ReSET

ソフトウェア進化パターン集

第 1.4 版（2014 年 4 月 16 日）

平成 25 年度 産学戦略的研究フォーラム

ソフトウェア進化技術の実践に関する調査研究

ReSET (Reengineering Software Evolution Technology)

執筆者一覧

丸山 勝久（立命館大学）
沢田 篤史（南山大学）
小林 隆志（東京工業大学）
大森 隆行（立命館大学）
林 晋平（東京工業大学）
飯田 元（奈良先端科学技術大学院大学）
門田 暁人（奈良先端科学技術大学院大学）
吉田 則裕（奈良先端科学技術大学院大学）
角田 雅照（近畿大学）

岩政 幹人（株式会社東芝）
今井 健男（株式会社東芝）
遠藤 侑介（株式会社東芝）
村田 由香里（株式会社東芝）
位野木 万里（東芝ソリューション株式会社）
白石 崇（東芝ソリューション株式会社）
長岡 武志（東芝ソリューション株式会社）
林 千博（株式会社とめ研究所）
吉村 健太郎（株式会社日立製作所）
大島 敬志（株式会社日立製作所）
三部 良太（株式会社日立製作所）
福地 豊（株式会社日立製作所）

1. ソフトウェア進化パターンとは

ソフトウェア進化の法則[1]によると，ソフトウェアは，本質的に出荷後も利用者の要求の変化や利用環境の変化に応じて，継続的に変更され続けなければならない．この法則の科学的根拠に関しては議論の余地はあるものの，ソフトウェア開発に関わる技術者や実務者にとって，ソフトウェアの進化が避けられないという実感は強い．実際，さまざまなシステムが，仕様変更への対応の繰り返しの結果として大規模化複雑化し，保守が困難となる状況に直面している．特に，運用システムのプラットフォームの陳腐化のため，オープンプラットフォームに移行しようとするものの，現状把握のための分析コストが多大に必要になり，移行がスムーズにいかないケースも増えている．

このため，ソフトウェア進化技術に対するソフトウェア開発現場からの期待は大きく，それに関するさまざまな研究が行われている[2]．また，情報システムの移行に関する研究や事例報告も存在する[3]．しかし，残念ながら，それぞれの進化技術が実際のソフトウェア開発においてどのように役に立つのかを明らかにするまでに至っていない．例えば，ソースコードの進化を実現するという目的に対して，数々のプログラム解析技術が活用されていることは分かるが，それらの技術が実際のソフトウェア開発や保守において，どのように役に立つのかについては未整理なままである．

このような状況を打破するためには，ソフトウェア進化活動において，どのような場面で，どのような技術を，どのように適用すればよいのかを示す指針や慣例が必須である．つまり，技術者や実践者にとって有益な指針や慣例に焦点を当て，進化技術を整理し直すことが望まれる．

本パターン集は，このような指針や慣例をソフトウェア進化パターンとして取りまとめたものである．ここでは，以下に示す 5 種類のパターンを提示する．

(a) **ソフトウェアプロダクトラインパターン**
(b) **コードクローンパターン**
(c) **ソフトウェア変更支援パターン**
(d) **プログラム理解支援パターン**
(e) **リファクタリングプロセスパターン**

それぞれのパターンは,大学側メンバおよび企業側メンバで構成されたチームにおいて，それぞれのメンバの持つ問題意識や進化技術に基づく議論を通して収集されたものである．ただし，実際のソフトウェア開発・保守現場における，パターンとしての有効性に関しては未検証であることを述べておく．

ここで，本パターン集で扱うソフトウェア進化，ソフトウェアパターン，ソフトウェア進化パターンに関して，簡単に用語を説明しておく．

ソフトウェア進化(Software Evolution)

ソフトウェア進化とは，一旦出荷されたソフトウェアに対する変更を受け入れる仕組みや活動を指す．通常，個体（生物など）に対して，それ自体が変化することを進化とは呼ばない．進化とは，互いに似ている一群の個体を指す種(species)に対して，世代を越えて発生する現象である．このような観点から，単一のソフトウェアが利用者の要求や利用環境の変化に追従できなくなることをソフトウェアの老化(software aging)と呼び，ソフトウェア進化とは区別するという考え方もある[4]．また，ソフトウェアにおける種をアーキテクチャ記述やプロダクトファミリのようなハイレベルなモデルで捉えることで，そのようなモデルに対する変更を進化と呼びという考え方も存在する[5]．

本パターン集では，進化と老化という 2 つの用語を区別していない．ソフトウェア開発においては，ソースコードの一部だけを書き直したり，入れ替えたりすることで，新たなソフトウェアを構築しているとみなす傾向にある．事実，開発者にとって単一のソフトウェアを変更しているだけでも，利用者から見た機能や振る舞いが変更される（リリース番号が更新される）ことで，世代が違うように見える．このような状況において，ソフトウェア進化の実践という観点からパターンを適用する場合，ソフトウェア進化と老化というレベルで明確に区別する必要はないと考えた．

また，ソフトウェア進化は，要求分析，アーキテクチャ設計，モジュール設計，コーディング，テストなどの活動と密接な関係を持つ．よって，これらの活動に関係するパターンも進化パターンと捉えることは可能である．しかしながら，本パターン集では，このような広義の意味の進化パターンではなく，ソフトウェアリエンジニアリング(software reengineering)活動に特化したものだけを進化パターンと呼ぶことにする．

ソフトウェアリエンジニアリングとは，既存のソフトウェアを作り直す作業を指す[6]．例えば，レガシーシステムのマイグレーション(legacy migration)がある．図 1.1 に示すように，リエンジニアリングは，リバースエンジニアリングとフォーワードエンジニアリングで構成される[7][7]．リバースエンジニアリングとは，ソースコードから設計図や要求仕様を回復する活動を指す．レガシーシステムの保守において，ソフトウェアの仕様書や設計書が利用できない状況や，度重なるソフトウェアの更新によってそれらとプログラムコードの整合性が維持できていない状況は頻繁に発生する．このような状況において，プログラムコードから，より抽象度の高い表現（あるいはモデル）が構築できると，ソフトウェア理解がより容易になる．フォーワードエンジニアリングとは，要求仕様書に基づきプログラムコードを実装する従来のソフトウェア開発作業を指す．リエンジニアリングにおいて，まずリバースエンジニアリングによって，既存のソフトウェアシステムから情報を抽出し（詳細を切り捨て），より抽象度の高いモデルを構築する．次に，この抽象モデルにおいて，ソフトウェアの改良，拡張，修正を適用する．最後に，フォーワードエンジニアリングにより，新しい抽象モデルから新しいシステムを作り出す．

ここで，ソフトウェアリエンジニアリング活動にはリストラクチャリング活動も含まれ

る．リストラクチャリングとは，外部から見た振る舞いを変えずに，同じ抽象レベルの別の表現に変換する活動を指す．特に，既存ソフトウェアの理解性や変更容易性を向上させることを目的とした上で，外部的挙動を保存したままで内部構造を改善する活動をリファクタリングと呼ぶ[8]．リファクタリングでは，大きな設計変更を小さな変換の繰り返しで実現することで挙動の保存を保証するのが特徴である．

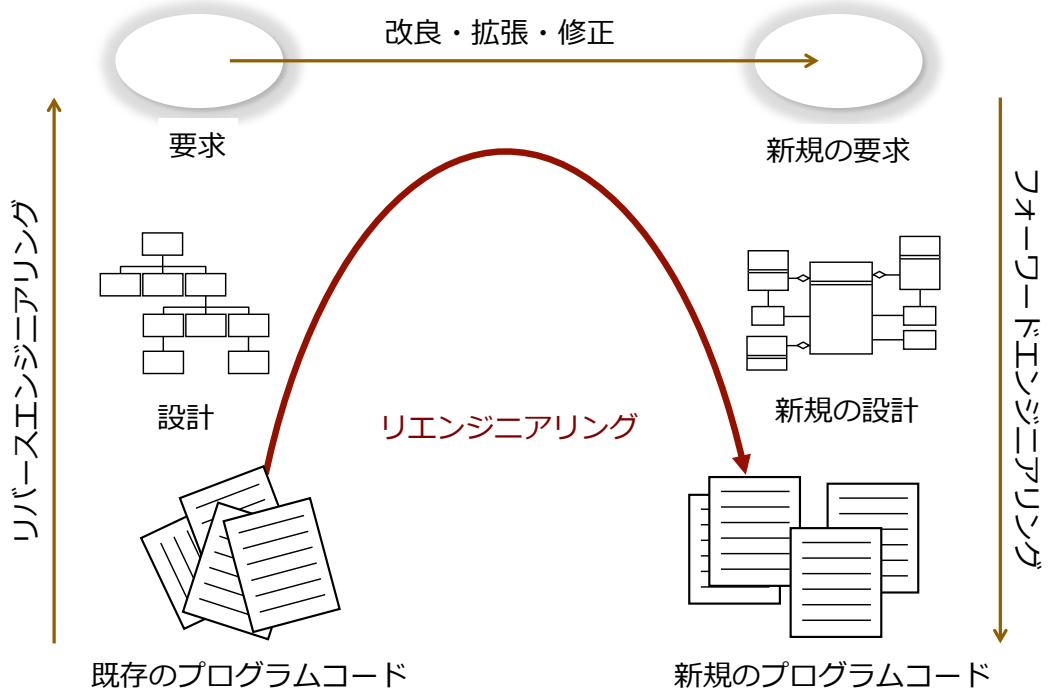

図 1.1：ソフトウェアリエンジニアリング[7]

ソフトウェアパターン(Software Patterns)

ソフトウェアパターンとは，ソフトウェア開発の特定の領域において繰り返し現れる問題とその解決策を目に見える形で体系的に表現したものである[9]．これにより，ソフトウェア開発において蓄積された経験やノウハウ(know-how)を(再)利用できるようになる．

パターンの概念を図 1.2 に示す．通常，パターンごとに，どのような状況（文脈）で，どのような問題に適用可能であり，どのように問題を解決するのか，適用によってどのような結果（利点や欠点）が引き起こされるのか，が記述されている． ただし，パターンとは，特定の状況で発生した個々の問題に対する解法を単に集めたものではなく，状況の共通性に基づき抽象化された問題とその解法を対にして集めたものである．

開発者は，現在直面している状況や問題に合致するパターンを，既存のパターン集合（パターンカタログやリポジトリ）から，フォース(force)に着目して選択する．フォースとは，パターンを適用する際に考慮すべき要件，制約，特性（利益や不利益）を指す．また，パ

ターン選択においては，パターンの構造や振る舞いといった機能的特性(functional property)だけでなく，性能，信頼性，保守性などの非機能的特性(non-functional property)を考慮することも重要である．

ここで，実際の開発ソフトウェア，開発プロセス，開発組織にパターンを適用する際には，具体的状況や問題に応じて，選択したパターンを具体化（修正や拡張）することが必要となる．つまり，パターンはそのまま組み込む部品ではないことを，開発者は強く意識する必要がある．また，過度にパターンを使用しないことも重要である．状況や問題に応じて適切なパターンを選択および適用しない限り，品質の高いソフトウェアを得ることはできない．パターンの選択や適用においては，フォースや結果を十分に検討してソフトウェア全体に与える影響を十分に把握する必要がある．また，検討結果によっては，パターンの適用を避けることも重要である．

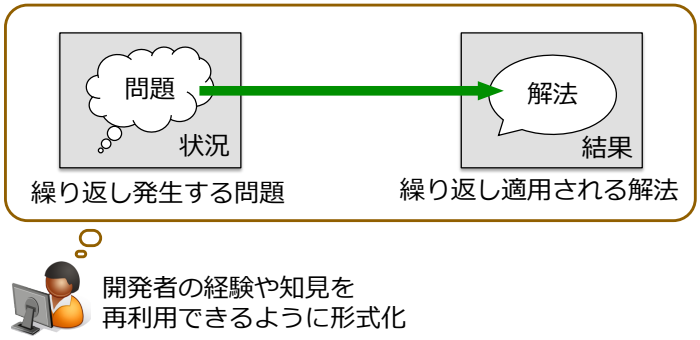

図 1.2: ソフトウェアパターン

ソフトウェア進化パターン(Software Evolution Patterns)

ソフトウェア進化を実践する上での指針や慣例がソフトウェア進化パターンである．Demeyer らは，ソフトウェアリエンジニアリング活動における問題とその解決方法をオブジェクト指向リエンジニアリングパターン（本パターン集では，OORP と呼ぶ）としてまとめている[7]．また，リファクタリング[8]におけるコード変換は，それを適用する場面（文脈や条件）が想定されており，設計やコードに関する問題を解決するために実施される．よって，リファクタリングは，既存ソフトウェアの改変を支援する進化パターンと見なすことができる．このようなパターンを開発者や保守者が知識として備えることで，過去のソフトウェア資産を有効に活用（改善や移植）することが見込める．

本パターン集では，Demeyer らの OORP を参考に，それらを補完する進化パターンの収集を試みた結果を示す．特に，ソフトウェアプロダクトライン，ソフトウェア変更支援における履歴やマイクロブログの活用などは，OORP の出版（2002 年）以降に急速に普及し

た概念を扱っている．さらに，コードクローン，リファクタリング，プログラム理解や変更に関する近年の技術進歩という観点から，進化パターンが追加されている．

最後に，本パターン集における進化パターンの記述形式を表 1.1 に示す．

表 1.1： パターンの記述形式

項目	説明
名前(Name)	パターンの内容を表す語 パターンを適用する目的(intent)を含む
問題(Problem)	パターンが対象とする問題 問題が発生する状況(context)や条件(condition)を含む
解法(Solution)	問題を解決するための方針や手順
議論(Discussion)	パターンの問題や解法に関する議論点
トレードオフ(Tradeoffs)	パターンの適用することで得られる良い点(pros) パターンを適用することにより発生する悪い点(cons) パターンの適用に関する困難(difficulties)
根拠(Rationale)	論理的根拠を示す資料や文献
実例(Example)	パターンが適用されている例
よく知られた使い方(Known Uses)	パターンが利用されている/説明されている資料
関連するパターン (Related patterns)	他に参考にするとよいパターン

2. ソフトウェアプロダクトラインパターン

ソフトウェアプロダクトライン(SPL: Software Product Line)とは，共通の特性を持ち，特定の市場やミッションのために，共通のコア資産から規定された方法で作られる，ソフトウェア中心システムの集合を指す[10][11].

ここで，ソフトウェアプロダクトラインパターンを紹介する前に，簡単に用語を整理しておく[12]．**コア資産**(core asset)とは，プロダクトラインの基盤を形成する成果物(artifact)のことを指す．コア資産は，複数の製品系列に対して適用可能なように，あらかじめ共通部分と可変部分を織り込んで構築される．**可変ポイント**(variation point)とは，コア資産において，特定の製品系列に固有な部分とすべての製品系列に共通な部分との分岐点を指す．**バリアント**(variant)とは，可変ポイントに対する選択肢を指す．**共通可変分析**(commonality and variability analysis)とは，特定の製品系列の固有部分とすべての製品系列に共通の部分を分析し決定することを指す.

また，**フィーチャー**(feature)とは，ソフトウェアシステムにおける，突出したまたは他と明確に区別でき目に見える側面,品質,特徴のことを指す[13].この定義は,The American Heritage Dictionary からの引用である．補足すると，フィーチャーとは，プロダクトが提供するサービス，操作性や性能などの非機能要求などのそのプロダクトの代表的な特性のことである．**フィーチャーモデリング**(feature modeling)とは，プロダクトライン型開発において，ドメインの共通／可変部分を抽出するための代表的な手法を指す[14]．フィーチャーモデリングと他のさまざまなモデリング技術を融合させ，プロダクトラインの構築，運用，進化への研究が取り組まれている．**フィーチャーマトリクス**(feature matrix)とは，フィーチャーとコア資産の要素との対応関係をトレースしやすくするための情報をマトリクス形式で記述した仕様のことを指す.

ここでは，プロダクトライン開発に特化した 5 つのパターンを紹介する.

パターン 2.1　プロダクトラインに移行すべきかを判断しよう
パターン 2.2　コア資産を浄化しよう
パターン 2.3　「石ころ」に投資してはいけない
パターン 2.4　レガシーの現状を分析しよう
パターン 2.5　コア資産への再生シナリオを決定しよう

パターン 2.1

プロダクトラインに移行すべきかを判断しよう
Launch Product Line

開発プロセスをプロダクトラインに移行すべきかどうかを，製品ライフサイクルに基づいて判断することで，投資対効果を高める．

問題

- 開発プロセスをプロダクトラインに移行すべきかどうかの意思決定ができない．

解法

- 自社製品が製品ライフサイクル[15]のどのステージにいるかを分析し，開発アプローチをプロダクトラインに変換すべき/すべきでないかを判断すべきである．
- 図 2.1 に示すように，製品ライフサイクルの段階は導入期(Introduction)，成長期(Growth)，成熟期(Maturity)，衰退期(Decline)の 4 つがある．導入期は，新製品を開発し，消費者に宣伝・紹介している段階である．成長期は，競合他社との差別化のために新機能を開発し，先進的な消費者に販売している段階である．成熟期は，製品の開発コストを削減し，多くの消費者に販売している段階である．衰退期は，最小限のプロモーションにより，販売を継続している段階である．
- プロダクトラインへの変換は成長期から成熟期への移行時期が望ましい．なぜなら，成熟期では，消費者ニーズに応じた機能を低コストで提供することが重要であり，コア資産と消費者個別機能を管理しながら開発するプロダクトラインが適しているためである．成長期と成熟器の比較を表 2.1 に示す．

トレードオフ

-利点

- 製品ライフサイクルの段階に応じて開発アプローチを変更することで，投資対効果を向上できる．

-欠点

- レガシー資産をコア資産に変換する技術は増えつつあるが，変換作業のリスクは小さくない．変換作業が失敗した場合，競争力が低下してしまう．

関連するパターン

コア資産に移行する場合には，**コア資産は今すぐリファクタリングしよう (パターン 6.1)** が利用できる．

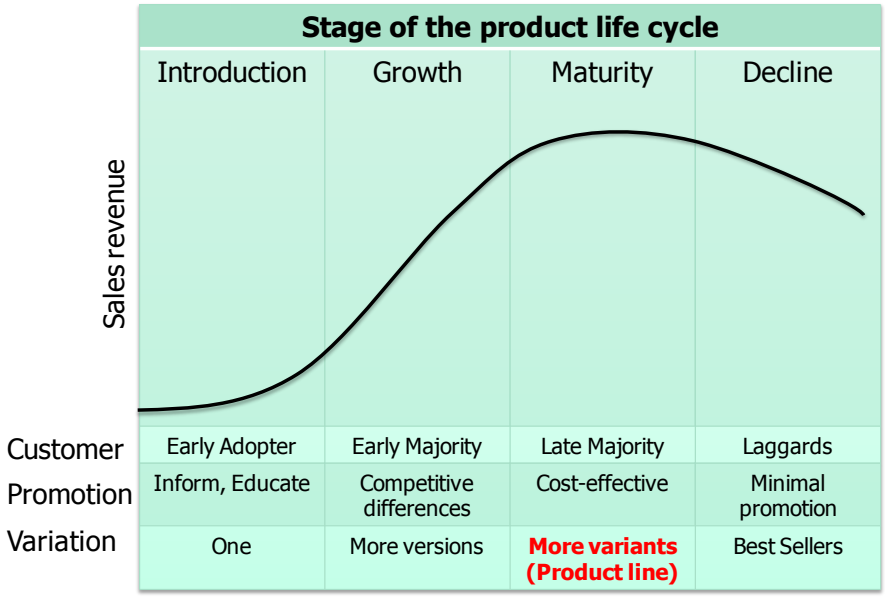

図 2.1: 製品のライフサイクルモデル

表 2.1: 成長期と成熟期の比較

	Growth	**Maturity**
Product competence	New features	Cost & time to market
Product roadmap	Short term	Long term
Development driver	Customers & competitors	Roadmap
Requirements	Drifting	Stable
Main development phase	Application engineering	Domain engineering
Derivation style	Clone & own	Binding variability (configuration)
Number of variations at same time	Small (<10)	Large

パターン 2.2

コア資産を浄化しよう
Detox Core Assets

肥大化したコア資産をスリムにして価値を高める.

問題

- プロダクトライン型の開発（コア資産の開発と進化）が 3 年以上などの長期間に渡り，技術，市場，組織の変化に応じて，コア資産を拡張し続けた結果，コア資産が肥大化し，コア資産内に製品開発で利用されない要素が存在するなどして，却って保守拡張のコストが増大している.
- プロダクトライン型の製品開発では，コア資産の可変ポイントから可変部分を選択または開発するが，コア資産が複雑化/肥大化して可変ポイントや可変部分が不明確になり，これらの選択が困難になっている.
- 複雑化/肥大化したままコア資産を保守拡張すると，一カ所の拡張の影響が多大な範囲に及ぶことや，そもそも影響範囲が把握できないまま拡張することになり，品質保証のためには膨大なテストを実行する必要があるなど，保守コストが増加している.

解法

- 不必要で製品開発に問題をもたらすコア資産を浄化(detox)し，製品開発時の可変ポイントや可変部分の選択をしやすくし，保守コストを下げる.
- 図 2.2 にコア資産を浄化する際の 3 つの方法を示す. いま，カバー率（コア資産が十分な要素を備えている/いない）と対応度（コア資産の要素間の整合がとれている/いない）の 2 つの指標をそれぞれ 2 値で表す. このとき，コア資産は次に示す A～D の 4 つのタイプに分けることができる.

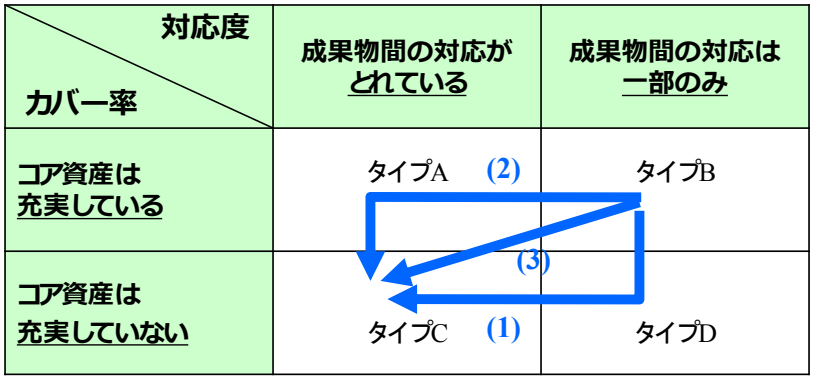

図 2.2: コア資産を浄化する 3 つの手順

タイプＡ：コア資産が十分な要素を備えている，かつ，コア資産の要素間の整合性がとれている

タイプＢ：コア資産が十分な要素を備えている，かつ，コア資産の要素間は整合性がとれていない

タイプＣ：コア資産が十分な要素を備えていない，かつ，コア資産の要素間は整合性がとれている

タイプＤ：コア資産が十分な要素を備えていない，かつ，コア資産の要素間は整合性がとれていない

- 問題で示したコア資産は，コア資産は肥大化するほどの十分な要素があるが，成果物間は整合が十分ではないため，タイプＢに分類できる．コア資産をDetoxするとは，不要な要素を減らした上で，整合がとれた状態にすることである．そのようなコア資産は，タイプＣに分類できる．
- 肥大化したコア資産，すなわちタイプＢのコア資産から，適切な状態であるタイプＣに状態を変化させるには，図 2.2に示す（１）～（３）のパスが考えられる．
  （１） コアアセットの削除 → 整理統合
  （２） コアアセットの整理統合 → 削除
  （３） 整理統合されたコアアセットの削除
- コア資産内は要素が複雑に結合している場合が多いので，不要な要素を単純に削除することは難しい．よって，コア資産の状態を把握するためにも，（２）のコアアセットを整理統合したのち，不要な要素を削除するパスを優先することがリスク回避のために有効である．

トレードオフ

-利点

- 長年活用されてきた複雑なコア資産を容易にスリム化することは困難であるので，整理統合の作業を実施して，コア資産の性質を把握することによって，安全に不要な要素を特定することが可能である．

-欠点

- 過去の製品開発で活用された要素がコア資産の浄化により削除される．これによって，既に納入された製品の保守管理が不可能になるリスクがある．

関連するパターン

コア資産の品質が不適切な場合には，コア資産を浄化しても効果が得られないケースがあるので，「石ころ」に投資してはいけない (パターン 2.3)を参照のこと．

パターン 2.3

「石ころ」に投資してはいけない
Don’t Invest in Stone

既存のコア資産の中から投資リスクが高いものを除外する.

問題

- ある組織内では，複数のプロダクトラインが維持管理されている．例を表 2.2 に示す．この表において，投資順位とは投資申請額の順位を指す．投資回収時期とは，ROI がプラスに転じるまでの想定期間を指す．ROI 順位とは，回収額の想定順位を指す．また，タイプ A～D とは，**コア資産を浄化しよう (パターン 2.2)**にて示したコア資産のタイプのことを指す.
- 事業計画において，既存のプロダクトラインを拡張して，今後の市場，技術，環境の変化に対応させて，事業を拡大していくことを望んでいる．しかし，投資額には限度があるため，どのプロダクトラインに投資すべきか，またはすべきでないかを決めなければならない.
- それぞれのプロダクトラインを持つビジネスユニットから，投資回収計画が出されたが，ROI 順位を投資判断の要素にすることが妥当であるのかどうかを判断できない.

表 2.2: 投資対象のプロダクトライン

ID	ドメイン	タイプ	投資順位	投資回収時期	ROI順位
PL1	製品情報トレーサビリティ	タイプC	3	3年	3
PL2	大規模情報編集支援	タイプB	1	3年	1
PL3	事務処理業務支援	タイプA	2	3年	2

解法

- ビジネスユニットが提出した ROI 順位よりも，コア資産の整合性の度合いに着目して，投資対象を判断すべきである.
- 各プロダクトラインのコア資産が肥大化して整合がとれていない場合には，コア資産の拡張のための投資を実行しても，肥大化した部分の整理統合にコストがかかり，想定している投資効果が得られないリスクが高まる．よって，このようなプロダクトラインに投資してはいけない.

トレードオフ

–利点

- 肥大化し整合性が保てていないコア資産は，拡張に着手しても，肥大化して複雑な部分の解析に時間がかかり，市場，技術，環境の変化に俊敏に対応することが困難になるため，こうしたプロダクトラインへの投資は，投資効果が期待できない．こうしたコア資産へ投資しない決断をすることは無駄な投資の防止につながる．

–欠点

- これまで実績を出してきたプロダクトラインへの投資を凍結することによって，対象製品の価値が低下し，新しい顧客獲得や既存顧客の囲い込みなどの機会を損失してしまう．

関連するパターン

SPL コア資産パターン全般を適用する前に，本パターンの適用を検討すべきである．

パターン 2.4

レガシーの現状を分析しよう
Analyze Legacy Software Assets

長期に渡って改造を繰り返してきたレガシーソフトウェア資産を再生し高品質なコア資産を構築したい．

問題

- 長期に渡って改造を繰り返してきたレガシーソフトウェア資産の中には，コア資産としてふさわしくない低品質な成果物が含まれている．
- 品質の良くない成果物をそのままコア資産に組み入れるとコア資産全体の品質が低下してしまう．

解法

- ソースコード解析技術などを利用し，ソースコードやドキュメントなどレガシーシステムの成果物の現状を客観的に分析・評価する．表 2.3 に現状分析の進め方を示す．

トレードオフ

–利点

- レガシー資産の成果物の現状を分析しておくことで，品質の低い成果物をそのままコア資産として組み込んでしまうリスクが低下する．

-欠点

- 分析対象の成果物が大量にある場合には，分析を効率的に行わないと分析自体のコストが大きくなりすぎてしまう．

表 2.3：現状分析の進め方：各種メトリクスと計測結果による分析

No	計測の目的	計測対象	メトリクス	計測結果により分析する内容
1	レガシーソフトウェア資産に含まれる成果物が充実し、整合がとれている	ソースコード	現行成果物とその属性（ファイル数、ファイルサイズ、LOC他）	ファイル数、サイズ等が妥当であるかどうか
2		ドキュメント	成果物のカバー率	レガシーソフトウェア資産に含まれる 成果物がどれほど充実しているか
3		ドキュメント	成果物の対応度 （トレーサビリティマトリクスの記述可能性／記述結果の妥当性）	成果物間の対応関係の整合がとれており品質が安定しているか
4	バリエーションを派生させやすい	ドキュメント及びソースコード	フィーチャーマトリクスによる可視化可能性	製品ロードマップを考慮した構造になっているか
5		ドキュメント及びソースコード	フィーチャーマトリクスによる可視化結果の妥当性	製品ロードマップに対して派生させやすい構造になっているか
6	変更・改善の影響範囲が局所化されている	ソースコード	クローン数	コピー／ペースト繰り返し冗長なソースコードになっていないか？ クローン部分への修正の影響が及ぶ範囲はどれほどか？
7		ソースコード	モジュール間依存度	想定したアーキテクチャに沿って整然とした構造になっているか？ 特定モジュールへの修正の影響が及ぶ範囲はどれほどか？

パターン 2.5

コア資産への再生シナリオを決定しよう
Decide Reengineering Scenarios

レガシーソフトウェア資産をコア資産として再生するためのシナリオを決定したい．

問題

- どのような方法でコア資産として再生すればよいかわからない．
- コア資産として再生するためのシナリオとしてどのような選択肢があるかわからない．
- 選択肢が提示されても，その中からどのシナリオを選択すれば良いかわからない．

解法

- レガシー資産をコア資産として再生するための代表的なシナリオを表 2.4 に示す．この表に示すシナリオからどのシナリオを選択するかについては，レガシー現状分析の結果や再生コスト，リスク，もたらされる効果を考慮して決定する．

トレードオフ

-利点

- 既存の成果物の現状態に応じて適切な再生シナリオを選択することができる．

-欠点

市場の成熟度，トレンドの変化などによっては，コスト，リスクの高いシナリオをとった場合に投資を回収できない恐れがある．

表 2.4: コア資産として再生するためのシナリオ[16]

シナリオ	アクティビティ	コスト	リスク	もたらされる効果
シナリオ1： 現状の可視化	以下の成果物の作成および調査。 ・コア資産一覧、 フィーチャーモデル作成 ・フィーチャーとソースコード対応 ・現状システムの評価 （サイズ、複雑さ、バグ残存状況）	L	L	・レガシー資産の現状の状態を把握できる ・シナリオ2～4を選択した場合の設計材料に利用できる
シナリオ2： 軽微な改善	既存成果物の軽微なリファクタリング。	L	M	・ソースコードへのコメントの挿入や、不要ソースコードの削除によりソースコードの可読性が高まる
シナリオ3： 可変ポイントの独立化	現状の仕組みを維持したまま、今後の変更・拡張が想定される部分をコンポーネント化。	M	H	・可変ポイントを独立化することで今後の製品開発およびメンテナンスの効率が向上する
シナリオ4： アーキテクチャの再構成	共通／可変部分を分離独立化しフレームワークを構築。	H	M	・現状分析によって洗い出した課題を解決し、本来あるべき姿へと近づけることができる。

L:Low、M :Middle、H:High

3. コードクローンパターン

コードクローン(code clone)とは，ソースコードのコピー&ペーストによって作られた（あるいは，作られたように見える）コードの断片を指す[17]．これは，重複コード(duplicated code)とも呼ばれる．一般的に，コードクローンの存在がソフトウェア保守を難しくするといわれており，Fowler や Beck もリファクタリングにおけるコードの悪臭のひとつであると指摘している[8]．

このような背景を受け，重複コードを発見するための 2 つのパターンが OORP[7]の 8 章に紹介されている．8.1 では，コードを機械的に比較しよう / Compare Code Mechanically (OORP パターン 8.1) が紹介されている．大規模なプログラムのコードを手動で調べ上げ，コードの重複を発見することは現実的でない．このパターンでは，ソースコードのテキストを行ベースで比較するスクリプトを利用することを奨励している．次に，8.2 では，散布図（ドットプロット）を用いてコードを視覚化しよう / Visualize Code as Dotplots (OORP パターン 8.2) が紹介されている．散布図とは，2 つのソースコードファイルの各行を X 軸と Y 軸に並べ，重複する行の位置にドットを表示した行列を指す．行列内に現れるドットの形状に応じて，4 つの解釈が提案されている．

ここでは，OORP の重複コード発見パターンに追加する形で，コードクローンに関する 5 つのパターンを紹介する．

パターン 3.1　着目すべき特徴を絞って散布図を分析しよう
パターン 3.2　類似したファイル群を抽出しよう
パターン 3.3　プロダクト中の類似部分を探そう
パターン 3.4　プロダクト間にまたがる類似部分を探そう
パターン 3.5　重複部分の一貫性を維持しよう

パターン 3.1

着目すべき特徴を絞って散布図を分析しよう
Focus on Particular Dotplots

散布図を解釈し，重複コードの存在によって生じるソースコードの問題点やその解決策を分析したい．

問題

- 散布図を使って重複コードを分析する際に散布図のどの部分に着目すればいいのかわ

からない.

- 分析者がコードクローンの専門家でない場合，散布図を見てもソースコードのどの部分に問題があるのか判断が難しい.
- 重複コードがどのような経緯で混入したか，その重複コードに対してどのような処置をすればよいのか，専門的な知識なしには分析ができない.

解法

- 散布図を分析する際，着目すべき部分の特徴として，次に示す 3 つの戦略を用いる.
  戦略 1：対角線に着目（図 3.1）
  戦略 2：分断された対角線に着目（図 3.2）
  戦略 3：破線・点線に着目（図 3.3）

トレードオフ

-利点

- どういう経緯で発生した重複コードであるか，リファクタリング適用の難しさ等について，コードクローンに関する専門知識がなくても分析することができる.

-欠点

- パターンを特定するための定量的な尺度がないため，分析結果が分析者の主観に依存する可能性がある.
- ソースコードが大規模である場合，描画領域のサイズの制約によって散布図の詳細を表示しきれず，特徴を発見できない可能性がある.

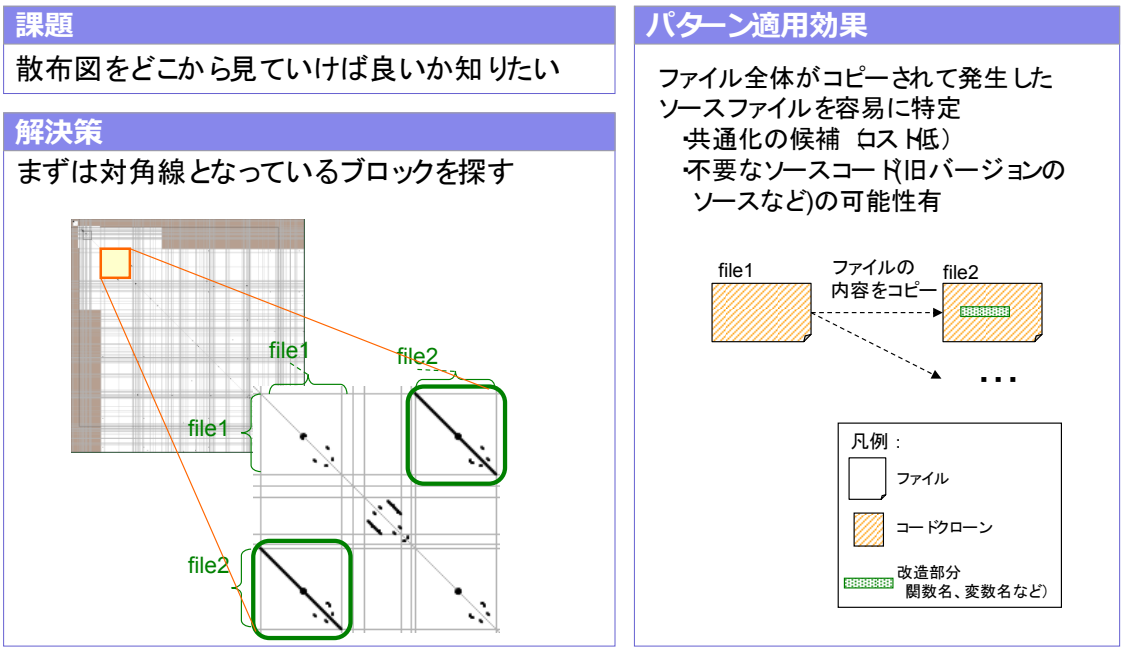

図 3.1：対角線に着目する（戦略 1）

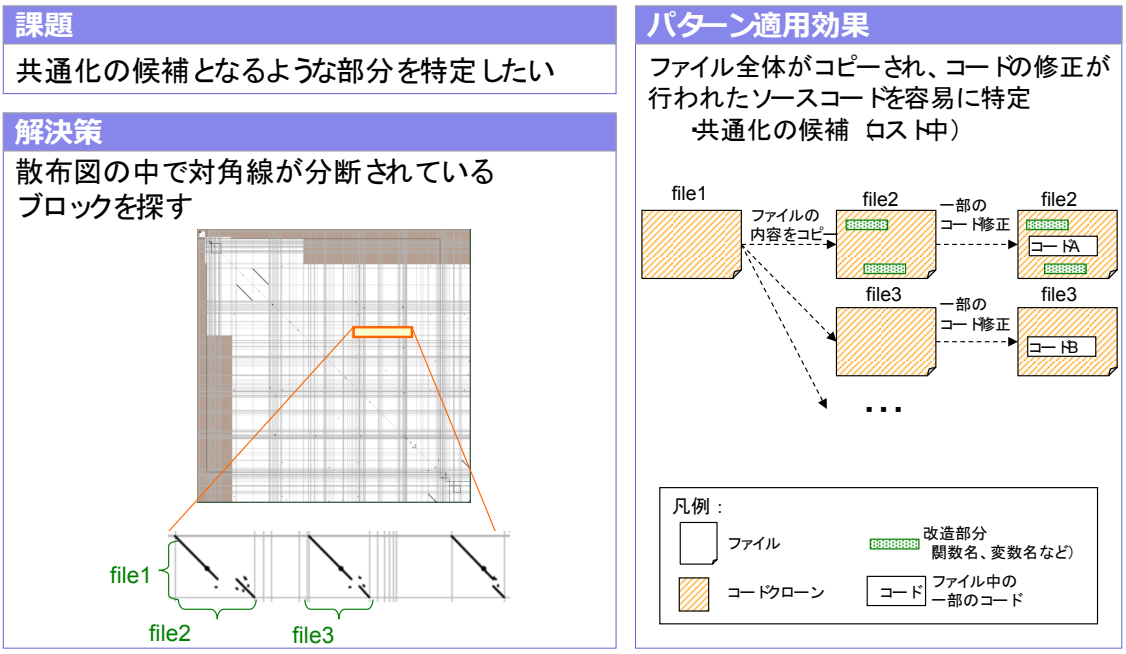

図 3.2：分断された対角線に着目する（戦略 2）

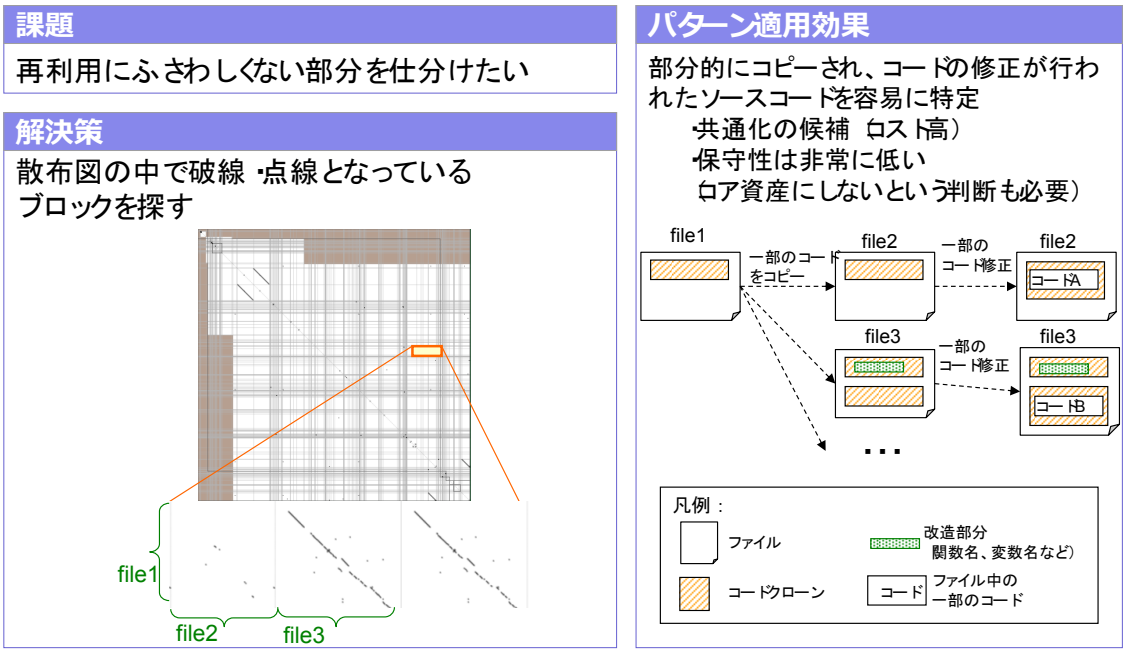

図 3.3：破線・点線に着目する（戦略 3）

関係するパターン

重複コードを，散布図を利用して解釈する方法は，OORP の散布図を用いてコードを視覚化しよう (OORP パターン 8.2) で示されている．本パターンは，散布図の解釈方法を詳細に示したものである．

パターン 3.2

類似したファイル群を抽出しよう
Extract Similar File Groups

大規模なソースコードの重複コードを，散布図を使って分析したい.

問題

- 分析対象のコードが大規模である場合，散布図上に詳細を表示できない.
- 分析対象ソースコードが大規模である場合, 図 3.4 のように散布図の詳細部分を表示できず,「着目すべき特徴を絞って散布図を分析する」で示した特徴を散布図から発見することができない．この問題については，OORP の散布図を用いてコードを視覚化しよう (OORP パターン 8.2) でも，*"The screen size limits the amount of information that can be visualized"* と指摘されている.

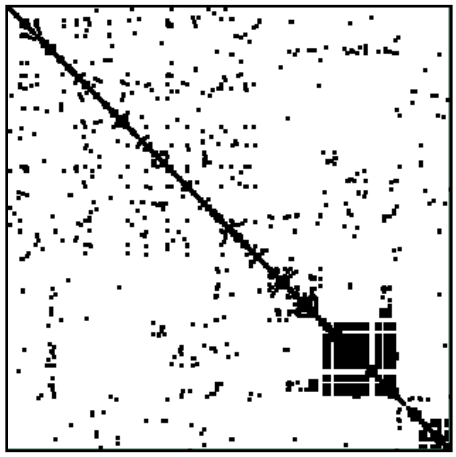

図 3.4: 解析対象ソースコードの規模が大きい場合の散布図の例
（ある OSS のソースコードの散布図）

解法

- 類似度の高いファイル群を機械的に抽出する.
- 分析対象ソースコード全体を散布図として表示するのではなく, 詳細な分析が必要なファイル群に絞込みを行った上で分析を行う.
- 互いに重複箇所の多いようなファイル群を抽出することで, 互いに重複箇所を持たないようなファイルを除外することができる.
- 絞込みを行った上で「着目すべき特徴を絞って散布図を分析する」を適用することで, 散布図の特徴に着目した分析が可能となる．図 3.5 は，ファイルの類似度に基づいて類

似ファイル群を抽出し，「着目すべき特徴を絞って散布図を分析する」を適用する流れを示したものである．

- コードクローン分析の目的によって抽出の粒度をディレクトリ，メソッドなどとすることも可能である．サブシステムレベルでの重複コードを特定したい場合には抽出の粒度をディレクトリにする，などが考えられる．

トレードオフ

-利点

- 散布図として表示するファイル数が少なくなるため，コードクローンの分布状況を詳細に表示することができ，散布図上での分析が容易になる．
- 散布図描画ツールのスケーラビリティのハードルが低下する．

-欠点

- 類似ファイル群のみの可視化になるため全体を俯瞰した分析ができない．
- 類似度の定義や類似度の閾値によって抽出される類似ファイル群が異なるため，コードクローン分析の目的によってこれらを決定する必要がある．

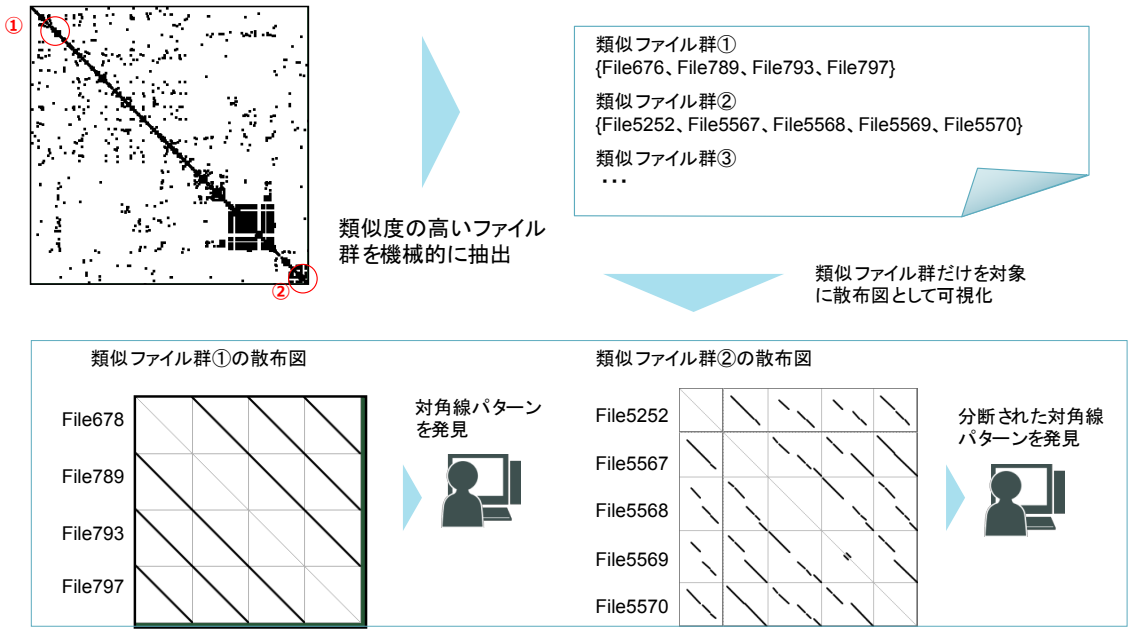

図 3.5　ファイルの類似度に基づいた絞込みの流れ

パターン 3.3

プロダクト中の類似部分を探そう
Identify Similar Parts Within a Product

レガシーソフトウェアの大規模な改修をする計画がある．その際，改修にあわせ共通化すべき部分を特定し，集約したい．改修にあわせることで，網羅的なテストを同時にできるため，共通化すべき部分の集約が受け入れられやすい．

問題

- 仕様書や設計書に記載された情報だけでは，類似性した部分を判断しづらい．また，それら文書の一部が欠如している，もしくは古いという問題がある場合もある．

解法

- ディレクトリなどの単位でソースコードの類似性を計測する．
- ソースコードを分析することで，仕様書や設計書の一部が欠落している，もしくは古いという問題があっても回避することができる．
- 類似性の計測には，CCFinder などのクローン検出ツールを使うことができる．最小一致トークン数は 100 トークンなど検出ツールのデフォルト値よりは大きめに設定するほうが，大規模なソースコードを対象とした場合であっても高速に検出できる．

トレードオフ

−利点

- 仕様書や設計書に問題があっても，共通化すべき部分を判断できる．

−欠点

- ソースコードの類似性のみで共通化を判断すると，将来の保守において独立して大規模な改修が行われるなどして，理解容易性が下がる可能性がある．

関連するパターン

類似部分がプロダクト間にまたがる場合は，**プロダクト間にまたがる類似部分を探そう** (**パターン 3.4**) の適用を検討する必要がある．また， 重複コードを発見するためには，OORP の**コードを機械的に比較しよう** (OORP パターン 8.1) と**散布図を用いてコードを視覚化しよう** (OORP パターン 8.2) が利用できる．発見した類似部分は，**重複部分の一貫性を維持しよう** (**パターン 3.5**) で管理すべきである．

パターン 3.4

プロダクト間にまたがる類似部分を探そう
Identify Similar Parts Between Products

発注先が計画通りの流用を行っているか確認したい.

問題

- 別プロジェクトのソースコードを流用した開発を行っており，発注額の見積りに流用率を用いている．流用しない計画になっているにもかかわらず，ほとんど流用している場合がある．また，流用する計画になっていないにもかかわらず，流用後にほとんど改修している場合がある.
- 別プロジェクトのソースコードをどの程度流用しているかを，週報や LOC 等の規模メトリクスで判断することは難しい.

解法

- 流用元のソースコードと流用先のソースコードの類似性を計測する．類似性の計測には，CCFinder などのクローン検出ツールを使うことができる.
- それぞれのソースコード内のみに存在するコードクローンは検出する必要がないため，検出しないようにオプションを設定し，検出速度を向上させる.
- 最小一致トークン数は 100 トークンなど検出ツールのデフォルト値よりは大きめに設定するほうが，大規模なソースコードを対象とした場合であっても高速に検出できる.

トレードオフ

−利点

- 週報や規模メトリクスでは判断できない流用を特定できる．流用部分についても特定できる.

−欠点

- ドメインに特化したコードクローン（データベース関連処理）が多く含まれる場合，それらコードクローンを流用と判断するおそれがある.

関連するパターン

類似部分がひとつのプロダクトに閉じているときは，**プロダクト中の類似部分を探そう (パターン 3.3)** の適用を検討する必要がある．また， 重複コードを発見するためには，OORP の**コードを機械的に比較しよう** (OORP パターン 8.1) と**散布図を用いてコードを視覚化しよう** (OORP パターン 8.2) が利用できる．発見した類似部分は，**重複部分の一**

貫性を維持しよう (パターン 3.5) で管理すべきである.

パターン 3.5

重複部分の一貫性を維持しよう
Keep the Consistency of Duplicated Code

水平展開（類似部分の同時修正）が必要な部分を特定したい.

問題

- ソースコードが大規模になると，水平展開のコストが大きく，水平展開漏れが起きるおそれがある.

解法

- 改修を行ったコード片と保守対象のソースコード間のコードクローンを検出する.
- CCFinder などのクローン検出ツールを使う場合は, 最小一致トークン数は 15 トークンなど検出ツールのデフォルト値よりは小さめに設定する.
- Visual Studio Premium/Ultimate を用いている場合は，コードクローン検索機能を用いることができる.
- grep などの文字列検索を合わせて用いることで, 水平展開漏れを減らすことができる.

トレードオフ

−利点

- 仕様書や設計書に基づいた横展開では見落としてしまう場合を防ぐことができる.

−欠点

- 必ずしも横展開が必要なコードが類似しているとは限らないため, 過去に同時修正した部分の記録及び分析も必要である.

関連するパターン

類似部分の発見には，コードクローンパターンを利用できる．同時修正箇所が必ずしも類似しているとは限らないため，変更履歴を記録しよう (パターン 4.1) や 活動履歴を記録しよう (パターン 4.2) の適用も検討する価値がある.

4. ソフトウェア変更支援パターン

ソフトウェア進化を実現するために，ソフトウェア変更は必須である．このような観点から，OORP[7]の 7 章では，システムの移行戦略に関する 12 個のパターンが紹介されている(**OORP パターン 7.1～7.12**)．さらに，9 章と 10 章では，設計変更（責任の再分配と条件から多態性への変換）に関するパターンが紹介されている(**OORP パターン 9.1～9.3, 10.1～10.6**)．これらは，リエンジニアリングにおける単一のソフトウェア変更を扱ったパターンである．

これに対して，ソフトウェア進化において，変更は繰り返し，かつ，継続的に発生する．このため，ソフトウェア進化を想定した環境では，ソフトウェア構成管理システム(software configuration management system)や版管理システム(version control system)の利用が一般的である[18]．また，IDE(Integrated Development Environment)や継続的インテグレーション(CI: Continuous Integration)[19]の普及も進んでおり，ソフトウェア変更を積極的に活用する技術が登場している．

このような状況を受け，ここでは， ソフトウェア変更を支援する 5 つのパターンを紹介する．

パターン 4.1　変更履歴を記録しよう
パターン 4.2　活動履歴を記録しよう
パターン 4.3　粒度を調整しよう
パターン 4.4　変更から意味的差分を抽出しよう
パターン 4.5　変更作業の手順書を作成しよう

パターン 4.1

変更履歴を記録しよう
Record Change Histories

変更履歴を記録することで，変更すべき箇所を知りたい．

問題

- ソフトウェアは複雑な依存性を内包しているため，一カ所を変更することで，他の多くの場所にも変更が必要となることが多い．
- ソフトウェアモジュール間の依存性は，詳細に解析をすることでそのほとんどを発見することは可能である．しかし，詳細な解析はコストが高く，また依存関係は膨大な量で

あるため，依存解析のみで変更が必要となる個所を特定することは困難である．

解法

- 変更履歴(change history)を記録・解析する．
- 変更の度に，ある目的のために変更したモジュール群の履歴を保存する．変更履歴を記録・解析することで，一度に変更するべきモジュール集合を特定することができる．
- 変更履歴には，依存解析の結果必要と判断された変更が残っている．この情報を蓄積，解析することによって，依存解析をせずに一度に変更するべき変更箇所を推薦することが可能となる．

トレードオフ

-利点

- 詳細な依存解析が削減できる．
- 一般的な構成管理ツールのログが利用できるため，過去の開発データを活用できる．

-欠点

- 一定の蓄積データがないと推薦できない．
- 蓄積されているデータに間違い（変更漏れ）があると正しく推薦できない．

関連するパターン

本パターンは，過去から学ぼう / Learn from the Past (OORP パターン 5.5) を補うものである．頻繁に変更が行われるプロジェクト，もしくは長期間の記録があり，変更候補が多数提示される場合には，活動履歴を記録しよう/ Record Interaction Histories (パターン 4.2) によって候補の順位付けを行うことができる．

パターン 4.2

活動履歴を記録しよう
Record Interaction Histories

活動履歴を記録することで，変更すべき箇所を知りたい．

問題

- 変更履歴を記録・解析する手法によって，詳細な依存解析をせずに必要な変更を推薦することが可能となる．しかし，変更履歴に含まれる情報は変更した結果のみであり，類似したモジュール集合に起こる変更を弁別することが難しい．

解法

- 活動記録(interaction history)を記録する.
- 変更履歴だけでなく，開発中にどのように活動（成果物の参照・変更）したかを記録する.
- 最終的な変更結果だけでなく，どのようなファイルを参照し，どのような順で変更を行ったかを区別することで，より精度の高い推薦が可能となる.

トレードオフ

-利点

- 多数の変更候補がある場合に，状況に応じた適切な推薦が可能となる.

-欠点

- 一般的な構成管理ツール以外に，特別な環境が必要となる．一定の蓄積データがないと推薦できない．蓄積されているデータに間違い（変更漏れ）があると正しく推薦できない.

パターン 4.3

粒度を調整しよう
Manage Granularity

ソフトウェア保守の際には，過去に行った変更の正しさを検証したり，また同チームの他の開発者が行った変更を分析したりする．こういった変更の利活用に役立つよう，変更の記録を行っておきたい.

問題

- ソフトウェア変更の際に，異なる意味の修正が同時に行われていたり，意味的に強い関わりのある修正が複数の別の変更として登録されていたりすると，理解の妨げとなる.
- また，計算機による変更の推薦を効果的に行うためには，関連性のあるモジュールを一度に修正しているような変更履歴が保持されている必要がある．そのためには，複数の意図の変更を混在させて行ったり，細切れに行ったりせず，存在していた関連性を正しく維持したまま変更を構成することが望ましい.

解法

- 適切な粒度で変更を記録する.
- 意味的に強いつながりのある修正を，他の意味の修正を含まないようにひとつの変更と

して変更履歴に登録する. バグや追加機能候補等を管理する問題管理システムを利用している場合, 問題管理システム上の票の粒度を適切に調整し, すべての変更を票と関連づけて登録することにより, 変更の粒度を調整することができる.

トレードオフ

-利点

- 意味的に関連のある修正をひとまとめに分析することが出来るため, その理解が容易となる. また, 変更の再利用・取り消しも容易となる.

-欠点

- 開発者はしばしば, 複数の変更を一度に登録したり, また変更漏れを起こしたりすることがすでに知られており, 適切な変更となるためには開発者に注意が必要である. また, 誤った粒度での変更を記録し直すこともしばしば必要となる.

-困難

- 互いに影響し合う変更を扱う場合, それぞれを分離することが困難となる. 変更の順番や依存関係を調整し, 分割を行えるようにすることが望ましいものの, 難しい場合もある.

根拠

- 複数の意図の修正が混在したソースコード変更の理解は, そうでないものに比べて難しいと言われている. 例えば, リファクタリングと機能変更の混ざった修正は, そうでないものに比べて理解が困難であったという実験結果が報告されている[20]. また, そういったもつれた変更が実際には多数行われていることも報告されている[21].

問題よく知られた使い方

オープンソースソフトウェア KDE では, 完了していない変更や原始的(atomic)でないコミットを構成しないよう, コミットポリシーに定めている[22].

関連するパターン

本パターンの実現は, 変更履歴を記録しよう / Record Change Histories (パターン 4.1) の欠点である, 記録に間違いがあると正しく変更を推薦できないことの防止に役立つ. また, 本パターンは構成管理パターンランゲージ[23]における Task Level Commit と同種のものである. ただし, 保守や進化のためのパターンランゲージとして再構成されている.

パターン 4.4

変更から意味的差分を抽出しよう
Extract Semantic Difference from Changes

過去に発生したソフトウェアの変更をなるべく容易に理解したい.

問題

- ソフトウェアの保守・進化の際には，過去に行った変更の正しさの検証や，同チームの他の開発者が行った変更の分析を行うために，変更内容を理解することが必須となる.
- 変更内容の理解には差分仕様書や変更履歴が助けになるが，これらの記録が散逸している，あるいは不十分な記述である場合，正確な変更内容を理解できない．あるいは，これらの記録が真に正しいものであるかを確認すべき場合が存在する.
- このような場合，往々にして頼りになるのはソースコードのみであり，作業者は 2 バージョンのプログラムテキストをつぶさに比較する必要に駆られる．しかし全コードを全て作業者が読んで比較するのは非常にコストが高い.
- そこでこのような場合において，diff などのテキストベースの差分抽出ツールが利用される．あるいは場合によっては，使用されているプログラミング言語を考慮した構文ベースでの差分抽出ツールが使われるかもしれない．差分抽出ツールを用いることにより，コードの内容に目を通す前にまず違いを見つけることができるようになり，作業コストをある程度軽減できる.
- しかし，こうしたテキスト／構文ベースの差分抽出ツールでは，以下に示す 2 つの点から，なお作業コストは高いといえる.
  （1） 実際にその変更がどのようなものか（ソフトウェアの挙動をどのように変更するものか）を理解するには，結局作業者が 2 バージョンのコードを詳しく読んで比較する必要がある.
  （2） リファクタリングを経ている場合，実際には処理内容が変化していないにも関わらずコードの記述が変化するが、テキスト/構文ベースの差分抽出ではそのような変更でも抽出されてしまう.

解法

- 変更に伴う意味的差分(semantic difference)を，変更前後のソースコードから取得し，比較する.
- 意味的差分において，ソフトウェアの変更とは，広義には，ソフトウェアの振舞いの変化を指す．狭義には，同一の入力に対して 2 つのバージョンが異なる出力をするようなソフトウェアの変化を指す．逆に言えば，このような変更を表すソフトウェアの差分を

意味的差分と呼ぶ.

議論

差分の表現形式に特に定型的なものはなく，技術/ツールによって様々なものがある．代表的なものを以下に挙げる．

- 変化が発生する入力の条件と，各バージョンでの対応する出力（文献[24]など）
- 入出力が変化するソースコード上のパス（文献[25]など）

トレードオフ

-利点

- 抽象度が高く要約された情報が得られる．
- リファクタリングなど，処理内容が実質的に変化しない変更を除外できる．

-欠点

- 関数／メソッド単位であるのが一般的で，またスケーラビリティに困難さが伴うことが多い．大規模なソフトウェアに適用する場合は適用方法に工夫が必要である．
- 理論的な限界や計算量の問題から，ツールで得られるのは本来の差分の「近似」である．多くの技術／ツールは「過大近似」となるように設計されており，本来の差分でないものも差分として報告される可能性がある（が、本来の差分は漏れなく報告される）．

よく知られた使い方

意味的差分を抽出する技術は，知られたところでは Jackson と Ladd によって最初に提案され（semantic diff と呼ばれる）[26]，その後も改良を加えた様々な技術が提案されている[24][25]．

関連するパターン

意味的に単一な変更でない場合，プログラム解析の結果も大きくなって得られる結果も取り扱いが難しくなる．**粒度を調整しよう / Manage Granularity (パターン 4.3)** の適用により変更の粒度に関して注意を払えば，本パターンの効果は大きくなる．

パターン 4.5

変更作業の手順書を作成しよう
Make Instructions for Software Modification

変更作業を実施する前に手順書を作成することで，ソフトウェアを変更する際に発生する作業効率を高める．

問題

- ソフトウェアを変更するためには，問題把握・修正分析，修正実施，レビュー・受入れの各作業が必要となり，この作業に時間が掛かりすぎる．例えば，小さな変更に数日を要することなどが問題となったとする．このことはソフトウェア変更の実施を阻害する要因となりうるため，組織として変更作業の効率を高める必要が生じる．しかし，この作業の効率をどのようにして高めるかが明らかではないという問題がある．
- ソフトウェア変更作業の効率に影響すると考えられる要因は数多く存在する．例えば，対象ソフトウェアの規模，システム構成，求められる信頼性，技術者の作業スキルなどがあり，作業効率を高めるためには，どの要因に着目し，改善すべきかが明らかではない．このため，作業効率の改善は容易であるとはいえない．

解法

- ソフトウェア変更作業の手順書を作成している（手順を標準化している）組織では，平均的に１人あたりの作業量が大きい，すなわち作業の効率が高い傾向がみられる．このことから，変更作業の手順書を作成することにより，作業効率の改善が期待できる．
- ソフトウェアの変更時に実施している作業（問題把握・修正分析，修正実施，レビュー・受入れ）の各手順を明確に定義し，変更作業の手順書を作成する．作業時にこの手順書を参照することにより，作業の効率を高めることを目指す．以下に手順を示す．
  （１） 各技術者（プログラマー）がソフトウェアの変更時に実施している作業，すなわち問題把握・修正分析，修正実施，レビュー・受入れの各作業の詳細な手順を，可能な限り列挙する．
  （２） 列挙されたソフトウェア変更作業をレビューし，複数の技術者がソフトウェア変更作業において共通して実施しており，かつ適切と考えられる手順（例えば，あるモジュールをレビューする際には，特定のチェックリストを利用するなど）を絞り込む．
  （３） 絞り込んだ手順をまとめた，ソフトウェア変更作業の手順書を作成する．
  （４） 各技術者はソフトウェア変更作業時に標準手順書を確認し，それに従って変更を行う．

トレードオフ

–利点

- 手順書を参照して作業することにより，ソフトウェア変更作業の効率が高まることが期待できる．
- 手順書を参照して作業することにより，各技術者の作業効率の差が小さくなる可能性がある．これにより作業時間見積もりの誤差が小さくなることが期待できる．

–欠点

- 各技術者のソフトウェア変更作業を列挙し，それに基づいて手順書を作成するためには，ある程度のコストが必要となる．
- ソフトウェアの進化により，最適な作業手順も変化する可能性がある．そのため，一定期間ごとに作業手順書の内容が適切であるか，検討を行う必要が生じる．この検討にもコストを要する．

–困難

- ソフトウェア作業手順書を適切に作成し，かつ実際に作業実施する技術者に取り入れてもらうためには，手順書に対する技術者の理解と同意が必要となる．
- 技術者がソフトウェアの変更作業を行う際，手順書を参照してその内容に従うことをしないで作業する可能性がある．

根拠

- 保守における作業効率（投入された人的資源に対する産出量）に関連を持つ（影響すると考えられる）特性を明らかにする研究がある[27]．
- ソフトウェアの保守作業は，各技術者が 1 人で大部分の作業を行っていることが多い．他の技術者と共同で作業しないため，各技術者は他の技術者の作業手順を知る機会が少ないといえる．そのため，各技術者が実施する変更作業の内容に差異が生じている可能性があり，さらに，作業内容に必ずしも効率の高くない手順が含まれていたとしても，技術者自身それに気づかない可能性もある．組織が作業手順書を作成し，各技術者が手順書に従って作業することにより，効率の高くない手順を取り除くことができる可能性が高まる．その結果，ソフトウェア変更の作業効率が高まると期待できる．

5. プログラム理解支援パターン

ソフトウェアシステムにおいて，実際に稼働している実体はプログラムである．このため，ソフトウェア進化を実現する際に，プログラムが唯一信用できる成果物と考える開発者や保守者は多い．また，実際の保守現場では，プログラムしかないという状況や，過去の変更が要求仕様書や設計仕様書に反映されていないという状況が頻繁に発生している．

このような状況において，既存のソフトウェアシステムを改変するためには，実際に稼働しているプログラムを理解することが不可欠である．特に，リバースエンジニアリングにより，プログラムコードを解析することで，プログラム理解に役立つ情報を抽出することができる可能性が高い．このような観点から，OORP[7]の 2〜5 章では，全部で 20 個のリエンジニアリングパターンが紹介されている(OORP パターン 2.1〜2.7, 3.1〜3.5, 4.1〜4.3, 5.1〜5.5)．

ここでは，OORP のリバースエンジニアリングパターンを補足する形で，プログラム理解に焦点を当てた 3 つのパターンを紹介する．

パターン 5.1　テストとモジュールの対応関係を作成しよう
パターン 5.2　マイクロブログを活用しよう
パターン 5.3　コードに付箋を付けよう

パターン 5.1

テストとモジュールの対応関係を作成しよう
Create a Mapping between Tests and Modules

既存のモジュールの利用法を理解したい．

問題

- ソフトウェアの保守・進化の際には，既存のモジュールを低コストで修正・再利用することが重要である．特に，モジュールごとの仕様，中でも API の使用方法を理解することが重要であるが，このドキュメントがそもそも存在しなかったり，存在しても内容が古くなってしまっていたりすることがしばしばある．当該モジュールの開発担当者と連絡できなくなっていることもしばしばある．
- API の名前とシグネチャからおおよその使用法を推測し，実際にいくつかの入力を与えて出力を見ることで確かめる方法がある．しかしこの方法は非常に正確さに欠ける方法であり．元の開発担当者の意図をつかむことは難しい．別の方法として，ソースコード

を精査して挙動を理解した上で使用方法を考えることもできるが，これは非常にコストが高い．

解法

- 当該モジュールを主なテスト対象としているテストケースを参照することで，モジュールの使用方法を大まかに把握できる．
- 当該モジュールの開発担当者は，そのモジュールが提供する機能を一通り把握しているはずであり，少なくとも機能ごとのテストは行うと考えられる．テストに書かれた入出力例は開発担当者の意図（当該モジュールの挙動だけでなく，使用のための前提条件も含む）が含まれているものであり，派生開発の担当者が試行錯誤して API の使用方法を調べるよりも確かな情報を把握できる．また，ソースコードを精査するほどのコストはかからない．

トレードオフ

−利点

- モジュールの大まかな使い方を手軽に理解できる．
- テストケースが正常系用か異常系用か判断できる場合は，想定されている入出力の範囲をある程度知ることができる．

−欠点

- テストはあくまで例でしかなく，保証された仕様がわかるものではない．
- 十分な数のテストケースがそろっていなければ利用が難しい．

パターン 5.2

マイクロブログを活用しよう
Exploit Micro-Blogging

ソフトウェア改変において，設計意図を有効に活用したい．

問題

- 既存のソースコードを改変しようと考えた際，当時の設計意図が分からない．このような状況で，改変を行った場合，過去に取り消したコードに戻ってしまうことや暗黙の了解事項に違反するコードに変更してしまうことがある．
- 通常，コメントはそのソースコードに関する最終的な内容を記入し，設計経緯や設計判断を記述しない．コメントに設計意図を混在させるとコメントが読みにくくなる．また，

コードに記述することで設計意図が外部に流出する可能性がある．

- ソフトウェアの寿命が長くなると，改変時に開発者を探しても見つからないことが多い．

解法

- 将来の改変のために，短い文章で設計意図を残しておく．その際，ソースコードに残す通常コメントと区別し，twitter のようなマイクロブログを活用するのが良い．
- マイクロブログを活用することで，保守者や開発者に対する質疑応答を機械処理できるようになる．これにより，設計意図を検索や推薦できる．
- ソースコードの内容に詳しいのは，そのコードを記述したプログラマである．また，通常，ソースコードに関する設計経緯や設計判断が存在しないまま，ソースコードが記述されたり，変更されたりすることはない．プログラマにとって，それを記述するのが面倒なだけである．

トレードオフ

−利点

- 設計の意図が復元でき，無駄な改変を避けることができる．

−欠点

- 記述のためのプログラマの負担が増える．

実例

GitHub などのソーシャルコーディング環境では，コミットやコミットされたコードにコメントを付与することができる．

関連するパターン

ソースコードの位置を指定したコメントが有用な場合，**コードに付箋を付けよう** (パターン 5.3) の利用を検討せよ．

パターン 5.3

コードに付箋を付けよう
Attach Post-it notes to Code

ソースコードに一時的な情報を残したい．

問題

- 通常，コメントはそのソースコードに関する恒久的な内容を記入し，揮発的なメモや TODO などの予定を記入しない.
- 恒久的な内容と一時的なコメントを同じ形式で混在させると，検索結果におけるノイズの発生確率が高くなる.
- マイクロブログやメールなどのメディアでは，ソースコードの特定箇所を指示するのが面倒である．ファイル名や行番号で位置を特定した場合，変更されたソースコードに対する位置を追従させるのが困難である.

解法

- メモや TODO を（開発環境で提供する）付箋としてソースコードに貼り付ける．これにより，ソースコードにおける位置の追従が容易になる.
- 不要になった付箋は簡単に削除することができるようにする．その際，ソースコードの変更には影響を与えないことが重要である.
- 理解作業や改変作業が休憩や翌日にまたがる場合に，付箋を作業の再開(resume)の参考に利用する．再開後に付箋は削除する.
- 付箋を特定のイベント（例えば，時刻，変数や関数の名前変更）に対応させて，付箋の表示（発火）を制御することができる.

トレードオフ

-利点

- 概念として慣れている（よく知られた）付箋を用いて，一時的な情報を管理できる.
- イベントドリブンなコメント機能が実現できる.

-欠点

- 付箋を管理する特別な開発環境が必要になる．異なる環境で付箋情報を共有することは難しい.

関連するパターン

本パターンは，コードと疑問を結びつけよう / Tie Code and Questions (OORP パターン 5.1) の具体的な実現方法のひとつである.

6. リファクタリングプロセスパターン

リファクタリングとは，既存ソフトウェアの理解性や変更容易性を向上させることを目的とした上で，外部的挙動を保存したままで内部構造を改善する活動を指す[8]．一般的には，リファクタリング前後で，同一の入力値集合（外部入力）に対して同一の出力値集合（外部出力）が得られることを，外部的挙動が保存されているという．また，リファクタリングでは，大きな設計変更を小さな変換の繰り返しで実現することで，挙動の保存を保証するのが特徴である．ソフトウェア進化では，ソフトウェアの改変を伴うため，リファクタリングは進化パターンでひとつである．

ここでは，リファクタリングにおける新たなコード変換を提案するのではなく，リファクタリングプロセスに関する 2 つのパターンを紹介する．

パターン 6.1　コア資産は今すぐリファクタリングしよう
パターン 6.2　リファクタリングのフレーム条件を定義しよう

パターン 6.1

コア資産は今すぐリファクタリングしよう
Refactor Your Core Assets, Just for Today

コア資産の品質を安定させるための適切なリファクタリング時期を逃さない．

問題

- コア資産のリファクタリングをいつやるべきかわからない．
- 何年もたって肥大化してしまったコア資産に対して，ようやくリファクタリングしようとしても，当時の開発者とは担当も異なり，リファクタリングは容易ではない．

解法

- コア資産のリファクタリングは，コア資産を開発している最中から積極的に実施し，その時点で可能なリソースの範囲で実施すべきである．
- プロダクトライン型の開発は長年に渡り，コア資産を維持管理，拡張させていく．製品開発とコア資産拡張の繰り返しによって，コア資産が肥大化，複雑化が進行する．長年の開発の過程では，当然，コア資産開発者も変化するため，逆に開発時点から継続的に，適切なタイミングでリファクタリングを実施しておかないと，例えば 3 年後では，当時のコア資産開発者は関わっておらず，過去の経緯は把握できなくなる．

- リファクタリングコストが確保されたプロジェクトも稀であるので，その時のコア資産の開発者がその時点でできることをタイムリーに実施していくことが必要である．

トレードオフ

−利点

- 製品承認（パッケージとして出荷されて）から，大掛かりなリファクタリングをする計画をしていても，技術や市場の変化が速い業界では，投資ができなくなるリスクが高いので，コア資産の裁量で，その場その場でリファクタリングしておくことは重要である．
- そのコア資産が，もっと大きなプロダクトラインの一部に組み込まれることもある．組み込まれてから，品質を高めることは手戻りコストがいっそう高くなる．

−欠点

- その時点でコア開発者のスキルに依存したリファクタリングとなり，リファクタリングした結果のコア資産の品質が均質化されないリスクがある．

根拠

エクストリーム・プログラミングに，YAGNI(You Aren’t Gonna Need It)がある．これは，機能は実際に必要となるまでは追加しないのがよいとする原則である．言い換えれば，今必要なことだけを行うべきである，という意味である．コア資産の拡張におけるリファクタリングは，まさに今必要なことである．追加要求は，コア資産の次の版で実現される要求であるから，その際に対応すれば良い．コア資産の品質を高めるためのリファクタリングは，まさに，今やるべきことである．

パターン 6.2

リファクタリングのフレーム条件を定義しよう
Define Frame Conditions before Refactoring

安全に（挙動の保存を担保して）リファクタリングを実施したい．

問題

- 挙動が保存されるかどうかが不明なため，リファクタリングの実施をためらうことがある．
- 保存する挙動が明確でないために，リファクタリング前後で挙動が保存できているかどうかが判定できない．保存する挙動を決めたとしてもそれを明示する方法がなく，挙動の保存を確認できない．

- これは, すでに存在するテストケースを用いれば良いという楽観的思考に基づく誤解である.
- リファクタリングが改善する内部的構造とテストケースが完全に対応するわけではない. つまり, 挙動の保存を保証するためのテストが不十分となる場合がある. このため, テストを行うことで挙動が保存されているという確信が持てない.

解法

- リファクタリングを行う前に，保存すべき挙動を明確に定義する．例えば，定義する内容として，範囲（システム全体，パッケージ，クラス，メソッド，API 集合など），要素間関係（継承や参照など），振る舞い（通常処理と例外処理，セキュリティなど）が考えられる．
- リファクタリングによって保存される挙動を形式的に記述し，検証可能とする．
- リグレッションテストだけに頼らず, 保存すべき挙動に応じたテストケースを必ず用意する.

トレードオフ

-利点

- 適用したリファクタリングが挙動を保存しているどうかを形式的に確認できる.

-欠点

- リファクタリングの費用が高くなる.
- 保存したい挙動を形式的に記述するためのスキルがプログラマに要求される.

関連するパターン

挙動の保存を確認するためには，OORP[7]の進化を実現するためのテストを書こう / Write Tests to Enable Evolution (OORP パターン 6.1) によるテストとの併用が重要である．これは，テストを用意することで，進化の際に機能の一部が保存されることを担保しようとするパターンである.

パターン一覧

ソフトウェアプロダクトラインパターン

コードクローンパターン

ソフトウェア変更支援パターン

プログラム理解支援パターン

リファクタリングプロセスパターン

参考文献

［1］M. M. Lehman, “Programs, Life Cycles, and Laws of Software Evolution”, Proc. IEEE, Vol.68, No.9, pp.1060–1076 (1980)

［2］大森隆行, 丸山勝久, 林晋平, 沢田篤史, “ソフトウェア進化研究の分類と動向”, コンピュータソフトウェア, Vol.29, No.3, pp.3–28 (2012)

［3］W. M. Ulrich, P. H. Newcomb, “Information Systems Transformation: Architecture-Driven Modernization Case Studies”, Morgan Kaufmann (2010)

［4］D. L. Parnas, “Software Aging”, Proc. ICSE'94, pp.279–287 (1994)

［5］M. Jazayeri, “Species Evolve, Individuals Age”, Proc. IWPSE'05, pp.3–12 (2005)

［6］E.J. Chikofsky, and J. H. Cross II, “Reverse Engineering and Design Recovery: A Taxonomy”, IEEE Software, Vol.7, No.1, pp.13–17 (1990)

［7］S. Demeyer, S. Ducasse, O. Nierstrasz, “Object-Oriented Reengineering Patterns”, Morgan Kaufmann (2002)

［8］M. Fowler, “Refactoring: Improving the Design of Existing Code”, Addison-Wesley Professional (1999), (訳) 児玉公信, 平澤章, 友野晶夫, 梅沢真史, “リファクタリング”, ピアソンエデュケーション (2000)

［9］パターンワーキンググループ, “ソフトウェアパターン入門”, ソフト・リサーチ・センター (2005)

［10］P. Clements and L. Northrop, “Software Product Line: Practices and Patters”, Addison-Wesley Professional (2001)

［11］K. Pohl, G. Böeckle, F. J. van der Linden, “Software Product Line Engineering: Foundations, Principles and Techniques”, Springer (2005) , (訳) 林好一, 吉村健太郎, 今関剛, “ソフトウェアプロダクトラインエンジニアリング ― ソフトウェア製品系列開発の基礎と概念から技法まで”, エスアイビーアクセス (2009)

［12］J. McGregor, D. Muthig, K. Yoshimura, P. Jensen, “Successful Software Product Line Practices”, IEEE Software, Vol.27, No.3, pp.16–21 (2010)

［13］K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, A. S. Peterson, “Feature-Oriented Domain Analysis (FODA) Feasibility Study”, CMU/SEI-90-TR-21 (1990)

［14］K. C. Kang, V. Sugumaran, S. Park, “Applied Software Product Line Engineering”, Auerbach Publications (2010)

［15］L. Gorchels, “The Product Manager's Handbook: The Complete Product Management Resource”, McGraw-Hill (2000)

［16］位野木万里, 杉本信秀, 深澤良彰, “ステークホルダの意思決定を支援するプロダクトライン再生シナリオの提案”, ソフトウェア工学の基礎ワークショップ(FOSE2008), pp.75–80 (2008)

［17］神谷年洋, 肥後芳樹, 吉田則裕, “コードクローン検出技術の展開”, コンピュータソフトウェア, Vol.28, No.3, pp.29–42 (2011)

［18］J. Humble, D. Farley, “Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation”, Addison-Wesley Professional (2010), (訳) 和智右桂, 高木正弘, “継続的デリバリー： 信頼できるソフトウェアリリースのためのビルド・テ

スト・デプロイメントの自動化”, アスキー・メディアワークス (2013)

［19］P. M. Duvall, S. Matyas, A. Glover, “Continuous Integration: Improving Software Quality and Reducing Risk”, Addison-Wesley Professional (2007), (訳) 大塚庸史, 丸山大輔, 岡本裕二, 亀村圭助, “継続的インテグレーション入門”, 日経 BP 社 (2009)

［20］S. Thangthumachit, S. Hayashi, M. Saeki, “Understanding Source Code Differences by Separating Refactoring Effects”, Proc. APSEC’11, pp.339–347 (2011)

［21］K. Herzig, A. Zeller, “The Impact of Tangled Code Changes”, Proc. MSR’13, pp. 121–130 (2013)

［22］KDE TechBase, “Policies/Commit Policy”, http://techbase.kde.org/Policies/Commit_Policy#Commit_complete_changesets

［23］S. Appleton, B. Berczuk, “Software Configuration Management Patterns”, Addison-Wesley (2002)

［24］S. Person, M. B. Dwyer, S. Elbaum, and C. S. Păsăreanu, “Differential Symbolic Execution”, Proc. FSE’08, pp.226–237 (2008)

［25］S. Lahiri, C. Hawblitzel, “Symdiff: A Language-Agnostic Semantic Diff Tool for Imperative Programs”, Proc. CAV’12, pp.712–717 (2012)

［26］D. Jackson and D. A. Ladd, “Semantic Diff: A Tool for Summarizing the Effects of Modifications”, Proc. ICSM’94, pp. 243–252 (1994)

［27］角田雅照, 門田暁人, 松本健一, 押野智樹, “受託開発ソフトウェアの保守における作業効率の要因”, コンピュータソフトウェア, Vol.29, No.3, pp.157–163 (2012)

謝辞

本パターン集を発行するにあたり,「ソフトウェア進化技術の実践に関する調査研究」の機会を頂きました産学戦略研究フォーラム(SSR: Joint Forum for Strategic Software Research)およびSSR会員企業の皆様に深く感謝いたします.