情報処理学会論文誌 Vol.55 No.2 1069–1078 (Feb. 2014)

プロダクトライン開発における アーキテクチャリファクタリングの意思決定法

牧 隆史[1,a)] 岸 知二[2]

受付日 2013年5月10日, 採録日 2013年11月1日

概要：ソフトウェア開発，特に同一アーキテクチャ上で製品群を開発するプロダクトライン開発においては，アーキテクチャの進化は重要な課題である．またアーキテクチャには設計の目標となる参照アーキテクチャと，個々の製品のソースコードとして具現されている実装アーキテクチャがあり，これが進化の問題をより複雑にしている．本稿ではこうしたアーキテクチャ進化の問題を広義にアーキテクチャリファクタリングの問題ととらえ，リファクタリングの意思決定に活用する手法を提案する．提案手法では，製品群の開発から得られるコスト上の気づきに関係する実装アーキテクチャと参照アーキテクチャ上の問題の大きさを定量化し，その特性傾向をアーキテクチャリファクタリングの方針立てに活用する．また，過去に行われた民生機器用のプロジェクトデータを用いて本手法の適用評価を行った．

キーワード：プロダクトライン開発，ソフトウェアアーキテクチャ，リファクタリング

A Decision Making Method for Product-Line Architecture Refactoring

Takashi Maki[1,a)] Tomoji Kishi[2]

Received: May 10, 2013, Accepted: November 1, 2013

Abstract: In software development, we have to evolve architecture along with the change of business and technical environment. In product-line development (PLD), this becomes more crucial because we have to develop various products on the shared architecture. We observe that there are two types of architecture; one is reference architecture at the design level, the other is implementation architecture as an actual implementation structure of the reference architecture. In architectural evolution, we have to consider these two types and this makes the problem more complicated. In this paper, we assume architecture evolution as a kind of architectural refactoring, and propose a decision making method for architecture refactoring in PLD. In our proposed technique, we quantify magnitudes of problems on reference and implemented architecture for 'bad smells' got from products project, then we utilize the result for decision making on architecture refactoring. We also evaluate the method based on actual data from our consumer products project.

Keywords: product-line development, software architecture, refactoring

1. はじめに

類似の特性を持つ製品群を効率的に開発するため，組み込みソフトウェア開発を中心にプロダクトライン型の開発手法が広がりを見せている [1], [2], [3], [4], [5], [6], [7], [8], [9]．プロダクトライン開発（Product-Line Development, PLD）とは，製品群に共通して適用されるソフトウェアアーキテクチャを定め，これに基づいて整備されたコア資産に基づいて複数製品の開発を行うことで開発効率向上を実現する手法である．このため，アーキテクチャの評価や決定のための手法についてはこれまで数多くの提案がなされてきた [6], [11], [12]．

しかしながらビジネス・技術環境の変化が厳しい製品開

[1] 株式会社リコーワークス・ソリューション開発本部
Ricoh Co., Ltd., Work Solutions Development Division, Yokohama, Kanagawa 224–0035, Japan
[2] 早稲田大学理工学術院
Waseda University, Faculty of Science and Engineering, Shinjuku, Tokyo 169–8555, Japan
[a)] takashi.maki@nts.ricoh.co.jp

発においては，当初の要求分析には含まれていなかった市場要求や技術環境の変化に対応するために，アーキテクチャを状況に応じて進化させることが必要となる．こうしたアーキテクチャ進化の問題はPLD以外のソフトウェア開発でも存在するが，1つのアーキテクチャ上で複数の製品群を（一般には）より長期間開発するPLDにおいてはその問題がいっそう顕在化する．

PLDには資産化のアプローチとしていくつかの方法が提案されている．たとえばプロアクティブな方法 [16], [17] ではビジネス・技術環境の変化によって当初のロードマップから変化が生じた場合，アーキテクチャの進化が必要となる．また，リアクティブな方法 [18] では必要となった箇所から順次コア資産に進化させていくためインクリメンタルなアーキテクチャの進化が必要となる．

これらのことからPLDにおいては，アーキテクチャを環境や状況に応じて進化させていくことが，プロダクトライン資産の効率的な整備や長期にわたる活用のための有効な手立ての1つであるといえる．

一般にソフトウェアのアーキテクチャには設計目標としての参照アーキテクチャと，個々の製品のソースコードとして具現されている実装アーキテクチャとがある．アーキテクチャの進化を考える際には参照の進化，実装の進化，さらに両者の整合という点を考慮する必要がある．一般に参照の進化は新たな状況に対応するための設計目標の変更を，実装の進化は実装レベルの品質の向上を目指すもので，実施のコストや影響範囲も異なる．したがって開発の中ではどの時点でどういう進化を実施するかという意思決定がプロジェクトに与える影響は大きい．しかしながら参照・実装を考慮した意思決定手法に関する研究はあまり行われていない．

我々はこうしたアーキテクチャ進化を広義のアーキテクチャリファクタリングの問題ととらえ，参照と実装の2種類のアーキテクチャの違いに注目したアーキテクチャリファクタリングの意思決定手法を検討・提案してきた [10]．以前の提案 [10] ではリファクタリング項目に対してポートフォリオ分析を行うことでこれらの優先度を判断する手法の提案を行ったが，この方法には検討の作業が複雑になるという問題があった．本稿ではプロジェクトへの適用を簡素化できるように以前の提案手法 [10] を改善し，実製品プロジェクトのデータを用いて手法の評価を行った．

以下本稿では，2章でアーキテクチャリファクタリングの概念について，3章では提案手法について説明する．4章ではプロジェクトデータを用いた提案手法の評価結果について，5章では結果の考察，6章では関連する研究との関連を，7章で今後の課題についてそれぞれ述べる．

2. アーキテクチャリファクタリング

本章では本稿におけるアーキテクチャおよびリファクタリングの概念について説明する．

2.1 参照アーキテクチャと実装アーキテクチャ

アーキテクチャとは構造を抽象化して表現したものであり，その抽象度には何段階かのレベルがあるが，本稿では関数など実装上の局所的な構造ではなく，設計段階でとらえるより粗粒度の機能単位であるサブシステムレベルでの抽象度を対象とし，それらの間の機能分割と依存関係に着目する．

アーキテクチャには，あるべき姿（設計目標）としての参照アーキテクチャと，あるものの姿（ソースコードで実現されたサブシステムレベルの構造）としての実装アーキテクチャがあるという見方ができる．本稿で扱う両アーキテクチャの差異は両者の位置づけの違いによるものであり，抽象度のレベルは同等である．現実の開発プロジェクトにおいてはこれらの間に乖離の問題があり，両者にはそれぞれに進化も起こりうる．このため，アーキテクチャ上の問題も実装アーキテクチャの問題と参照アーキテクチャの問題に分解して検討することで，問題の構造をより明確にできることが期待される．

2.2 リファクタリング

ソフトウェアの保守性や移植性の観点から，システムの振舞いを変えることなくプログラムの内部構造を変更する技術としてリファクタリングが知られている．ここではアーキテクチャに対する類似のアプローチ，すなわちそのアーキテクチャ上で実現が想定される主要な機能を維持しつつ保守性などの品質特性を向上させるためにアーキテクチャを進化させることをアーキテクチャリファクタリングと呼ぶことにする．ビジネス・技術環境の変化に対応するためのアーキテクチャ進化は，アーキテクチャ上で想定される機能の広がりをともなうことがあり，機能を変えないソースコードのリファクタリングなどとはやや異なる側面を持ちうる．しかしながら，本稿では過去のプロダクト群との連続性を保つ状況を考えており，本質的かつ大幅な想定機能の変更までは考慮しないことから，あくまで想定機能のための品質特性の改善・向上を意図するものとして，リファクタリングという用語を用いている．

ソフトウェア開発におけるアーキテクチャとは通常，開発において全体最適を維持するために設ける設計上の制約の集まりを指す．一般には参照となるアーキテクチャをまず決め，それを実際に実装する形で開発が行われるので，アーキテクチャリファクタリングについてもそれぞれについて考慮することが有用と考えている．

実装アーキテクチャのリファクタリングとは，リファクタリングの影響が全体構造に及ぶような大規模リファクタリングのことである．Roock ら [15] は，大規模リファクタリングにおいてはFowler [14] が提唱する局所的なリファク

タリングとは異なる注意，たとえばリファクタリング実施時期への配慮や製品プロジェクト上であらかじめリファクタリングのための工数確保などが必要になることを述べている．しかしながら Roock ら [15] の問題提起はいずれも実装の大規模なリファクタリング，つまり実装アーキテクチャのリファクタリングに関するものであり，参照アーキテクチャとの関係に及ぶ配慮がなされていない．

参照アーキテクチャのリファクタリングは設計における制約事項の変更であり，この変更により新たに実装上の不適合箇所が生じる場合には速やかに実装をこれに合わせることが望ましい．参照アーキテクチャの変更によって生じる不適合箇所は，サブシステム間の関係のルール変更が原因であるため，変更内容によっては大がかりな実装の修正が必要となりうる．

Roock らの指摘にもあるように実装アーキテクチャの修正には大きなコストがかかる．参照アーキテクチャの変更はさらに大きなコストや影響を持つ．一般に開発のある時点で考えられるリファクタリングの候補は様々なものがありうるが，それらのうちのどれをどのタイミングで行うかによって，プロジェクトに対する効果や影響が異なってくる．我々はどのようなリファクタリングを行うべきかを決定する問題を意思決定の問題ととらえ，その手法を提案する．

3. 提案手法

本章では，前章で指摘した問題をふまえ，参照・実装を考慮したリファクタリングの意思決定手法を提案する．本手法は過去の提案 [10] を基に手法中におけるポートフォリオ分析の位置づけをより使いやすくするよう改善したものである．

3.1 基本的な概念

提案手法の説明に必要となる不吉な匂い，構造上の要因，そしてリファクタリング項目の概念について以下に述べる．

(1) 不吉な匂い

ソフトウェア開発時に観測される，将来的に問題を引き起こしそうな兆候として経験的に知られているソースコード上の特徴は「不吉な匂い（Smell）」と呼ばれている．Fowler [14] は「重複したコード」などソースコード上で観測される 22 種類の不吉な匂いについて言及している．Fowler によるものはいずれもその原因と対策がソースコードの問題に閉じたものであるが，不吉な匂いの概念はアーキテクチャについても拡張が可能である．たとえば Roock らは特定サブシステムへの機能の集中や依存関係の循環といった，アーキテクチャレベルで問題と見られる現象を Architecture Smell として取り上げ，大規模リファクタリングの対象箇所を見つける手がかりとすることを提唱している [15]．Architecture Smell については Pollack [19] もビジネスロジックと非機能要件の密結合など 5 種類をあげて言及している．しかしながら，これらはいずれも実装アーキテクチャを対象とした考察であり，PLD において参照アーキテクチャも含めたリファクタリングの戦略立てを行うには十分とはいえない．

提案手法では PLD に代表される類似製品群の開発に際し，アーキテクチャに関わる諸活動（機能の追加・変更や非機能要件や品質特性の達成など）において，継続的に観測されている開発コストおよび開発成果物の品質などへの悪影響を「不吉な匂い」として抽出する．抽出した不吉な匂いは，アーキテクチャリファクタリングの戦略立ての出発点として用いる．以下に不吉な匂いのカテゴリ（S1～S3）および具体例を示す．

- 機能追加・変更コストへの影響（S1）
  - 例 1）機能能追加時に影響範囲を確認すべき箇所が多く工数がかかっている．
  - 例 2）特定のサブシステムに多くの機能が集中しているため，変更箇所の理解に時間がかかる．
- 品質への影響（S2）
  - 例 1）他のサブシステムと比較して不具合多発の傾向が見られる．
  - 例 2）同種の不具合が繰り返し発生する．
- 検証コストへの影響（S3）
  - 例 1）テストの組み合わせが多い．
  - 例 2）レグレッションテストの項目が多い．

(2) 構造上の要因

現象として観測されている不吉な匂いのうち，ソフトウェアの構造に起因するものを詳細に分析（ソースコードやアーキテクチャドキュメントに立ち入った調査）することにより，それらの元となっているアーキテクチャ上の問題点を明らかにする．本研究ではこの問題点を構造上の要因と呼ぶことにする．具体的な調査方法としては，問題現象に日々直面している開発担当者へのインタビューが有効である．構造上の要因は，参照アーキテクチャに起因するものと実装アーキテクチャに起因するものに分けられる．全体像としては，既存研究で知られている Code Smell [14] および Architecture Smell [15] を参考とし，下記に示すカテゴリに基づいて具体的な問題を特定する．以下に構造上の要因のカテゴリ（Pr1～Pr2，Pi1～Pi3）および具体例を示す．

参照アーキテクチャに起因する構造上の要因

- 機能の集中によるもの（Pr1）
  - 例 1）特定サブシステムの役割定義が広すぎ，特定サブシステムにコードが偏在している．
  - 例 2）全体から参照されるものが多く，他サブシステムからの依存が集中している．
- 機能の分散によるもの（Pr2）
  - 例 1）サブシステム間の依存関係が多い．

表 1 問題の大きさの評価指標
Table 1 Classification of magnitude of the structural problem.

構造上の要因	問題の在処	計測対象の例	5段階評価指標 小 ←(問題の大きさ)→ 大				
			1	2	3	4	5
サブシステムの役割定義が広すぎる	参照	サブシステムのソースコード行数(LOC)	<(L1)	<(L2)	<(L3)	<(L4)	(L4)≦
全体から参照されるものが多い	参照	依存関係の数	<(N1)	<(N2)	<(N3)	<(N4)	(N4)≦
サブシステム間の依存関係が多い	参照	依存関係の数	<(N1)	<(N2)	<(N3)	<(N4)	(N4)≦
コンパイルスイッチが多い	実装	コンパイルスイッチの出現密度	<(R1)%	<(R2)%	<(R3)%	<(R4)%	(R4)%≦
関数が長い	実装	関数の平均行数	<(L1)	<(L2)	<(L3)	<(L4)	(L4)≦
複雑度が高い	実装	サイクロマチック複雑度の平均値	<(C1)	<(C2)	<(C3)	<(C4)	(C4)≦
結合度が高い	実装	依存関係の数	<(N1)	<(N2)	<(N3)	<(N4)	(N4)≦
サブシステムの構成要素が異なる	実装	機能差分の数	<(F1)	<(F2)	<(F3)	<(F4)	(F4)≦
依存関係の齟齬	実装	依存関係の数	<(N1)	<(N2)	<(N3)	<(N4)	(N4)≦

実装アーキテクチャに起因する構造上の要因

- 実装技術の種類によるもの（Pi1）
  例 1）コンパイルスイッチが多いために保守性が低い．
- 実装の巧拙によるもの（Pi2）
  例 1）個々の関数が長い．
  例 2）複雑度が高い．
  例 3）構成要素間の結合度が高い．
- 参照アーキテクチャとのアライメント（Pi3）
  例 1）サブシステムの構成要素が異なる．
  例 2）依存関係の齟齬が存在する．

構造上の要因の大きさを定量化するため，各要因に応じたメトリクスを取得する．

参照アーキテクチャのメトリクスについては，アーキテクチャドキュメントに対してスコアリングを行う方法と，フォルダ構成のような設計意図を反映した実装構造やソースコードなど実装成果物の計測を通して間接的に行う方法がある．前者は直接にアーキテクチャの評価が行えるが，ドキュメントの記述精度に問題がある場合や評価結果のばらつきが大きい場合には誤差も大きくなる．後者では実装と参照に差異がある場合にはこれが誤差の原因となるが，評価対象物や評価方法の不正確に起因する誤差は生じない．どちらを選択するかはプロジェクトの状況と測定対象の特性によって判断する．たとえば，実装がほぼ参照に近い形で行われており，両者に多少の差異があっても，メトリクスで得られる特徴傾向が十分顕著であって，両者の差異の影響が及ばないと判断される場合には，後者を用いるなどである．

実装アーキテクチャのメトリクスについては，ソースコードなど実装成果物の計測により行う．

表 1 は参照アーキテクチャのメトリクスとして後者の方法を選択した場合のメトリクス選択例である．この例では，結果として依存関係の数が参照と実装の両方の評価で用いられているが，計測はそれぞれの構造上の要因を反映する箇所で行う．表 1 中で変数表記としている評価指標の具体的な数値については対象プロジェクトでの経験値に基づいてあらかじめ決めておく．得られたメトリクス値をこうした経験に基づき問題の大きさの小さい方から順に 1 から 5 の 5 段階のスコアを与えて正規化することで，異なる構造上の要因の問題の大きさを相対的に比較できるようにする．

(3) リファクタリング項目

個々の構造上の要因を解決するために行う一連のリファクタリング作業をリファクタリング項目と呼ぶことにする．リファクタリング項目は構造上の要因に対応するものであるため，その粒度はソースコード上の特定箇所のリファクタリング作業を具体的に示すというよりはむしろ，問題を解決するためのリファクタリング方針ととらえてもよい．参照アーキテクチャに対しては構成要素間の機能分割を改善する視点で立案する．また，実装アーキテクチャに対しては，対応する構造上の要因を解消するためのコードリファクタリングを立案する．以下にリファクタリング項目のカテゴリ（Rr1～Rr3，Ri1～Ri2）および具体例を示す．

参照アーキテクチャに対するリファクタリング項目

- サブシステムを分割または作成するもの（Rr1）
  例 1）特定の関心事に基づく処理を新しいサブシステムに移動する．
  例 2）機能追加に備えてサブシステムを新規に作成する．
- サブシステムを併合するもの（Rr2）
  例 1）類似した複数サブシステムを 1 つにまとめる．
  例 2）サブシステムの役割を拡張して新しい役割を収容する．
- サブシステムの役割を変更するもの（Rr3）
  例 1）サブシステム間で機能分割の境界を変える．

実装アーキテクチャに対するリファクタリング項目

- 実装技術を変更するもの（Ri1）
  例 1）変動点の実現方法をコンパイルスイッチから別の方式に変更する．
  例 2）関数ポインタを用いて処理を抽象化する．
- 実装技術は変えずに改善を行うもの（Ri2）
  例 1）長い関数を小さい単位に分ける．
  例 2）条件分岐のネストを浅くする．
  例 3）不要なコンパイルスイッチを削除する．
  例 4）サブシステム間で処理を移設する．

不吉な匂い，構造上の要因，リファクタリング項目のカテゴリ間には一般に**図 1** のように多対多の関係がある．すなわち，不吉な匂いとして問題が露見する背後にはその元となる構造上の要因が複数あり，さらにこれらの要因を取り除くためには，一般に具体的ないくつかのリファクタリング項目を組み合わせることになる．

3.2 ポートフォリオを用いた特性分析

それぞれの不吉な匂いに関して，対応する参照アーキテクチャと実装アーキテクチャの問題の大きさの評価結果を

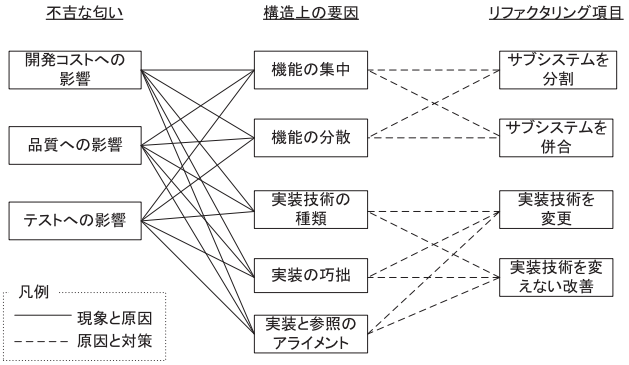

図 1 基本的な概念間の関係
Fig. 1 Relationship between general ideas.

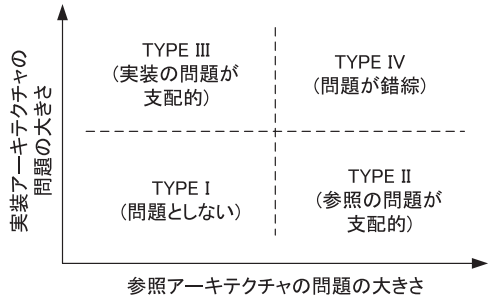

図 2 構造上の要因のポートフォリオ分析図
Fig. 2 Portfolio analysis of structural problem factor.

それぞれ X 軸と Y 軸とした平面上（**図 2**）にプロットした図を用いて，不吉な匂いの特性分析を行う．問題の特性傾向は大きく 4 区分に分けて考える．

プロットされた複数の不吉な匂いの相対的な位置関係は，対応の優先度判断の参考にすることができる．以下にそれぞれの区分について説明する．

TYPE I—問題としない：他の TYPE と比較して実装上も参照上も問題が小さい場合である．リファクタリングの優先度は他の区分よりも低いと判断できる．

TYPE II—参照の問題が支配的：相対的に参照アーキテクチャの問題が大きい場合である．この状況が発生するケースとしては，変更があまり発生しない技術領域において規格や法令の変化など要求仕様が，大きく変わる場合や，TYPE IV に分類される不吉な匂いに対して実装の対応が先行して行われた場合があげられる．実装アーキテクチャの問題は少ないため，実装は参照アーキテクチャに忠実に行われているが，参照アーキテクチャ自体が陳腐化して変更要求に耐えられなくなってきている状況が考えられる．

TYPE III—実装の問題が支配的：実装アーキテクチャの問題が相対的に大きい場合である．この状況が発生するケースとしては，参照アーキテクチャの意図に対して実装技術が及ばなかった場合や，リアクティブなアプローチによって参照アーキテクチャが変更されたことによって実装との間に乖離が生じた場合があげられる．参照アーキテクチャの問題は大きくなく，実装が参照から乖離しているケースがこれに含まれる．実装の修正により問題は解消することが多い．

TYPE IV—問題が錯綜：実装も参照も問題が大きい場合である．これは参照アーキテクチャが適切でない状態において，実装の劣化も進行してしまったときに見られる．この状況においては参照アーキテクチャをこのままにして実装の修正のみを行っても参照アーキテクチャの問題が残るので，結局は参照の改善とそれに合わせた再度の実装の改善を行うことになる．したがって解決するにはまず参照アーキテクチャの修正を行うべきである．

3.3 提案手法の手順

以下に，3.1 節，3.2 節に基づき，製品開発におけるリファクタリングの意思決定手法を以下に述べる．

STEP 1 プロジェクト上の不吉な匂いを検出する．

プロジェクトの経験から，プロジェクト進行にとってコスト上問題と見られ，同一アーキテクチャ上で開発される複数製品の開発において繰り返し発生している現象を集める．

STEP 2 構造上の要因へのマッピング

STEP 1 で集めた不吉な匂いを分析し，問題となる現象を引き起こす原因を列挙し，それぞれの原因が参照アーキテクチャに起因するものか実装アーキテクチャに起因するものかを判断する．

STEP 3 リファクタリング項目を決める

STEP 2 で得た各問題を解決するためのリファクタリング項目群を決め，それに必要な工数を見積もる．

STEP 4 問題の大きさの定量化

STEP 2 で得た各問題の大きさを，問題の内容に応じて選定したメトリクスを用いて定量化する．異なった計測手段による結果どうしを比較可能とするため，各メトリクスはあらかじめ過去の経験に基づいて 5 段階に正規化したものを用いる．

STEP 5 ポートフォリオによる分析

STEP 4 で求めた参照アーキテクチャと実装アーキテクチャ上との問題の大きさをそれぞれ X 軸，Y 軸として STEP 1 で集めた不吉な匂いをポートフォリオ分析図（図 2）上にプロットし，問題の特性傾向を把握する．

STEP 6 リファクタリング優先度判断

STEP 3 で決めた各リファクタリングにかかる工数，STEP 4 で求めた問題の大きさ，STEP 5 で求めた不吉な匂いの特性傾向，およびプロジェクトの状況を総合的に判断し，リファクタリングの着手優先度を判断する．判断方法の基本的な考え方は以下である．

- プロジェクトにおいて許容される工数の範囲内でリファクタリングを計画する．
- 工数が同じであれば解決される問題の大きさの大きなものを優先する．

- 近い将来に大規模な変更が予定されている場合には参照のリファクタリングは控える．

STEP 7 リファクタリングの実施

STEP 6 で得た判断結果に基づいてリファクタリングを実施する．

4. 提案手法の評価

本研究ではすでに実施されたプロジェクトに対して提案手法を過去に遡って適用し，プロジェクトで実際に行われたアーキテクチャリファクタリングと比較することで提案手法が示唆するリファクタリング優先度の妥当性について評価を行った．

4.1 手法評価のアプローチ

図 3 に本研究で行った提案手法評価のアプローチを示す．

実プロジェクトでは，製品群の開発時点で観測されていた開発上の問題に基づいて経験者の判断によってリファクタリングを実施した結果，開発効率の向上など一定の成果が得られた．これが本研究で前提とした事実である．本研究では開発時に観測されていた開発上の問題を出発点として用い，提案手法に基づいて構造上の問題を定量的に分析することでリファクタリング項目とその優先度を求め，実際に行ったリファクタリングとの比較を行った．すなわち手法による判断で，実際のプロジェクトと同様の判断を行うことができることを示すことで手法の妥当性を示す．さらに，リファクタリング実施前後におけるアーキテクチャの改善結果の可視化分析により，特性傾向の変化という視点から，問題現象の元となる複数の要因の寄与について明らかにする．

4.2 プロジェクトの概要と課題

対象としたプロジェクトは民生機器の組み込み制御ソフト開発プロジェクトであり，全体規模は 50 万 LOC 程度である．同一プラットフォーム上で複数機種の開発を並行して行っており，年間 5 機種前後の製品がリリースされている．製品のリリース間隔は 4〜6 カ月程度である．

類似の製品を複数機種開発する中で，既存ソフトウェアのアーキテクチャが原因となって開発効率が上がらないと思われる状況が見られるようになる．こうしたアーキテクチャ上の問題を解決するためにフルスクラッチでの全体再設計を行う方法も有効であるが，対象プロジェクトの開発規模を考慮すると通常の製品リリースの間に再設計することには困難があった．このため，全体を一度に修正するのではなく，問題箇所を順次修正していくアプローチ，すなわちアーキテクチャリファクタリングが有効と判断した．

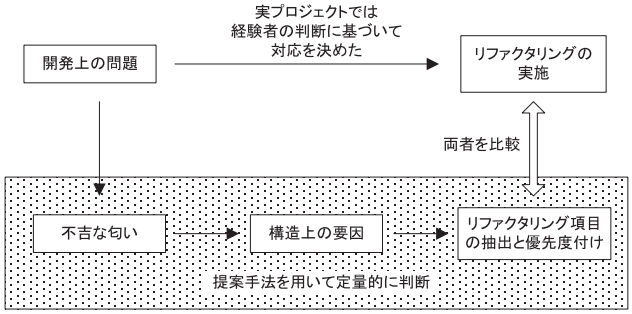

図 3 評価のアプローチ
Fig. 3 Approach of the evaluation.

4.3 実プロジェクトでのリファクタリング

図 4 にプロジェクトにおけるリファクタリング前後の参照アーキテクチャを示す．全体としてレイヤアーキテクチャ [13] に準じた形となっているが，いわゆる厳密なレイヤリングは前提としていない．レイヤアーキテクチャでは一般にレイヤの上下関係に対応した依存関係の制約が設けられる．対象プロジェクトのアーキテクチャは 3 層レイヤに準じた構造であり，各サブシステムはいずれかのレイヤ（または共通部）に属する．上位層にはユーザ I/F などハードウェアから見て抽象度の高いもの，中間層には製品の機能を実現するアプリケーションロジック，下位層にはハードウェア制御に関するものが配置されており，これら全体から参照される共通部がある．実システムでは図 4 に記載されている以外にもサブシステムがいくつか存在するが，今回の説明に関係しない部分については記載を省略している．

図 4 において共通部以外の依存関係は基本的には上位側から下位側への参照で構成するルールだが，実装上は逆方向の参照も一定数存在しており，これがサブシステム間の循環参照をなすことが保守性を低下させる一因となっていた．

表 2 に当該プロジェクトで当時観測されていた主な開発上の問題を示す．

(1) apl での機能追加にともなう修正工数が多い．
(2) 機能追加に必要な変更が特定サブシステムに閉じていない．
(3) ui で不具合が比較的多く発生する．

これらの問題に対し，実プロジェクトで計測が行われていたサブシステム間の参照数の情報を参考として，実装における循環参照の解消に取り組むことを中心に，アーキテ

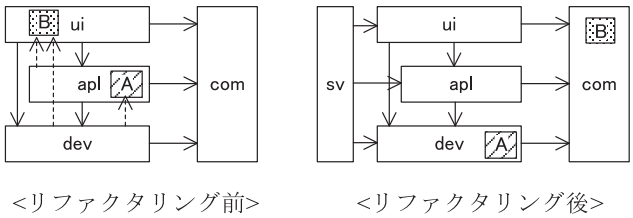

図 4 参照アーキテクチャのリファクタリング
Fig. 4 Refactoring of reference architecture.

表 2 プロジェクトデータの適用結果
Table 2 Adoption result by using project data.

不吉な匂い		問題の在処	構造上の要因	問題の大きさの計測対象	問題の大きさ		リファクタリング項目
					リファクタリング前	リファクタリング後	
#1	aplの変更時に確認すべき関連箇所が多く，修正に工数がかかっている．(S1)	実装	①下位層(dev)からの逆参照が多く，apl層修正の影響範囲が読めない．(Pi3)	dev→aplへの逆参照数	5	2	①dev→aplの逆参照を解消する．(Ri2)
		参照	②ハードウェア制御部分が混在しているため，apl層の行数が多い．(Pr1)	apl層の総行数	5	4	②ハードウェア依存部分はdevに移動させる．(Rr3)
#2	一つの修正のために複数サブシステムに修正が必要となるため，修正に手間がかかっている．(S1)	参照	③全体から参照されるべきもの，全体を参照するものが各部に点在しているため，全体にサブシステム間参照が多い．(Pr2)	単位行数あたりのドメイン間参照数	4	3	③全体に関与するものを外部に分離する．(Rr1)
#3	uiにおいて機能追加に工数がかかっており，変更時のミスが発生しやすい．(S2)	実装	④個々の関数の複雑度が高いために保守性が低い．(Pi2)	サイクロマチック複雑度の平均値	5	4	④単純な関数に分割し、関数の複雑度を下げる．(Ri2)
			⑤コンパイルスイッチが多いため保守性が低い．(Pi1)	コンパイルスイッチ出現密度	5	5	⑤不要なコンパイルスイッチ及び関連コードを除去する．(Ri2)
		参照	⑥設定情報のデータをuiが持っているため，下位(特にapl)からの逆参照が多い．(Pr1)	apl→uiへの逆参照数	4	2	⑥設定情報は共通アクセスエリアに移動する．(Rr3)
			⑦機能を実現するためのコードがuiに集中しているため，機能の増加に伴ってコード量が増える．(Pr1)	uiのコード量	4	4	今回は計画せず．

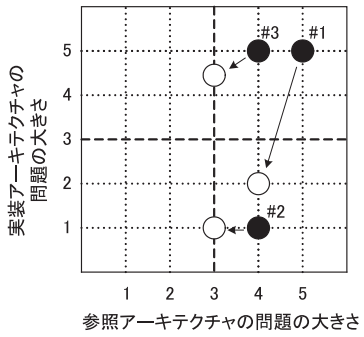

図 5 ポートフォリオ分析図
Fig. 5 Portfolio chart.

クチャを図 4 の右側に示す形に変更することにした．具体的には (1) に対しては apl と dev の間の循環参照を解消するように A の処理を dev へ移動し，(2) に対しては全体に関与する部分を sv として独立させ，(3) に対しては B で示した全体から参照されるデータを共通エリア（com）に移動した．実プロジェクトでは後述する提案手法の適用で明らかになった問題の在処（表 2）については必ずしも明示的に意識していなかったが，構造上の要因の一部については定性的に認識があり，これに基づいて参照アーキテクチャの変更を実施している．実プロジェクトでは問題の大きさについての概念は持っていなかった．

4.4 提案手法の適用検証結果

プロジェクトでのリファクタリングの参考にしたものと同じ開発上の問題を出発点とし，3.3 節に示した提案手法の手順における STEP 1 から STEP 4 までに対応して詳細に分析した結果を表 2 に，STEP 5 のポートフォリオ分析を行った結果について**図 5** に示す．

表 2 に記載した情報はいずれも開発当時のソースコードおよび設計ドキュメントの分析に基づいて得られたものである．表 2 に示す問題の大きさは，それぞれのメトリクスで得られた結果を表 1 の評価指標に基づいて正規化して得られた値である．

図 5 の黒丸がリファクタリング前の不吉な匂いに対して手法を適用して得られたポートフォリオ分析の結果である．なお白丸は，リファクタリング後に再度手法を適用して問題の変化をとらえたもので，意思決定の段階では参照しない．提案手法におけるリファクタリングの意思決定は，図 5 における黒丸の位置（リファクタリング前の状態）が属するタイプの種類に基づき，3.2 節に記載のタイプ別の方針に従って行う．

図 5 のポートフォリオ分析図において，不吉な匂い#1 と#3 は 3.2 節で示した特性傾向の TYPE IV（問題が錯綜）に該当するため，実装の問題も大きいがまず参照から手を付けることが望ましいことが示唆されている．表 2 によれば，不吉な匂い#1 に関する参照の構造上の要因は apl の総行数が多いことであり，リファクタリング項目としてはハードウェア依存部分を dev に移動させることである．この判断はプロジェクトにおいて図 4 の A の部分を移動したことに符合している．また，不吉な匂い#3 の参照に対応するリファクタリング項目は設定情報を共通アクセスエリアに移動することであり，この判断はプロジェクトにおいて図 4 の B の部分を移動したことに符合している．不吉な匂い#2 については，実装上の問題は大きくないが，参照アーキテクチャの改善が示唆されている．不吉な匂い#2 の参照に対応するリファクタリング項目は全体に関与するものを外部に分離することであり，この判断はプロジェクトにおいて全体に関与する部分を sv として独立させたことに符合する．

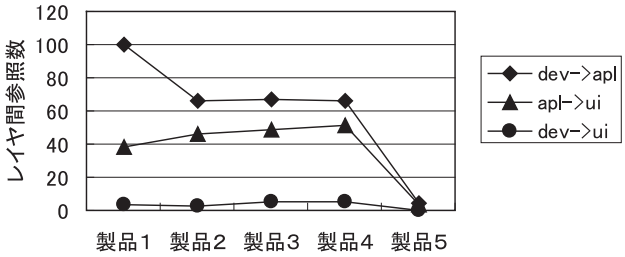

図 6 レイヤ間逆参照数の推移
Fig. 6 Transition of reversed access count.

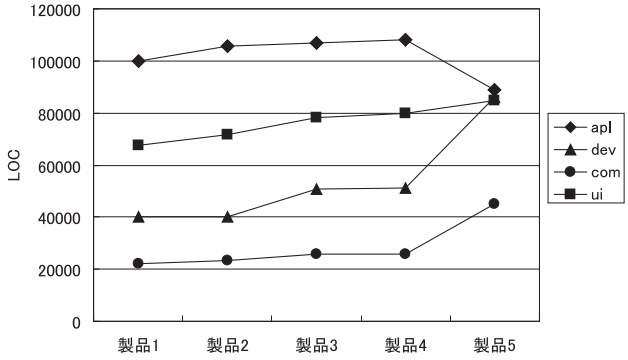

図 7 レイヤごとの LOC
Fig. 7 Transition of LOC in each layer.

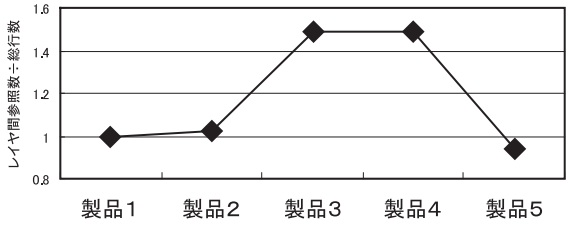

図 8 単位行数あたりのレイヤ間参照数
Fig. 8 Transition of access count.

4.5 メトリクスによる意思決定の可視化

表 2 に示した不吉な匂いに関係するアーキテクチャリファクタリングについて，プロジェクトデータを用いて可視化分析を行った結果を以下に示す．

不吉な匂い#1

実装および参照上の問題のいずれも大きいため，3.2 節の特性傾向では TYPE IV（問題が錯綜）が相当すると判断される．すなわち，両軸の問題が大きいため，解決するにはまず参照アーキテクチャに手をつけたほうが望ましいというものである．**図 6** に，レイヤ間逆参照の推移を示す．横軸は製品の開発順，縦軸はレイヤ間の参照数である．参照数は，レイヤを超えて参照される変数，関数，およびマクロの数の合計である．縦軸は製品 1 における dev→apl の参照数を 100 として正規化している．製品 1〜4 は参照アーキテクチャリファクタリング前の製品であり，製品 4 と製品 5 の間で参照アーキテクチャの変更と対応する実装の変更を行っている．ここでは構造上の要因①に対応するメトリクスの推移として dev→apl の逆参照数の推移に着目する．

実プロジェクトでは逆参照箇所が問題であることが認識されており，可能な範囲において随時修正を行っていた．製品 1 と製品 2 の間で dev→apl の逆参照数をいったん大きく減らすことができたのはこの効果であった．しかしその後しばらくは改善が見られず，次に大きく改善したのは製品 5 の時点であり，これは参照アーキテクチャを変更したタイミングと一致している．

次に，参照アーキテクチャのリファクタリング効果についてメトリクス値を確認する．**図 7** はレイヤごとの LOC の推移を示したものである．縦軸 LOC はコメントと空行を除いたソースコードの行数，横軸が製品開発の時間軸であり，製品 1 における apl 層の LOC を 100,000 として正規化している．不吉な匂い#1 の参照アーキテクチャに関する構造上の要因は apl 層の総行数のメトリクスであるため，図 7 の apl レイヤの LOC の変化に注目すると，アーキテクチャリファクタリングを実施した製品 5 において改善していることが確認できる．

製品 5 で行ったリファクタリング項目②の効果として，apl レイヤ中に含まれていたハードウェアへの依存部分（斜線の A の部分）を dev に移動することにより，apl の規模が小さくなると同時に dev→apl への逆参照数も改善することが期待されていた．実際に製品 5 では図 6 に示したように dev→apl への逆参照がほぼ解消する結果となった．つまり，不吉な匂い#1 への対応としては，プロジェクトで最初に実施した実装アーキテクチャのリファクタリング効果は限定的であり，問題の根本的な解消には参照アーキテクチャのリファクタリングを待つ必要があったと見ることができる．このことは 3.2 節の特性傾向での TYPE IV（問題が錯綜）の示唆，すなわち先に参照アーキテクチャの進化が必要，という特性傾向が当てはまる．

不吉な匂い#2

不吉な匂い#2 は 3.2 節の特性傾向によると TYPE II（参照の問題が支配的）が相当すると判断される．すなわち，実装は参照どおり行われているが，改善には参照アーキテクチャのリファクタリングが必要と見られるケースである．計測対象のメトリクスとしては単位行数あたりのドメイン間参照数を用いた．**図 8** に，単位行数あたりのレイヤ間参照数の推移を示す．縦軸は製品 1 での値を 1 として正規化している．

レイヤ間参照数は製品開発の影響により製品 3 と 4 の時点で増加傾向にあったが，参照アーキテクチャの変更を実施した製品 5 では，増加前の製品 1 の状態と比較しても若干の改善が見られることが確認できた．リファクタリング項目③は，サブシステム間の制御など全体に関与するものが各サブシステム内に散在することによって起こっていた問題への対応であるため，図 4 の右図に示すように全体制

御にかかる部分を sv として分離することにより，独立性をより高めた実プロジェクトでの判断と一致する．

不吉な匂い#3

不吉な匂い#3 も 3.2 節の特性傾向によると TYPE IV（問題が錯綜）に相当すると判断される．ここでの参照アーキテクチャに対する構造上の要因として⑥と⑦の 2 つをあげている．要因⑥については製品 1〜4 のアーキテクチャにおいて ui への参照はその大半が ui の持っている設定情報データであるため，設定情報データの置き場所（図 4 における B の部分）を ui から共通エリアに移動することで解消することが期待される．これは実際にプロジェクトで実施した参照アーキテクチャリファクタリングとも一致している．メトリクスである他サブシステムから ui への参照数の推移については図 6 に示したとおり製品 5 の時点でほぼ解消していることが分かる．なお，要因⑦についてはレイヤ型アーキテクチャを採用する限りある程度避けられない問題と判断し，今回は対応の検討を行っていない．実装アーキテクチャに関する構造上の要因④と⑤については，構造上の要因がいずれも他サブシステムとの関係がさほど強くなかったものであるため，参照アーキテクチャの変更を待たずに改善できた部分もあった．構造上の要因が他サブシステムとの関係が強くない場合には必ずしも TYPE IV の示唆が当てはまらなかったといえる．

5. 考察

実プロジェクトでは，参照アーキテクチャのリファクタリングは，本手法におけるポートフォリオ分析を用いず専門家の経験に基づく判断によって行ったが，ポートフォリオ分析が示唆する結果はおおむねこれと方向性が合っていることが確認できた．

さらに，図 5 に示すポートフォリオ分析の結果から，リファクタリングによる問題傾向の変化も観測されている．たとえば不吉な匂い#1 に関しては TYPE IV（問題が錯綜）から TYPE II（参照の問題が支配的）に遷移しており，今回の参照アーキテクチャリファクタリングの結果，実装上の問題は大きく改善したが，参照アーキテクチャの問題はまだ残されていると読み取ることができる．このように不吉な匂いに対応する問題の特性のポートフォリオ分析結果は，継続的にアーキテクチャリファクタリングを行う際，次のアクションについての判断材料としても用いることが可能となる．

6. 関連研究との関係

PLD においてアーキテクチャの評価や決定を行う手法については，これまで数多くの提案がなされてきた [6], [11], [12]．これらは個別のアーキテクチャについて評価を行う技術としては有効であるが，開発と並行してアーキテクチャリファクタリングを行う目的に対しては直接に解を与えるものではない．また，組み込みシステムでのプロダクトライン開発におけるアーキテクチャの改良事例として，Kolb らの事例 [20] が報告されている．彼らのアプローチは製品開発と並行してアーキテクチャを進化させていくという点，および管理指標としてレイヤ間の依存関係に着目して順次実装を参照アーキテクチャにあわせていくという点において本研究と類似しているが，目標とするアーキテクチャのゴールは PuLSE 手法 [6] を用いてあらかじめ定めたものであり，開発途中で発生するアーキテクチャ進化の要求に対応するには不十分である．また，Bourquin ら [21] はアーキテクチャ規約違反箇所の検出に基づいた大規模リファクタリングによって，製品開発を行いながらアーキテクチャリファクタリングの実施例を報告している．しかしながらここでアーキテクチャ修正の主たる動機となっているのはもっぱら実装上で検出されたアーキテクチャ違反であり，PLD において重要である参照アーキテクチャも視野に入れた上で総合的にリファクタリングの優先順位を判断するものではない．

これに対して我々の手法は，開発時に観測されるプロジェクト上の課題（開発コストなど）を出発点とし，その原因となっているシステム上の問題（アーキテクチャ上の問題）を実装と参照に分けて検討することにより，製品開発の事情に合わせてリファクタリングの実施を判断する手段を提供することが可能である．

7. 今後の課題

今回は，過去のプロジェクトデータを用いることで，実施したアーキテクチャリファクタリングの意思決定についてプロジェクトで行われた判断との比較を行うことができた．異なったメトリクスの正規化は一般には困難であるが，個別のプロジェクトの経験に基づくことである程度比較が可能であり，今回は一定の妥当性のある正規化が行えたといえる．しかしながら定量的な精密性という観点では今後の工夫による改善の余地があるものと考えている．また，現実の局面ではリファクタリングにかかる工数も重要な要素であり，実施に必要な工数とそれによる改善効果とのバランスについても考慮する必要がある．今後は，改善効果や定量的な正確性の向上にも考慮して本手法をさらに進化させていきたい．

8. おわりに

本稿では，PLD においてアーキテクチャのリファクタリングを行う際の意思決定手法を提案し，プロジェクトデータを用いて評価を行った．アーキテクチャ上の問題を参照アーキテクチャの問題と実装アーキテクチャの問題に分解し，その構造上の要因をポートフォリオ上で表現することで，アーキテクチャリファクタリングの意思決定に対して有益な判断材料となる可能性があることを確認した．実施

による改善効果も考慮に入れた総合的な意思決定手法については今後の課題である．

参考文献

[1] Clements, P. and Northrop, L.M.: Software Product Lines: Practices and Patterns, Addison-Wesley (2001).
[2] Boeckle, G., Clements, P., et al.: Calculating ROI for Software Product Lines, *IEEE Software*, Vol.21, No.3 (2004).
[3] Phol, K., Boeckle, G. and Linden, F.V.D.: Software Product Line Engineering: Foundations, *Principles and Techniques*, Springer-Verlag New York Inc. (2005).
[4] Linden, F.V.D.: Software Product Families in Europe: The Esaps & Café Projects, *IEEE Software*, Vol.19, No.4 (2002).
[5] Atkinson, C., Bayer, J., et al.: Component-based Product Line Engineering, Addison-Wesley (2002).
[6] Bayer, J., Flege, O., et al.: PuLSE: A Methodology to Develop Software Product Lines, *Proc. 5th ACM SIGSOFT Symposium on Software Reusability* (*SSR '99*), pp.122–131 (1999).
[7] Gomaa, H.: Designing Software Product Lines with UML, Addison-Wesley (2005).
[8] 渡辺晴美：組み込みソフトウェア開発技術：4. プロダクトライン開発技術，情報処理，Vol.45, No.7, pp.694–698 (2004).
[9] 岸　知二, 野田夏子, 位野木万里ほか：特集 ソフトウェア再利用の新しい波—広がりを見せるプロダクトライン型ソフトウェア開発，情報処理，Vol.50, No.4, pp.265–310 (2009).
[10] Maki, T. and Kishi, T.: Problem Factor Portfolio Analysis for Product Line Architecture Refactoring, *17th Asia-Pacific Software Engineering Conference* (*APSEC 2010*): Industry Papers (2010).
[11] Kazman, R., Klein, M., Barbacci, M., et al.: The Architecture Tradeoff Analysis Method, Software Engineering Institute, Technical Report CMU/SEI-98-TR-008 (1998).
[12] Kazman, R., Abowd, G., Bass, L. and Webb, M.: SAAM: A Method for Analyzing the Properties of Software Architectures, *Proc. 16th International Conference on Software Engineering*, Sorrento, Italy, pp.81–90 (May 1994).
[13] Buschmann, F., Meunier R., Rohnert, H., et al.: Pattern-Oriented Software Architecture: A System of Patterns, John Wiley & Sons (1996).
[14] Fowler, M.: Refactoring: Improving the Design of Existing Code, Addison-Wesley (1999).
[15] Roock, S. and Lippert, M.: Refactoring in Large Software Projects: Performing Complex Restructurings Successfully, John Wiley & Sons (2006).
[16] Clements, P.: Being Proactive Pays Off, *IEEE Software*, pp.28–30 (July/Aug. 2002).
[17] Krueger, C.: Easing the Transition to Software Mass Customization, *Proc. 4th International Workshop on Software Product Family Engineering*, pp.282–293, Springer (2001).
[18] Deelstra, S., Sinnema, M. and Bosch, J.: Product derivation in software product families: A case study, *Journal of Systems and Software*, Vol.74, No.2, pp.173–194 (2005).
[19] Pollack, M.: Architecture Refactoring: Improving the Design of Existing Application Architectures, Presentation slides of Microsoft TechEd North America (2009).
[20] Kolb, R., John, I., Muthig D., et al.: Experiences with Product Line Development of Embedded Systems at Testo AG., *Proc. 10th International Software Product Line Conference*, pp.172–181 (2006).
[21] Bourquin, F.: High-Impact Refactoring based on Architecture Violations, *11th European Conference on Software Maintenance and Reengineering*, pp.149–158 (2007).

牧 隆史 （正会員）

1987 年名古屋大学工学部応用物理学科卒業．1989 年名古屋大学大学院工学研究科応用物理学専攻博士前期課程修了．1989 年（株）リコー入社．LSI システム設計，組み込みソフトウェア設計，応用システムの研究開発に従事．2013 年北陸先端大・情報科学研究科博士後期課程修了．博士（情報科学）．日本物理学会会員．

岸 知二 （正会員）

1982 年京都大学・情報工学専攻修了．1982 年 NEC 入社．2002 年北陸先端大・情報科学研究科博士課程修了．博士（情報科学）．現在，早稲田大学経営システム工学科教授．ソフトウェアアーキテクチャ，ソフトウェアプロダクトライン，ソフトウェア設計，設計検証の研究に従事．