Vol.2014-SE-184 No.1
Vol.2014-EMB-33 No.1
2014/5/19

産学連携によるソフトウェア進化パターン収集の試み

丸山 勝久[1,a)]　沢田 篤史[2]　小林 隆志[3]　大森 隆行[1]　林 晋平[3]　飯田 元[4]　吉田 則裕[4,†1]
角田 雅照[5]　岩政 幹人[6]　今井 健男[6]　遠藤 侑介[6]　村田 由香里[6]　位野木 万里[7,†2]　白石 崇[7]
長岡 武志[7]　林 千博[8]　吉村 健太郎[9]　大島 敬志[9]　三部 良太[9]　福地 豊[9]

概要：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．我々は，Demeyer らのオブジェクト指向リエンジニアリングパターンを補完することを目的に，産学連携でパターンの収集を試みた．本稿では，その結果として，ソフトウェアプロダクトライン，コードクローン，ソフトウェア変更支援，プログラム理解支援，リファクタリングプロセスに関する進化パターンを提案する．

1. はじめに

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，ソフトウェア開発や保守に関わる技術者にとって，ソフトウェアの進化が避けられないという実感は強い．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・保守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ことが報告されている [2], [3], [4]．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や保守においてどのように役に立つのかが整理されているとは言い難い．たとえば，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，それらの技術の活用に関する明確な指針は示されていない．

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明確にすることが重要である．このような観点から，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（本稿では，OORP と呼ぶ）としてまとめている [5][*1]．著者らは，ソフトウェア開発・保守の実践者にとって有益な指針や慣例に焦点を当て，Demeyer らの OORP を補完する進化パターンの収集に取り組んだ．本稿では，その結果として，次に示す 5 種類，20 個の進化パターンを提案する．

(a) ソフトウェアプロダクトラインパターン (5)
(b) コードクローンパターン (5)
(c) ソフトウェア変更支援パターン (5)
(d) プログラム理解支援パターン (3)
(e) リファクタリングプロセスパターン (2)

括弧内の数字はパターンの数を指す．進化パターンの収集においては，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，OORP の出版（2002 年）以降に急速に普及した概念を積極的に取り込んだ．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，進化パターンを追加した．

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．

著者らの提案する進化パターンの記述形式を**表 1** に示す．ソフトウェアにおけるパターンとは，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

1 立命館大学
2 南山大学
3 東京工業大学
4 奈良先端科学技術大学院大学
5 近畿大学
6 株式会社東芝
7 東芝ソリューション株式会社
8 株式会社とめ研究所
9 株式会社日立製作所
†1 現在，名古屋大学
†2 現在，工学院大学
a) maru@cs.ritsumei.ac.jp

*1 http://scg.unibe.ch/download/oorp/ から PDF 版をダウンロード可能である．

表 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や条件を含む
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 パターンを適用することにより発生する悪い点 パターンの適用に関する困難
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

を目に見える形で体系的に表現したものである [6]．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．

紙面の都合上，本稿では，パターンの名前，問題，解法（の概要）だけを提示する．その他の項目に関する記述を含む進化パターン集は，以下のサイトで公開されている．

http://www.fse.cs.ritsumei.ac.jp/ssr2013/patterns.html

以降，2 ～ 6 章では，20 個の進化パターンを 5 種類の分類に基づき紹介する．

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，共通の特性を持つソフトウェア中心システムの集合を指す [7]．これは，特定の市場やミッションのために，共通のコア資産 (core asset) から，規定された方法で作られる．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

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 [8] 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．
- 製品ライフサイクルの段階は導入期, 成長期, 成熟期, 衰退期の 4 つがある．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からである．

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が長期間に渡る場合，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する．
- 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．その際，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となる．
- 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んだり，影響範囲が把握できないまま拡張することになる．その結果，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，保守コストが増加する．

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する．
- 肥大化した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，まず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戦略を優先することがリスク回避のために有効である．

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

- ある組織において，複数のプロダクトラインが維持管理されている場合を考える．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，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（回収額の想定順位）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．

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．

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．

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

- 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よいかわからない．

解法

- レガシー資産をコア資産として再生するための，代表的なシナリオは [9] で提案されている．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のがよい．

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 [10]．これは重複コードとも呼ばれ，重複コードを発見するためのパターンが OORP の 8 章で紹介されている．ここでは，OORP の重複コード発見パターンに追加する形で，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

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: 対角線となっているブロックを探すことで，ファイル全体がコピーされて発生したソースファイルを特定できる．

戦略 2: 対角線が分断されているブロックを探すことで，ファイル全体がコピーされ，その後コードの修正が行われたソースファイルを特定できる．

戦略 3: 破線・点線となっているブロックを探すことで，再利用にふさわしくない部分が仕分けられる．

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，散布図の詳細部分を表示できず，**パターン 3.1** で示した特徴を散布図から発見することができない．

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**パターン 3.1** 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．
- 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，メソッドなどと変える．例えば，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ことが考えられる．

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場合がある．

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．

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，それを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．

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．

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 の 7, 9, 10 章では，システムの移行戦略や設計変更に関するパターンが紹介されている．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や版管理システムの利用が一般的である [11]．このような状況を受け，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

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，一か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．

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

- **パターン 4.1** を適用して変更履歴を記録し，それを解析することによって，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．

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．

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．これ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しなければならない場合が存在する．
- 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．
- テキストベースの差分抽出ツールや構文ベースの差分抽出ツールが使えるかもしれない．しかし，こうした差分抽出ツールを用いても，実際に変更内容を理解するためには，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しなければならない．また，リファクタリングのように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変更が差分として抽出される．よって，作業コストは高いままである．

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．

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．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．
- 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) 開発者や保守者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ることで，ソフトウェア変更作業の手順書を作成する．

5. プログラム理解支援パターン

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 の 2 ～ 5 章では，リエンジニアリングパターンが紹介されている．ここでは，これらの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

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，なかでも 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．また，当該モジュールの開発担当者と連絡ができないこともしばしば発生する．

- 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．
- 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．

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よい．
- 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．

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．

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 [12]．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．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

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．
- 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，過去の経緯を把握しているコア資産開発者が存在するうちに，リファクタリングを実施すべきである．

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．
- 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．

7. おわりに

本稿では，著者らが収集した 20 個の進化パターンを紹介した．パターンのなかには，未来の開発環境を想定したものもあり，現時点での有用性は未知である．今後，それぞれのパターンの適用場面をさらに調査し，より開発現場や保守現場に適した記述に洗練していく予定である．

謝辞 本稿における進化パターンの収集は，平成 25 年産学戦略研究フォーラム (SSR: Joint Forum for Strategic Software Research) の支援を受けた「ソフトウェア進化技術の実践に関する調査研究」のもとで実施しました．SSR および SSR 会員企業の皆様に深く感謝いたします．

参考文献

[1] Lehman, M. M.: Programs, Life Cycles, and Laws of Software Evolution, *Proc. IEEE*, Vol. 68, No. 9, pp. 1060–1076 (1980).
[2] Madhavji, N. H., Fernández-Ramil, J. and Perry, D. E.: *Software Evolution and Feedback: Theory and Practice*, Wiley (2006).
[3] Mens, T. and Demeyer, S.: *Software Evolution*, Springer (2008).
[4] 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).
[5] Demeyer, S., Ducasse, S. and Nierstrasz, O.: *Object-Oriented Reengineering Patterns*, Morgan Kaufmann (2002).
[6] 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).
[7] Clements, P. and Northrop, L.: *Software Product Line: Practices and Patters*, Addison-Wesley Professional (2001).
[8] Gorchels, L.: *The Product Manager's Handbook: The Complete Product Management Resource*, McGraw-Hill (2000).
[9] 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).
[10] 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).
[11] Humble, J. and Farley, D.: *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*, Addison-Wesley Professional (2010).
[12] Fowler, M.: *Refactoring: Improving the Design of Existing Code*, Addison-Wesley Professional (1999).