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．

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．

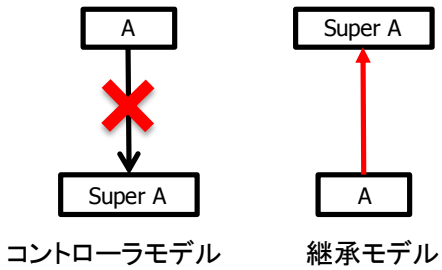


図 52　R4 に反するモデル内の構造例

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 53 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 の呼び出しクラスである User からは，共通インタフェースを介して 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 5 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 4 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．

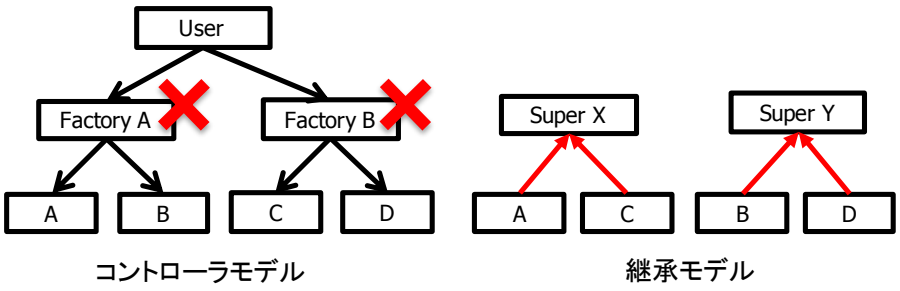


図 53　R5 に反するモデル内の構造例

6.5. 拡張性の強化ガイド

表 12 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．

表 12 拡張性の強化ガイド

ID	拡張性強化ガイド
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

Factory 階層の挿入（G1）

図 49 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．

抽象インタフェースへの変更（G2）

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．

上位クラスの使用箇所を下位クラスへ移動（G3）

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．

使用箇所の集約および Abstract Factory の導入（G4）

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．

テンプレートメソッド化（G5）

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方法である．

委譲関係への変更（G6）

オブジェクト指向言語において，機能を分散する方法として，本研究で対象とした継承関係の他に委譲関係がある．親クラスと派生クラスとの継承関係ではなく，別のクラスの機能を利用する関係になる．この場合，継承関係のような静的に構成された関係ではなく，動的に利用関係を構築することも可能である．

親クラスが想定している使用方法以外の振舞いを行う子クラスを必要とする場合や，特定の子クラスにのみ必要とされる機能の場合は，親クラスへの変更よりも，他クラスの機能を再利用する方がよい．

抽象インタフェースの導入（G7）

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．

6.6. 進化型組込みソフトウェアへの適用

組込みソフトウェアの進化に対する提案手法の適用検証を実施した．対象とした組込みソフトウェアについては，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を用いた．なお，対象コードは代表的なオブジェクト指向言語である C++で記述されている．

6.6.1. 対象コード

5.11.1 項同様，ソフトウェア進化過程における対象コードの一覧を表 13 に示す．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．

表 13 対象コード一覧

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

6.6.2. 拡張性強化箇所の特定および強化ガイド

表 11 に示した 5 つのルールにもとづく拡張性強化箇所の特定結果推移を図 54 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．これは，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ことを示している．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．

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

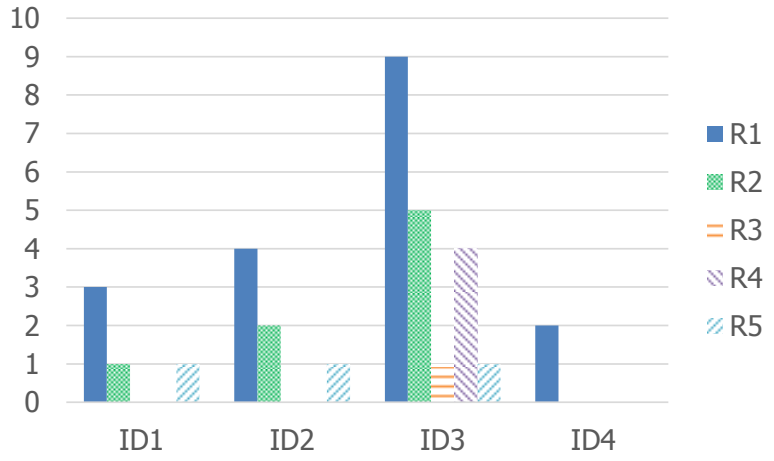


図 54　拡張性強化箇所の特定結果推移

拡張性強化ガイドを用いて，ID3 のコードに対する拡張性強化に要した追加および修正クラス数を表 14 に示す．

表 14 拡張性ルールごとの追加および修正クラス数

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

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，追加クラス数が多い．一方，その他のルールでは，修正クラス数の方が多くなった．

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 54 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．

拡張性強化箇所の特定と拡張性ガイドについての具体例を ID3 と ID4 における変化で示す．

クラスの生成処理を使用者から隠蔽する（R1）

図 54 に示した R1 の拡張性ルールに対し，9 カ所の拡張性改善箇所が ID3 にて特定されている．継承モデルにおける ID3 から ID4 への変化を図 55 に示す．

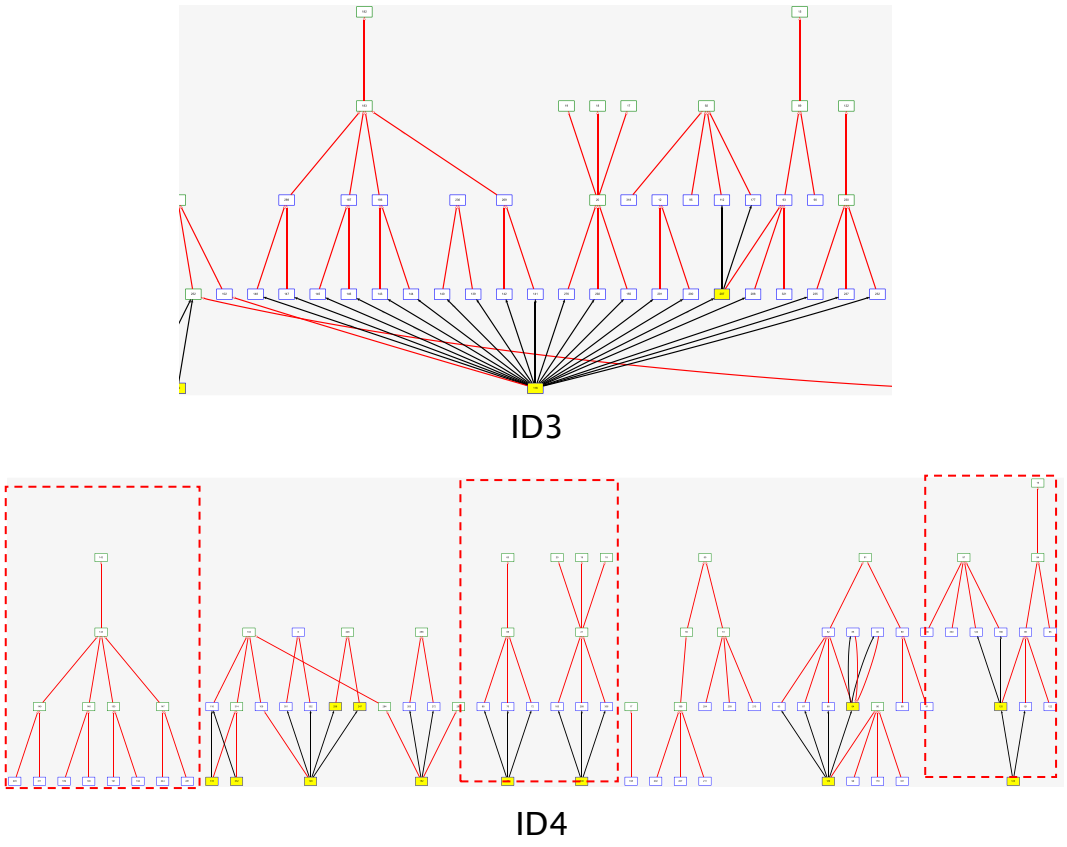


図 55　R1 の拡張性強化による継承モデルの変化

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，さらに抽象 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．これにより，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．

隠蔽された具象クラスにアクセスしない（R2）

図 54 に示した R2 の拡張性ルールに対し，5 カ所の拡張性改善箇所が ID3 にて特定されている．コントローラモデルにおける ID3 から ID4 への変化を図 56 に示す．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．これにより，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による呼び出し箇所が ID4 のコントローラモデル上では削除されている．

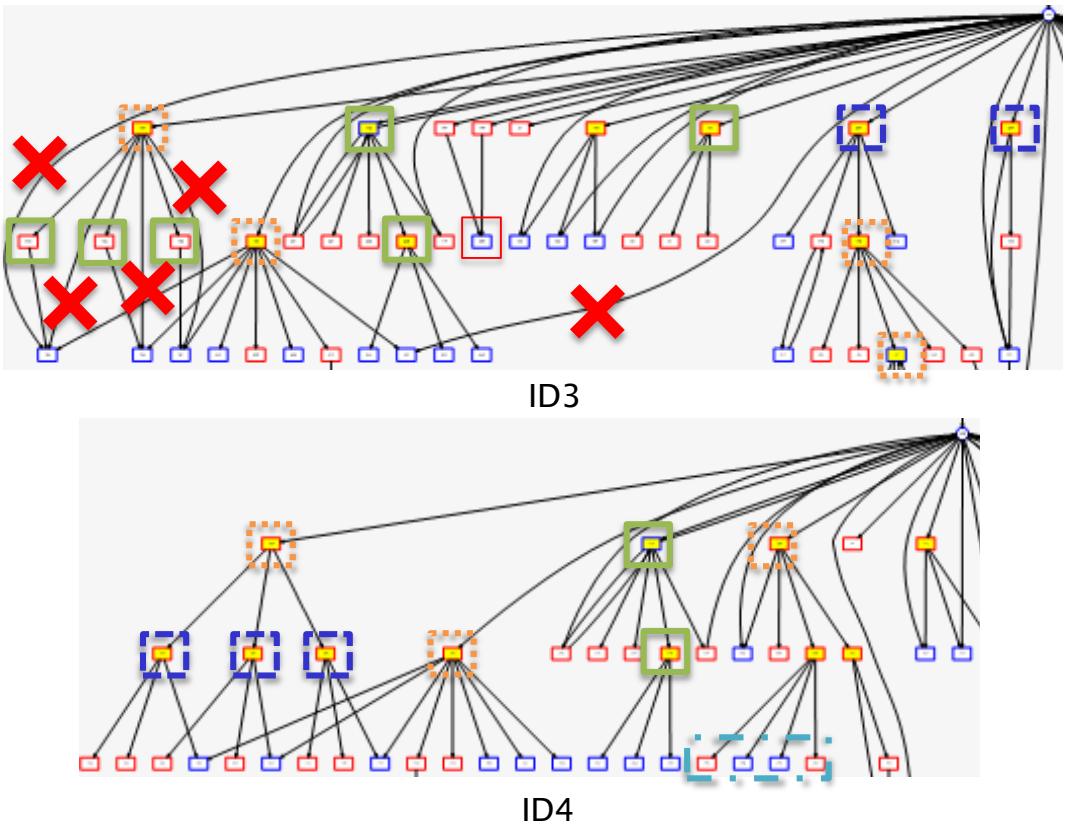


図 56　R2 の拡張性強化によるコントローラモデルの変化

同一デバイスに関する生成操作は 1 カ所に隠蔽する（R3）

図 54 に示した R3 の拡張性ルールに対し，1 カ所の拡張性改善箇所が ID3 にて特定されている．コントローラモデルにおける ID3 から ID4 への変化を図 57 に示す．拡張性ガイドの G4 に沿って，ソフトウェア内に分散していた同一デバイスに関するクラス生成判断処理を 1 カ所に集約し，Abstract Factory を導入した．

ID3 のコントローラモデルには，デバイス種類に依存しクラスを切替える 3 カ所の生成処理対象がある（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．

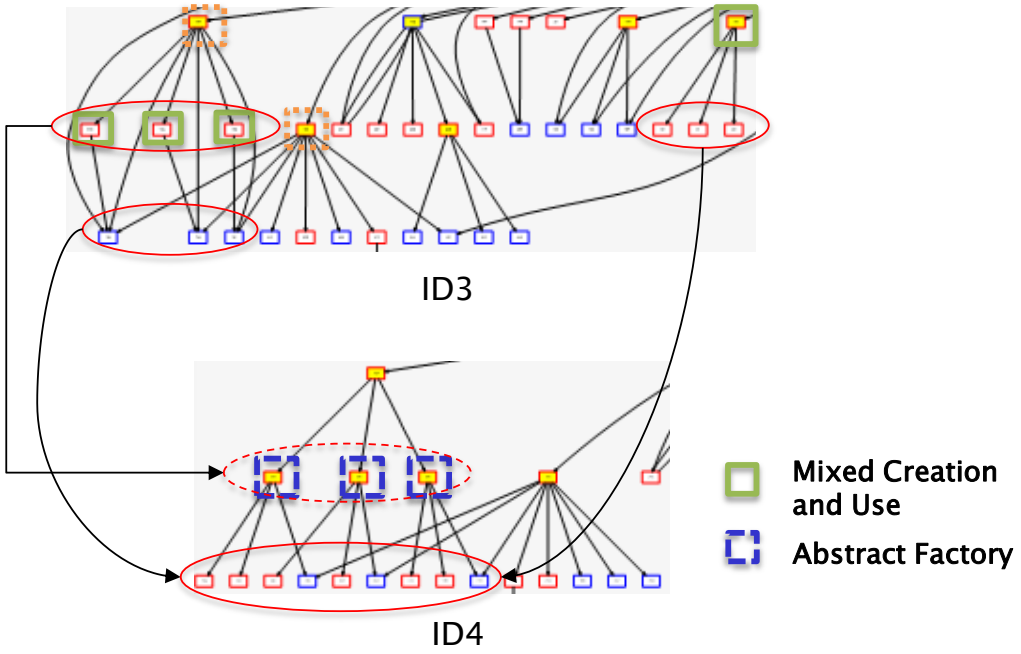


図 57　R3 の拡張性強化によるコントローラモデルの変化

テンプレートメソッドの逆パターンを排除する（R4）

図 54 に示した R4 の拡張性ルールに対し，4 カ所の拡張性改善箇所が ID3 にて特定されている．コントローラモデルにおける ID3 から ID4 への変化を図 58 に示す．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．これにより，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．

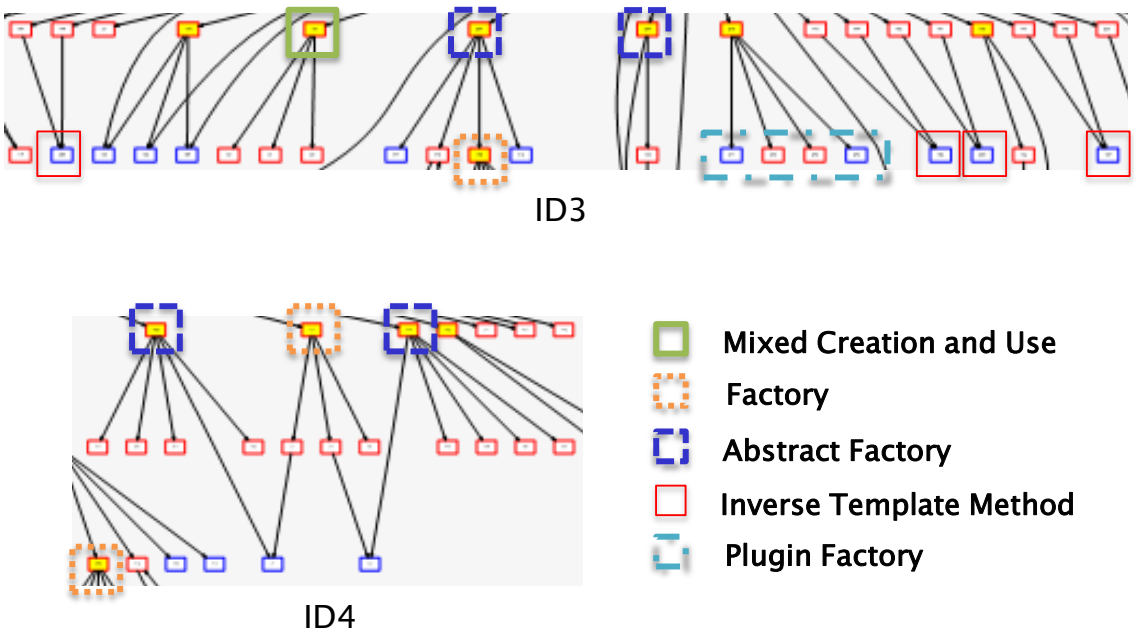


図 58　R4 の拡張性強化によるコントローラモデルの変化

明示的なインタフェースを導入する（R5）

図 54 に示した R5 の拡張性ルールに対し， 1 カ所の拡張性改善箇所が ID3 にて特定されている．継承モデルにおける ID3 から ID4 への変化を図 59 に示す．

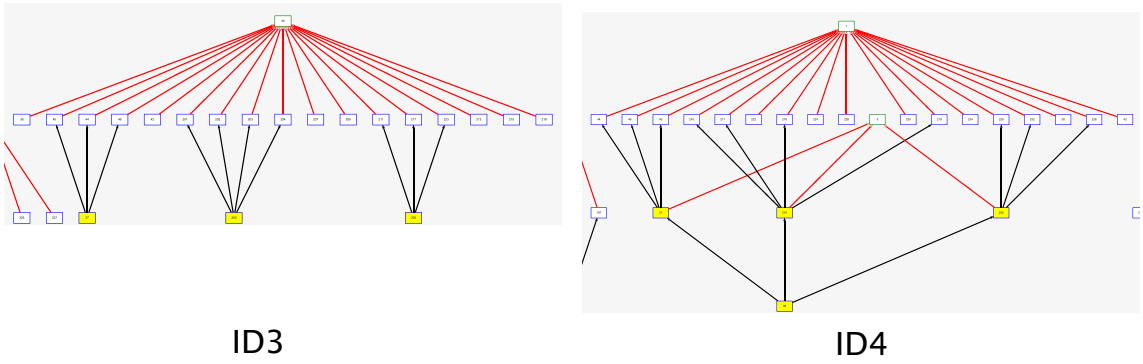


図 59　R5 の拡張性強化による継承モデルの変化

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．これにより，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．

6.6.3. 拡張性の強化前後における機能追加時の変更量比較

提案手法による拡張性強化の有効性を検証した．検証では，今後のソフトウェア進化の予想にもとづき，拡張性強化前後におけるソースコードに対する変更量のシミュレーションを行った．拡張性強化前後のコードに対し，同一の変更を行った場合の追加・修正クラス数を変更量とする．対象とする変更は，開発者へのインタビューにより，今後予想される当該製品における進化を 4 つの変更タイプとして分類した．

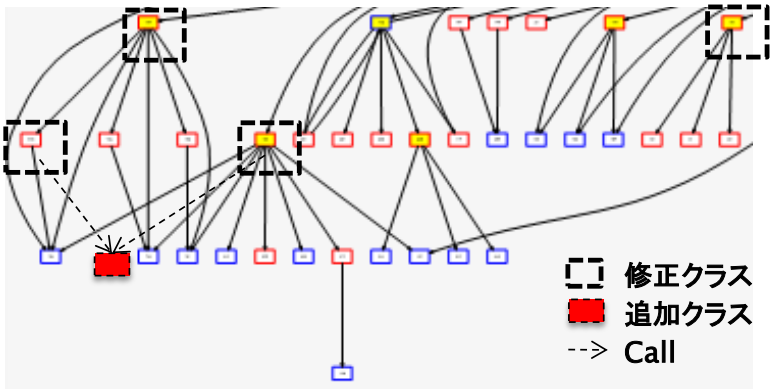
表 15 に変更タイプごとの拡張性強化前（ID3）と拡張性強化後（ID4）の比較結果を示す．シミュレーションに表現されている機能とは，レイヤ構造における第 1 階層のクラスに対して設定された番号であり，同一番号の機能は，同一のクラス構造を対象としていることを示している．また，対応ルールとは，各変更タイプにおいて関連する拡張性の強化ルールである．差異とは，拡張性が強化されたことによる，ID3 と ID4 の違いである．

表 15 機能追加シミュレーション比較結果

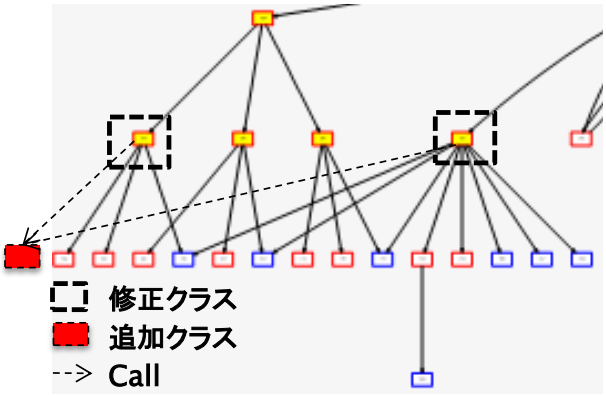
変更タイプ		シミュレーション	対応ルール	比較項目	ID3	ID4	差異
デバイスの追加	類似デバイス追加（メーカ違い等）	機能11にクラスを追加	R2	追加クラス数	1	1	上位層へのデバイス隠蔽能力
				修正クラス数	4	2	
	新規デバイス追加（方式違い等）	機能7にクラスを追加	R1	追加クラス数	1	1	コントローラへの変更影響
				修正クラス数	1	1	
	設定方式が異なるデバイス追加	機能11にデバイス操作クラス，設定クラスを追加	R3	追加クラス数	3	4	具象デバイスに対する変更影響の局所化
				修正クラス数	3	2	
デバイスの多機能用途化		複数機能からの呼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

表 15 に示した類似デバイス追加時における変更内容を図 60 に示す．ID3 では，デバイスの追加により，四角の破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 カ所の破線で囲った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デバイス

隠蔽能力が向上したことで修正が不要となる．



ID3



ID4

図 60　類似デバイス追加時の変更内容比較

表 15 に示した設定方式が異なるデバイス追加時の変更内容を図 61 に示す．ID3 では，図 60 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ソフトウェア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を少なくすることが可能であり，具象デバイスに対する変更影響の局所化がなされている．

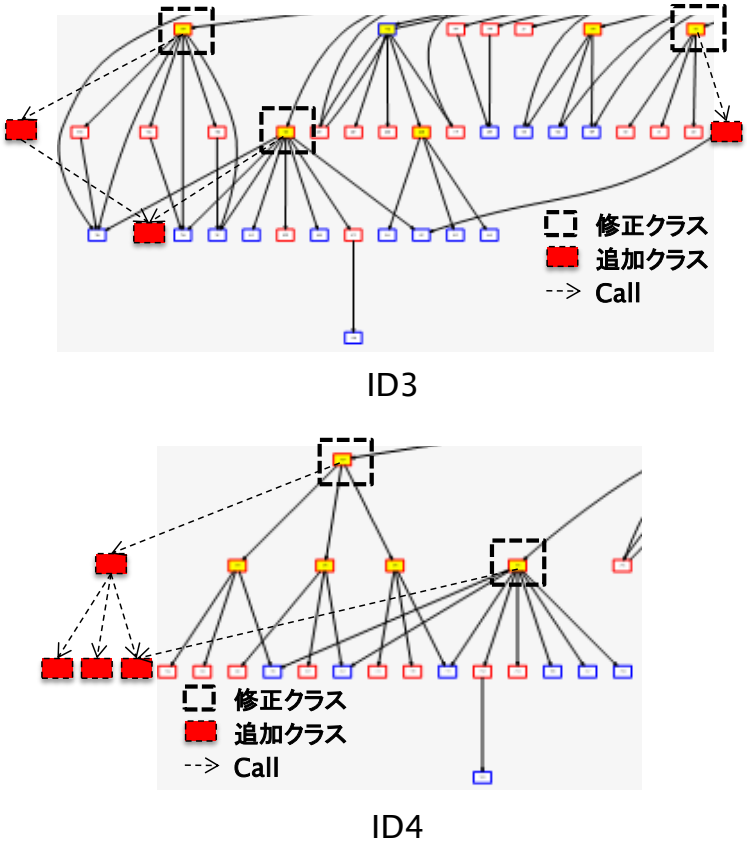


図 61　設定方式が異なるデバイス追加時の変更内容比較

表 15 に示した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して導入されたコントローラクラス以下への Factory 階層の有無である．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る一方，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 15 に示したデバイスの多機能用途化とは，特定機能向けに提供していたデバイスを，他の機能でも使用するという要件変更を想定している．たとえば，多くの機器に搭載されている姿勢センサやカメラからの画像処理等，ソフトウェアの進化過程においてユースケースが拡大する可能性がある．このようなデバイスや処理に関しては，独立性を高くし，さらに並行動作できるよう進化させる必要がある．本研究において定義した拡張性強化ルールでは，当該箇所に対して，拡張性の問題はなかったと判定したために，ID3 と ID4 に違いが現れなかった．そのため，デバイスの多機能用途化という進化に対して，有効性を発揮できなかった．

6.7. 考察

拡張性強化手法についての評価を行った．評価観点としては，拡張性強化手法に求められる要件が満たされているかである. したがって 6.2 節で定義した 2 つの課題を解決していることを評価した.

6.7.1. 進化過程における問題検出・強化の有効性

6.2 節の（1）に示した課題については，自動的に，問題箇所を検出できているかを評価した．6.6 節に示したように，5 つのルール全てにおいて，問題を特定でき，ガイドラインに沿って，拡張性を強化することができた．本提案手法により，開発者の経験によらず拡張性の問題箇所を認識することができる．しかしながら，図 54 に示したように，20 件の問題検出のうち，2 件の誤検知となるケースも存在した．この 2 件は，上位階層に対して隠蔽を要する箇所ではなく，各々のインスタンスが必要な箇所であった．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，このように可変性や拡張性なのか，または類似のインスタンス生成なのかについて判定が難しい.

上記の誤検知件を除き，提案手法は，自動的かつ定量的に，拡張性の問題箇所を特定することができ，拡張性を強化できる手法であるといえる.

6.7.2. 拡張性強化の有効性

6.2 節の（2）に示した課題については，拡張性の強化により，メンテナンス性が向上したかを評価した．6.6.3 項に示したように，デバイス追加を目的とした変更に対しては，提案手法を用いた拡張性の強化による効果がみとめられた．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かった．本研究では，明らかにメンテナンス性が低下するような構造を拡張性の問題箇所としてルール化しているため，当該ケースのようなソフトウェア進化においては対応できない．そこで，より複雑な進化における拡張性の向上にも対応可能な拡張性強化ルールの追加が必要である.

なお，6.6.3 項より得られた，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で修正ミスによる不具合防止，テスト工数の削減

また，6.6.3 項は拡張性強化箇所 1 カ所ごとについて比較した結果であるが，図 54 で示したように，実際のコードでは，多くの拡張性強化箇所を含むため，さらに大きな効果を

得ることができると考えられる．

ソフトウェア開発の現場では，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，提案手法を用いて継続的，自動的にコードから拡張性強化箇所を示すことは，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．

これらにより，提案手法は，デバイス追加を目的とした変更に対して，拡張性強化に効果的な手法であるといえる．

6.8. まとめ

本章では，進化型組込みソフトウェアが満たすべき要件の一つに対応する手法について説明した．ソースコードの可視化によって算出された拡張性構造に対し，拡張性の強化ルールを用いて，拡張性の問題箇所を特定し，拡張性の強化ガイドに従い，適切な拡張性の強化を行う手法について述べた．さらに，進化型組込みソフトウェアの進化過程における実際の製品コードに対する適用実験によって，提案手法の有効性を評価した．

本章で提案した拡張性強化手法が満たすべき要件は“要件 2 : メンテナンス性の構造的な問題を自動で検出し，正しく対策できなければならない”であり，これを満たすためには，“問題箇所の発見が難しい“という課題と”拡張性の強化方法が異なる“という課題がある．そこで，提案手法では，5 つの拡張性に関するルールを定義し，ルールに違反する構造を可視化手法で抽出された 2 つの観点による拡張性構造を用いて特定する．さらに，各ルールは，7 つの拡張性強化ガイドラインのいずれかと対応付けられており，開発者は，ガイドラインに従って，拡張性を強化することができる．

提案手法を進化型組込みソフトウェアへ適用した実験では，自動的かつ定量的に，18 件の拡張性の問題箇所を特定することができ，拡張性の問題箇所がソフトウェアの進化にあわせ増加していることが分かった．よって提案手法は，自動的かつ定量的に，拡張性の問題箇所を特定することができ，拡張性を強化できる手法であるといえる．

さらに，提案手法による拡張性強化の有効性を検証するため，今後のソフトウェア進化の予想にもとづき，進化におけるソースコードの変更量をシミュレーションしたところ，デバイス追加を目的とした変更に対しては，提案手法を用いた拡張性の強化による効果がみとめられた．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かった．本研究では，明らかにメンテナンス性を低下させるような構造に対して拡張性の問題箇所としてルール化しているが，より複雑な進化における拡張性の向上に対応すべく，提案手法のルールベースに，新しい拡張性の向上パターンを追加することで，今後対応していく必要がある．

第7章

拡張性評価手法

7.1. はじめに

前章では，提案手法における全体構成のうち，拡張性強化手法について説明した．本章では，第 5 章で説明した可視化手法で得られた結果を分析し，拡張性の評価を行う手法について説明する．また，本章は，4.5 節で挙げた要件の一つに対応する手法についての説明である．まず，7.2 節では，提案手法が満たすべき要件に対する課題について述べる．7.3 節では，要件の一つに対応する拡張性の評価手法について概要を説明する．7.4 節では，要件変更の類型化と提案手法が対象とする要件変更の種類について定義する．7.5 節では，拡張性の評価指標である変更許容性について説明し，7.6 節では，同じく，拡張性の評価指標である変更容易性について説明する．

7.7 節では，提案手法を実際の進化型組込みソフトウェア製品に適用した実験内容について示す．7.8 節で考察をし，7.9 節で本章のまとめを示す．

7.2. 拡張性評価手法の課題

拡張性評価手法が“要件 3 : 将来の進化にあった柔軟なリファクタリングができなければならない“を満たすための課題について述べる．

(1) メンテナンス性向上コストの必要性判断が難しい

進化型組込みソフトウェアは，市場ニーズにあわせて進化するわけであるが，将来的に進化する必要のない箇所に対してメンテナンス性を向上させることは，トータルとしてメンテナンスコストの削減にはならない．さらには，利用されない拡張性構造が不具合混入の原因になってしまうことさえありうる．しかしながら，一般的には，将来どこが拡張されるのかを事前に知ることは難しく，Jim Johnson [28]によると，事前に作っていた機能のうち，ほとんど，または，全く使用されていない機能が 64%にのぼるという．

そこで，機能ごとの進化に対応するメンテナンス性の向上コストを知ることができれば，対象ソフトウェアにおける進化の頻度や方向性にあわせたメンテナンス性と向上コストのトレードオフ判断により，柔軟なリファクタリングが可能である．なお，拡張性に問題がある箇所については，第 6 章で述べた手法により，メンテナンス性を強化することができるが，拡張性の問題がない箇所についてもソフトウェア進化の対象になることがある．したがって，拡張性に関する問題の有無に関わらずリファクタリングによるメンテナンス性向上の必要性を客観的に判断可能な手法が必要である．

(2) メンテナンス性と性能のトレードオフ判断が難しい

組込みソフトウェアではリアルタイム性についても重要な観点である．そこで，機能ごとのリアルタイム性要件にあわせたメンテナンス性構造を設定することができれば，メンテナンス性とリアルタイム性のトレードオフ判断により，柔軟なリファクタリングが可能となる．

しかしながら，機能毎にメンテナンス性構造をどの程度保有しているか知ることは難しい．

7.3. 拡張性評価手法の概要

7.3.1. 構成

対象ソフトウェアにおける現状のメンテナンス性を把握するには，備えている拡張性の構造について定量的に評価する必要がある．そこで，本手法は，拡張性構造について“変更の受け入れやすさ”と“変更のしやすさ“という 2 つの観点で定量化を行う．さらに，類型化した要件変更に対して，メンテナンス性に関する問題の有無にかかわらず，機能ごとの拡張性を評価・比較する手法である．

提案手法は，第 5 章で述べた可視化手法の出力を入力とし，拡張性の構造について相対的な定量化をし，要件変更の程度を類型化することで，図 62 に示すように，変更許容性と

変更容易性という 2 つの評価を出力する．これにより，開発者は，変更発生頻度が高い機能や変更が予定されている機能のうち，変更に対する許容性の低い機能があれば，当該機能に対するリファクタリングを検討することができる．さらに，変更容易性を用いて，どのような可変メカニズムを導入すべきかについて判断することができる．このように，ソースコードをベースとし，戦略的に拡張性を強化していくことができる．

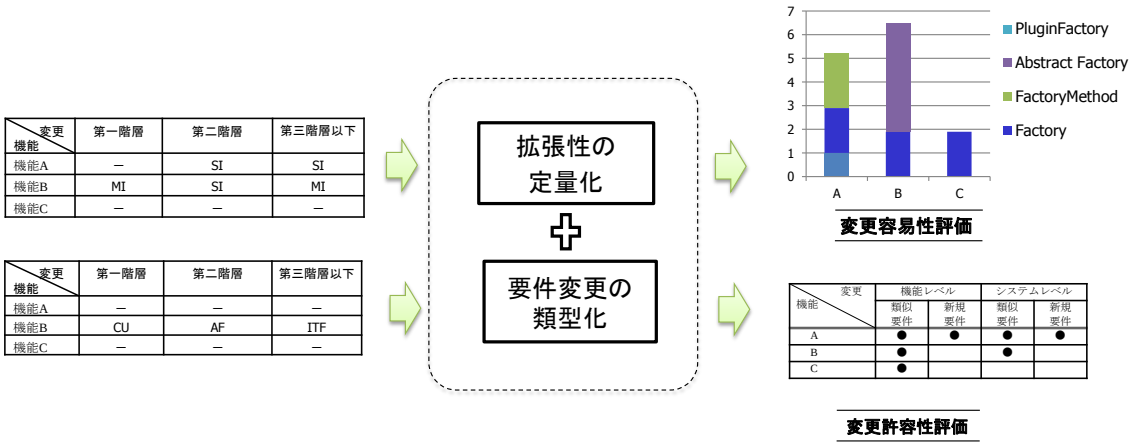


図 62　拡張性評価手法の概要

7.3.2. 特徴

提案手法においては，進化の表現方法とメンテナンス性の指標が特徴的である．

(1) 将来における進化の表現方法

進化における要件変更を 2 種類の“進化による影響の大きさ”で表現する．指標毎のメンテナンス性を示すことで，どのような進化に対して，メンテナンス性が高いか，または低いかを評価できる．詳細については，7.4 節に示す．

(2) 現在のメンテナンス性の指標

下記，2 種類のメンテナンス性に関する指標を定義した．

変更許容性：将来の進化において，どの程度の影響に対する備えを持っているかの指標を定義．詳細については，7.5 節に示す．

変更容易性：進化により，変更が発生した場合に，どのような仕掛けを持っているかの指標．詳細については，7.6 節に示す．

拡張性構造とリアルタイム性能の関係としては，幅広い進化に対応できる構造ほど，多くの冗長な構造が組み込まれているため，リアルタイム性能は劣化する可能性がある．ま

た，変更時の作業量を低く抑えるような構造ほど冗長となり，やはりリアルタイム性能が劣化する可能性があることを前提とする．

7.3.3. 期待される効果

メンテナンス性が悪化したことで，歪んだアーキテクチャになった図 4 で示した例に対して，拡張性評価手法による評価対象を図 63 に示す．拡張性評価手法を適用すると，各々の機能に対するメンテナンス性を定量化することができる．これにより，進化の予測や要求されるリアルタイム性能を踏まえ，メンテナンス性を向上させることの必要性を検討することが可能になる．

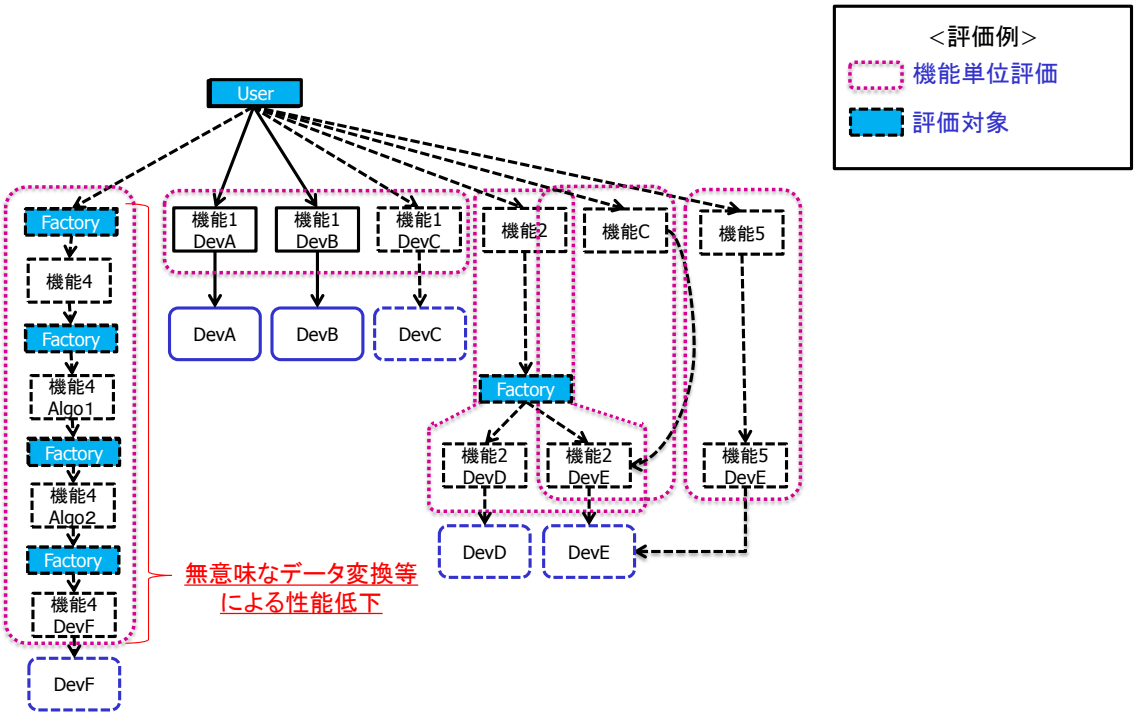


図 63　拡張性評価手法による評価対象

7.4. 要件変更の類型化および要件変更種類

7.4.1. 要件変更の類型化

本手法では，進化による影響の大きさを抽象度と変更度で表現し，さらに各々について 2 種類のレベルを定義する．要件変更の分類を表 16 に示す．

抽象度は，要件変更の影響範囲を示している．新機能の追加等は，システムに影響を及ぼす要件の変更であるため，システムレベルの抽象度と定義する．また，既存の単一機能

のみに影響を及ぼす要件の変更は，機能レベルの抽象度として定義する．なお，システムレベルの抽象度における変更は，図 8 に示した組込みソフトウェアにおける基本的なレイヤ構造上に定義されているコントローラクラスのインタフェースを壊さない範囲において有効であるとする．

変更度は，要件変更の大きさを示している．既存の可変メカニズムが持つインタフェースで吸収可能な要件の変更を類似要件と定義し，吸収不可能な要件変更を新規要件として定義する．なお，変更度は，各抽象度に対し定義できるものとする．

表 16 要件変更の分類

分類	レベル	内容
抽象度	システム	システム全体の挙動に影響を及ぼす要件の変更
	機能	単一機能の挙動にのみ影響を及ぼす要件の変更
変更度	類似	既存のインタフェースで吸収可能な要件の変更
	新規	新規インタフェースの追加が必要な要件の変更

7.4.2. 対象とする要件変更の種類

本研究で対象とする要件変更項目を表 17 に示す．処理速度やリソース消費量等の非機能要件に関する要件変更ついては，進化型ソフトウェアにおいて，重要な観点ではあるが，動的な解析によって，現状の問題点を検出する手法が一般的であり，また変更発生時に，計測および修正が可能であるため，本手法の対象外とする．また，ユースケースに関する要件の変更については，機能の実行順序のみに影響を及ぼすような変更以外を対象とする．

表 17 要件変更の種類に対する対応表

抽象度	要件変更	対応
システム レベル	機能要件の追加	○
	非機能要件の追加	－
	ユースケースの追加/変更	△
機能 レベル	デバイスの追加/変更	○
	アルゴリズムの追加/変更	○

7.5. 変更許容性

7.5.1. 変更許容性定義

既存のソフトウェアに対して，要件の変更が生じた場合に，現在の継承構造を保った状

態で変更を吸収することができる程度を変更許容性として定義する．

拡張性を評価する指標の一つとして，要件変更に対し，機能ごとに評価する．対象とする要件変更は表 16 で示した抽象度および変更度による組合で表現できるものとする．

要件変更の抽象度に関する変更許容性の評価では，抽象度によって，異なるレイヤを対象とした評価となる．システムレベルに影響を及ぼす要件変更は，主にシステムへの機能追加等に関する変更であるため，第 1 階層に含まれるクラスを切替え対象とするコントローラクラスの可変メカニズムに対する評価となる．一方，機能レベルに影響を及ぼす要件変更は，主に単一の機能におけるハードウェアやアルゴリズムに関する変更であるため，第 2 階層以下のレイヤに含まれるクラスを切替え対象とする第 1 階層のレイヤにマッピングされている可変メカニズムに対する評価である．

要件変更の変更度に関する変更許容性の評価は，可変メカニズムによって生成されるクラスにおける継承構造に対する評価となる．多階層継承構造は，単純継承構造に比べ，異なる抽象クラスの追加および具象クラスの追加が容易なため，可変メカニズムによって生成されるクラスが多階層継承構造を持つ場合，新規要件に対する許容性があるとする．一方，単純継承構造の場合は，類似仕様にのみ許容性があるとする．また，継承構造を持たない場合や，可変メカニズムが抽出されない場合は，許容性なしとなる．なお，新規要件に対して許容性がある場合は，類似要件についても許容性があるとする．

表 18 に機能ごとの変更許容性の評価例を示す．

表 18 機能ごとの変更許容性評価例

変更 / 機能	機能レベル		システムレベル	
	類似要件	新規要件	類似要件	新規要件
A	●	●	●	●
B	●		●	
C	●			

変更許容性評価では，許容性ありを“●”印で示す．たとえば，機能 A は，すべての要件変更タイプにおいて“●”印がついているため，多階層継承構造を持つ可変メカニズムが第 1 階層および第 2 階層に導入されていると分かる．一方，機能 C は，機能レベルの類似要件のみに“●”印がついているため，単純継承構造を持つ可変メカニズムが第 2 階層に導入されているだけであると分かる．

7.5.2. 変更許容性の定量化

抽象度と変更度の各組合せで表現される要件変更に対するクラス構造の追加シミュレー

ションを図 64 および図 65 に示す．図中の青色鎖線表記部は各構造に対し，左列の構造から追加される箇所である．なお，類似要件や新規要件変更に対する許容性を高めるため，既存実装済みの構造に対し可変メカニズムを導入する場合，コントローラモデルおよび継承モデルの双方への変更が必要となる．また，許容性のない構造に対して，切替え可能なクラスを追加する場合は，許容性ありの構造へ拡張後，追加する必要がある．

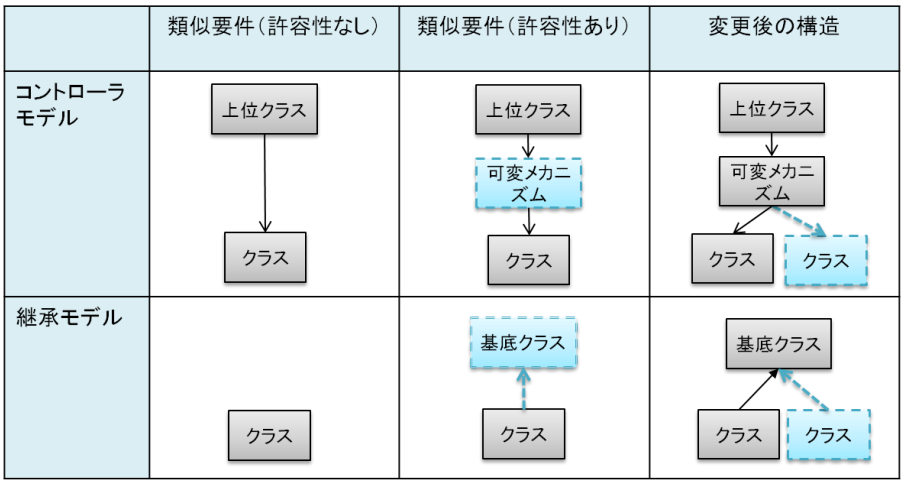


図 64　類似要件変更時のクラス構造追加シミュレーション

図 64 には，既存実装済みの要件と類似している要件を追加する場合について示す．

まず，既存機能に対応するクラスの Base クラスを作成し，継承構造を導入する．さらに，既存機能に対応するクラスを呼び出しているクラスに対し，可変メカニズムを介して呼び出すよう変更を行う．その後，追加される要件に対応するクラスを前述の継承構造から派生することによって作成し，導入した可変メカニズムを介して呼び出す．

図 65 では，既に類似要件の変更に耐えうる構造を導入している箇所において，新規の要件を追加する場合について示す．

まず，切替え対象クラスを構成している継承構造の Base クラスに対し，さらに抽象化したクラスを抽出する．この抽象クラスを用いてクラスの切替えを行うように可変メカニズムを変更する．その後，追加される要件に対応するクラスを前述の抽出した抽象化クラスから派生させることで作成し，可変メカニズムを介して呼び出す．

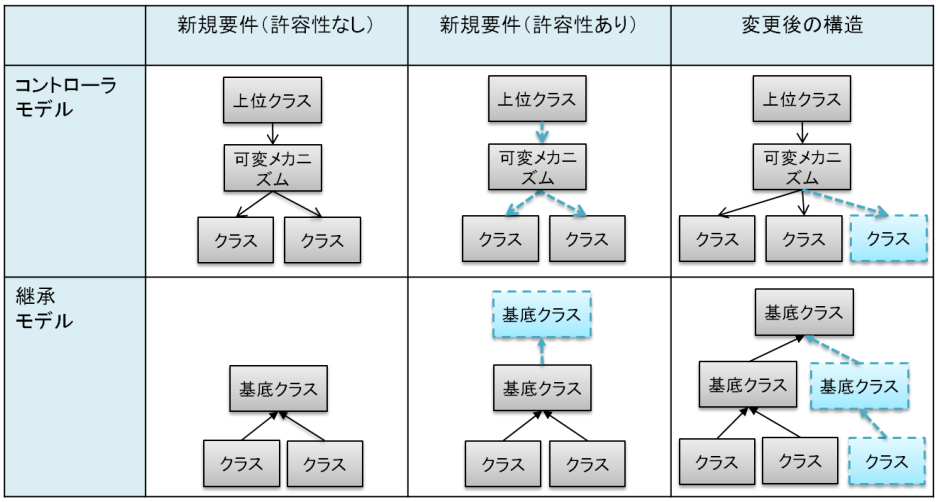


図 65　新規要件変更時のクラス構造追加シミュレーション

また，要件変更に応じて，作成する構造に含まれる関数の種類を表 19 に定義した．

表 19 シミュレーションで追加される関数

	可変メカニズム	Base クラス	抽出抽象クラス	具象クラス
コンストラクタ	○	○	○	○
デストラクタ	○	○	○	○
クラスの生成	○			
機能実装の抽象		○	○	
クラス切替え	○			
機能実装			○	○

任意の機能における変更許容性評価について，変更度および許容性の違いごとの変更関数の数を表 20 に示す．括弧内の数字は，類似要件かつ許容性ありを 1 とした場合の比率である．

表 20 変更許容性評価における変更関数の数

変更 / 機能	類似要件	新規要件
許容性あり	4 (1)	7 (1.8)
許容性なし	11 (2.8)	13 (3.3)

対象とするソフトウェアにおいて，機能レベルの類似要件に対する変更工数（$Cost_B$）を計測することで，変更許容性評価における変更関数比率（$Ratio(a,b,c)$）を用い，任意の機能系統の要件変更に対するメンテナンスコスト（$Cost$ ）を算出することができる．算出式

を式(１)に示す．なお，追加・変更される関数の導入/ 修正コストは一定だと仮定する．

$$Cost=\boldsymbol{Cost_B} \times Ratio(\boldsymbol{a},\boldsymbol{b},\boldsymbol{c}) \quad (1)$$

$Cost_B$：機能レベルの類似要件に対する変更工数

$Ratio(a,b,c)$:変更許容性評価における変更関数比率

a:抽象度，b:変更度, c:許容性

これにより，どの機能に，どのような要件変更が生じた場合に，メンテナンスコストがどの程度発生するか予測することができる．なお，本研究では，絶対的なメンテナンスコストの算出ではなく，機能ごとの相対的なメンテナンスコストを予測するものである．

7.6. 変更容易性

7.6.1. 変更容易性の定義

既存のソフトウェアに対して，要件の変更が生じた場合に，現在の可変メカニズムによって簡単に変更ができる程度を変更容易性として定義する．これは，拡張性を評価する指標の一つとして，要件変更に対し，機能ごとに評価する．変更容易性の評価では，変更を隠蔽する能力の高い可変メカニズムを多く導入している機能ほど変更容易性が高いとする．変更時の影響を隠蔽する方法として，クラスの使用側と生成側を分離しておく方法が有効である．また，生成処理の複雑さは可変メカニズムごとに異なる．変更容易性の評価は，5.8.2 項で示した可変メカニズムについて個々の隠蔽度を算出することで行う．

図 66 に機能ごとの変更容易性評価例を示す．

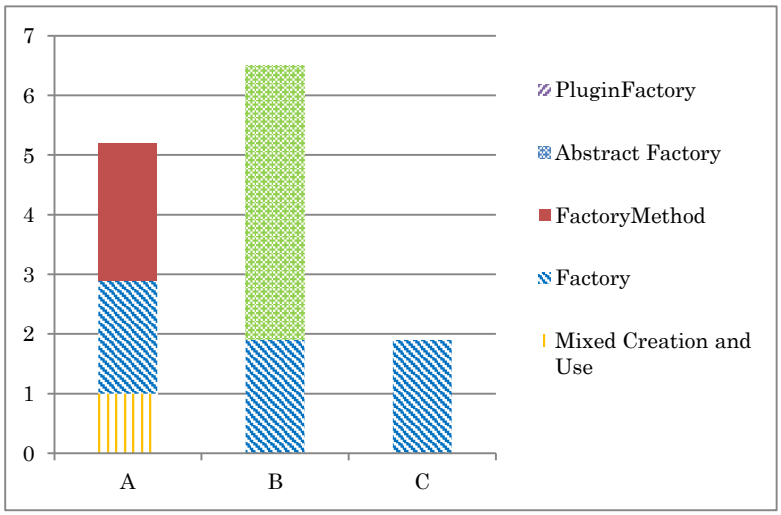


図 66　機能ごとの変更容易性評価例

図 66 では，各機能に導入されている可変メカニズムの数と種類を知ることができる．機能 B は，可変メカニズム数が機能 A より少ないが，変更容易性の高い可変メカニズムを導入しているため，機能 A より変更時の既存コードへの影響は小さくなることが分かる．

7.6.2. 変更容易性の定量化

隠蔽度は,”ある機能が，2 種類の異なる部品を持ち，各部品は 3 種類の異なる材料のうち一つが選ばれる組合せで構成されている場合”における，各可変メカニズムに加えられる変更についてシミュレーションを行って算出する．シミュレーションによって算出した可変メカニズムごとの隠蔽度を表 21 に示す．隠蔽度は，部品を使用するクラスに加える変更行数と Factory クラスに加える変更行数，それ以外の箇所に加える変更行数の合計に対し，変更が加えられる箇所ごとに規定した重み付けを加算した合計値とし，合計値が低いほど隠蔽度が高いとした．変更行数は，使用するための呼び出し行および生成クラスを切替えるための条件分岐行を合計した値である．重み付けは，使用クラスに対し 2，Factory クラスに対し 0.4，それ以外に対し 0.2 とする．

表 21 可変メカニズムごとの隠蔽度

パターン	使用クラス	Factory クラス	他箇所	合計
Mixed Creation and Use	6	0	0	12
Factory	2	6	0	6.4
Factory Method	2	0	6	5.2
Abstract Factory	1	0	3	2.6
Plugin Factory	1	0	0	2

可変メカニズムごとの評価値は，Mixed Creation and Use パターンにおける隠蔽度を 1 とし，各可変メカニズムと Mixed Creation and Use パターンとによる隠蔽度の比率により算出した．表 22 に算出した可変メカニズムごとの評価値を示す．

表 22 可変メカニズムごとの評価値

パターン	評価値
Mixed Creation and Use	1
Factory	1.9
Factory Method	2.3
Abstract Factory	4.6
Plugin Factory	6

各機能系統における変更容易性の評価（*Score*）は，導入されている可変メカニズム種類ごとの評価値（$Score_P$）を合計することで求める．機能系統ごとの変更容易性評価式を式（2）に示す．

$$\mathbf{Score} = \sum_{i=0}^{n} Score_P(i) \quad (2)$$

$Score_P$：*可変メカニズム種類ごとの評価値*

i:*抽出された可変メカニズム*

これにより，どの機能に，どのような可変メカニズムが入っており，変更をどのように隠蔽させるのがよいかを検討することができる．

7.7. 進化型組込みソフトウェアへの適用

組込みソフトウェアについては，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を用いた．なお，対象コードは代表的なオブジェクト指向言語である C++で記述されている．

7.7.1. 対象コード

表 23 に，開発時期の異なるソフトウェアを示す．ID1～2 は，同一製品に対する機能追加前後のコードである．

表 23 対象コード一覧

ID	内容
1	初期製品コード
2	ID1 のコードへの機能追加

対象としたコードには，下記の機能系統が含まれていた．

- 3 個の機能系統はデバイスからの入出力を担う
- 3 個の機能系統はデータ処理を担う
- 2 個の機能系統はウインドウの描画とシステムの管理機能を担う

ウインドウ描画やシステム管理のような，デバイス入出力またはデータ処理と関連の薄い機能系統は，ハードウェアの変更や機能要件変更による影響が少ないため，評価対象から除外した．

7.7.2. 変更許容性評価

ID1 における機能系統ごとの変更許容性評価を表 24 に示す.

表 24 機能ごとの変更許容性評価（ID 1）

変更 / 機能	機能レベル		システムレベル	
	類似要件	新規要件	類似要件	新規要件
入力デバイス 1	●	●	●	●
入力デバイス 2	●			
出力デバイス				
データ処理 1	●			
データ処理 2				
データ処理 3				

入力デバイス 1 は，すべての要件変更種類に対して現構造で対応可能である．一方，出力デバイス，データ処理 2 および 3 は，すべての変更に対して，現構造で対応することができない．また，入力デバイス 2 とデータ処理 1 は，共に同レベルの変更に対してのみ対応することができる.

機能追加後の ID2 における変更許容性評価結果を表 25 に示す．データ処理 2 の許容性が向上したことが分かる．これは，ID1 の機能切替えをともなわない Factory Method パターンから，新たな派生が追加され，切替えをともなう Factory Method へ変更されたことが評価に反映された結果である.

表 25 機能ごとの変更許容性評価（ID 2）

変更 / 機能	機能レベル		システムレベル	
	類似要件	新規要件	類似要件	新規要件
入力デバイス 1	●	●	●	●
入力デバイス 2	●			
出力デバイス				
データ処理 1	●			
データ処理 2			●	
データ処理 3				

7.7.3. 変更容易性評価

図 67 および図 68 には，ID1 における変更容易性の評価結果を示す．ハードウェアの入出力を担う機能群と，取得したデータを加工するデータ処理機能群に分けて結果を示す．

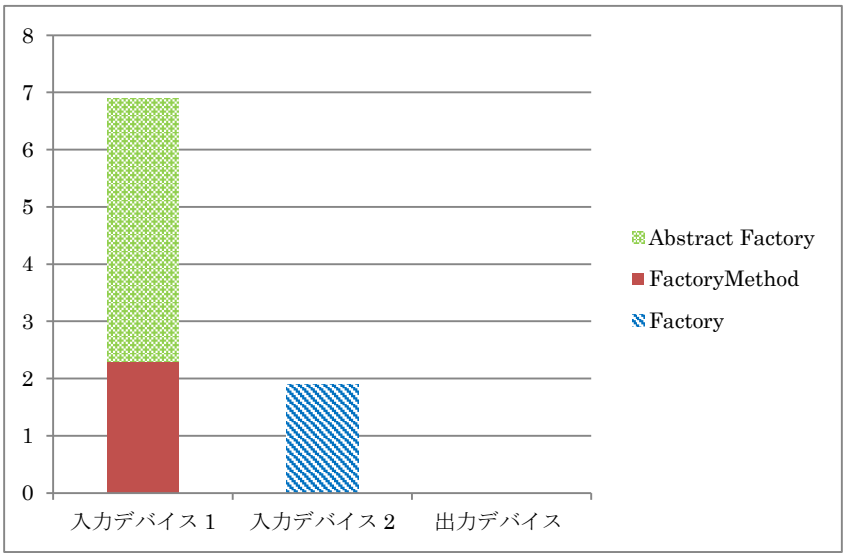


図 67　ハードウェア入出力関連機能における変更容易性評価(ID 1)

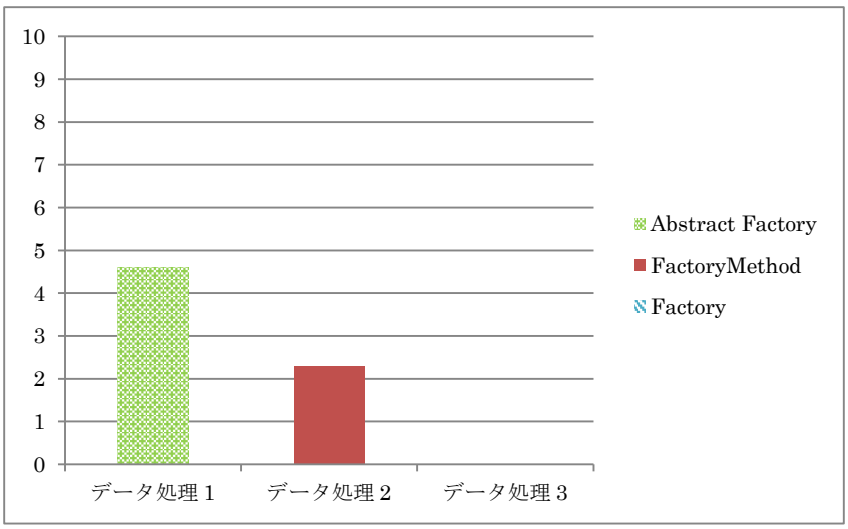


図 68　データ処理関連機能における変更容易性評価(ID 1)

入力デバイス 1 は，Factory Method パターンを用い，異なるオブジェクトの生成を可能にしており，さらに Abstract Factory パターンを用いることで，複数のハードウェアへの対応も可能になっていることが分かる．

変更許容性評価では同じ評価であった入力デバイス 2 とデータ処理 1 は，変更容易性評

価により進化発生時における変更の容易さに違いがあることが分かる．

機能追加後の ID2 における変更容易性評価結果について図 69 に示す．

データ処理 2 の機能系統に新しい可変メカニズムが導入されたことで，評価値が向上している．これは，進化前では，当該機能系統に存在しなかった Plugin Factory が追加されることで，柔軟性の高い機能系統になったことが評価に反映された結果である．

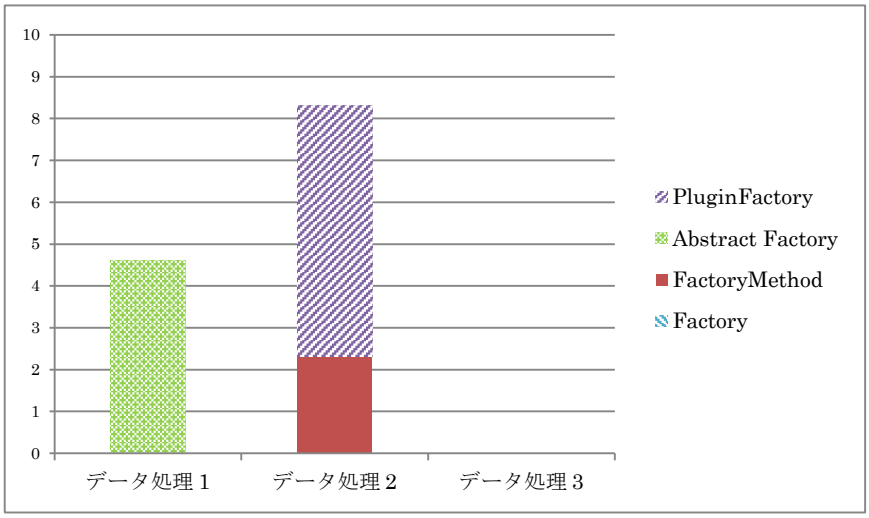


図 69　データ処理関連機能における変更容易性評価（ID 2）

7.7.4. 変更許容性の変更関数比率

全ての変更に対し，変更許容性を持つ入力デバイス 1 の構造を，図 64 および図 65 に示した構造へ分解し，変更される関数の数を比較した結果を図 70 に示す．

要件変更の抽象度，変更度にかかわらず，許容性なしの場合における変更関数の数が，許容性ありよりも多く，最大で 10 倍以上の修正量が発生している．

変更度に関しては，要件変更に対する許容性ありの場合，類似要件に対する変更関数の数に比べ，新規要件に対する変更関数の数の方が多くなる．一方，要件変更に対する許容性なしの場合，許容性ありの場合とは逆に，類似要件に対する変更関数の数が，新規要件に対する変更関数の数より多くなる．これは，要件変更に応じて新たに追加される類似の構造とは別に，継承構造や，インタフェースとなる抽象クラスを現行構造から分離させ，さらに可変メカニズムの実装を要するからである．

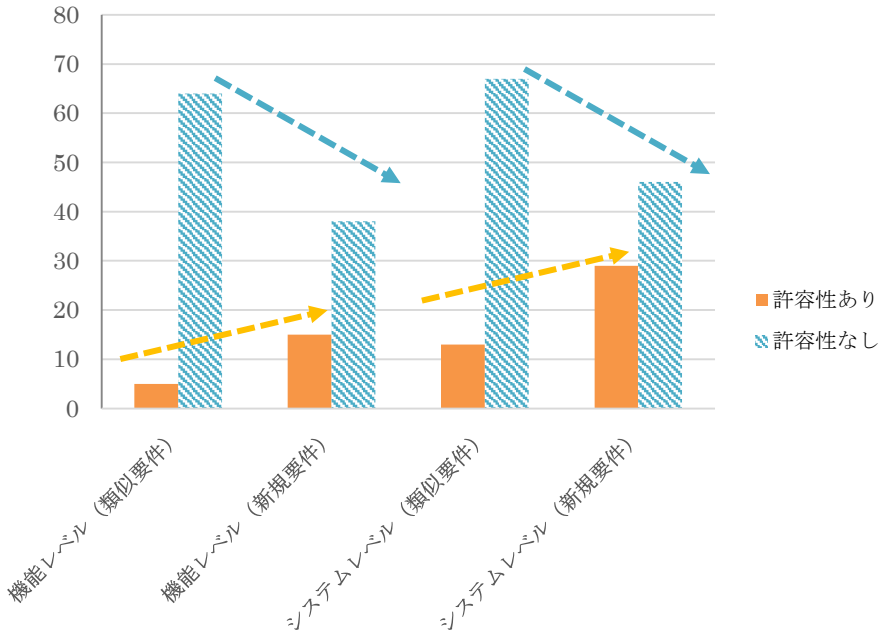


図 70　修正が必要となる関数の数

機能レベルの類似要件に対する変更量に生じるコストを 1 とした場合における，変更コスト比率を表 26 に示す．表 20 に示した変更コスト比率と実験結果を比較すると，ほぼ同様の傾向になっていることが分かる．

表 26 変更許容性評価における変更関数比率

変更 / 機能	機能レベル		システムレベル	
	類似要件	新規要件	類似要件	新規要件
許容性あり	1	3	2.6	5.8
許容性なし	12.8	7.6	13.4	9.2

7.7.5. メトリクスの変化

表 23 に示した ID1 および ID2 におけるメトリクスの比較検証を行った．

提案手法における拡張性評価においては，可変メカニズムを導入することで，使用と生成の分離を促進するため，複雑度の低下が想定される．そこで検証では，CK メトリクスの一つである Weighted Method per Class (WMC) (3)を用いて行うこととした．

$$\mathrm{WMC} = \sum_{i=1}^{n} c_i \qquad (3)$$

c：Cyclomatic complexity

i:クラスメンバ関数

WMC の値をコントローラモデル図へ重ね合わせた結果を図 71 および図 72 に示す．図中では，WMC の値が高くなるほど，各クラスの表示色を緑～赤まで変化させることで WMC の悪化程度を表現する．なお，図 71 および図 72 は鎖線表示，補足説明の記述を除き，作成したツールによる出力である．

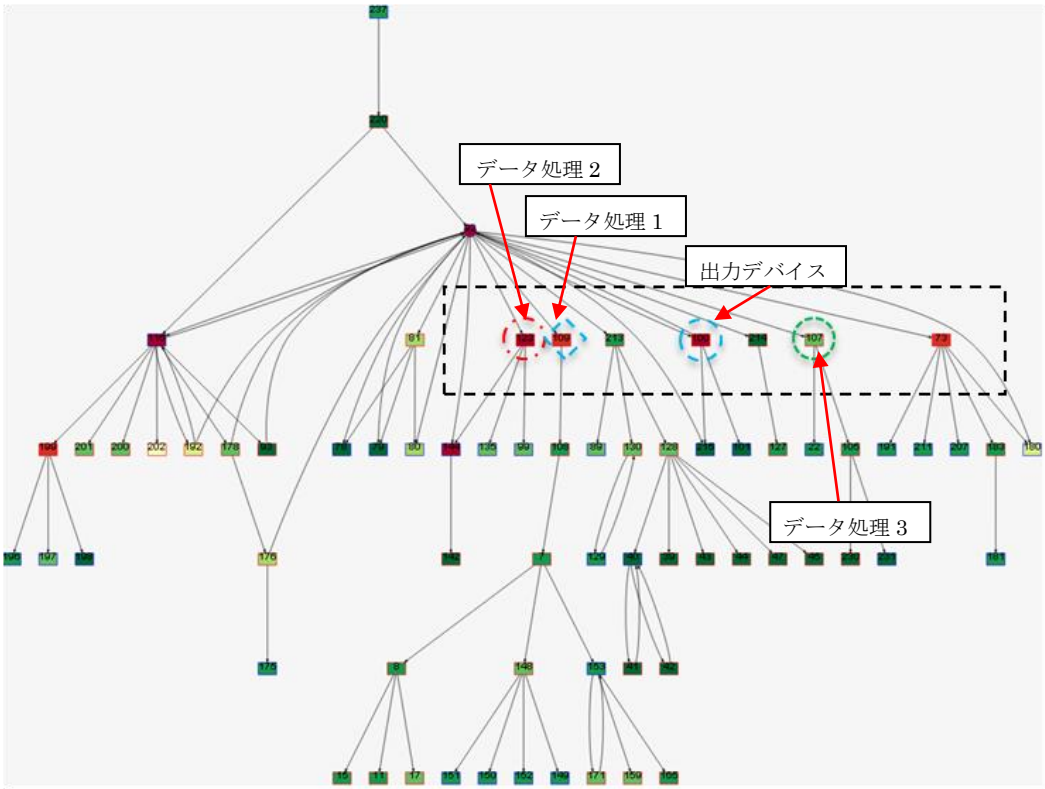


図 71　WMC 値（ID1）

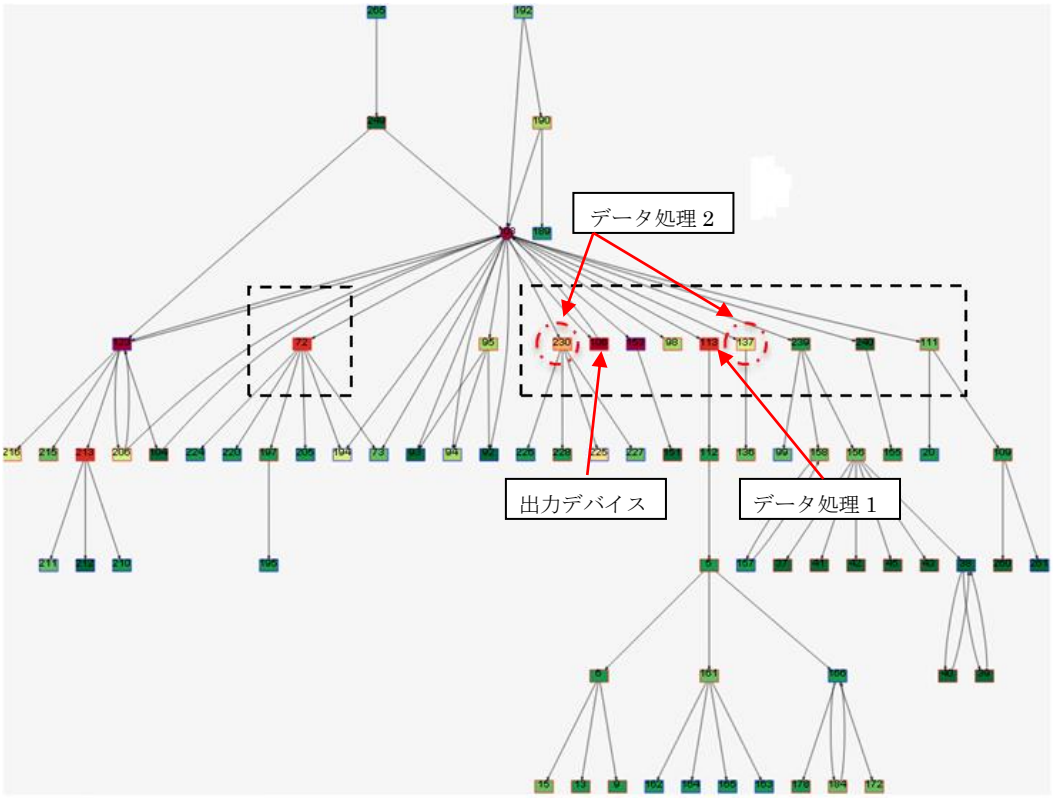


図 72　WMC 値（ID2）

ID1 では 8 クラスが WMC の高いクラスである赤色で示され，ID2 では一つ削減され 7 クラスとなった．

また，WMC の高いクラスは第 1 階層が最も多く，第 3 階層以下では存在しなかった．データ処理 2 において，図 71 では，WMC の値は赤く表示されているが，図 72 では，橙または緑への改善がみられる．これは，図 71 で第 1 階層にある一つのクラスに集約されていた処理が，図 72 ではリファクタリングにより，Factory Method パターンが改良され，2 つのクラスに分割された効果である．

一方，改善が見られない図 71 の出力デバイス（水色の荒い鎖線で示した丸枠）については，第 1 階層内のクラスで複数の制御切替えが必要であるが，図 67 の変更容易性の評価で示される通り可変メカニズムが導入されていない．システムレベルの類似要件に対する変更許容性を向上させるリファクタリングを実施することで，第 1 階層におけるクラスの WMC を低減することができる．同様に図 71 のデータ処理 1（水色の鎖線で示した菱枠）については，表 24 の変更許容性評価で示される通り，機能レベルの類似要件に対応可能な可変メカニズムを持っているが，システムレベルの要件変更に対応可能な可変メカニズムを備えていないため，WMC が高い状態である．

また，図 71 のデータ処理 3（水色の細鎖線で示した丸枠）も，図 68 の変更容易性の評価によると，出力デバイスと同じく可変メカニズムを保有していない．しかしながら，低

い WMC となっている．これは仕様が単純，かつコード規模も小さいため，WMC が低いものと考えられる．

7.8. 考察

拡張性評価手法についての評価を行った．評価観点としては，拡張性評価手法に求められる要件が満たされているかである．したがって，7.2 節で定義した課題を解決していることを評価した．

7.8.1. メンテナンス性向上コストの評価妥当性

7.2 節に示した課題について，まず，進化に対するメンテナンス性の向上コストが評価できているかを検証した．具体的には，変更の大きさ（抽象度，変更度）に対するコスト的な差異を検証した．7.7.4 項に示したように，変更許容性として評価された拡張性の構造と要件変更の抽象度や変更度には，変更に必要となる関数の比率に関連があることが分かった．許容性がない場合，要件変更の大きさによらず，許容性のある構造よりも 10 倍前後の変更量が発生した．また，許容性ありの場合，類似要件より，新規要件時の変更関数の数が多いが，許容性なしの場合は逆となった．これは，要件変更に応じて新たに追加される類似の構造とは別に，継承構造や，インタフェースとなる抽象クラスを現行構造から分離させ，さらに可変メカニズムの実装を要するからである．これにより，変更の大きさに対する備えの有無には明確なコスト的差異があるため，当該評価手法は妥当であるといえる．

7.8.2. メンテナンス性定量化の妥当性

さらに，7.2 節に示した課題であるメンテナンス性を定量化できているかについて検証した．7.7.2 項に示したように，変更許容性評価を用いることにより，現行構造で許容可能な進化の大きさを定量化することができた．これにより，実際の組込みソフトウェアに対する進化の前後で，どの機能における要件変更種類のメンテナンス性が向上したのかを把握することできた．

また，7.7.3 項に示したように，変更容易性による指標を用いることにより，進化発生時における，変更の容易さを定量化することができた，これにより，実際の組込みソフトウェアに対する進化の前後で，特定の機能において，変更容易性の値が改善したことを確認することができた．

さらに，開発プロジェクトへのヒアリングにより，当該進化の目的は，データ処理機能の一つをプラグイン化し，データ処理機能の変更に対する柔軟性を高め，自由にカスタマイズ可能にすることであった．当該評価データによって，この目的が達成されていることを裏付けることができた．また，出力デバイスについては，処理性能向上を重視している

ということであり，これも，拡張性の評価結果と一致していた．これらにより，提案手法は，機能毎に 2 種類の評価指標を用い，必要とするメンテナンス性について，その他の優先的な設計事項とのトレードオフを検討できる手法であるといえる．

例えば，優先的な設計事項がリアルタイム性であれば，機能ごとのリアルタイム性に対する要件に基づき，リアルタイム性がクリティカルではない機能については，メンテナンス性を向上させる仕組みを導入し，クリティカルな場合は，メンテナンス性を犠牲にするという客観的な判断が可能となる．また，優先的な設計事項が，プロジェクトの予算であれば，機能ごとの変更予測に基づき，最小のコストでメンテナンス性を確保可能な拡張性構造を客観的に選択することができる．

このように，提案手法を用いることで，戦略的な判断を行うための客観的なデータを取得できる．なお，提案手法の利用者がトレードオフの判断をする場合は，トレードオフの判断対象である優先的な設計事項に関する機能ごとの値を知っていることが前提となる．

7.8.3. 変更容易性評価の妥当性

本提案では，表 22 で示した可変メカニズムの評価値を得るために，特定の条件におけるクラスの使用と生成の分離に着目した変更量の違いにもとづき，可変メカニズムの隠蔽度を算出している．本提案において，算出の前提条件となっている特定の条件や，変更が加わる箇所ごとの重み付けを変えると，シミュレーション結果は異なる．そのため，本提案での値は，絶対的な比率ではない．しかしながら，上述の値を変化させても，各可変メカニズムの隠蔽程度の順序が基本的に入れ替わることはないため，相対的な比率の順序関係は正しいといえる．

7.8.4. 他手法に対する優位性について

7.7.5 項に示したように，メトリクスを用いた分析手法との比較検証を行った．WMC メトリクスを進化前後で計測することで，一部変化を確認することができた．しかしながら，WMC の変化はわずかであり，WMC の変化は，拡張性以外の要因でも変化するため，メトリクス分析では，メンテナンス性について問題の検出や定量的評価は難しいといえ，提案手法の方が優れている．

7.9. まとめ

本章では，進化型組込みソフトウェアが満たすべき要件の一つに対応する手法について説明した．進化型組込みソフトウェアにおいて，繰り返し発生する要件変更に対して，要件変更が影響を及ぼす抽象度と変更度で類型化を行うことにより，要件変更の影響レベル

に関連付けられた拡張性の評価手法について述べた．さらに，進化型組込みソフトウェアの進化過程における実際の製品コードに対する適用実験によって，提案手法の有効性を評価した．

本章で提案した拡張性評価手法が満たすべき要件は“要件 3 : 将来の進化にあった柔軟なリファクタリングができなければならない”であり，これを満たすためには，“将来の進化に対するメンテナンス性の向上コストがわからない“という課題と” 現在のメンテナンス性を定量的に知ることは難しい“という課題がある．そこで，進化における要件変更を 2 種類の“進化による影響の大きさ”で表現し，要件変更に対する許容能力による変更許容性と，要件変更を隠蔽することができる可変メカニズムの種類による変更容易性という 2 つの観点でメンテナンス性を評価する手法を提案した．

提案手法を進化型組込みソフトウェアへ適用した実験では，メンテナンス性の評価として，現行構造で許容可能な進化の大きさを定量化することができた．これにより，実際の組込みソフトウェアに対する進化の前後で，どの機能における要件変更種類のメンテナンス性が向上したのか把握することできた．さらに，変更の作業量について，各可変メカニズムの隠蔽度を定量化し，変更容易性による指標を用いて拡張性を評価した．メンテナンス性の評価として，進化発生時における，変更の容易さを定量化することができた，これにより，実際の組込みソフトウェアに対する進化の前後で，変更容易性の値が改善したことを確認することができた．

また，他の手法に対する優位性について，メトリクスを用いた分析手法との比較検証を行った．WMC メトリクスを進化前後で計測することで，一部変化を確認することができたが，WMC の変化はわずかであり，拡張性以外の要因でも変化する．故に，メトリクス分析では，メンテナンス性についての問題検出や定量的評価は難しいといえ，提案手法の方が優れている．

第8章

関連研究

8.1. はじめに

本章では，本研究で用いた各要素技術に対する関連研究について述べる．まず，8.2 節で拡張性の可視化に関する研究について述べ，続いて 8.3 節では拡張性の強化，8.4 節では拡張性の評価について述べる．8.5 節で本章をまとめる．

8.2. 拡張性の可視化に関する研究

可視化に関する研究として，ソースコードから特定の関心事に関する情報を抽出する研究，ソースコード内の特定の関心事に関する情報を定量化する研究，ソースコードから抽出した情報をビジュアル化し，理解容易性を向上させる研究についてとりあげる．

8.2.1. 機能抽出

Walkinshaw ら[32]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，ランドマーク関数とバリア関数として定義付けられた関数を抽出し，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．本研究では，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．

Ra'Fat ら[33]は，プロパティで記述されたオブジェクトセットを構成する理論的なフレームワークである FCA（Feature Concept Analysis）を用いた機能実装箇所の特定方法を提案しており，複数の製品間において，共通かつ複数の機能で共有されているオブジェクト指向の言語構造を抽出することで，機能を特定する．一方，本研究の提案方法では，レイヤ構造を持った組込みソフトウェアにおいて，クラス間の呼び出し関係をレイヤ上にマップすることで機能の抽出を行っている．

8.2.2. 可変メカニズムの抽出

Bosch ら[34]は，SPL において，より柔軟な構成を行うために，動的バインディングを可能にする可変メカニズムが必要であるとしている．これにより，開発後に進化させることができるとしており，特にシステムに必要な動的進化方法について述べている．本研究の提案手法は，オブジェクト指向言語による仕組みを用いた可変メカニズムを扱っている．

Lee ら[35]は，可変メカニズムとその実装についてまとめている．13 の可変メカニズムとそれらの実装として，オブジェクト指向プログラミングやコンポーネントベースプログラミングのような広く使われる手法から，新しい仕組みとして，アスペクト指向プログラミング，ジェネリックプログラミング，KobrA，MDA，CVL，AUTOSAR 等のような DSL もあるが，単一の開発で使われることはあっても，SPL として使われるものはあまりないと述べている．本研究では，継承をベースにしたデザインパターンによる可変メカニズムについての抽出を行っている．

8.2.3. デザインパターン抽出

Fant ら[36]は，フィーチャーモデルとデザインパターンを関連付ける手法を提案している．デザインパターン，フィーチャーモデルを各々作成し，それらをマッピングした後，デザインパターンをカスタマイズし，ドメイン固有のシステムを構築する．本研究では，フィーチャーの抽出およびパターンの抽出もソースコードベースに行っている．

Keepence ら[37]は，フィーチャー指向ドメイン分析で使われる必須，選択，オプション，類似なデザインパターン定義を行っている．本研究では，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．

Niere ら[38]は，ソースコードのリバースエンジニアリングによって抽象構文木を作成し，アノテーションを追加した UML に類似の記述法によって，デザインパターンを表現している．デザインパターンは，抽象構文木でルール化し，グラフ変換法によって，システム全体に含まれるデザインパターンを抽出，抽象化した抽象構文木を作成する．また，鷲崎ら[39]

は，ソースコードに対して，パターンを適用するためのコードが満たすべき前提条件を利用して，パターンを検出する手法を提案している．当該手法によれば，適用前条件と一致しないバージョンにおいて，適用後条件と一致した場合，当該バージョンでパターンが適用されたと検出することができる．この手法は，従来の手法よりも精度を向上させる手法である．本研究では，クラス生成に関するデザインパターンに限定し，継承とクラス呼び出し関係を用いてデザインパターンを抽出する．

8.2.4. 可視化

Denier ら[40]は，サンバーストレイアウトを用いたクラス継承関係の可視化手法を提案した．大規模ソフトウェアにおいても，可視化された図に示された箱の数や，配置された領域により，継承に関するメトリクスを算出することも可能である．本研究においては，継承モデルによって継承関係を可視化しているが，継承関係と同時に，派生クラスをインスタンス化するクラスの表現も行っている．

Chatzigeorgiou ら[41]は，クラス間の呼び出し関係から強い結合度を示すクラス関係を特定し，ハブ化しているクラスを可視化する手法を提案している．本研究においては，コントローラモデルによって，クラスの呼び出し関係を可視化しているが，デザインパターンの抽出を目的としているため，レイヤ構造の観点で可視化している．

8.2.5. メトリクスによる定量化

Bansiya ら[42]は，オブジェクト指向設計の要素をデザインメトリクスや品質属性へマップし，抽象度の高い設計品質属性の評価に対する階層的モデルを強化した．構造と振舞いのデザインと信頼性，柔軟性，複雑性について継承情報を用い関連を定義した．

Junior ら[43]は，単一製品に対するアーキテクチャの評価手法はあるが，再利用できる資産評価手法がないため，SPLA の複雑度と拡張性メトリクスを提案した．SPLA 複雑度メトリクスは SPLA におけるそれぞれのコンポーネント複雑度の合計であり，SPLA 拡張性メトリクスは SPLA におけるそれぞれのコンポーネント拡張性の合計であるとした．

Makkar ら[31]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．本研究では，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．

Kaur ら[44]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，本研究で述べたような，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.

Breesam ら[45]は, オブジェクト指向言語における継承の仕組みについて, 継承の深さ, 派生の数, 抽象クラス, 多重継承の使用, メンバ関数継承の 5 つの視点におけるメトリクスを検証し, デザインの複雑度を計測するのにそれぞれ有効であると結論付けた.

Hudli ら[46]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. 本研究は, 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.

Thaw [47]は, XML スキーマ文章に対して, 9 つのメトリクスとバイナリエントロピー関数にもとづいた再利用性と拡張性のメトリクスを提案した. 拡張性のメトリクスは, XML スキーマにおいて拡張と制限キーワードで定義される継承タイプの数, XML 要素の合計数, ユーザ定義タイプの合計数等の XML に特化し算出される.

Kayarvizhy ら[48]は, C＋＋を使ったオブジェクト指向メトリクスの実験的な適用による評価を行った. 8 種類の結合度に関するメトリクス, 6 種類の凝集度に関するメトリクス, 10 種類の継承に関するメトリクスを 2 つのプロジェクトで計測し, これらプロジェクトは, 高凝集度で, 低結合度となっており, オブジェクト指向の機能がよく使われていることをメトリクス値から確認できたとしている.

Kaur ら[49]は, CK メトリクスや MOOD メトリクスについて, 6 つのプロジェクトに適用し, 検証を行っている.

8.3. 拡張性の強化に対する研究

拡張性の強化に関する研究として, ソースコードから強化すべき箇所を特定する方法に関する研究と, どのように強化すべきかを示唆する研究についてとりあげる.

8.3.1. コードスメル

Andrade ら[50]は, コードスメル（不吉な匂い）は必ずしも問題となるわけではないため, SPL のコンテキストにおけるコードスメルの特徴を明らかにする実験的研究を行い, アーキテクチャのスメルとして, Connector Envy, Scattered Parasitic Functionality, Ambiguous Interface, Extraneous Adjacent Connector, Feature Concentration の 5 つを提案した. Andrade らは, ほぼ手動でソースコードや, 文章をまとめることで, アーキ構造やフィーチャーモデルを作成し, 手動で定義したスメル個所を見つけ出していたが, 本研究では, 定義した拡張性に関する問題個所については, 自動でソースコードから抽出することができる.

Danphitsanuphan ら[51]は, 統合開発環境である Eclipse にも搭載されている 7 種類のコードスメルの検出（巨大クラス, 長い関数, 多重継承, 多数の引数, 怠惰なクラス, ス

イッチ文，データクラス）とソフトウェアの構造的バグ（直列化可能なクラス内での非直列化可能なクラス定義，例外の無視，コンストラクタ内での未初期化，未使用なローカル変数，例外の発生しない例外処理構文の使用，非効率なオブジェクト生成）の関係についてデータマイニングを使って分析し，コードスメルと構造には関係性があることを示した．本研究は，Eclipse に搭載されているような，局所的なコード実装に起因するコードスメルとその構造ではなく，拡張性を作り出すための構造的設計に起因している問題をとらえる研究である．

8.3.2. 問題特定

田畑ら[52]は，コードスメルのカテゴリに対して，メトリクス値のマッピングを定義し，ソースコードからコードスメル箇所をリファクタリングすべき問題箇所として自動で特定する手法を提案した．また，Simon ら[53]は，メトリクスベースのリファクタリング方法を提案しており，メトリクスの値を可視化することで，リファクタリングすべき候補の特定を提案している．本研究では，メトリクスのような局所的かつ値でしかないメトリクスではなく，構造とコードスメルのマッピングにより，ソースコードからリファクタリングすべき問題箇所を特定する．

三宅ら[54]は，リファクタリングすべき箇所の識別，適用すべきリファクタリングパターンの決定，リファクタリング効果の予測について自動化する手法を提案した．リファクタリングすべき箇所の特定では，サイクロマチック複雑度，コード行数，定義されたローカル変数と実際に使用されたローカル変数の比率という 3 つのメトリクスを用いて，コードスメルを特定することで行った．また，Fontana ら[55]は，システムの設計，実装品質の評価における支援を目的としたメトリクス計測と，アンチパターン検出の使用方法を提案した．アンチパターンの検出では，依存関係の数を計測することで 6 つの構造アンチパターン（Local breakable, Global breakable , Local butterfly, Global butterfly, Local hub, Global hub）を検出し，クラス内の変数，関数の数や，それらの宣言の方法を計測することで，オブジェクト指向の設計概念から外れた実装である 4 つのアンチパターン（Cobol like, Pool, Pseudo class, Record）を検出する．本研究では，メトリクス値のみによる誤検知を避けるために，構造の可視化情報を用い，問題箇所を特定している．

8.3.3. 強化方法提案

Munro [56]は，コードスメルをモデル化し，コードメトリクスを用いて，コードスメル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，これらメトリクスが規定された基準値を超えると当該コードの改善を促す．本研究では，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，拡張性の問題箇所

を自動特定し，強化方法を提示する．

Prechelt ら[57]は，小規模な C++プログラムの修正作業におけるパターン利用の効果を，実験に基づいて定量的に評価している．複数のソフトウェア開発者を被験者として，パターンを使用する場合と，より素朴な解決策を用いる場合とで修正作業にかかる作業時間と品質を比較した結果，多くの作業内容について，パターンを使用した方が同等もしくは短い作業時間であり，品質も高かったと報告している．また，想定外の新しい要求に対応するためにも，柔軟性をもたらすパターンを使用した方がよいと結論づけている．

Ribeiro ら[58]は，SPL のモジュール化改善時に，多くの可変メカニズム（継承，コンフィグ，アスペクト）から間違った選択をすると悪影響が出るのを防ぐため，SPL の可変性再構築に関し，開発者が機構を選択するためのディシジョンモデルを提案した．これにより，テストの可変性モジュール化を改善し，重複コードのようなコードスメルを削減することができる．

Rajasree ら[59]は，SPL の可変性をモデル化するために，デザインパターンを繰り返し選択できるフレームワークを提案した．コスト，再利用性，保守性の観点における可変性の選択においては，デザインパターンにおける 4 つの属性（サイズ，静的適合性，動的適合性，拡張性）を定量化し，SPL の様々な製品へ容易に適用することができる．

8.4. 拡張性の評価に対する研究

評価に関する研究として，メンテナンス性を算出する研究と，ソフトウェアの評価方法に関する研究についてとりあげる．

8.4.1. メンテナンス性評価

Oman ら[60]は，3 つの保守性算出式（Halstead’s E をもとにした単一メトリクスモデル，Halstead’s E，拡張サイクロマチック複雑度，コード行数，コメント行数をもとにした 4 つのメトリクス多項式モデル，Halstead’s E，拡張サイクロマチック複雑度，コード行数，コメント行数，外部ドキュメントとビルド容易さの評価をもとにした 5 つのメトリクス線形再帰モデル）を提案している．

Granja-Alvarez ら[61]は，プロジェクトでの実験データから，理解容易性，変更容易性，テスト容易性のメトリクスを定義し，COCOMO をもとにメンテナンスコストの推定モデルを提案した．

Briand ら[62]は，ソフトウェアコスト予測では，システムサイズが多く使われており，また，他の構造的設計属性としては，結合度，凝集度，複雑度は追加のコスト要因を示していることが多いが，工数データを使って，クラスサイズとクラス開発工数との関係や，工数に対する結合度等の属性が何に対して追加的な影響を与えるかについて実験的な調査

をした.

Subramanyam ら[63]は, CK メトリクスのサブセットである, Size, WMC, CBO, DIT, CBO*DIT を用いたバグ予測の算出式を作成し，メトリクスと不具合の関係を示した.

ソフトウェアのメンテナンス性は意味が広い言葉であるため，多くの研究が存在するが，本研究では，ソフトウェア進化にともなう，シリーズ製品化を容易に作り出すことができる拡張性を備えることが重要という観点で，メンテナンス性をとらえ，変更の大きさや，変更のしやすさといった指標を定義して算出する方法を提案した.

8.4.2. 評価手法

Babar [64], Bengtsson ら[65], Castaldi ら[66]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いる．また，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．本研究では，既存コードからのアプローチを提案している.

Punter ら[67]は，製品評価の現状を整理し，品質プロファイルの構築，品質プロファイルを解決するための正しい実装，評価行動のデザインと実行が，今後進展すべき項目であるとして説明した．本研究では，可変性を作り出す仕組みについての拡張性を継続的にソフトウェア進化プロセスの中で評価していく手法を構築した.

Sheldon ら[68]は，設計と保守におけるクラス継承階層のメトリクスに着目し，クラス継承階層の理解容易性と変更容易性についての新しいメトリクスを提案した．理解容易性は，プログラム構造やクラス継承構造の理解のしやすさを表し，任意のクラスを呼び出すクラスの数＋1 として定量化している．また，変更容易性は，プログラム構造やクラス継承構造の変更しやすさを表し，任意のクラスの理解容易性＋呼び出すクラスの数の半数として定量化した．これらは，クラスの関連があると，その数だけコードの理解が必要という考え方である．本研究における変更容易性は，継承構造やデザインパターンという変更するための構造が既に組み込まれているかを計測することにより行っている.

Antinyan ら[69]は，機能追加時においては，不具合を埋め込んだり，可読性を低下させたりするかもしれないため，アジャイルやリーン開発におけるコード開発時のリスク評価とリスクの識別手法を提案し，複雑度とコードファイルのリビジョンがリスク評価には有効であるとした．本研究では，構造を可視化し，要件に対して，どのような変更を行うべきかを示すことによって，機能追加時の不具合混入や，メンテナンスコスト増加というリスクを回避することを目的とした評価法を提案した.

Hattori ら[70]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．本研究では，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.

Xue ら[71]は，GenericDiff によって検出される製品機能モデル（PFM）間の違いにもとづき，異なる製品バリアントの製品機能に起こりうる進化的な変更タイプを自動的に推測する手法を提案した．変更のタイプとしては，追加，削除，リネーム，移動，分割，結合

がある．また，Etxeberria ら[72]は，SPL において，製品系列の全アーキインスタンスを評価することは費用がかかるため，品質特性について可変性を考慮したコスト効果の高い SPL 品質評価手法を提案した．この手法は，品質に影響を与える可変性を識別するための拡張フィーチャーモデルを使っている．本研究では，一つの実装コードから進化に対する許容能力と進化の容易性を評価するものである．

藤原ら[73]は，アジャイル開発におけるイテレーション回数，実施テストケース数，実装開発サイズ，実績開発工数，実施レビュー回数，発見フォールト数というプロセス系のメトリクスを用い信頼性・品質を定量的に評価する手法を提案した．本研究では進化型組込みソフトウェア開発のデメリットになりうる長期的なメンテナンス性低下を防ぐために，拡張性を評価する手法である．

8.5. まとめ

本章では，本研究で提案する 3 つの手法に関したソフトウェアの可視化，強化，評価についての研究について述べた．本章で挙げた関連研究では，拡張性に関する研究に限定せず，ソフトウェアの分析手法として，ソースコードからリバースエンジニアリングを行うことで実施する研究についてまとめた．

ソフトウェアの強化に関する研究では，コードスメルやメトリクスを用いた問題の抽出手法を本研究に組合せることで，さらに提案手法を強化できる可能性があると考えられる．

第9章

結論

9.1. 本研究の成果と得られた知見

メンテナンス性が悪化し，歪んだアーキテクチャになった図 4 で示した例に対し，提案した 3 つの手法を適用することで，解析される構造を図 73 に示す.

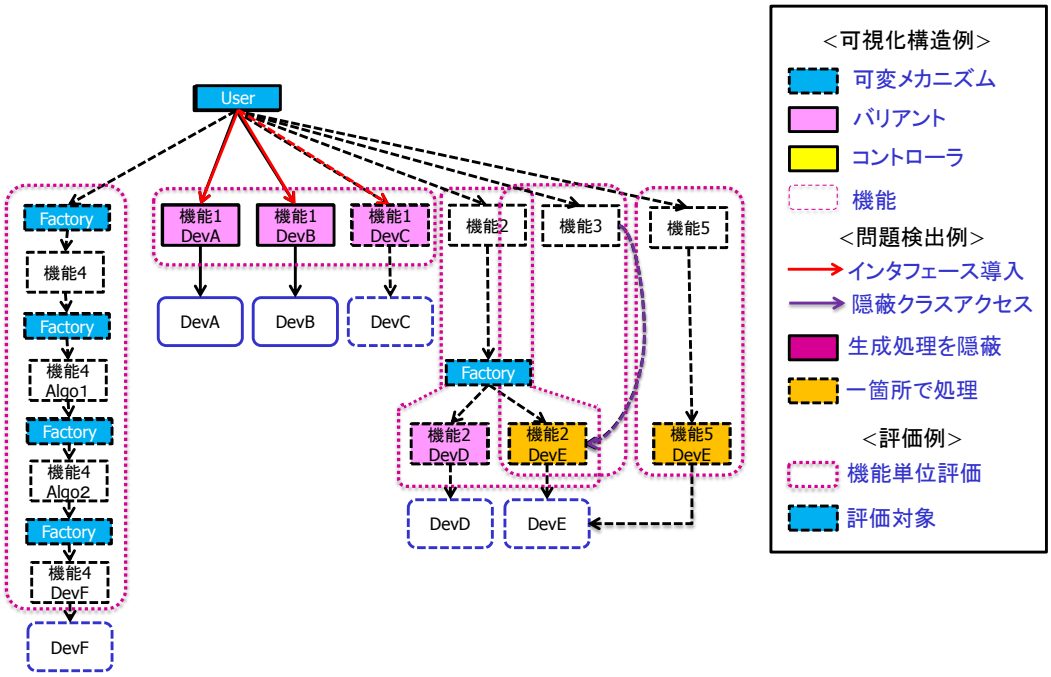


図 73　3 つの手法適用時に解析される構造

本研究では，ソースコードから拡張性に関する構造の可視化，拡張性の強化，拡張性の評価によって，進化型組込みソフトウェアのメンテナンス性向上をともなった開発を行うことができた．これらの手法を用いることで，対象とする組込みソフトウェアのメンテナンス性に関する構造を網羅的に解析・評価することができるようになった．

また，本研究においては，実装コードを可視化するために利用した組込みソフトウェアのレイヤ構造や，拡張性の評価における性能とメンテナンス性のトレードオフ関係等，組込みソフトウェアを前提とした手法により構築することができた．また，拡張性についても組込みソフトウェアの製品特性を考慮した特定の拡張性構造を観察対象にすることで計測が可能になった．

9.1.1. 拡張性の可視化

進化型組込みソフトウェアの開発において，“メンテナンス性に関する実装構造を把握できなくてはならない”という要件を満たす拡張性の可視化手法を提案した．本提案手法では，メンテナンス性に関する構造を特定するために，オブジェクト指向言語の言語構造に含まれる性質と組込みソフトウェアに特有のレイヤ構造に含まれる性質を利用することが特徴的である．

本手法では，コントローラモデルと継承モデルという異なるモデルを定義し，デザインパターンによる拡張性構造と継承による拡張性構造を明らかにした．評価実験では拡張性構造を抽出することにより，対象となる組込みソフトウェアがどのような拡張性の構造を有しており，どのように変化したかについて定量化できた．これらより，提案手法は，継承，デザインパターンで表現される拡張性構造を定量的に可視化できる手法であるといえる．

また，コード規模増大に対する提案手法の優位性を評価した結果，提案した可視化手法は，一般的なクラス図を用いた手法よりも 50%程度に捨象することができ，手作業で作成した概念モデルと比べ，クラスの抽出漏れがなかった．これらより，提案手法は，手作業による概念モデルや，ツールによる自動生成された一般的なクラス図に比べ，捨象率，精度の面において，コード規模増大に対し実用的で優位性のある手法といえる．

なお，本提案手法は組込みソフトウェア以外のドメインでは，拡張性の構造を識別することは困難であった．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．

9.1.2. 拡張性の強化手法

進化型組込みソフトウェアの開発において，“メンテナンス性の構造的な問題を自動で検出し，正しく対策できなければならない”という要件を満たす拡張性の強化手法を提案し

た．本提案手法では，メンテナンス性に関する問題箇所の定義と拡張性に関する問題箇所の検出方法が特徴的である．

提案手法は，拡張性の問題箇所を検出するルールおよびそのガイドラインで構成されており，評価実験では，5 つのルール全てにおいて，問題を特定でき，ガイドラインに沿って，拡張性を強化することができた．これにより提案手法は，自動的かつ定量的に，拡張性の問題箇所を特定・強化できる手法であるといえる．また，デバイス追加を目的とした変更に対しては，提案手法を用いた拡張性の強化による効果がみとめられた．

9.1.3. 拡張性の評価手法

進化型組込みソフトウェアの開発において，“将来の進化にあった柔軟なリファクタリングができなければならない”という要件を満たす拡張性の強化手法を提案した．本提案手法では，進化における要件変更を 2 種類の“進化による影響の大きさ”で表現した．本提案手法では，どの程度の影響がある将来の進化に対する備えを持っているかという変更許容性の指標と，進化による変更が発生した場合に，どのような仕掛けを持っているかという変更容易性の指標を定義したことが特徴的である．

評価実験では，現行構造で許容可能な進化の大きさや進化発生時における変更の容易さを定量化することができた．これにより提案手法は，機能系統ごとの変更に対する許容性と容易性を定量的に比較することができ，機能毎に 2 種類の評価指標を用い，必要とするメンテナンス性と優先的な設計事項とのトレードオフを検討できる手法であるといえる．

9.1.4. 進化型組込みソフトウェアに対する提案手法の効果

図 74 には，5.11 節で説明した製品コードに対するメンテナンス性の問題検出数の推移を示す．提案手法を用いない従来の開発では，ID1～ID3 のように，進化のたびにメンテナンス性の問題箇所が増加していた．特に，“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）”が単調増加しやすい．一方，本研究で提案した 3 つの手法を組み込んだ進化型組込みソフトウェアに対する開発では，ID4 のようにメンテナンス性の問題箇所が改良された．

このように，短期的かつ局所的なソフトウェア変更の繰り返しによって，最終的にメンテナンス性の低いソフトウェアが構築される状態となる前に，提案した進化型組込みソフトウェアにおける開発手法を用いることで，現状の拡張性を把握し，各開発者依存ではなく，開発チームとして，メンテナンス性に関する同一の基準を満たすことができる．それにより，市場ニーズの変化やハードウェア進化の速度に対応した開発を行うことが可能になる．

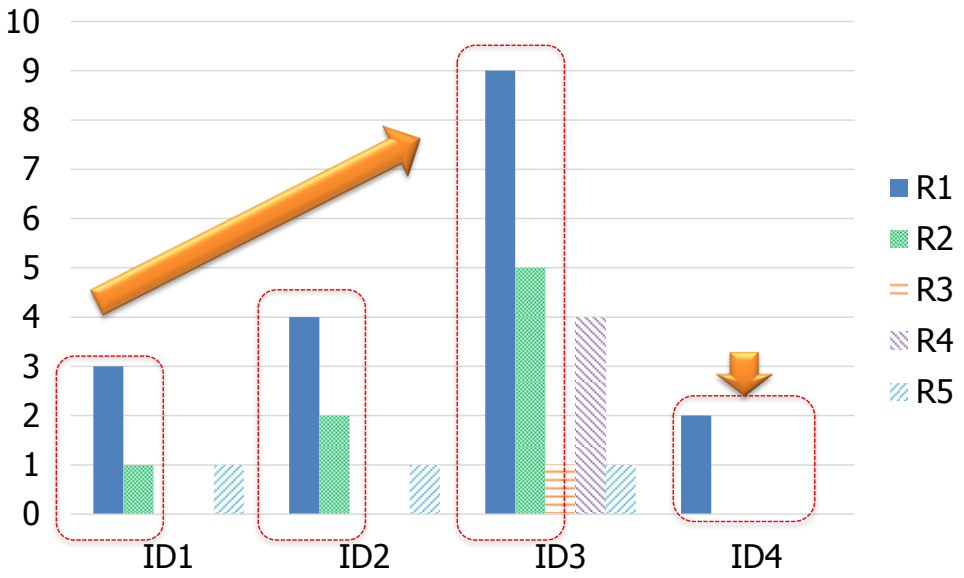


図 74　提案手法によるメンテナンス性の向上結果

9.2. 今後の展望

本研究では，進化型組込みソフトウェアにおいて，継続的なメンテナンス性の向上を行うために，実装コードに対する拡張性の可視化，強化，評価という一連の手法を提案した．しかしながら，まだいくつかの課題が残っている．以下に，本研究の今後の展望について述べる．

9.2.1. 拡張性強化ルールの拡充

本研究では，メンテナンス性に問題のある拡張性の構造を検出することができるルールを設定し，拡張性の問題箇所を改善することで，メンテナンス性を向上させながらソフトウェア進化を促すことができた．しかしながら，拡張性の構造が含まれていないコードや拡張性の問題が生じていないコードでは，提案手法が有効ではなかった．これらについても，現在より進化しやすい拡張性の構造を提案手法が開発者へ示唆できることが必要である．

9.2.2. 異なる組込みソフトウェアへの適用

本研究では，複数の異なるドメインにおけるソフトウェアへの適用検証により，拡張性を考慮したレイヤ構造を持つ組込みソフトウェアに対し，可視化手法が有効であることを示した．さらに，組込みソフトウェアの進化過程における適用検証により，本提案手法全

体が有効であることも示すことができた．今後は，複数の異なる組込みソフトウェアに対して提案手法を適用し，比較する必要がある．

9.2.3. 非オブジェクト指向言語へ対応

IPA の調査報告書[74] によると，組込みソフトウェアにおいては，C 言語の使用率が60.3%と最大ではあるが，C++/C#/Java 等のオブジェクト指向言語についても 29.7%の割合で使用されている．そこで本研究では，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で記述されたコードを対象とした．しかしながら，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特徴を理解する仕組みとして，関数インタフェースによる内部処理の隠蔽や，グローバル変数へのアクセス，構造体の類似性，関数ポインタの切替えを利用することが考えられる．このように，本研究の提案手法は，非オブジェクト指向言語へ拡張可能である．

9.3. まとめ

ソフトウェア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，技術の陳腐化，急速な製品価値低下が起きている．そこで，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ている．このような状況において，製品リリース後も積極的に市場ニーズの変化やハードウェアデバイスの進化に対し，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進化型組込みソフトウェアにおける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．本研究では，ソースコードから拡張性に関する構造を可視化し，拡張性の強化および拡張性の評価を行うことで，進化型組込みソフトウェアが必要とするメンテナンス性を向上させるための要件を満たす手法の提案を行った．さらに，実際の製品ソフトウェアを用いて提案手法の妥当性を評価した．

実際の開発現場において，多くのソースコードがメンテナンスに不安を抱えながらも，機能追加，拡張を繰り返している状況において，本研究が，ソフトウェアの資産価値を高めることができる有効な手段となれば幸いである．

研究業績

本研究は，以下の論文で発表済みである．

国内論文誌（査読あり）

1. 情報処理学会
   佐々木隆益，吉岡信和，田原康之，大須賀昭彦．(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，情報処理学会論文誌 57 巻 2 号．

国際会議（査読あり）

1. JCKBSE
   Sasaki, T., Yoshioka, N., Tahara, Y., Ohsuga, A. (2014). Evaluation of Flexibility to Changes Focusing on the Variable Structures in Legacy Software. In Knowledge-Based Software Engineering. (pp. 252-269). Springer International Publishing.

2. ICSTE
   Sasaki, T., Yoshioka, N., Tahara, Y., Ohsuga, A. (2015). A Method for Efficient Extensibility Improvements in Embedded Software Evolution. In 2015 7th International Conference on Software Technology and Engineering, Hong Kong. Journal of Software, 10 (12), 1375-1388.

下記論文は，本研究の範囲ではないが，著者の研究活動過程において，発表された論文である．

国内研究会（査読なし）

1. 電子情報通信学会
   佐々木隆益，吉岡信和，田原康之，大須賀昭彦．(2013). 割り込み処理に着目した組込みソフトウェアへのモデル検査適用の検討．電子情報通信学会技術研究報告. KBSE，知能ソフトウェア工学, 113(215), pp.19-24.

参考文献

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

[2] Coad, P., Luca, J. D., and Lefebvre, E. (1999). Java Modeling Color with Uml: Enterprise Components and Process with Cdrom. Prentice Hall PTR.

[3] Haber, A., Rendel, H., Rumpe, B., Schaefer, I., and Van Der Linden, F. (2011, August). Hierarchical variability modeling for software architectures. In Software Product Line Conference (SPLC), 2011 15th International (pp. 150-159). IEEE.

[4] Gacek, C., and Anastasopoules, M. (2001, May). Implementing product line variabilities. In ACM SIGSOFT Software Engineering Notes (Vol. 26, No. 3, pp. 109-117). ACM.

[5] Fritsch, C., Lehn, A., Strohm, T., and Bosch, R. (2002, October). Evaluating variability implementation mechanisms. In Proceedings of International Workshop on Product Line Engineering (PLEES) (pp. 59-64).

[6] Coleman, D., Lowther, B., and Oman, P. (1995). The application of software maintainability models in industrial software systems. Journal of Systems and Software, 29(1), 3-16.

[7] Ogheneovo, E. E. (2013). Software Maintenance and Evolution: The Implication for Software Development. West African Journal of Industrial and Academic Research, 7(1), 81-92.

[8] 玉井哲雄, 中谷多哉子. (2004). ソフトウェア進化プロセスの統計モデル. コンピュー

タ ソフトウェア, 21(3), 159-168.

[9] Chapin, N. (2004, September). Agile methods' contributions in software evolution. In Software Maintenance, 2004. Proceedings. 20th IEEE International Conference on (p. 522). IEEE.

[10] ISO/IEC (2011). ISO/IEC 25010 Systems and software engineering -- Systems and software Quality Requirements and Evaluation (SQuaRE) -- System and software quality models. ISO/IEC.

[11] ISO/IEC (2013). ISO/IEC 26550 Software and systems engineering -- Reference model for product line engineering and management. ISO/IEC.

[12] Kang, K. C., Lee, J., and Donohoe, P. (2002). Feature-oriented product line engineering. IEEE software, (4), 58-65.

[13] Nguyen, T., and Colman, A. (2010, July). A feature-oriented approach for web service customization. In Web Services (ICWS), 2010 IEEE International Conference on (pp. 393-400). IEEE.

[14] Suresh, Y., Pati, J., and Rath, S. K. (2012). Effectiveness of software metrics for object-oriented system. Procedia Technology, 6, 420-427.

[15] Chidamber, S. R., and Kemerer, C. F. (1994). A metrics suite for object oriented design. Software Engineering, IEEE Transactions on, 20(6), 476-493.

[16] Melo, W. (1996, March). Evaluating the impact of object-oriented design on software quality. In Software Metrics Symposium, 1996 Proceedings of the 3rd International (pp. 90-99). IEEE.

[17] 後藤祥, 吉田則裕, 藤原賢二, 崔恩瀞, 井上克郎. (2014). 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, 31(3), 3_318-3_324.

[18] Martin, R. (1994). OO design quality metrics. An analysis of dependencies, 12, 151-170.

[19] Budgen, D. (2013). Design Patterns: Magic or Myth? IEEE software, (2), 87-90.

[20] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1994). Design patterns: elements of reusable object-oriented software. Pearson Education. (本位田真一, 吉田和樹 (監訳): オブジェクト指向における再利用のためのデザインパターン, 改訂版, ソフトバンククリエイティブ, 1999.)

[21] Koenig, Andrew (1995 March/April). Patterns and Antipatterns. Journal of Object-Oriented Programming, 8 (1). 46-48.

[22] Brown, W. H., Malveau, R. C., and Mowbray, T. J. (1998). AntiPatterns: refactoring software, architectures, and projects in crisis. (岩谷宏 (訳): アンチパターン: ソフトウェア危篤患者の救出，新装版，ソフトバンクパブリッシング，2002.)

[23] Ujhelyi, Z., Horvath, A., Varró, D., Csiszár, N. I., Szoke, G., Vidács, L., and Ferenc, R. (2014, February). Anti-pattern detection with model queries: A comparison of approaches. In Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week-IEEE Conference on (pp. 293-302). IEEE.

[24] Fowler, M. (1997, June). Refactoring: Improving the Design of Existing Code. In 11th European Conference. Jyväskylä, Finland.

[25] Duszynski, S., Knodel, J., and Becker, M. (2011, October). Analyzing the source code of multiple software variants for reuse potential. In Reverse Engineering (WCRE), 2011 18th Working Conference on (pp. 303-307). IEEE.

[26] Duszynski, S., and Becker, M. (2012, June). Recovering variability information from the source code of similar software products. In Proceedings of the Third International Workshop on Product Line Approaches in Software Engineering (pp. 37-40). IEEE Press.

[27] Xue, Y. (2011, May). Reengineering legacy software products into software product line based on automatic variability analysis. In Proceedings of the 33rd International Conference on Software Engineering (pp. 1114-1117). ACM.

[28] Mens, T., and Tourwé, T. (2004). A survey of software refactoring. In Software Engineering: (pp.126-139). IEEE

[29] Thomas, D., and Jan, W. (2002). Tool-supported discovery and refactoring of structural weaknesses in code. In Unpublished doctoral dissertation, Technical University of Berlin, Germany.

[30] Jim, J. (2002 May). Standish Group International, Inc. 2003b. Standish Group Study. In The XP 2002 Conference. Sardinia.

[31] Makkar, G., Chhabra, J.K., and Challa, R.K. (2012). Object oriented inheritance metric-reusability perspective. In *International Conference on Computing, Electronics and*

Electrical Technologies: (pp. 852-859). Kumaracoil

[32] Walkinshaw, N., Roper, M., and Wood, M. (2007). Feature Location and Extraction using Landmarks and Barriers. In *Proceedings of International Conference on Software Maintenance:* (pp. 54-63). Paris

[33] Ra'Fat, A., Seriai, A., Huchard, M., Urtado, C., Vauttier, S., and Salman, H. E. (2013). Feature location in a collection of software product variants using formal concept analysis. In Safe and Secure Software Reuse (pp. 302-307). Springer Berlin Heidelberg.

[34] Bosch, J., and Capilla, R. (2012). Dynamic variability in software-intensive embedded system families. Computer, (10), 28-35.

[35] Lee, J., and Hwang, S. (2014). A review on variability mechanisms for product lines. International Journal of Advanced Media and Communication, 5(2-3), 172-181.

[36] Fant, J. S., Gomaa, H., and Pettit, R. G. (2013, January). A pattern-based modeling approach for software product line engineering. In System Sciences (HICSS), 2013 46th Hawaii International Conference on (pp. 4985-4994). IEEE.

[37] Keepence, B., and Mannion, M. (1999). Using patterns to model variability in product families. IEEE software, (4), 102-108.

[38] Niere, J., Schafer, W., Wadsack, J.P., Wendehals, L., and Welsh, J. (2002). Towards pattern-based design recovery. In *Proceedings of the 24rd International Conference on Software Engineering:* (pp. 338-348). Orlando, FL

[39] 鷲崎弘宜, 深谷和宏, 久保淳人, 深澤良彰. (2010). パターン適用前のソースコードを用いたデザインパターン検出. コンピュータ ソフトウェア, 27(2), 136-141.

[40] Denier, S., and Sahraoui, H. (2009, October). Understanding the use of inheritance with visual patterns. In Empirical Software Engineering and Measurement, 2009. ESEM 2009. 3rd International Symposium on (pp. 79-88). IEEE.

[41] Chatzigeorgiou, A., Xanthos, S., and Stephanides, G. (2004, May). Evaluating object-oriented designs with link analysis. In Proceedings of the 26th International Conference on Software Engineering (pp. 656-665). IEEE Computer Society.

[42] Bansiya, J., and Davis, C. G. (2002). A hierarchical model for object-oriented design quality assessment. Software Engineering, IEEE Transactions on, 28(1), 4-17.

[43] Junior, E. A. O., Maldonado, J. C., and Gimenes, I. (2010, September). Empirical validation of complexity and extensibility metrics for software product line architectures. In Software Components, Architectures and Reuse (SBCARS), 2010 Fourth Brazilian Symposium on (pp. 31-40). IEEE.

[44] Kaur, K., and Singh, H. (2011). Determination of Maintainability Index for Object Oriented Systems. ACM SIGSOFT Software Engineering Notes, 36(2), 1-6.

[45] Breesam, K.M. (2007). Metrics for Object-Oriented Design Focusing on Class Inheritance Metrics. In *2nd International Conference on Dependability of Computer Systems:* (pp. 231-237). Szklarska

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

[47] Thaw, T. Z. (2011, December). Measuring and evaluation of Reusable quality and extensible quality for XML schema documents. In Research and Development (SCOReD), 2011 IEEE Student Conference on (pp. 473-478). IEEE.

[48] Kayarvizhy, N., and Kanmani, S. (2011, April). Analysis of Quality of object oriented systems using Object Oriented Metrics. In Electronics Computer Technology (ICECT), 2011 3rd International Conference on (Vol. 5, pp. 203-206). IEEE.

[49] Kaur, A., Singh, S., Kahlon, K. S., and Sandhu, P. S. (2010). Empirical Analysis of CK & MOOD Metric Suit. Int. Journal of Innovation, Management and Technology, 1(5), 447-452.

[50] de Andrade, H. S., Almeida, E., and Crnkovic, I. (2014, April). Architectural bad smells in software product lines: An exploratory study. In Proceedings of the WICSA 2014 Companion Volume (p. 12). ACM.

[51] Danphitsanuphan, P., and Suwantada, T. (2012, May). Code Smell Detecting Tool and Code Smell-Structure Bug Relationship. In Engineering and Technology (S-CET), 2012 Spring Congress on (pp. 1-5). IEEE.

[52] 田畑敦史, 鈴木正人. (2008). メトリクスの測定によるリファクタリング支援の自動化 (メトリクス (学生セッション)). 情報処理学会研究報告. ソフトウェア工学研究会報告, 2008(29), 131-138.

[53] Simon, F., Teinbruckner, F. S., and Lewerentz, C. (2001). Metrics based refactoring. In Software Maintenance and Reengineering, Fifth European Conference on (pp. 30-38). IEEE.

[54] 三宅達也, 肥後芳樹, 井上克郎. (2008). ソフトウェアメトリクスとメソッド内の構造を用いたリファクタリング支援手法の提案. 信学技報, SS2008-25, July.

[55] Fontana, F. A., and Maggioni, S. (2011, June). Metrics and antipatterns for software quality evaluation. In Software Engineering Workshop (SEW), 2011 34th IEEE (pp. 48-56). IEEE.

[56] Munro, M.J. (2005). Product Metrics for Automatic Identification of "Bad Smell" Design Problems in Java Source-Code. In *Software Metrics, 2005. 11th IEEE International Symposium:* (pp. 15-15). Como, Italy

[57] Prechelt, L., Unger, B., Tichy, W. F., Brossler, P., and Votta, L. G. (2001). A controlled experiment in maintenance: comparing design patterns to simpler solutions. Software Engineering, IEEE Transactions on, 27(12), 1134-1144.

[58] Ribeiro, M., and Borba, P. (2009, March). Improving guidance when restructuring variabilities in software product lines. In Software Maintenance and Reengineering, 2009. CSMR'09. 13th European Conference on (pp. 79-88). IEEE.

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

[60] Oman, P., and Hagemeister, J. (1994). Construction and testing of polynomials predicting software maintainability. Journal of Systems and Software, 24(3), 251-266.

[61] Granja-Alvarez, J. C., and Barranco-García, M. J. (1997). A method for estimating maintenance cost in a software project: a case study. Journal of Software Maintenance, 9(3), 161-175.

[62] Briand, L. C., and Wust, J. (2001). The impact of design properties on development cost in object-oriented systems. In Software Metrics Symposium, 2001. METRICS 2001. Proceedings. Seventh International (pp. 260-271). IEEE.

[63] Subramanyam, R., and Krishnan, M. S. (2003). Empirical analysis of ck metrics for object-oriented design complexity: Implications for software defects. Software Engineering, IEEE Transactions on, 29(4), 297-310.

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

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

[66] Castaldi, M., Inverardi, P., and Afsharian, S. (2002). A case study in performance, modifiability and extensibility analysis of a telecommunication system software architecture. In Modeling, Analysis and Simulation of Computer and Telecommunications Systems, 2002. MASCOTS 2002. Proceedings. 10th IEEE International Symposium on (pp. 281-290). IEEE.

[67] Punter, T., Solingen, R. V., and Trienekens, J. (1997, October). Software Product Evaluation. In 4th Conference on Evaluation of Information Technology.

[68] Sheldon, F. T., Jerath, K., and Chung, H. (2002). Metrics for maintainability of class inheritance hierarchies. Journal of Software Maintenance and Evolution: Research and Practice, 14(3), 147-160.

[69] Antinyan, V., Staron, M., Meding, W., Osterstrom, P., Wikstrom, E., Wranker, J., and Hansson, J. (2014, February). Identifying risky areas of software code in Agile/Lean software development: An industrial experience report. In Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week-IEEE Conference on (pp. 154-163). IEEE.

[70] Hattori, L., Guerrero, D., Figueiredo, J., Brunet, J., and Damasio, J. (2008). On the precision and accuracy of impact analysis techniques. In *Proceedings of the Seventh International Conference on Computer and Information Science*: (pp.513-518). Portland, Oregon

[71] Xue, Y., Xing, Z., and Jarzabek, S. (2010, October). Understanding feature evolution in a family of product variants. In Reverse Engineering (WCRE), 2010 17th Working Conference on (pp. 109-118). IEEE.

[72] Etxeberria, L., and Sagardui, G. (2008, September). Variability driven quality evaluation in software product lines. In Software Product Line Conference, 2008. SPLC'08. 12th International (pp. 243-252). IEEE.

[73] 藤原隆次, 山田茂. (2007). アジャイル開発における品質メトリクスにもとづくソフト

ウェア信頼性評価に関する一考察 (一般セッション). プロジェクトマネジメント学会研究発表大会予稿集, 2007, 200-205.

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

謝辞

本論文をまとめるにあたり，終始あたたかいご指導を賜りました電気通信大学 大須賀 昭彦 教授，田原 康之 准教授に厚く御礼申し上げます．社会人学生として，右も左もわからない中，論文執筆について丁寧にご指導頂き，本当にありがとうございました．

また，国立情報学研究所におけるトップエスイー在籍中の研究活動から電気通信大学における本論文執筆に至るまで長期間に渡りご指導いただいた，国立情報学研究所 GLACE センターおよび総合研究大学院大学 吉岡 信和 准教授に心より感謝いたします．

大学での研究活動を常にサポート頂きました清 雄一 助教，利 百合子 女史，システム設計基礎学講座（大須賀・田原研究室）の学生の皆様にも深く感謝申し上げます．

さらに，ソフトウェア工学についての学びの場を提供して頂きました国立情報学研究所 本位田 真一副所長および GRACE センター長，石川 冬樹 准教授，鄭 顕志 助教，田辺 良則 特任教授をはじめとするトップエスイー教員の方々に心より感謝の意を表します．

トップエスイーおよび博士後期課程進学を薦めて頂き，ソフトウェア工学について研究の機会を下さいましたキヤノン株式会社 ソフトウェア基盤技術開発センター 佐々木 誠司 部長および木村 岳男 室長をはじめとする同部員の皆様にも深く感謝いたします．

最後に，仕事と研究の両立で悩んでいる私を励まし，支えてくれた家族，友人，愛犬に心から感謝いたします．

著者略歴

佐々木隆益

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．2011 年国立情報学研究所トップエスイー5 期修了．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．情報処理学会会員．