Javaプログラムにおける
コーディングパターンの分析に関する研究

提出先　　大阪大学大学院情報科学研究科

提出年月　2015 年 1 月

伊達　浩典

内容梗概

ソフトウェアの大規模化複雑化に伴い，ソフトウェア開発のコストが大きくなってきている．ソフトウェア開発において，成果物を再利用することがコストの削減につながると期待されている．

ソフトウェアのソースコードには，様々なコーディングに関するパターンが含まれている．本論文で扱うコーディングパターンは，Java のソースコードからパターンマイニングにより機械的に取り出したものである．抽出したパターンを，次の開発に再利用できれば，さらなる開発コストの削減につながる．コーディングパターンを再利用対象として扱うには，次のような問題がある．まず，検出したコーディングパターンにどのような種類があるのか判明していない．次に，コーディングパターンを種類ごとに分類するための手法が存在しない．そして，多数の類似パターンが検出されるため，再利用に適したパターンの選択が困難である．

これらの問題を解決するために，3 つの研究を行った．まず，コーディングパターンの種類を調査するために，6 種類のオープンソースソフトウェアを調査した．この調査により，コーディングパターンには，アプリケーションの特定機能の実装，例外処理の定型的な記述，偶然の一致などの種類があることが判明した．

次に，コーディングパターンを分類するために，6 種類のメトリクスを定義した．そして，4 種類のオープンソースソフトウェアから抽出したコーディングパターンのメトリクスを計測し，コーディングパターンの種類とメトリクス値の関係について，分析した．

そして，類似パターンの中から，再利用に適するパターンを選択するために，コーディングパターンの構成要素が変化しないパターンが再利用に適していると考え，10 種類のオープンソースソフトウェアの複数バージョンから検出したコーディングパターンを調査した．結果として，コーディングパターンの全バージョンで変化せずに登場している物は，極少数であり，コーディングパターンの絞り込みに有効であると判明した．

本論文では，コーディングパターンの性質を明らかにした．これにより，コーディングパターンを使った開発支援につながる．

論文一覧

主要論文

Hironori Date, Takashi Ishio, Makoto Matsushita, Katsuro Inoue: “Analysis of Coding Patterns over Software Versions”, コンピュータソフトウェア, 日本ソフトウェア科学会, Vol.32, No.1, 2015.

Hironori Date, Takashi Ishio, Katsuro Inoue: “Investigation of Coding Patterns over Version History”, Proceedings of the 4th International Workshop on Empirical Software Engineering in Practice (IWESEP 2012), pp.40-45, 2012.

伊達 浩典, 石尾 隆, 井上 克郎: “オープンソースソフトウェアに対するコーディングパターン分析の適用”, ソフトウェアエンジニアリング最前線 2009, pp.173-178, 2009.

石尾 隆, 伊達 浩典, 三宅 達也, 井上 克郎: “シーケンシャルパターンマイニングを用いたコーディングパターン抽出”, 情報処理学会論文誌, Vol.50, No.2, pp.860-871, 2009.

Takashi Ishio, Hironori Date, Tatsuya Miyake, Katsuro Inoue: “Mining Coding Patterns to Detect Crosscutting Concerns in Java Programs”, Proceedings of the 15th IEEE Working Conference on Reverse Engineering (WCRE 2008), pp.123-132, 2008.

Takashi Ishio, Hironori Date, Tatsuya Miyake, Katsuro Inoue: “Mining Application-Specific Coding Patterns for Software Maintenance”, Proceedings of the Linking Aspect Technology and Evolution Workshop (LATE 2008), 2008.

関連論文

ブャンネメフ オドフー, 眞鍋 雄貴, 伊達 浩典, 石尾 隆, 井上 克郎: “コードクローンの動作を比較するためのコードクローン周辺コードの解析”, 情報処理学会研究報告, Vol.2013-SE-180, No.2, pp.1-8, 2013.

工藤 良介, 伊達 浩典, 石尾 隆, 井上 克郎: “コードクローンに含まれるメソッド呼び出しの変更度合の調査”, 情報処理学会研究報告, Vol.2013-SE-179, No.15, pp.1-8, 2013.

中野 佑紀, 伊達 浩典, 渡邊 結, 石尾 隆, 井上 克郎: “プログラム実行履歴を用いたオブジェクト生成関係の可視化”, 情報処理学会論文誌, Vol.53, No.3, pp.1166-1176, 2012.

工藤 良介, 伊達 浩典, 石尾 隆, 井上 克郎: “リファクタリング支援のためのコードクローン間の識別子名の対応関係分析”, 情報処理学会研究報告, Vol.2011-SE-173, No.8, pp1-8, 2011.

中野 佑紀, 伊達 浩典, 渡邊 結, 石尾 隆, 井上 克郎: “オブジェクト生成関係抽出ツール ROBIN”, 日本ソフトウェア科学会 FOSE2010, ソフトウェア工学の基礎 XVII, pp.187-189, 2010.

伊達 浩典, 関山 太朗, 石尾 隆, 井上 克郎: “コーディングパターンに基づくコード補完ツールの試作”, 名阪和ソフトウェア工学ミニワークショップ 2010, 2010.

関山 太朗, 伊達 浩典, 石尾 隆, 井上 克郎: “コーディングパターンとキーワードを用いて生成したコードスニペットの推薦”, 第 72 回情報処理学会全国大会講演論文集 (1), pp.553-554, 2010.

悦田 翔悟, 伊達 浩典, 石尾 隆, 井上 克郎: “分散処理を用いたコーディングパターン検出ツールの実装”, 第 71 回情報処理学会全国大会講演論文集 (1), pp.339-340, 2009.

石尾 隆, 伊達 浩典, 市井 誠, 井上 克郎: “大規模パターンマイニングを用いた高品質ソースコードの検索”, ウィンターワークショップ 2009・イン・宮崎 論文集, 情報処理学会シンポジウムシリーズ, Vol.2009, No.3, pp.9-10, 2009.

伊達 浩典, 三宅 達也, 石尾 隆, 井上 克郎: “コーディングパターンの分類に用いるソフトウェアメトリクスの検討”, 平成 20 年度 情報処理学会関西支部 支部大会 講演論文集, B-05, pp.59-62, 2008.

目 次

図 目 次

表 目 次

第1章　まえがき

情報技術の発展により，社会の様々な部分で情報システムが利用されるようになってきた．情報システムが社会基盤の重要な位置を占めるようになり，また高い信頼性が求められている．さらに，従来情報システムはそれぞれが独立して動作していたが，情報システム同士がネットワークを通じて，互いに情報を交換し合い連携して動作するようになった．それに伴い，情報システムを構成するソフトウェアも大規模複雑化の一途をたどっている．また，ソフトウェアを取り巻く社会状況も激しく変化しており，社会状況に適合したソフトウェアを素早く開発し，改良を続ける必要がある．

ソフトウェアの再利用を行うことで，ソフトウェアの開発期間を短縮でき，コストの削減につながる．ソフトウェアを再利用するためには，ソフトウェアの設計やソースコードなどの成果物を，あらかじめ再利用対象として部品化しておくことが望ましい．

ソフトウェア開発においては，様々なフェーズで様々な種類の成果物を再利用する．本章では，まず，再利用対象の種類毎にソフトウェアの再利用について説明を行う．次に，ノウハウの再利用として，パターンを取り上げ説明する．そして，再利用可能な部品を既存のソースコードから抽出する技術として，API マイニング，アスペクトマイニングを取り上げる．

1.1　ソフトウェアの再利用

ソフトウェア開発における再利用とは，ひとつの汎用的に使えるように設計されたデザインや仕様，ソースコード，ドキュメント，テストスイート，作業手順などの成果物を，複数の問題解決において使用することである [62].

ソフトウェア開発においては，様々なフェーズで様々な成果物を再利用できる．過去に開発したシステムと類似したシステムを開発する場合には，まず，設計や実装の枠組みの再利用がおこなわれる．そして，実装段階では，この機能を実現する部品単位での再利用が行われる．以降，設計段階での再利用，ソフトウェアの全体の枠組みの再利用，細かな部品としての実装の再利用について述べる．

1.1.1　設計の再利用

過去に開発したシステムと類似したシステムの開発を行う場合には，その設計情報についても再利用が可能な場合がある．設計を再利用することで，過去の優れたアーキテク

チャを採用できる．近年では，ソフトウェアプロダクトライン開発として，共通の基本構造をもつ複数の類似プロダクトを効率的に開発する手法が採用されるようになっている．ソフトウェア開発開始時点から，プロダクトラインとして設計されることもあるが，既存のソフトウェアから，新規ソフトウェアと共有できる基本構造を抽出し再利用する場合も多い [43]．また，データ構造やデータベースの設計についても類似する場合が多く，再利用可能である．

1.1.2 枠組みの再利用

ソフトウェアの実装の再利用に関しても，マクロ的に見た全体の構成の再利用と，それぞれの細かな機能の実装の再利用が考えられる．ここでは，ソフトウェアの枠組みとしてのフレームワークについて説明する．

特定分野で使用されるソフトウェアは，同一の構造を持つことが知られている．たとえば，Web アプリケーションであれば，ユーザに提供する機能部分はアプリケーションごとに異なるが，次のような共通の処理手順を持つ．

1. ユーザからのリクエストを受け取る．
2. 受け取ったリクエストに応じて処理を行う．
3. 処理結果をユーザに送信する．

このような，共通する処理の枠組みを再利用する方法は，フレームワークと呼ばれる．

フレームワーク

フレームワークとは，特定分野に有効な，一連の協調動作する再利用可能な部品集合のことである．分野で共通する処理は，フレームワーク側であらかじめ実装されている．そのため，開発者は，そのソフトウェア固有の機能の実装のみを行うことで，短期間で高品質なソフトウェアの開発が可能になる．

Web アプリケーションの分野で用いられるフレームワークとしては，プログラミング言語 Java 用の Apache Struts[5] や Spring Framework[63]，Ruby 用の Ruby on Rails[59] 等が有名である．

1.1.3 実装の再利用

ソースコードを作成する段階では，過去に作成されたコードを行単位，関数単位，ファイル単位などで再利用できる．既存のコードを再利用することにより，新たにコードを記述し，テストを行うコストを削減できる．

コピー・アンド・ペースト

ソフトウェア開発者は，既存のソースコードの一部分をコピーして，作成中のファイル中にペーストすることでコードの再利用を行っている [39, 74]．このような再利用を行うと，同一のコードが複数個所に存在することになる．複数個所に存在するコードは，コードクローン検出手法 [76] やパターンマイニング手法による検出が期待される．

ライブラリ

ソフトウェア開発で必要になる機能を，全て独自に開発することは現実的ではない．そのため，ソフトウェアを構成する汎用的な機能は，プログラミング言語であらかじめ実装されており，開発者に対して標準ライブラリとして提供されている．また，様々な専門分野に特化したライブラリも提供されており，必要に応じて利用できる．

ライブラリをプログラム中から呼び出すための API（Application Programming Interface）の集合と捉えることができる．ライブラリは，C 言語では関数の集合として，Java ではクラスの集合として提供される．

1.2　ソフトウェアのパターン

パターンとは，繰り返し起きる問題とその問題の解決方法を示したもので，建築学の分野で Alexander ら [3] により作成されたものである．通常，複数のパターンをまとめてカタログ化された形で提供されている．再び同様の問題が発生した場合に，パターンカタログを参照し，適切なパターンを適用することにより問題を素早く解決できる．

このパターンの考え方をソフトウェアの分野に適用したものがソフトウェアのパターンである．

1.2.1　ソフトウェアの開発工程

ソフトウェア開発工程は，たとえば，要求分析，基本設計，詳細設計，実装の順に進んでいく．まず，要求分析では，システム化対象をモデル化してソフトウェアが実現すべき機能を洗い出し，ソフトウェアの要求仕様を作成する．次に，基本設計では，要求仕様を満たすために必要なソフトウェア全体の構成を決定する．そして，詳細設計では，ソフトウェアのそれぞれの機能を実現するための詳細な設計を行う．最後に，実装では，ソースコードを作成し実際に動作するソフトウェアとなる．

1.2.2　ソフトウェアの開発工程ごとのパターン

ソフトウェアの開発工程とそれぞれの工程で用いられるパターンには，次のようなものがある．

要求分析に関するパターン

要求分析の段階で用いられるパターンは，**アナリシスパターン**と呼ばれる．分析対象をモデル化する際に，どのような視点でとらえて，どのように表現すれば有用なモデルとなるかをパターン化したものである．

アナリシスパターンに該当するパターンは，Fowler[22] によるものが存在する．

基本設計に関するパターン

基本設計では，ソフトウェア全体の基本的な構造，すなわちソフトウェアのアーキテクチャを設計する．Buschmann らは，POSA (Pattern-Oriented Software Architecture) の**アーキテクチャパターン**を提案している [15]．POSA のアーキテクチャパターンの中では，GUI 作成時にプログラムを Model，View，Controller の 3 要素に分割して全体を構成する MVC パターンが有名である．

詳細設計に関するパターン

詳細設計の段階で利用されるパターンとしては，**デザインパターン**が挙げられる．アーキテクチャパターンが，ソフトウェアのマクロ的な構造に関するパターンであるのに対し，デザインパターンは，もう少し細かい部分の設計に用いられるパターンである．

Gamma らは，生成，構造，振る舞いの 3 つの視点に立った合計 23 種類のデザインパターンを提案している [24]．この他にも，マルチスレッド関連に特化したデザインパターン [78] も提案されている．

実装に関するパターン

実際にソースコードを記述する段階において登場するパターンとしては，**コーディングのパターン**や**イディオム**が存在する．コーディングのパターンやイディオムは，特定のプログラミング言語や環境に依存する抽象度が低いパターンである．

Java によりプログラムを記述する場合の Iterator を用いた複数オブジェクトへの反復処理の実装や StringBuilder や StringBuffer クラスを利用した文字列構築処理は，イディオムに該当する．

Coplien は，C++言語における実装上のテクニックをイディオムとしてまとめている [18].

1.3 コーディングパターン

コーディングパターンは，API やプログラム内部の機能を利用する場合に，繰り返し登場する定型的な処理のことである．コーディングパターンとして検出されるものとして，API の利用方法やアスペクトが挙げられる．

ソフトウェア開発において成果物を再利用するためには，その成果物の利用方法を知る必要がある．API に典型的な利用方法がある場合には，そのような利用方法は，ソースコードの複数個所に記述されることになる．そのため，パターンマイニングにより，どのようなメソッド呼び出しの組み合わせが順序制約を持って用いられているかを調査できる．

また，アスペクトは，その実装がソースコード中に分散して存在する特徴があるため，コーディングパターンとして検出する対象となりうる．

1.3.1 API の利用方法

ライブラリ中の複数の API を呼び出す場合に，API の呼び出す順番に制約が存在する場合がある．この制約に違反すれば，ライブラリは正しく動作しない．たとえば，ファイルにデータを書きだす時，open, write, close の 3 つのメソッドを用いる場合，まず open メソッドによりファイルを開き，次に write メソッドによりファイルにデータを書き込む，そして最後に，close メソッドによりファイルを閉じる必要があり，この 3 つのメソッドを順番通りに呼び出す必要がある．もし，close メソッドを呼び出した後に，write メソッドを呼び出せばエラーとなる．また，open メソッドにより開いたファイルは，close メソッドを用いて閉じなければならない．

開発者が，このようなライブラリの利用方法を学習する場合，ライブラリに付属するドキュメントを参照したり，そのライブラリを実際に利用しているソースコードを参考にする．

しかし，ドキュメント中に API 全ての利用方法が記述されているとは限らない．また，開発者はオブジェクトを作成する場合に，オブジェクトの作成方法を調査するのではなく，まず引数を持たないデフォルトのコンストラクタを呼び出してオブジェクトを作成しようと試みることが報告されている [64]．そのため，他人の作成したソースコードを理解し，ライブラリの利用方法を学習するコストは大きいと考えられる．

1.3.2 API の利用方法のマイニング

複数の API の呼び出し順序に制約が存在する場合，これらの API の呼び出しが制約にしたがった順番で登場するはずである．そのため，多数の API の利用事例を調査すれば，API の利用方法を特定できる．

パターンマイニングの手法を利用して，API の呼び出し順序を解析する研究が行われている．Acharya らは，手続き間の制御フローを解析し，API の呼び出しにおける半順序関係を抽出する手法を提案している [1]．この手法は，半順序関係の候補を削減するために，どの関数群が 1 つのグループであるのかを決定しておく必要があり，異なるグループに属する API 間の関係は発見しない．

Xie らは，開発者があらかじめ興味あるライブラリやクラス名をクエリとして入力し，コード検索エンジンから抽出してきたソースコードに対するパターンマイニングを行う

ツール MAPO を提案している [72].

1.3.3 アスペクト

オブジェクト指向では，データとそのデータに関する操作を，ひとまとめとしてクラスを作成する．しかし，データ以外の側面からクラスを観察すると，1 つのクラスに複数種類の関心事が混ざり合っているととらえることもできる．また，ある関心事に着目すると，その関心事は，複数のクラスに散らばっている状況になる．たとえば，データ構造を担う複数のクラスが，マルチスレッド環境下で安全に動作させる目的で，それらのクラスの各メソッドの先頭と末尾に，スレッド間の同期処理を行う処理を挿入する場合が考えられる．このような場合，各クラスが持つメソッドの名前などは，それぞれ異なるため，共通の親クラスなどに処理をまとめることは困難であり，同期したい処理の開始と終了を示す enter と exit といったメソッドを外部に用意し，該当する全てのメソッドの先頭と末尾でこれらのメソッドを呼び出すといった実装になる．このように，オブジェクト指向ではモジュール化が困難な横断的関心事が存在し，横断的関心事をクラスから抽出してモジュール化を行うためにアスペクトの枠組みが考案されている．

アスペクト指向プログラミング

アスペクト指向プログラミング [38] では，横断的関心事をモジュールとして定義して，コンパイル時にクラスに統合し利用可能となる．アスペクトは，それ単体で利用されるものではなく，オブジェクト指向プログラミング言語の弱点を補完する立場にある．Java をベースにしたアスペクト指向プログラミング環境 AspectJ[8] や，C++をベースにした環境 AspectC++[7] が存在する．

アスペクトマイニング

アスペクトマイニングは，経験的な指標を用いて，ソースコードから横断的関心事の候補を探索する手法の総称である．

Marin らは，ソフトウェアの機能を表しているメソッドはソフトウェア中の様々な場所から呼び出される傾向にあることを利用して，メソッドの被呼び出し回数（Fan-In）の値を用いてアスペクトを検出する解析手法を提案している [48].

Breu らは，版管理システムに蓄積された開発履歴を用いて，一貫した振る舞いを実装するためのメソッド群を抽出する手法を提案している [13]．このアプローチは，同じ開発者によって追加された，あるいは短期間に連続して追加されたメソッドは，意図的に一箇所に配置されたと考える．たとえば,「enter と exit は一対で使用しなくてはならない」といった実装上の制約を発見するという用途に適している一方で，開発履歴が十分に蓄積されるまでは適用できないという制限もある．

Krinke は，常にメソッドの先頭あるいは末尾にのみ配置されるようなメソッド呼び出しを抽出し，アスペクトの候補とする手法を提案している [42]．この手法は，AspectJ の `before/after` アドバイスによるアスペクト抽出を前提としている．

1.4 コーディングパターンに関する問題点とその解決

本論文で対象としているパターンは，ソフトウェアの「実装」工程における，コーディングのパターンやイディオムである．コーディングパターンの詳細な説明は，第 2 章で行う．

コーディングパターンは，本論文ではソースコードからシーケンシャルパターンマイニングの手法を用いて抽出する．抽出されたパターン数は，数千から数万という膨大な数となり，検出結果の中には大量の類似パターンが含まれる．機械的に抽出したコーディングパターンの中から，開発者にとって有益なパターンを選び出す作業を支援するために，次の一連の研究を行った．

コーディングパターンの検出結果から，再利用可能な要素としてのパターンを取り出すために問題となる点を挙げる．

問題点 1 コーディングパターンには，どのような種類があるのか判明していない．

問題点 2 コーディングパターンを分類するには，開発者による手作業の調査が必要である．

問題点 3 類似するパターンが多数存在するため，類似パターンの中から再利用可能な部品を選択することが困難である．

これらの問題点を解決するために，次に示す 3 つの研究を行った．

コーディングパターンに含まれるパターンの種類の分析

プログラミング言語 Java により記述された 6 種類のオープンソースシステムのソースコード中からコーディングパターンを検出した．検出されたコーディングパターンの中から，サンプルを抽出して手作業で調査を行った．その結果，アプリケーションの特定機能の実装に関連したコーディングパターンの存在を確認できた．その他にも，ログ出力に関連したパターン，例外処理に関連したパターン等を確認した．

メトリクスによるコーディングパターンの分析

コーディングパターンの分類作業を支援するために，コーディングパターンの特徴を数値的に表すメトリクスを提案し，コーディングパターンのメトリクス計測ツールを実装した．4 つの Java のオープンソースプロジェクトのソースコードに対してパターンマイニングを行い，コーディングパターンを検出した．コーディングパターンについてメトリクスを計測し，特徴的なパターンを分類できた．

コーディングパターンは，要素の列として検出されるため，要素の順序が入れ替われば別パターンとして検出される．また，基本パターンに，要素が付加された派生パターンが大量に検出される．パターンの再利用を試みる場合には，大量の類似パターンの中から再利用に必要十分な要素列のパターンを選択する必要がある．ソフトウェアの開発履歴において，あるコーディングパターンが長期間登場していれば，そのパターンは安定しており，再利用に適していると考えられる．そこで，コーディングパターンの安定性を，コーディングパターンが登場するソフトウェアのバージョン数という形で定量的に定義した．そして，10 種類のオープンソースソフトウェアから検出された，コーディングパターンの登場するバージョン数を調査した．その結果，全バージョンに登場するコーディングパターンは少数であり，大部分のコーディングパターンは不安定であることが判明した．

以降，第 2 章ではコーディングパターンについて解説し，第 3 章ではコーディングパターンの種類の調査を行い，第 4 章ではメトリクスを用いたコーディングパターンの分析を行い，第 5 章ではコーディングパターンの不変性について調査を行った．そして最後に，第 6 章では，全体のまとめを述べる．

第2章　ソースコードから抽出されるコーディングパターン

2.1　コーディングパターンの検出

Java では，オブジェクトの初期化に必要な処理をコンストラクタとして記述し，一連の処理をメソッドとして記述する．コーディングパターンの検出では，コンストラクタやメソッドを，ひとつの完結した処理の単位であると考える．

コーディングパターン抽出の流れを図 2.1 に示す．コーディングパターンの抽出では，まず，解析対象のソースコードを，特徴シーケンスを抽出する単位であるコンストラクタやメソッドに分割して，それぞれ独立したシーケンスへと変換し，シーケンスデータベースを構築する．シーケンスへの変換時にソースコード中から抽出する特徴情報は，次のとおりである．

- メソッド呼び出し情報
- コンストラクタ呼び出し情報
- 繰り返し情報
- 条件分岐情報
- 例外処理情報
- スレッド同期処理情報

次に，特徴データベースに対して，パターンマイニングを適用しコーディングパターンを抽出する．そして，不要パターンの除去を行って最終的なコーディングパターンが決定される．

本研究では，プログラミング言語で記述されたソースコードから，コーディングパターンの検出を行う．本研究で用いるパターンマイニングツールは，Java SE 6 環境向けのソースコードに対応している．他の言語で記述されたソースコードからコーディングパターンを行う場合には，その言語に対応したソースコードの正規化ルールを作成する必要がある．

2.1.1　ソースコードの正規化

ソースコードの正規化では，Java プログラムの各メソッドをそれぞれ独立したコード片とみなし，メソッド単位で，メソッド呼び出し要素と制御構造要素の列へと変換する．1

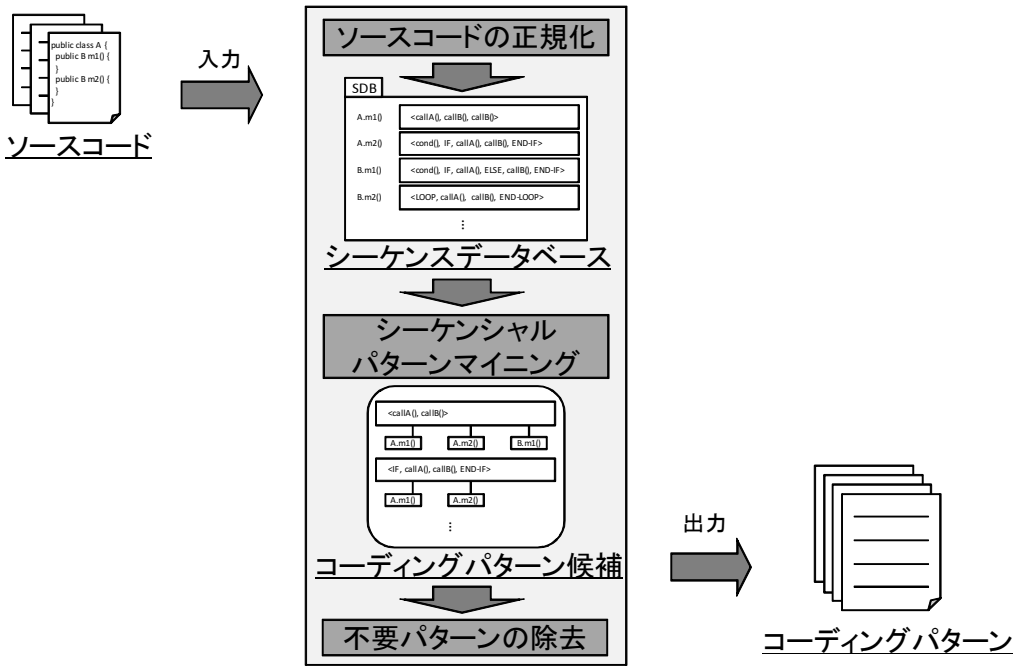


図 2.1: コーディングパターン抽出の流れ

つの Java プログラムは，一般に多数のメソッドから構成されているため，この正規化によって，Java プログラムは，要素列のデータベース（Sequence Database）へと変換される．この要素列データベースが，次のステップであるパターンマイニングの対象となる．

メソッドの正規化処理は，次の 3 つの正規化によって構成される．

- メソッド呼び出しの正規化
- コンストラクタ呼び出しの正規化
- 制御構造の正規化

これらの正規化処理について順に説明する．

メソッド呼び出しの正規化

ソースコード中に登場するメソッド呼び出し式を，メソッド呼び出し要素へと変換する．メソッド呼び出し要素は，メソッド名，戻り値の型，引数の型名の列を保持する．

ここで重要な点は，メソッドが所属するクラス名を持たないことである．Java プログラムでは，メソッド呼び出しは動的束縛によって解決される．すなわち，あるクラス A のメソッド m を呼び出す，とソースコード上に記述されているとき，A を継承したサブクラスのメソッド m が実行される可能性がある．そのため，ソースコードから得られるクラス名を保持していても，実際に呼び出されるクラスの名前とは一致しないことがある．デー

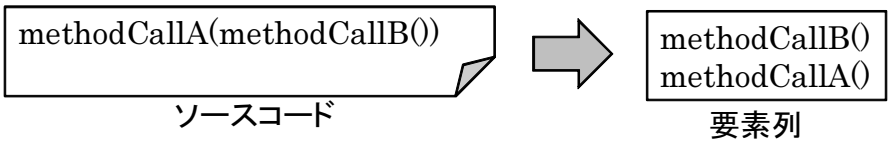


図 2.2: 引数中でのメソッド呼び出し

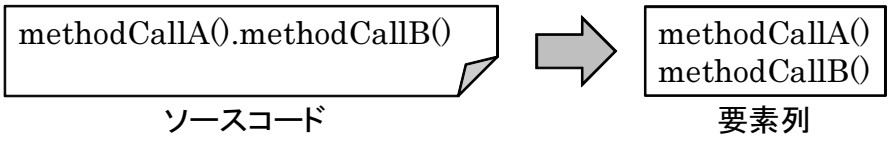


図 2.3: 戻り値に対するメソッド呼び出し

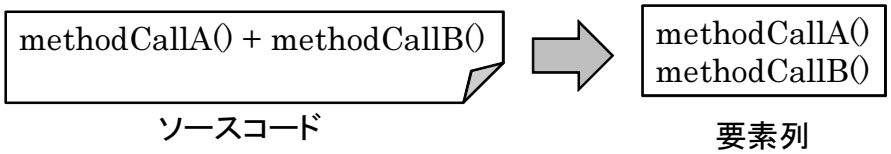


図 2.4: 式中の複数メソッド呼び出し

タフロー解析を用いると，実際に起こりうる動的束縛を解決することができる [65] が，このような計算は，プログラム全体のソースコードを解析する必要があるため，計算コストが大きい．そこで，クラス名を持たないという解決策を採用している．なお，クラス名を持たないことは，継承関係にないクラスに同一のコードが複製されている場合にも対処できるという点で有益である [29].

メソッド呼び出しは，構文上は，式（Expression）であり，他の式の部分式として登場しうる．これに対しては，以下の 2 つのルールを用いて対処する．

- メソッド呼び出し要素の順序は，原則として，演算子の優先順位などによって定まる式の評価順序に従う．
- 式の評価順序が言語仕様で定義されていないとき，ソースコードの配置順序に従う．

たとえば，図 2.2 のように,「methodCallA()」の引数が「methodCallB()」であるときは,「methodCallB を呼び出して，その戻り値を methodCallA に渡す」という順序が規定されている．そのため，正規化された要素列は「methodCallB, methodCallA」となる．

図 2.3 のように,「methodCallA()」の戻り値として得られたオブジェクトに「methodCallB()」を呼び出す場合は，正規化された要素列は「methodCallA, methodCallB」となる．

一方，図 2.4 のように，methodCallA と methodCallB の結果を単純に加算している場合，どちらを先に呼び出すかは規定されていない [26]．本研究では，このときは，ソースコード上で先に登場する methodCallA のほうが先に呼び出されると考え，メソッド呼び出し要素を配置する．

コンストラクタ呼び出しの正規化

ソースコード中に登場するコンストラクタ呼び出し式を，コンストラクタ呼び出し要素へと変換する．コンストラクタ呼び出し要素は，呼ばれたコンストラクタを識別できるように，クラスの完全修飾名と引数の型名の列を保持する．コンストラクタには，メソッドのような個別の識別子が存在しないため，パッケージ名とクラス名からなるクラスの完全修飾名を識別のために使用する．

メソッド呼び出しの場合と同様に，式の評価順序，ソースコード上での配置順序に従い要素を抽出する．

制御構造の正規化

ソースコードの正規化処理では，条件分岐，繰り返し，例外処理に関連した特殊な要素を使用する．最新のコーディングパターン検出ツールで対応している正規化ルールは，表 2.1 に示すとおりである．表 2.1 の source 中の “<” と “>” で囲まれた部分は，sequence 中の該当部分に展開される．

繰り返し処理の正規化

メソッド中に登場した for 文，while 文，do-while 文を正規化する．繰り返し処理の正規

表 2.1: ソースコードの正規化ルール

No.	Source	Sequence
1	for (<init>; <cond>; <inc>) <body>	⟨<init>, <cond>, LOOP, <body>, <inc>, <cond>, END-LOOP⟩
2	for (: <init>) <body>	⟨<init>, LOOP, <body>, END-LOOP⟩
3	while (<cond>) <body>	⟨<cond>, LOOP, <body>, <cond>, END-LOOP⟩
4	do <body> while (<cond>)	⟨LOOP, <body>, <cond>, END-LOOP⟩
5	if (<cond>) <then> else <else>	⟨<cond>, IF, <then>, ELSE, <else>, END-IF⟩
6	<cond> ? <then> : <else>	⟨<cond>, IF, <then>, ELSE, <else>, END-IF⟩
7	try <try> catch ( ) <catch> ... finally <finally>	⟨TRY, <try>, CATCH, <catch>, ..., FINALLY, <finally>, END-TRY⟩
8	throw <exp>	⟨<exp>, THROW⟩
9	synchronized(<exp>) <body>	⟨<exp>, SYNCHRONIZED, <body>, END-SYNCHRONIZED⟩
10	synchronized *return-type method-name*( ) <body>	⟨SYNCHRONIZED, <body>, END-SYNCHRONIZED⟩

化規則を表 2.1 の No. 1〜4 に示す.

制御構造要素 LOOP，END-LOOP は，繰り返し実行される処理の範囲に対応する．たとえば for 文の場合，繰り返し実行の対象である body 部を実行し，カウンタのインクリメントなどが記述される update 部を実行したのち，条件式が評価される，という一連の処理がループ 1 回の実行単位であると考えている．この正規化ルールは，まったく同一のループ構造を，for や while など，異なる制御文を使って書き直した場合に対応するよう定義している．

条件分岐の正規化

ソースコード中の，if 文と三項演算子は条件分岐として正規化する．条件分岐の正規化規則は，表 2.1 中の No. 5〜6 が該当する．条件分岐の正規化を行うことで，パターンマイニングの段階で if 文によって表現された条件分岐と，三項演算子によって表現された条件分岐を同一視してパターンの抽出を行うことができる．

Java の論理演算子，`&&` や `||` は，短絡評価を引き起こす．たとえば 2 つの式 a, b を `a && b` というように連結した場合，式 a が偽であったときは，式 b を評価しない．これは条件分岐の一種とも考えることができるが，`if` 文などの条件式に頻繁に出現する記法であることから，条件分岐ではない単純な式と同様に扱い，正規化の結果には影響を与えない．

また，switch 文を用いた条件分岐には対応していない．ソースコード中に switch 文が出現した場合は，その中に含まれている文がそのまま並べられているものとして，要素列を抽出する．

例外処理の正規化

try 文のような制御フローに影響を与える構文が存在する．try 文の正規化には，表 2.1 の No. 7 を利用する．try 文中に複数の catch 節が存在する場合は，順番に catch 節の数だけ”CATCH, <catch>”を展開する．

たとえば，”try {a();} catch (AException ex1) {b();} catch (BException ex2) {c();} finally {d();}” というソースコードは，$\langle TRY, a(), CATCH, b(), CATCH, c(), FINALLY, d(), END-TRY \rangle$ と正規化される．

また，プログラム中で意図的に例外を発生させる場合に throw 文を用いる．throw 文の正規化には，表 2.1 の No. 8 を用いる．ソースコード中の”<exp>”部分に，発生させる例外を生成する処理等が記述されるため，例外の生成と例外の発生（THROW）の実行時の順序関係を考慮して，$\langle < exp >, THROW \rangle$ という順序のに正規化する．

スレッド同期処理の正規化

Java では，マルチスレッドの同期処理を行うために，synchronized キーワードを用いる．synchronized 文を正規化する場合は，表 2.1 の No. 9 を，synchronized 修飾子が付加されたメソッドを変換する場合は，No. 10 を使用する．synchronized 修飾子が付加されたメソッ

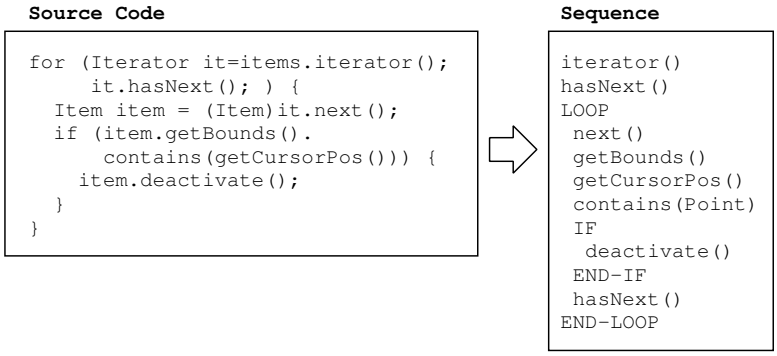


図 2.5: ソースコードの正規化例

ドでは，メソッド内の処理すべてが同期処理の対象となるため，シーケンスの最初と最後に，SYNCHRONIZED と END-SYNCHRONIZED を置く．

ソースコードの正規化例

図 2.5 に正規化の例を示す．図のコード片を実行すると，まず最初に `for` 文の初期化子で `iterator` メソッドが呼び出される．続いて，ループ本体を実行するかどうかを判定するために `hasNext` が呼び出され，ループ本体の実行に進む．ループの本体では，まず `next` を呼び出し，続く一連の処理を実行する．そして，ループ本体の末尾において，ループの継続を判定するために再度 `hasNext` を呼び出し，継続するのであればループ本体の先頭，すなわち `next` 呼び出しに戻る．図 2.5 の右側に示す要素列の順序は，このような制御フローを反映したものとなっている．

2.1.2 コーディングパターンのマイニング

シーケンシャルパターンマイニング

シーケンシャルパターンマイニング [2] は，与えられたシーケンス集合から頻出するすべてのサブシーケンスを抽出する手法である．

定義

要素集合 $I = \{i_1, i_2, \cdots, i_n\}$ が与えられた時，長さ l のシーケンス s は，$s_1 \in I, s_2 \in I, \cdots, s_l \in I$ を満たす要素の列 $s = \langle s_1, s_2, \cdots, s_l \rangle$ と表される．そして，パターンマイニングの対象となるシーケンスデータベース SDB は， シーケンス ID（sid）とシーケンス (s) のタプル (sid, s) の集合である．

2 つのシーケンス $s = \langle s_1, s_2, \cdots, s_l \rangle$ と $s' = \langle s'_1, s'_2, \cdots, s'_m \rangle$ が与えられたとき，$e_1 = s'_{I_1}, s_2 = s'_{I_2}, \cdots, s_l = s'_{I_l}$ を満たす整数 $1 \leq I_1 < I_2 < \cdots < I_l \leq m$ が存在するならば，s

```
sequential-pattern-mining(SDB, min_sup, FS)
    Input: a sequece database SDB, a minimum support threshold min_sup
    Output: frequent sequences FS
1:      FS ⇐ ∅;
2:      call frequent-sequences(nil, SDB, min_sup, FS);
3:      return FS;

frequent-sequences(s_p, PSDB, min_sup, FS)
    Input: a prefix sequence s_p, a s_p-projected sequence database PSDB,
           a minimum support threshold min_sup
    Output: frequent sequences FS
4:      if s_p is not nil
5:          FS ⇐ FS ∪ s_p;
6:      FI ⇐ frequent-items(PSDB, min_sup);
7:      if FI is empty
8:          return;
9:      for-each item in FI
10:         item_SDB ⇐ projected-database(PSDB, item);
11:         call frequent-sequences(s_p + ⟨item⟩, item_SDB, min_sup, FS);
```

図 2.6: シーケンシャルパターンマイニングアルゴリズムの概要

は s' のサブシーケンスである．また，s' は s のスーパーシーケンスである．これらのスーパーシーケンス，サブシーケンスの関係にある 2 つのシーケンスを，$s \sqsubseteq s'$ と表現する．たとえば，$\langle a, c, d\rangle$ は，$\langle a, b, c, d, e\rangle$ のサブシーケンスであるため，$\langle a, c, d\rangle \sqsubseteq \langle a, b, c, d, e\rangle$ と表記する．

s_α の SDB 中での出現回数をサポート値と呼び，次の式で定義される．

$$support^{SDB}(s_\alpha) = |\{(sid, s)|(sid, s) \in SDB \wedge s_\alpha \sqsubseteq s\}|$$

そして，サポート値の閾値 (min_sup) が与えられた時，s_α は，$support^{SDB}(s_\alpha) \geq min\_sup$ の時，頻出するという．

シーケンシャルパターンマイニングアルゴリズムの概要を図 2.6 に示す．sequential-pattern-mining 関数は，シーケンスデータベースとサポート値の閾値を入力として受け取り，内部で frequent-sequences 関数を呼び出す（2 行目）ことにより，頻出パターンを計算し出力する．frequent-sequences 関数は，自分自身を再帰的に呼び出すことで，パターンマイニングを進める．6 行目で呼び出している frequent-items 関数は，与えられたシーケンスデータベース中で頻出する要素の集合を返す．これらの頻出するそれぞれの要素について，projected-database 関数を呼び出し（10 行目）て射影データベースを作成し，11 行目で frequent-sequences 関数を再帰的に呼び出し，さらに長いパターンが存在するか調査する．

飽和系列の抽出

サポート値が同じパターンのシーケンスのうち，スーパーシーケンスとなるシーケンス S_β が存在しない場合，S_α は，飽和系列である．具体的には，以下の条件式となる．

$$\nexists s_\beta \quad | \quad s_\alpha \sqsubset S_\beta \quad \wedge \quad support^{SDB}(s_\alpha) = support^{SDB}(s_\beta)$$

飽和系列のみを求めるアルゴリズム [71, 73] も提案されているため，これらのアルゴリズムを採用すれば，より効率的に飽和系列のみを検出できる．

本研究では，すべてのパターンを検出したのちに，飽和系列のみ取り出している．

パターンマイニングの例

本研究では，シーケンシャルパターンマイニングを，コーディングパターンの抽出に使用する．シーケンシャルパターンマイニングとは，与えられた配列データベースから頻出する部分列をパターンとして抽出する手法である [2]．頻出する部分列の個々の出現は，順序が同一でなければならないが，不連続なものでよい．

本研究では，シーケンシャルパターンマイニングのアルゴリズムの 1 つである PrefixSpan を使用する．このアルゴリズムは，入力として配列データベース SDB と最小出現回数 min_sup を取り，以下の手順でシーケンシャルパターンの集合を抽出する [58].

まず最初に，配列データベースに出現する各要素の出現回数（その要素を含む配列の数）を数え上げ，最小出現回数 min_sup 以上の要素を，長さ 1 のパターンとする．たとえば，$SDB = \{(1, \langle a, b, c, d\rangle), (2, \langle a, e, a, d, e\rangle), (3, \langle d, a, f, b\rangle), (4, \langle a, c, d\rangle)\}$，$min\_sup = 2$ が与えられたとき，4 つの長さ 1 のパターン $\langle a\rangle : 4, \langle b\rangle : 2, \langle c\rangle : 2, \langle d\rangle : 4$ が得られる．ここで，各パターンは，パターンの要素列 $pattern$ と出現回数 $support$ を用いて $\langle pattern\rangle : support$

表 2.2: PrefixSpan アルゴリズムが作成する射影データベース

prefix	*prefix*-Projected Database	Patterns
	$(1, \langle a, b, c, d\rangle), (2, \langle a, e, a, d, e\rangle),$ $(3, \langle d, a, f, b\rangle), (4, \langle a, c, d\rangle)$	$\langle a\rangle : 4, \langle b\rangle : 2,$ $\langle c\rangle : 2, \langle d\rangle : 4$
$\langle a\rangle$	$(1, \langle b, c, d\rangle), (2, \langle e, a, d, e\rangle),$ $(3, \langle f, b\rangle), (4, \langle c, d\rangle)$	$\langle a, b\rangle : 2, \langle a, c\rangle : 2,$ $\langle a, d\rangle : 3$
$\langle b\rangle$	$(1, \langle c, d\rangle)$	
$\langle c\rangle$	$(1, \langle d\rangle), (4, \langle d\rangle)$	$\langle c, d\rangle : 2$
$\langle d\rangle$	$(2, \langle e\rangle), (3, \langle a, f, b\rangle)$	
$\langle a, b\rangle$	$(1, \langle c, d\rangle)$	
$\langle a, c\rangle$	$(1, \langle d\rangle), (4, \langle d\rangle)$	$\langle a, c, d\rangle : 2$
$\langle a, d\rangle$	$(2, \langle e\rangle)$	
$\langle c, d\rangle$	ϕ	
$\langle a, c, d\rangle$	ϕ	

と表記している．この例では，ただ 1 つの配列にしか出現しない $\langle e \rangle$ と $\langle f \rangle$ がパターンから除外されている．

PrefixSpan は，続いて，長さ k のパターンから長さ $k+1$ のパターンを構築する手順を，新たなパターンが抽出されなくなるまで繰り返し実行する．まず長さ k のパターンに対し，**射影データベース**（projected database）を作成する．これは，配列データベース SDB の各配列に対し，パターンの最初の出現以降の要素列だけを取り出す操作である．たとえば，パターン $\langle a \rangle$ に関する SDB の射影データベースは，$\{(1, \langle b, c, d \rangle), (2, \langle e, a, d, e \rangle), (3, \langle f, b \rangle), (4, \langle c, d \rangle)\}$ という 4 つの配列となる．次に，得られた射影データベース中での各要素の出現回数を数え上げ，min_sup 個以上の配列に出現した要素を用いて長さ $k+1$ のパターンを構築する．a に関する射影データベースでは，b，c が 2 つの配列に，d が 3 つの配列に出現していることから，$\langle a, b \rangle : 2$，$\langle a, c \rangle : 2$，$\langle a, d \rangle : 3$ というパターンが得られる．SDB から抽出される長さ k のパターン（*prefix*），それに関する射影，そして新たに得られる長さ $k+1$ のパターンを，表 2.2 に示す．記号 ϕ は，射影の結果が空であることを意味している．

2.1.3 不要パターンの除去

パターンマイニングの結果には，パターンとして不適切なものが含まれている．以下のような，パターンは，不適切であると判断し，パターンマイニングの結果から取り除く．

パターン長が指定された閾値より短いパターン

パターンマイニングツールの引数として指定された，パターン長の閾値を下回るパターンは結果から取り除く．

飽和系列でないパターン

パターンのシーケンスが飽和系列でない場合は，別のパターンのシーケンスの部分列となっており，冗長であるため結果から取り除く．

制御構造の開始要素と終了要素の間にメソッド呼び出しが含まれないパターン

制御構造は，その制御構造が関連しているメソッド呼び出しのを実行するかどうかを決定したり，関連しているメソッド呼び出しを繰り返し実行するというように，メソッド呼び出しに関連して，初めて意味を持つので，制御構造の開始要素と終了要素の間にメソッド呼び出しが含まれないパターンは，不適切であるため結果から取り除く．

制御構造の割合が 70% を超えるパターン

パターンの構成要素中の制御構造の割合が，メソッド呼び出しに関連したパターンである可能性は低いため削除する．

この 70%という割合は，経験的に定めたものである．

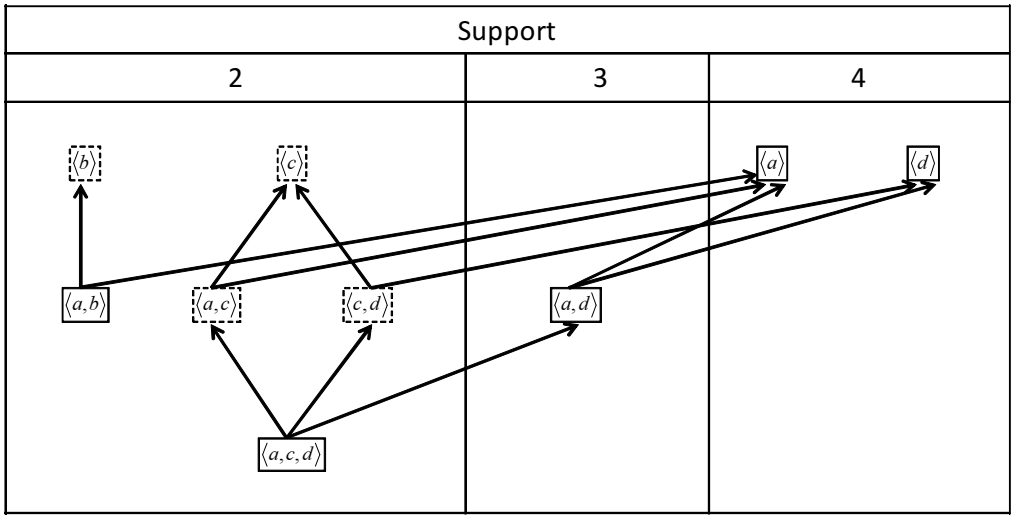


図 2.7: パターン間の関係

制御構造の開始要素と終了要素の対応がとれていないパターン

制御構造の開始要素と終了要素は，必ず対応して登場するため，この対応関係の崩れたパターンは意味をなさないため削除する．

制御構造の中間要素が対応する開始要素と終了要素に囲まれていないパターン

条件分岐の ELSE 要素は，その ELSE 要素が対応する，IF 要素と END-IF 要素の間に登場しなければ意味を成さない．同様に，例外処理の CATCH 要素，FINALLY 要素は，TRY 要素と END-TRY 要素の間に登場しなければならない．ELSE 要素，CATCH 要素，FINALLY 要素のような中間要素が，単独で登場しているパターンは，削除対象となる．

2.2 パターン間の関係

表 2.2 中の Pattern 列に登場するパターンを pattern の要素列を基準として $subpattern \longleftarrow superpattern$ の関係をグラフ化したものを，図 2.7 に示す．なお，推移的に到達可能なノード同士を結ぶ辺は省略している．

2 つの異なるパターン α と β が，$\alpha \sqsubset \beta$ の関係にある時，サポート値の関係は $support(\alpha) \geq support(\beta)$ となる．また，α，β のパターン長の関係は $length(\alpha) < length(\beta)$

図 2.7 中の破線で囲まれたパターンは，サポート値が等しいパターン中にスーパーパターンが存在している（$\langle b\rangle \sqsubset \langle a,b\rangle$，$\langle c\rangle \sqsubset \langle a,c\rangle$，$\langle c\rangle \sqsubset \langle c,d\rangle$，$\langle a,c\rangle \sqsubset \langle a,c,d\rangle$，$\langle c,d\rangle \sqsubset \langle a,c,d\rangle$）ため，飽和系列でない．

よって，最終的に出力されるパターンは，$\langle a\rangle$，$\langle d\rangle$，$\langle a,b\rangle$，$\langle a,d\rangle$，$\langle a,c,d\rangle$ の 5 つである．

第3章　コーディングパターンの種類の分析

3.1　はじめに

近年，多くのソフトウェア開発にオブジェクト指向プログラミングが使用されている．オブジェクト指向プログラミングの利点として，継承や多態性など，モジュール化された部品を活用するための機構がある．しかし，ソフトウェアのすべての機能を完全にモジュール化することはできず，例えば，ロギングや同期処理といった機能は，横断的関心事と呼ばれ，複数のモジュールに分散した定型的なコードとして実装されることが知られている [38, 46].

複数のモジュールに分散配置されるコードは，元となるソースコード片を開発者が複製し，配置先の状況に応じて適宜改変を加えるという方式で作成されることが多く，一群の定型的なコード片，すなわち**コーディングパターン**を構成する．コーディングパターンに属するコード片は互いに類似しており，また，多くは同一の機能を実現している．そのため，コード片の 1 つを変更する場合，開発者は，同一のパターンに属する他のコード片に対しても一貫した変更を適用するべきか，検討する必要がある [10, 23].

本研究では，コーディングパターンを，メソッド呼び出し要素と，それに付随する制御構造要素（条件分岐と繰り返し文）の定型的な列と捉え，コーディングパターンをソースコードから自動的に抽出するパターンマイニング手法を提案する．具体的には，適用対象として選択した Java のためのソースコード正規化ルールを用意し，Java の各メソッドを，特徴列へと変換する．その結果得られた特徴列データベースに，シーケンシャルパターンマイニングのアルゴリズムの 1 つである PrefixSpan[58] を適用し，頻出する部分列をコーディングパターンとして抽出する．シーケンシャルパターンマイニングは，マイニング対象である要素の順番が同じであれば，無関係な要素がいくつ間に追加されても，抽出されるパターンは影響を受けない．横断的関心事のように複数のモジュールに分散したコードは，しばしば他の機能に属するコードと混ざり合うことが知られている [38] が，そのようなコードからもパターンの抽出が可能である．

本手法で抽出されるコーディングパターンが横断的関心事などの保守に有用であるかを調査するため，提案手法をツールとして実装し，6 つの Java プログラムに対して適用した．頻出する 55 のパターンを手作業で調査した結果，ロギングのパターンを始めとする，モジュール化が困難なアプリケーションの機能をコーディングパターンとして抽出していることを確認した．

以降，3.2 節では，研究の背景と，本研究で使用したパターンマイニングのアルゴリズム

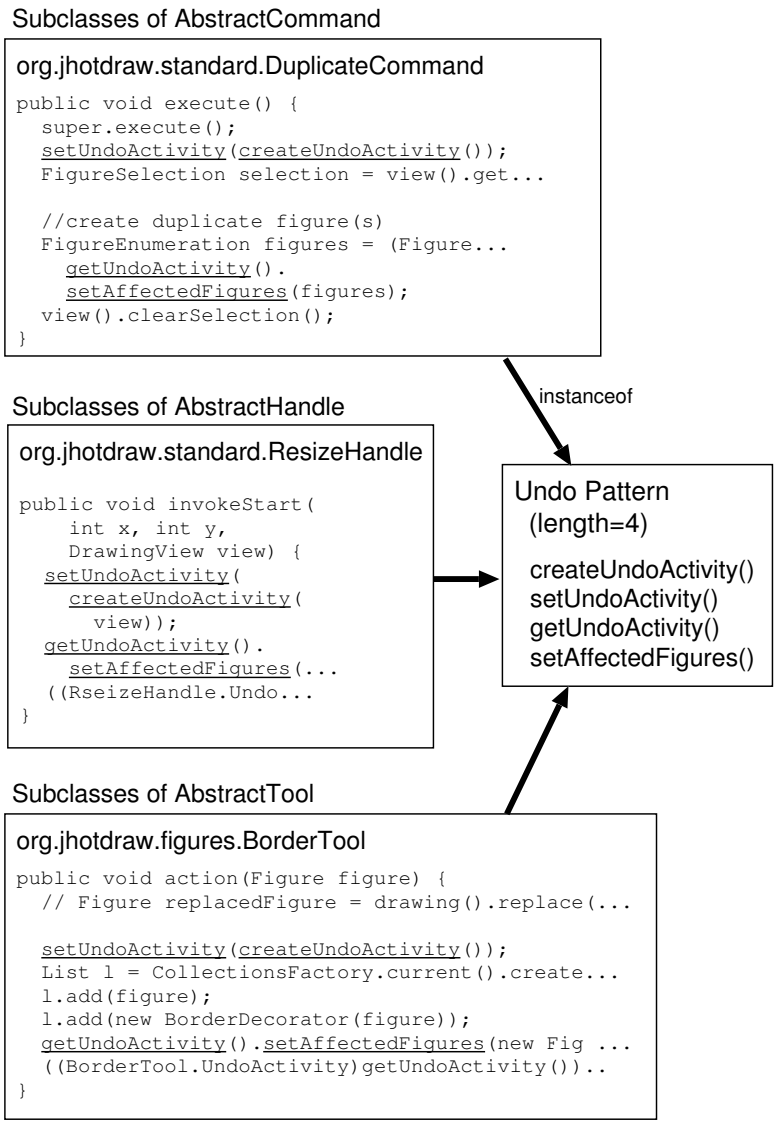


図 3.1: JHotDraw 5.4b1 の編集内容を「元に戻す」実装パターン

について述べる．3.3 節では 6 つの Java プログラムへの適用結果を述べ，3.4 節では，ケーススタディの結果について考察する．3.5 節で関連研究について述べ，3.6 節でまとめと今後の課題を述べる．

3.2 コーディングパターン

ソフトウェアの機能をモジュールとして分割，実装することは，保守性や拡張性を向上するために重要である．しかし，ソフトウェアのすべての機能を完全にモジュール化することはできず，そのような機能は，複数のモジュールに分散した定型的なコードとして実

装される [46]．本研究では，そのように分散して実装される定型的なコードを，コーディングパターンと呼ぶ．

図 3.1 は，図形エディタ JHotDraw 5.4b1 における，様々な編集操作を「元に戻す」ことを可能とするための実装の一部である．編集操作ごとに実行している処理は異なるが，下線で示されるメソッド呼び出し列が共通している．このようなメソッド呼び出しのパターンを知ることは，どのように「元に戻す」仕組みが実装されているかを理解するために，また，新たな編集操作を実装するときに有用である．

定型的なメソッド呼び出し列は，しばしば，特定の制御構造を伴った記述となる．たとえば，サーバソフトウェアである Apache Tomcat には，デバッグ用のメッセージを出力する `debug` メソッドの呼び出しが多数記述されている．各呼び出しには，メッセージを記録するかどうかを判定する `isDebugEnabled` メソッドの呼び出しと `if` 文による条件分岐が付属しており，条件分岐はメッセージ出力処理の重要な構成要素となっている．そこで，本研究では，コーディングパターンをメソッド呼び出しとそれに付随する制御構造要素の列として捉える．

既存の定型的なコードに基づいて新たなコードを記述するとき，開発者は，しばしば既存のコード片を複製し，改変を加えるという形式での記述を行う [39]. このようなソースコードの複製は，互いに類似したコード，すなわちコードクローン [10, 37] の一種であると考えられる．しかし，コードクローン検出ツール，たとえば CCFinder [37] は，図 3.1 の下線で示されたメソッド呼び出しのように，他の機能の一部として埋め込まれた短いコード片を発見するには不向きであり，コードクローン検出手法のみでは横断的関心事の実装を特定することが困難であると指摘されている [14].

3.2.1 アスペクトマイニング

本研究で抽出することを目標としているコーディングパターンのように，モジュール化することが困難な機能，すなわち横断的関心事を実装するコードを発見することに特化された手法は，アスペクトマイニングと呼ばれる．

従来のアスペクトマイニング手法は，いずれも，横断的関心事の典型的な実装法に関する経験的な指標を用いて，横断的関心事の候補を探索する．Marin らは，1 つの横断的関心事，たとえばロギング処理を実装する多数のコード片が，`Logger.log()` のような特定のメソッドを呼び出すことから，被呼び出し回数が多いメソッドを横断的関心事の候補とする手法を提案した [48]．また，Breu らは，あるメソッド呼び出しが単独ではなく，他のメソッドと一対になって初めて意味を持つものかを調べる手法を提案している [13]．一方，Krinke は，あるメソッドの呼び出しが，必ず他のメソッドの先頭あるいは終端に配置されている，という制御フロー上の特徴を使用することを提案している [42].

本研究で着目するコーディングパターンは，Marin らの着眼点と同様，類似したコードが多数書かれていることを手がかりとするが，メソッド呼び出し列を探索し，またそれに付随する制御構造要素をパターンの一部として含める点で，従来手法とは異なる．

3.2.2 コーディングパターンのグループ化

フィルタリングを適用した後，パターンのグループ化を行う．これは，あるパターンが存在するとき，そのパターンと少なくとも同数のインスタンスを持つ部分パターン（より短いパターン）が存在するためである．たとえば，4 要素からなるパターン $\langle a, b, c, d\rangle$ が存在するとき，3 要素からなる部分パターン $\langle a, b, c\rangle$，$\langle a, b, d\rangle$，$\langle a, c, d\rangle$，$\langle b, c, d\rangle$ もまた存在する．これらのパターン群を同一グループに含めるため，あるパターン p_1 と p_2 が相互に重なるとき，すなわち，p_1 のインスタンスの少なくとも 1 つの要素が p_2 のインスタンスの要素でもあるとき，それらのパターンは同一のグループとする．

3.3 適用実験

3.3.1 実験方法

提案手法の有効性を確認するため，提案手法の一連の手順をツールとして実装し，6 つの Java プログラム（JHotDraw[32]，jEdit[31]，Azureus[1][9]，Apache Tomcat[6]，ANTLR[4]，SableCC[60] ）に対して $min\_sup = 10$，$min\_len = 4$ という設定で適用した．対象プログラムの名称，バージョン，規模（LOC），抽出されたパターン数およびグループ数を表 3.1 に示す．

抽出されたパターングループのうち，JDK 標準ライブラリに含まれるメソッド呼び出しのみからなるパターンを除いて，出現回数での上位 5 グループを調査した．出現回数での上位を調査したのは，Marin らの手法において，被呼び出し回数が多いメソッドに横断的関心事との関連性があることが指摘されているためである [48]．抽出されたグループすべてを手作業で調査することは現実的ではなく，調査の労力の都合上，上位 5 件と決定した．

6 つのソフトウェアから上位 5 件のパターングループを選択したところ，Apache Tomcat には出現回数が第 5 位のパターングループが 2 つ存在したため，調査対象となったパターングループの総数は 31($6 \times 5 + 1$) となった．各グループからは調査対象として，インスタンス数が最大であるパターンと，最長（要素数が最大）のパターンを抽出した．インスタ

表 3.1: 対象ソフトウェア

Name	Version	LOC	#Pattern	#Group
JHotDraw	7.0.9	90,166	137	37
jEdit	4.3pre10	168,335	747	33
Azureus	3.0.2.2	552,021	4,682	128
Apache Tomcat	6.0.14	313,479	1,415	85
ANTLR	3.0.1	59,687	352	29
SableCC	3.2	35,388	162	18

[1]Azureus は，ソフトウェアの名称が Vuze へと変更されている．

ンス数が最大のパターンが，同時にグループ内で最長であるようなグループが 7 個あったため，調査したパターンの数は 55 個 ($31 \times 2 - 7$) となった．

3.3.2 抽出されたコーディングパターン

表 3.2: 調査したコーディングパターン

ID	Sup.	Len.	Elements	Type
JHotDraw 8S	19	4	LOOP / willChange / changed / END-LOOP	(3)
JHotDraw 8L	11	5	LOOP / willChange / transform / changed / END-LOOP	(3)
jEdit 1S	55	4	openNodeScope / jjtreeOpenNodeScope / closeNodeScope / jjtreeCloseNodeScope	(3)
jEdit 1L	10	12	openNodeScope / jjtreeOpenNodeScope / jj_consume_token / Expression / jj_consume_token / IF / clearNodeScope / ELSE / popNode / END-IF / closeNodeScope / jjtreeCloseNodeScope	(3)
jEdit 3S	34	4	IF / getToolkit / beep / END-IF	(2)
jEdit 3L	10	6	isEditable / IF / getToolkit / beep / END-IF / remove	(2)
jEdit 9S	25	4	isEditable / IF / beep / END-IF	(2)
jEdit 9L	10	5	isEditable / IF / beep / END-IF / setCaretPosition	(2)
Azureus 2S	151	4	enter / iterator / next / exit	(3)
Azureus 2L	10	10	enter / iterator / hasNext / LOOP / next / IF / add / END-IF / END-LOOP / exit	(3)
Azureus 4S	140	4	size / LOOP / printStackTrace / END-LOOP	(4)
Azureus 4L	10	7	size / LOOP / get / printStackTrace / END-LOOP / size / get	(4)
Azureus 5S	119	4	isEnabled / IF / log / END-IF	(1)
Azureus 5L	10	13	log / isEnabled / IF / log / END-IF / isEnabled / IF / log / END-IF / isEnabled / IF / log / END-IF	(1)
Azureus 6S	97	4	getDataSource / setSortValue / isValid / setText	
Azureus 6L	12	5	getDataSource / setSortValue / isValid / getText / setText	
Azureus 8S	85	4	iterator / hasNext / next / printStackTrace	(4)
Azureus 8L	14	7	iterator / hasNext / next / iterator / hasNext / next / printStackTrace	(4)
Tomcat 1S	304	4	isDebugEnabled / IF / debug / END-IF	(1)
Tomcat 1L	10	24	isDebugEnabled / IF / debug / END-IF / ... (the same sequence is repeated 6 times.)	(1)

(続く)

表 3.2: 調査したコーディングパターン（続き）

ID	Sup.	Len.	Elements	Type
Tomcat 6SL	46	4	isPackageProtectionEnabled / IF / doPrivileged / END-IF	(2)
Tomcat 8S	42	4	IF / debug / END-IF / debug	
Tomcat 8L	11	6	IF / debug / END-IF / IF / debug / END-IF	
Tomcat 11S	38	5	isInfoEnabled / IF / getString / info / END-IF	(1)
Tomcat 11L	11	8	getName / getString / isInfoEnabled / IF / getName / getString / info / END-IF	(1)
Tomcat 12S	38	7	createManagedName / findManagedBean / getDomain / IF / getDefaultDomain / END-IF / createObjectName	
Tomcat 12L	19	13	createManagedName / findManagedBean / getDomain / IF / getDefaultDomain / END-IF / createMBean / createObjectName / isRegistered / IF / unregisterMBean / END-IF / registerMBean	
ANTLR 1S	107	4	setErrorListener / newTool / setCodeGenerator / genRecognizer	
ANTLR 1L	10	8	setErrorListener / newTool / setCodeGenerator / genRecognizer / getRecognizer / indexOf / substring / assertEquals	
ANTLR 2S	69	4	setErrorListener / newTool / translate / assertEquals	
ANTLR 2L	10	5	setErrorListener / newTool / translate / assertEquals / checkError	
ANTLR 3S	38	4	LT / match / reportError / recover	(4)
ANTLR 3L	10	11	LA / LT / match / LT / match / LT / match / LT / match / reportError / recover	(4)

調査した 55 個のパターンのうち，特定の機能の実装に関わりがあると判断した 33 個を表 3.2 に示す．残る 22 個は，アプリケーションの具体的な機能に属すると判断できず，実装におけるイディオム，あるいは無意味なパターンであると判断し，表 3.2 からは除外した．たとえば，図 3.2 は，JHotDraw 7.0.9 から抽出されたパターンで，メソッドの戻り値が `null` かどうかを検査し，`null` でなければもう一度そのオブジェクトを取得，処理を続行するという定型的な処理である．

表 3.2 の *ID* 列は，本文中でパターンを識別する ID である．ID は，ソフトウェア名，パターンが所属するグループの番号，そしてアルファベットからなり，アルファベット “S” は，それぞれグループ中で最もインスタンス数が多い（Supported）パターンを，“L” は最も長い（Longest）パターンを意味する．たとえば，パターン “JHotDraw 8L” は，JHotDraw から検出された，8 番目にインスタンスが多いパターンのグループの中で，最も長いパターンである．*Sup* 列はパターンのインスタンス数（そのパターンを含むメソッドの数）を，

```
public class AttributeFieldEventHandler ... {
  protected Set<Figure> getCurrentSelection() {
    if (getCurrentView() != null) {
      return getCurrentView().getSelectedFigures();
    } else {
      return Collections.emptySet();
    }
  }
}
```

図 3.2: JHotDraw から抽出された null 値チェックのパターン

```
public class CompleteWord extends CompletionPopup {
  public static void completeWord(View view) {
    JEditTextArea textArea = view.getTextArea();
    Buffer buffer = view.getBuffer();
    int caretLine = textArea.getCaretLine();
    int caret = textArea.getCaretPosition();
    if(!buffer.isEditable()) {
      textArea.getToolkit().beep();
      return;
    }
    :

public class TextArea extends JComponent {
  public void backspaceWord(boolean eatWhitespace) {
    if(!buffer.isEditable()) {
      getToolkit().beep();
      return;
    }
    :
```

図 3.3: jEdit から抽出されたパターン 9S のソースコード

Len 列はパターンの要素数を，それぞれ示す．*Elements* 列は，パターンの要素を “/” で区切ったリストである．*Type* 列は，複数のプログラムについて調査した結果，複数のプログラムで共通の実装方法だと推測された 4 つのカテゴリの番号を記述している．以下，カテゴリの区分と合わせて，コーディングパターンの例を記述する．

(1) **特定の条件が成立しているとき，複数のメソッド内で追加の処理を行う**．このパターンのカテゴリは，`boolean` を戻り値とするメソッドと，その戻り値によって条件分岐を行う `if` 文，それによって実行される追加の処理が組となっている．具体例としては，パターン Azureus 5S や，Tomcat 1S が該当する．なお，ロギングは横断的関心事の典型例として知られているが，Azureus や Tomcat におけるロギングは多様なメッセージを記録するため，モジュール化は困難である．たとえば，Azureus のソースコードをキーワード検索で調査したところ，`DHTLog.log` というメソッドは 55 箇所で呼び出されているが，51 種類の異なるメッセージが引数として使用されており，`Logger.log` というメソッドの 200 箇所の呼び出しでは 148 種類の異なるメッセージが引数として使用されていた．

(2) **特定の条件が成立しているとき，複数のメソッドの処理を切り替える**．このカテゴリのパターンは，構成要素自体は (1) のパターンと同様であるが，条件分岐の結果がメソッドの処理を完全に切り替えるというものである．たとえば，jEdit 3S，3L，9S，9L のパターン群は， 読み取り専用のバッファに対してテキスト編集操作を実行しようとしたとき，`beep` 音を鳴らして動作をキャンセルする処理を実装している．パターンに該当するソースコードの一部を，図 3.3 に示す．

(3) **複数のメソッドに共通した前処理と後処理を実行する**．このカテゴリのパターンは，それぞれ異なる処理を実行するメソッド群に含まれる，共通の前処理と後処理であった．たとえば，jEdit 1S では構文解析処理の前後に `openNodeScope` と `closeNodeScope` が使用されていた．また，Azureus は，マルチスレッドで処理を実行するため，様々な共有データへのアクセスの前後に `AEMonitor.enter` と `AEMonitor.exit` というメソッドを用いた同期処理を挿入しており，それが Azureus 2S，2L というパターンとして抽出された．

(4) **例外に対して，特定の処理を実行する**．このカテゴリのパターンは，一群のメソッドに対して，`try`/`catch` ブロックと同時に出現し，共通の例外処理を実装していた．たとえば，パターン Azureus 4S は `Debug.printStackTrace` というメソッドによって例外発生の状態を記録しており，パターン ANTLR 3S は `reportError` と `recover` というメソッドによってエラーへの対応を行っている．

その他の，カテゴリ分類を持たないパターンは，それぞれアプリケーション固有の処理を実装している．たとえば，Tomcat 12L は `createMBean` という名前を持つメソッド群にのみ存在する，Managed Bean の処理に関連するパターンである．また，Tomcat 8S，8L は，(1) に分類されたロギングの実装と異なり，`if` 文などを伴わずに `debug` メソッドを呼び出しているケースであった．

なお，本実験で付与したカテゴリ分類は，発見されたパターン群に基づいて便宜的に定めたものであり，網羅的ではない．また，(1) と (2) は相互に排他的であるが，(3) や (4) とは排他的とは限らない．

3.3.3 コーディングパターンを用いた保守支援

本実験で調査した 4 つのカテゴリは，いずれもプログラムの複数のメソッドに一貫した振る舞いを持たせるためのものである．「一貫した振る舞い（Consistent Behavior）」は，Marin らが分類した横断的関心事の 1 つである [47]．一貫した振る舞いは，モジュール化されていない機能を実現するための実装上のルールであり，ソフトウェアの変更を行うときには重要となる．たとえば，jEdit に新たな編集機能を追加する開発者にとっては，「バッファが変更不可能であれば beep 音を鳴らして利用者に通知する」といったルールを知っておくことは重要であるし，また，beep 音を鳴らす基準が変化したときは，これらすべてのコードを一貫した状態に保った変更が必要となる．一方，Tomcat や Azureus においては，新たなコードを追加するごとに，パターンに従って，たとえば `isDebugEnabled` メ

ソッドと debug メソッドを組み合わせて新たなログ収集を行うことになる．このように，カテゴリに分類されたコーディングパターンは，実装上の暗黙のルールを表しており，ソフトウェア保守において有用な情報である．これらのパターンの情報は，開発者がソースコードを理解する，あるいは編集する際に開発者が直接読むことができるように，Marin らの提案する横断的関心事のドキュメント記述手法 [47] を用いてドキュメント化することが有効であると考えられる．

そのほかにも，コーディングパターンの情報は，以下のような既存の手法への入力データとして使用することが考えられる．

パターンの自動リファクタリング 複数のメソッドの一貫した振る舞いは，AspectJ のアドバイスや，デザインパターンの 1 つである Template Method によってリファクタリングできる可能性がある．たとえばカテゴリ (2) には `around` アドバイスを，カテゴリ (1)，(3) には `before`/`after` アドバイスを，それぞれ適用することが考えられる．このようなアスペクト抽出の自動的なリファクタリングは既に研究されており [12, 70]，本研究で得られたコーディングパターンは，これらの手法への入力となりうる．また，カテゴリ (4) についても，例外をアスペクト化するための手順が提案されている [21].

パターンに該当するコードの一括編集 すべてのパターンが必ずしもリファクタリングとは限らない．たとえばロギング処理は,「プログラムが何をしているか」を抽象的に表現した文字列をファイルに記録するため，そのような文字列をアスペクト側で自動的に作成することは困難である．リファクタリングが困難なコーディングパターンを開発者がそのまま保守し続ける場合，本手法で得られたパターン情報をもとに，コード片の共通部分をエディタで一括編集する [28]，あるいはテンプレートを用いたソースコードの同時変更手法 [61] を適用するという対応が考えられる．

現状では，カテゴリが網羅的なものではないため，カテゴリ (1)〜(4) に該当しないパターンであっても，一概に有用でないと判断することはできない．たとえば ANTLR 1S のパターンは，テスト実行用のデータを生成するための手順を表現している．このようなコードは一種の重複コードであり，リファクタリングの対象となりうる [23] ため，パターンとして情報を提示することは有用であると考えられる．一方で，表 3.2 からは取り除いた 22 個のパターンは，特定の機能と結びつかなかった．たとえば，図 3.2 のコード片は，単独でも意味の読み取りが容易であり，また，`null` 値の検査が欠落している可能性は，静的解析手法によって検出することが可能である [57]．このようなパターンは，保守において悪影響を与えるとは考えられないため，パターンマイニングの結果から自動的に除外できることが望ましい．今後，多数のコーディングパターンをカテゴリに自動分類するなどして，カテゴリ (1)〜(4) に該当しないパターンにどのようなものがあるか，調査を行っていく必要がある．

3.4 考察

3.4.1 コーディングパターンの自動分類

実験では，抽出したパターンに対し，個々のインスタンスをソースコード上で確認する作業を行った．このとき，パターンの意味を理解するには，以下の情報が有用であった．

- パターンの要素の一覧．
- パターンのインスタンスを持つメソッドの一覧．
- 上記 2 項目の情報の，同グループ内のパターン間での差分．

ソースコードを正規化する段階では，メソッドが属するクラス名を取り除いているが，パターンを理解する段階では，クラス名もまた有用な手がかりであった．また，一部のパターンは，特定の名前を持つメソッドにのみ存在している．たとえば，Tomcat 12L のパターンは，`createMBean` という名前のメソッドにのみ所属していたことから，同メソッド群に固有の実装であることが推測可能であった．

抽出されたコーディングパターンを効果的に分析するための環境を構築するにあたっては，コーディングパターンのインスタンスが配置されているメソッドやクラスに共通の部分文字列が含まれているかどうか，パターンの要素として出現するメソッド呼び出しの名前に何らかの規則が存在するか，といった，命名規則を自動的に調査するツールが有効であろうと考えている．また，コードクローン検出手法に関する研究では，類似したコード片のファイルあるいはディレクトリ単位での分布情報が有用なクローンを抽出するための基準として提案されており [75]，コーディングパターンに対しても，インスタンスの分布情報に基づくフィルタリングが適用できる可能性がある．

コーディングパターンをカテゴリ分けしていく段階では，ソースコード上の制御フローの参照が必要であった．たとえば機能を挿入するカテゴリ (1) のパターンと機能を切り替えるカテゴリ (2) のパターンは，いずれも `boolean` を戻り値とするメソッドと `if` 文からなっている．制御構造要素をパターンの一部として加えたことで，これらのカテゴリを容易に認識することが可能であった．また，JHotDraw 8S のように 1 つのループ内部で実行される処理と，Azureus 2S のようにループの前後に登場する処理とを区別することができ，後者はカテゴリ (3) へと分類することが可能であった．

パターン情報が与えられたとき，どのカテゴリに属するかを自動的に認識するためには，追加でのメソッド単位での制御フロー解析，データフロー解析が必要である．たとえばカテゴリ (1) と (2) のパターンを認識するには，メソッドの戻り値が `if` 文で使用されるかというデータフロー情報と，`if` 文に関する制御フロー情報が必要となる．制御フロー情報は各メソッドのソースコードから計算できるため，計算コストも少なく，容易に実現可能である．また，メソッド呼び出しの結果が条件分岐に使用されたかどうかを調査するには，局所的な情報に基づく簡易データフロー解析 [68] が有効であると考えている．カテゴリ

(3) には制御フロー解析で対応でき，また，カテゴリ (4) は `try/catch` ブロックに関する情報を付与するだけで抽出できるため，正規化ルールの拡張のみで対応可能であると考えている．

3.4.2 ソフトウェア設計のパターンへの影響

本研究は，モジュール化が困難なソースコードを発見することで，それらのコードを同時に変更する必要のある開発者を支援することを目的としている．しかし，ある機能をどのようにメソッドに分割し，それを呼び出すかは，開発者の判断に大きく依存する．提案手法はメソッド単位でのパターンマイニングを行っているため，たとえば，長いメソッドをいくつかの短い作業用メソッドに分割すると，それによってパターンの出現回数に影響が出る可能性がある．ただし，ソースコードが複製されるときはブロックなど意味のあるまとまりが単位となる [39] ことから，複製の単位がコーディングパターンの要素となりやすく，また，そのような複製の単位が，開発者によって異なるメソッドへと分割される可能性は低いと仮定している．なお，設計そのものに問題がある場合については，コーディングパターン分析よりも，たとえば設計の欠陥候補を検出するツール [49] の適用が有効であると考えられる．

3.4.3 手法の拡張性

本研究では，条件分岐と繰り返しのみを制御構造要素として正規化したが，Java における `synchronized` や `try` 構文に対するルールを追加することで，同期処理や例外処理のパターンも発見できる可能性がある．また，識別子の同義語を取り扱える正規化辞書を構築できれば，複数のプログラムを入力としてパターンを抽出し，ソフトウェアのドメイン依存，あるいはソフトウェア開発組織依存のコーディングパターンを抽出できる可能性がある．

本研究で使用した PrefixSpan アルゴリズムは，パターンに該当するコードが 1 つのメソッドに複数回出現する可能性を考慮しない．そのようなコード片を発見したい場合には，コーディングパターンの抽出後，インスタンスを持つメソッドに対して個別に検索を行うことで対応可能である．

PrefixSpan の計算時間は，プログラムに含まれるパターンの総数によって大きく変動する．適用実験のために作成したツールは，Intel Core2 Duo 1.86GHz の環境で JHotDraw の解析を約 40 秒で完了する一方，Azureus の解析には数時間を要した．より大規模な解析対象に本手法を適用するために，*Parallel Modified PrefixSpan* という，PrefixSpan の拡張アルゴリズムを並列計算する方法 [66] のツールへの取り込みを検討している．

3.5 関連研究

3.5.1 コードクローン検出手法

トークン列比較によるコードクローン検出法 [37] と比較した場合，本手法で適用したシーケンシャルパターンマイニングは (1) 多数の複製が存在するものだけが検出対象である，(2) 順序関係を維持される限り，ソースコード上で不連続のコードを発見することができる，(3) パターンのインスタンスを 1 メソッドにつき 1 つしか検出しない，という違いがある．

構文木を比較するコードクローン検出手法 [33] やプログラム依存グラフを比較する手法 [41] は，文の追加や順序の入れ替えにある程度対応することができるが，コードクローン検出手法のみで横断的関心事の実装を特定することは困難であると指摘されている [14]．これは，いずれのクローン検出手法も，たとえば連続した 30 トークンの一致 [37] といったように，一致するコード片のサイズを検出条件としており，短いコード片から構成されるパターンの検出には適していないためである．

図 3.3 に示すコード例では，`completeWord` と `backspaceWord` という 2 つのメソッドが，それぞれ，バッファが編集可能かどうかを `isEditable` メソッドの呼び出しによって判定し，編集不可能であった場合は `beep` メソッドを呼び出したのち，処理を中断している．このようなコード片のトークン列あるいは構文木を比較したとき，一致するのは `if` 文のみである．また，プログラム依存グラフを比較すると，変数 `buffer` のデータ依存関係が異なる．具体的には，`completeWord` メソッドでは引数である `view` から取得される戻り値であるが，一方の `backspaceWord` ではフィールドとなっている．このような違いによって，これら 2 つのメソッドに埋め込まれたパターンは，コードクローンとして検出されない．なお，適用実験では最小要素数 $min\_len = 4$ としてパターンを抽出しているため，多くのパターンはソースコード上でも短いコード片となっており，上記の例と同様にコードクローン検出手法では検出対象外となると考えられる．このような検出対象の差異から，開発者が複製したソースコードをすべて探索するといった用途に対しては，提案手法とコードクローン検出手法を併用することで相互に弱点を補完できると考えている．

3.5.2 デザインパターン

本研究では，コーディングパターンを定型的なソースコードと定義した．現状では，特定のソフトウェアに固有のパターンを検出しており，デザインパターン [24] のような一般性は持っていない．しかし，デザインパターンを用いて設計されたソフトウェアの実装は，コーディングパターンの出現に何らかの関係を持っていると考えられる．たとえば，Java 標準ライブラリに含まれた `Iterator` の使用は，`hasNext` メソッド，`next` メソッドや `LOOP` 要素からなるコーディングパターンとして抽出される．また，図 3.1 のパターンの例は，JHotDraw の様々な編集操作が `Command` パターンとして実装された際に生じたものである．

Gil らは，クラスの実装上の特徴をマイクロパターン（Micro Patterns）と呼び，一部のデザインパターンの実装上の特徴を整理している [25]．また，Hannemann らは，デザインパターンを実装するためのクラスに特定の名前のメソッドが実装されることを利用して，ソースコードからデザインパターンの実装を発見する手法を提案している [27]．

今後，複数のソフトウェアを横断した解析を行うことで，デザインパターンの典型的な実装を行うコーディングパターンを発見できる可能性がある．ただし，デザインパターンの実装に関するメソッドの名称はソフトウェア間で異なっている可能性があるため，たとえば同義語辞書によって対応付けを行うといった提案手法の拡張が必要であると考えられる．

3.6 まとめ

本研究では，複数のモジュールに分散した定型的なコードを検出するために，Java プログラムを正規化するルールを定義し，シーケンシャルパターンマイニングを適用する手法を提案した．提案手法を 6 つの Java プログラムに対して適用した結果，アプリケーションの様々な機能を実装するパターンを検出し，パターン情報がソフトウェアを保守する際に有用な手がかりとなることを確認した．

今後の課題としては，開発者が注目したいコーディングパターンだけを絞り込むためのコーディングパターンの自動分類や検索手法の検討，ツールの性能改善が挙げられる．

第4章　コーディングパターンのメトリクスの分析

4.1　まえがき

近年，ソフトウェア開発へのオブジェクト指向プログラミングの採用が増加している．オブジェクト指向の特徴である継承や多態性の仕組みを利用することで，ソフトウェアの再利用性や保守性を向上させることができる．しかし，オブジェクト指向の枠組みでは，モジュール化が困難な機能が存在し，これらの機能の実装はソースコード中に繰り返し登場する [46]．このような機能の代表例としては，ロギングや同期処理が挙げられており，機能に該当するソースコードが複数のモジュールに横断的に出現することから，横断的関心事とも呼ばれる [38]．

このような複数のモジュールに分散配置されるコードは，元となるソースコード片を開発者が複製し，配置先の状況に応じて適宜改変を加えるという方式で作成されることが多く，一群の定型的なコード片，すなわち**コーディングパターン**を構成する．コーディングパターンに属するコード片は互いに類似しており，また，多くは同一の機能を実現している．そのため，コード片の 1 つを変更する場合，開発者は，同一のパターンに属する他のコード片に対しても一貫した変更を適用するべきか，検討する必要がある [10, 23]．

これまで，我々の研究グループでは，ソースコードに対するパターンマイニングを用いたコーディングパターン検出手法を提案し，いくつかのオープンソースソフトウェアに対して適用を行ってきた [77, 50, 29]．その結果，コーディングパターンには，プログラムの横断的関心事に該当するパターンだけでなく，ライブラリの定型的な使い方なども含まれていることが判明している．

しかし，大規模なソフトウェアからは多数のコーディングパターンが検出される一方で，従来はその分析を手作業に頼っていたことから，調査可能なパターンの総数がきわめて限られていた．Marin らによる，被呼び出し回数が多いメソッドには横断的関心事との関連性があるという指摘 [48] に基づき，パターンに該当するソースコード片の数（インスタンス数）が大きいものほど重要なパターンである，という一元的な評価尺度により，分析対象のパターンを選択していた．パターンに該当するソースコード片の数は，しばしば有用なパターンの発見に役立つが，言語仕様あるいはコーディングスタイルなどから，偶然パターンに該当するコード片が発生することもあり，意味のないパターンを手動で取り除く必要もあった．

本研究では，開発者が注目したいパターンのみを効率的に分析可能な環境を構築する

ため，新たな評価尺度として導入可能なメトリクスの評価を行った．パターンの長さやインスタンス数，1 パターンに含まれる要素の種類の数などといった単純なメトリクス以外に，コードクローン検出法 [37, 45] で用いたソースコード片の出現位置に関するメトリクス [75] についても評価を行った．

メトリクス間の値の関係と，実際のパターンの特徴を分析した結果，パターンのインスタンス数，インスタンスの分布の広さ，パターンの要素中に含まれる繰り返し構造の比率といったメトリクスなどが，分析すべきパターンの選択にとって有用であることを確認した．

以降，4.2 節では，コーディングパターンとコーディングパターンの検出手法について述べる．次に，4.3 節では，コーディングパターンの特徴，出現位置について述べる．そして，4.4 節では，行った実験とその結果についての述べる．最後に，4.5 節で，まとめと今後の課題を示す．

4.2 コーディングパターン

コーディングパターンとは，複数のモジュールに分散した定型的なコードである．我々は，コーディングパターンを，メソッド呼び出し要素とそれに付随する制御構造要素（条件分岐と繰り返し文）の定型的な列と捉えたパターンマイニング手法を提案している [29]．

4.2.1 コーディングパターンの例

コーディングパターンは，その出現するメソッド呼び出し要素の種類によって，大きく 2 種類に分けられる．1 つは，解析対象ソフトウェアの外部で定義されているメソッド，すなわちライブラリを使用することを主とするパターンである．もう 1 つは，解析対象ソフトウェア中の特定の機能を実現するために，ソフトウェア内で定義されているメソッドを呼び出しているものである．

イテレータを用いたループ処理のパターン

Java では，コレクションオブジェクトに含まれる各要素に対して処理を行うために，デザインパターン [24] の一種である Iterator パターンを利用できる．

Iterator パターンの利用は，次の手順で行われる．

1. コレクションオブジェクトから，繰り返し処理のための iterator オブジェクトを取得する．
2. 処理を行う要素がコレクション内に残っているかを調査する．
3. コレクションから要素を取り出し処理を行う．

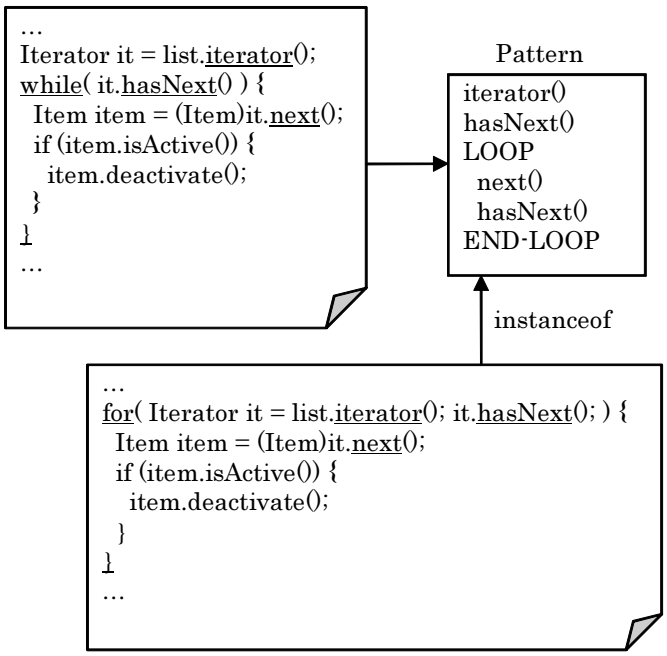


図 4.1: Iterator を使用したループ処理のパターン

これらの一連の処理が，図 4.1 に示すように Iterator を使用したループ処理のパターンとして抽出される．

繰り返し処理は，実際のソースコード上では，for 文や while 文として記述されるが，繰り返し処理の正規化処理により,「LOOP」と「END-LOOP」の組により表現される．

Undo 機能の実現に関するパターン

図 4.2 は，図形エディタ JHotDraw 5.4b1 から抽出された Undo パターンである．

この Undo パターンは，JHotDraw 5.4b1 のユーザが行った操作を元に戻す「Undo 処理」を実現するパターンである．Undo パターンのインスタンス部分で実際に行っている処理は異なるが，下線で示されるメソッド呼び出し列が共通している．このようなメソッド呼び出しパターンを開発者が知ることは，この「Undo 処理」がどのように実装されているかを理解するために有用である．また，新たな編集操作実装するときに役に立つ．

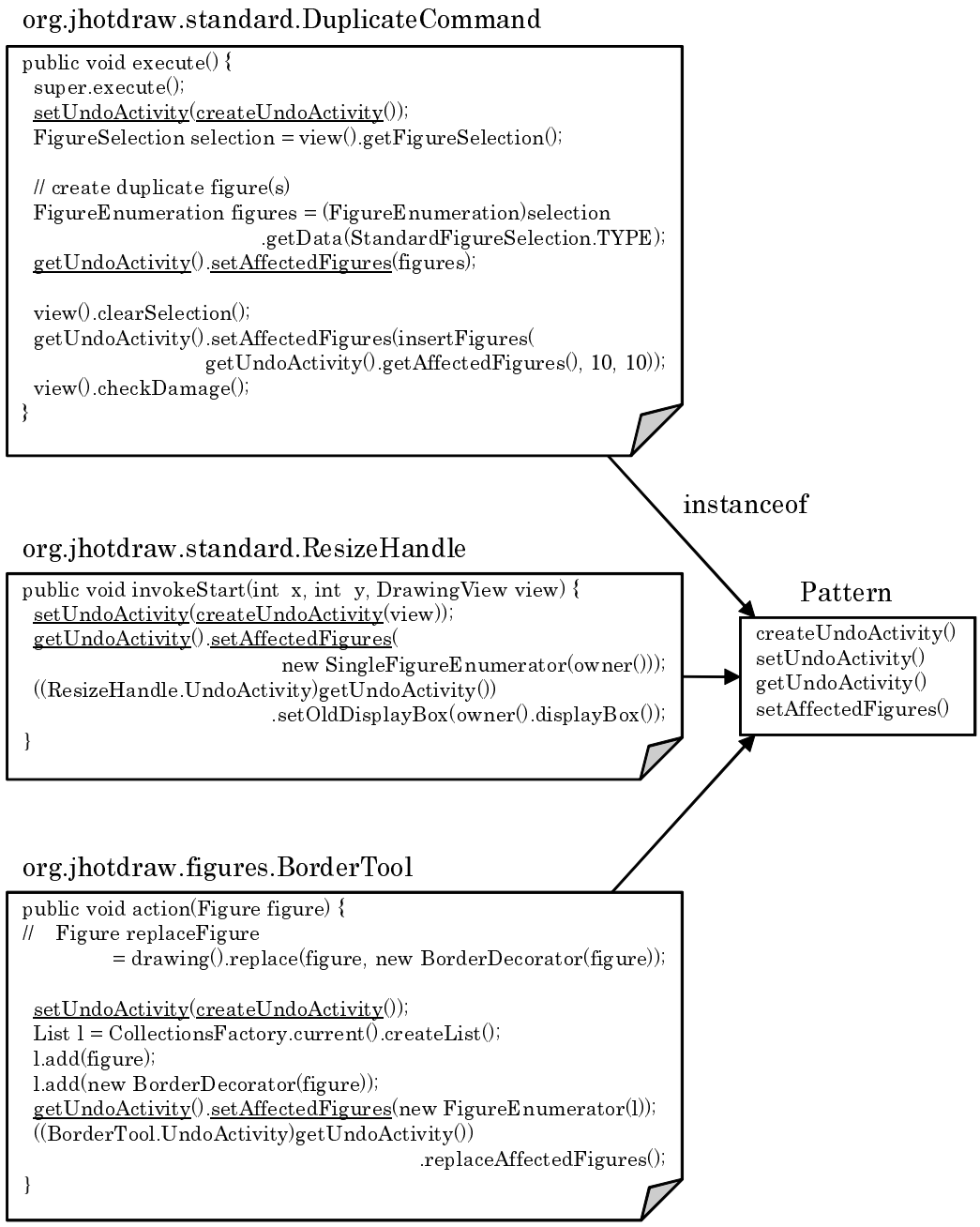


図 4.2: JHotDraw 5.4b1 から抽出された Undo パターン

4.3 コーディングパターンの特徴と出現位置

コーディングパターンは，ソースコードに頻出するメソッド呼び出しの列である．過去の研究では，開発者が部品化することが困難な「横断的関心事」に該当する機能を発見するためにコーディングパターンの分析を行ったが，1 つのプログラムからは数百，数千のコーディングパターンが抽出されるため，その分析対象はインスタンス数が多いパターンから順番に選んだ，ごく少数のパターンに限られていた [29]．インスタンス数が多いパターンを優先的に調査したのは，パターン中で呼び出されているメソッドは被呼び出し回数が多いという事実と，横断的関心事は被呼び出し回数が多いメソッドによって構成されていることが多いという Marin らの指摘に従ったものである [48]．

コーディングパターンを効果的に分析するためには，分析者が注目すべきコーディングパターンやそのインスタンスだけを自動的に抽出することが重要である．本研究では，その基盤として使用することができるソフトウェアメトリクスの候補を選定し，メトリクス間の相関や，コーディングパターンの特徴を調査した．

コーディングパターンから得られる情報には，大きく分けて以下の 2 種類がある．

- パターン自身の情報．パターンに含まれる要素数や，パターンのインスタンス数がこれに該当する．
- パターンのインスタンスから得られる情報．パターンに該当するソースコード片の位置などはこちらに含まれる．

本研究では，これらの情報を表現したメトリクス値として 6 種類を選定した．パターンの分析を行う都合という目的から，すべて，パターン P を引数に取り，整数あるいは実数値を返す関数 $f(P) : Pattern \rightarrow value$ という形式で定義した．以下，それらのメトリクスの定義を述べるが，パターン P に対して，それらのインスタンス i は $i \in P$ というように集合 P の要素として記載し，パターン P のインスタンス数は $|P|$ という形式で記述するものとする．

4.3.1 パターン長：LEN（Pattern Length）

コーディングパターン P のパターン長 $LEN(P)$ は，パターン P に含まれている要素の数を示す整数値である．

コーディングパターン検出ツールにより，パターン長の閾値に満たないパターンは，検出結果から取り除かれている．

4.3.2 パターンのインスタンス数：NOI（Number of Instances）

コーディングパターン P のインスタンス数 $NOI(P) = |P|$ は，パターン P を構成する要素列が，マイニング対象のソースコード中に出現した回数を示す整数値である．

本研究で使用しているパターンマイニングのアルゴリズム PrefixSpan では閾値としても用いられている．

4.3.3 制御構造要素の割合：RCE（Ratio of Control Elements）

コーディングパターン P の制御構造要素の割合 RCE(P) は，パターン P に含まれる全要素数に対する，制御要素数の割合として計算される実数値である．

RCE(P) の値が大きいパターン P は，メソッド呼び出しを含まない if 文や for 文の単純な入れ子関係だけを表現している可能性が高い．そのため，パターンマイニングの終了後，経験的に定めた閾値 RCE(P) ≤ 0.7 という条件で，パターンのフィルタリングを行っている．

4.3.4 パターンの密度：DEN（Density）

コーディングパターン P の密度 DEN(P) は，パターン P の各要素が，ソースコード上でどれだけ密に配置されているかを示す実数値である．具体的には，以下の式によって計算する．

$$DEN(P) = \frac{\sum_{i \in P} DEN_{inst}(i)}{|P|}$$

ただし，$DEN_{inst}(i)$ は，パターンのインスタンスに対して定義される密度である．パターンに該当する要素を含むメソッドの要素列が与えられたとき，

$$DEN_{inst}(i) = \frac{LEN(P)}{i \text{ の末尾要素の位置} - i \text{ の開始要素の位置} + 1}$$

インスタンスの密度の計算例を図 4.3 に示す．図 4.3 右に示されているコーディングパターンが，図 4.3 左のようなインスタンスとして出現しているとき，先頭要素 iterator() から末尾要素 END-LOOP まで，パターンに該当する 6 要素が 12 要素の列中に出現しているとみなし，このインスタンスに対する密度は $DEN_{inst}(i) = 6/(12 - 1 + 1) = 0.5$ となる．

パターン P の密度 DEN(P) は，P のすべてのインスタンスの密度の平均として定義されており，各インスタンスが他の要素を間に含まない単一のコード断片に近づいていくほど，値が 1 に近づいていく．

4.3.5 非繰り返しの要素割合：RNR（Ratio of Non-Repeated elements）

パターン P の「非繰り返し」を意味する RNR(P) は，パターン P の要素中に含まれる繰り返し構造を検出，取り除いた後に残る要素の割合を示した実数値である．

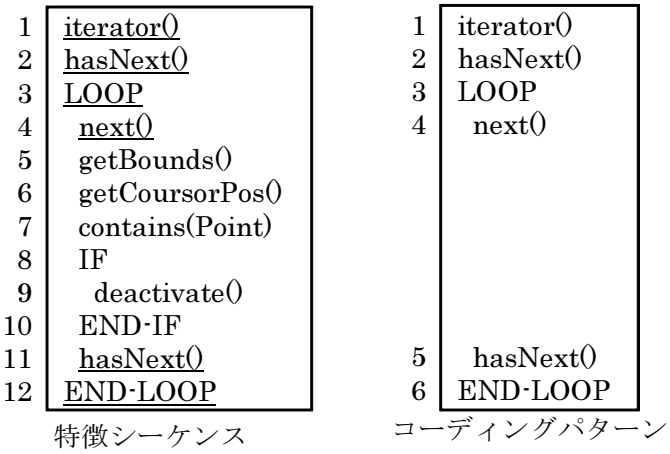


図 4.3: メトリクス DEN

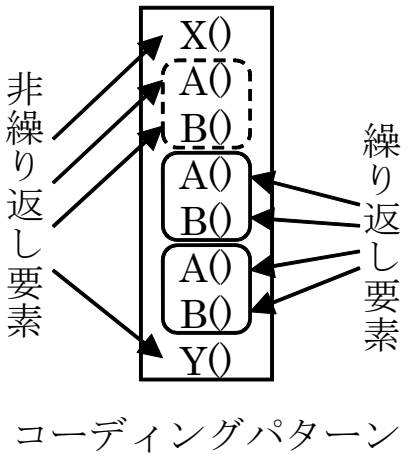


図 4.4: メトリクス RNR

本研究では，繰り返しの検出には，SEQUITUR アルゴリズム [52] を用いた．このアルゴリズムは，ある要素列が与えられたときに，連続した 2 要素の組が 2 回以上出現した場合を，繰り返しとして検出する．図 4.4 は，8 要素のコーディングパターン「X, A, B, A, B, A, B, Y」が与えられたときの非繰り返し要素を示している．このパターンは,「A，B」という組が 3 回繰り返されていることから，その 2 回目以降の出現を繰り返し要素と認識し，RNR は 0.5 となる．

4.3.6 パターンインスタンスの分散：RAD（Radius）

パターンインスタンスの分散 RAD(P) は，パターン P のインスタンスが配置されているソースコードの範囲を示す整数値である．

Java 言語では，ソースコードをパッケージという階層的構造によって管理しており，開発組織のドメイン名とソフトウェア名，さらにソフトウェア中のサブシステム名などを用

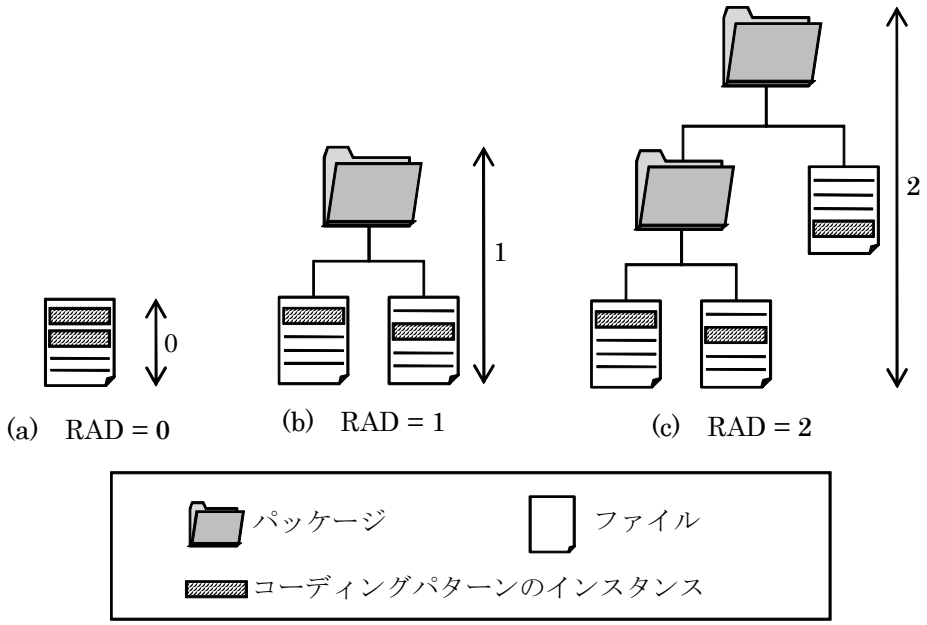


図 4.5: メトリクス RAD

いて，次のように重複しない名称を使用する [26].

```
jp.ac.osaka_u.ist.sel.pattern_mining
jp.ac.osaka_u.ist.sel.pattern_mining.metrics
org.eclipse.jdt.core
```

RAD は，パッケージ階層中でのコーディングパターンのインスタンスの分散度合いを表すメトリクスである．図 4.5(a) では，コーディングパターンのすべてのインスタンスが，同一のファイル内に存在しているため，RAD 値は 0 とする．また，図 4.5(b) のように，コーディングパターンのインスタンスが，同一パッケージ内の複数のファイルに分散している場合には，RAD 値は 1 とする．さらに，コーディングパターンのインスタンスが複数パッケージに分散している場合には，それぞれのインスタンスの存在するパッケージをルートノードに向かってたどり，すべてのインスタンスを子孫として持つパッケージにたどり着いたら，そのパッケージを基準として，すべてのインスタンスまでの距離を計測し最大のものを RAD とする．図 4.5(c) の例では，RAD は 2 となる．

4.4 メトリクスを用いたコーディングパターン分析

コーディングパターンの特徴を調査するために 4.3 節で定義した 6 種類のメトリクスを，オープンソースソフトウェアから抽出したコーディングパターンに対して適用した．

4.4.1 対象ソフトウェア

調査対象として，図形描画ソフトウェア JHotDraw[32]，テキストエディタ jEdit[31]，アプリケーションサーバ Apache Tomcat[6]，パーサジェネレータ SableCC[60] を用いた．

調査対象としたソフトウェアの一覧と実験に使用したバージョン，ソフトウェアの規模，実際に検出されたコーディングパターンの数を表 4.1 に示す．

コーディングパターンを抽出する際のパラメータ設定は，次の通りである．

- インスタンス数のしきい値：10
- パターン長のしきい値：4

4.4.2 メトリクス間の関連分析

本研究では，6 種類のメトリクスのすべての組み合わせに対して関連性の調査を行った．本節では，関連性の認められたメトリクスの組み合わせについて述べる．

非繰り返し要素の割合と制御構造要素の割合

制御構造要素を含むパターンと制御構造要素を含まないパターンそれぞれの数を表 4.2 に示す．

制御構造要素を含むパターンと含まないパターンの数を比較した場合，今回解析したソフトウェアすべてで，制御構造要素を含むパターンの数が制御構造を含まないパターンの数より多く検出されている．これは，制御構造要素の種類が IF，ELSE，END-IF，LOOP，

表 4.1: 対象ソフトウェア

ソフトウェア名	バージョン	規模（LOC）	パターン総数
JHotDraw	7.0.9	90,166	375
jEdit	4.3pre10	168,335	2,902
Apache Tomcat	6.0.14	313,479	8,782
SableCC	3.2	35,388	450

表 4.2: 制御構造を含むパターン含まないパターンの数

ソフトウェア名	制御構造要素無し	制御構造要素有り
JHotDraw	140	235
jEdit	1,212	1,690
Apache Tomcat	2,489	6,293
SableCC	120	330

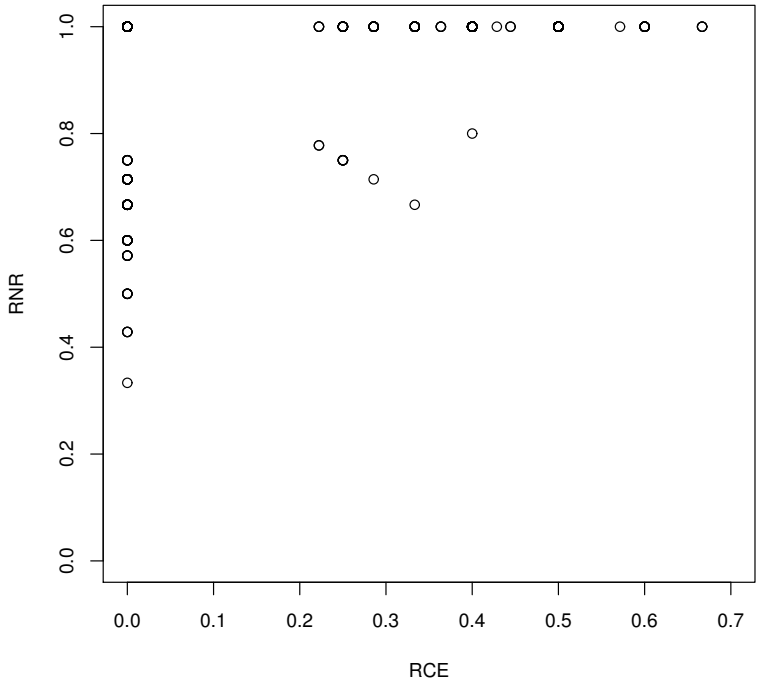


図 4.6: 制御構造要素の割合と非繰り返し要素の割合 (JHotDraw)

END-LOOP の 5 種類と少なく，プログラムを作成するために多用されるため，パターンとして検出さやすいことが要因として考えられる．

図 4.6〜図 4.9 に，X 軸に制御構造要素の割合，Y 軸に非繰り返し要素の割合をとった散布図をパターンが検出されたソフトウェアごとに示す．これらの図中では，制御構造要素を含まないパターンは，Y 軸上に配置される．また，制御構造要素を含んでいるパターンは，制御構造要素の割合に応じてグラフ上に表示されている．

制御構造要素を含まないパターンは，非繰り返し要素の割合に偏りがなく広く分布しているため，制御構造要素を含むパターンのみに着目する．JHotDraw（図 4.6），jEdit（図 4.7），SableCC（図 4.9）に関しては，制御構造要素を含むパターンは，グラフの上部に集中して登場する傾向がある．しかし，Apache Tomcat（図 4.8）の場合には，制御構造要素を含むパターンの Y 軸方向への偏りは小さい．

ここで，Apache Tomcat に関する事例についてさらに分析する．制御構造要素は，条件分岐の要素（IF，ELSE，END-IF），繰り返し処理の要素（LOOP，END-LOOP）の 2 種類に分かれる．そこで，Apache Tomcat から検出されたコーディングパターン中から，条件分岐の要素を含んでいるパターンと，繰り返し処理の要素を含んでいるパターンを別々にプロットした．条件分岐の要素，繰り返し処理の要素を別々にプロットした散布図を図 4.10 に示す．

図 4.10 の結果では，条件分岐の要素を含む要素は全体に広がっているが，繰り返し要素は，非繰り返し要素の割合が 1 に近い側に偏って分布している．

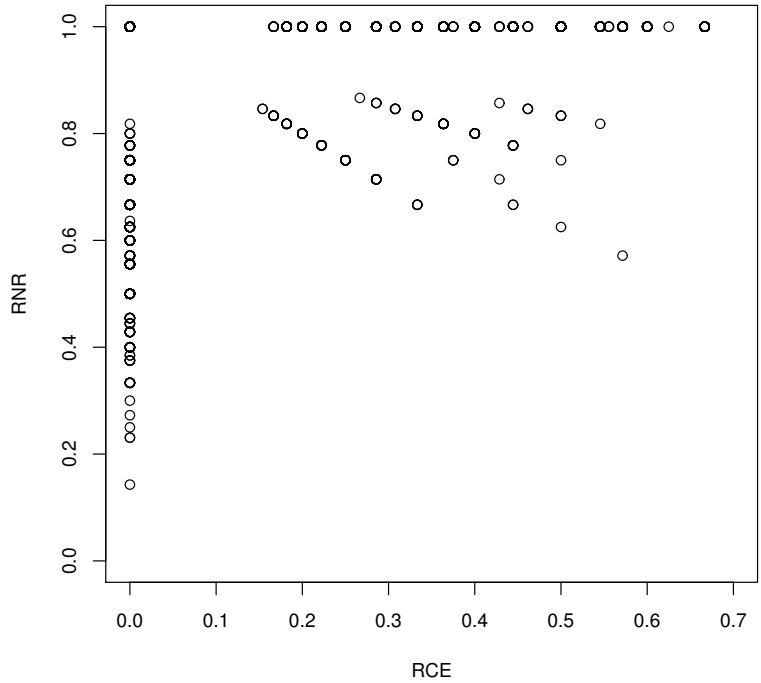


図 4.7: 制御構造要素の割合と非繰り返し要素の割合 (jEdit)

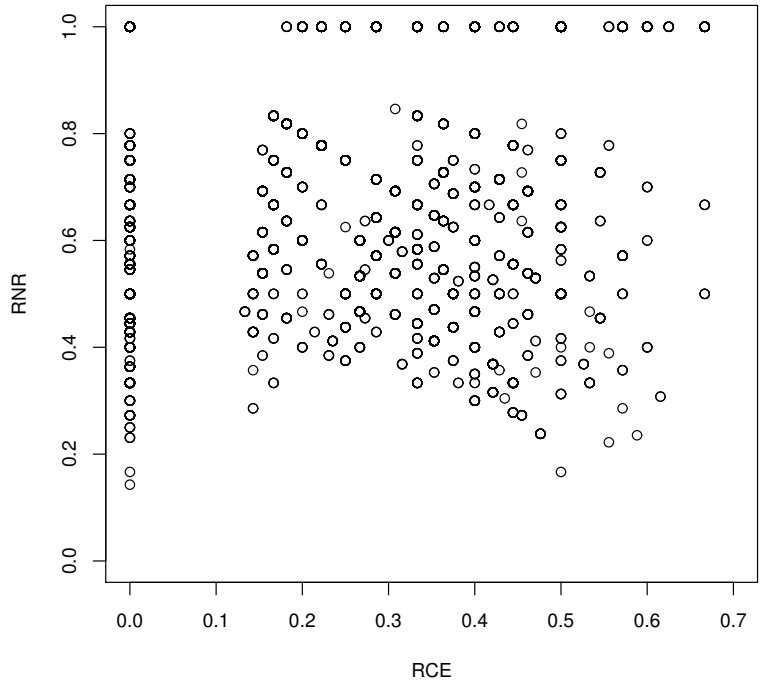


図 4.8: 制御構造要素の割合と非繰り返し要素の割合 (Apache Tomcat)

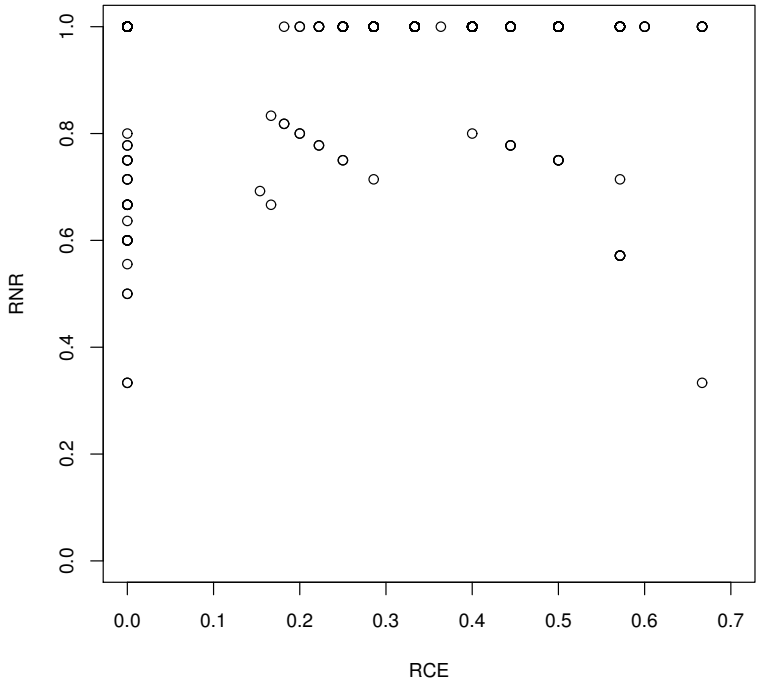


図 4.9: 制御構造要素の割合と非繰り返し要素の割合 (SableCC)

これらのことを総合して考えると，制御構造要素，特にその中でも LOOP 構造をの要素を含むパターンは，非繰り返し要素の割合が高くなる傾向がある．パターン内に LOOP 構造を持つということは，繰り返し処理がその LOOP 構造により集約さることを意味するので，繰り返される要素が減少し，非繰り返し要素の割合が高くなると考えられる．

イテレータと出現位置の関連

本項では，イテレータの利用とパターンインスタンスの出現位置の関連性を分析する．パターンインスタンスの出現位置を表すメトリクスとしては，パターンインスタンスの分散を用いる．

イテレータの利用と出現位置の関連を分析するためには，コーディングパターンの中からイテレータの利用を含むパターンをび出す必要がある．

イテレータの利用パターンの特定

イテレータの利用は，メソッド名に「next」や「hasNext」という特徴的な文字列を持つメソッド呼び出しを含んでいる．また，イテレータの利用では，複数のオブジェクトに対して処理を繰り返す必要がある．そこで，コーディングパターン中から，次の条件を満たすコーディングパターンを，イテレータを利用しているパターンとして抽出した．

- メソッド名に「next」を含むメソッド呼び出し要素を持つ．

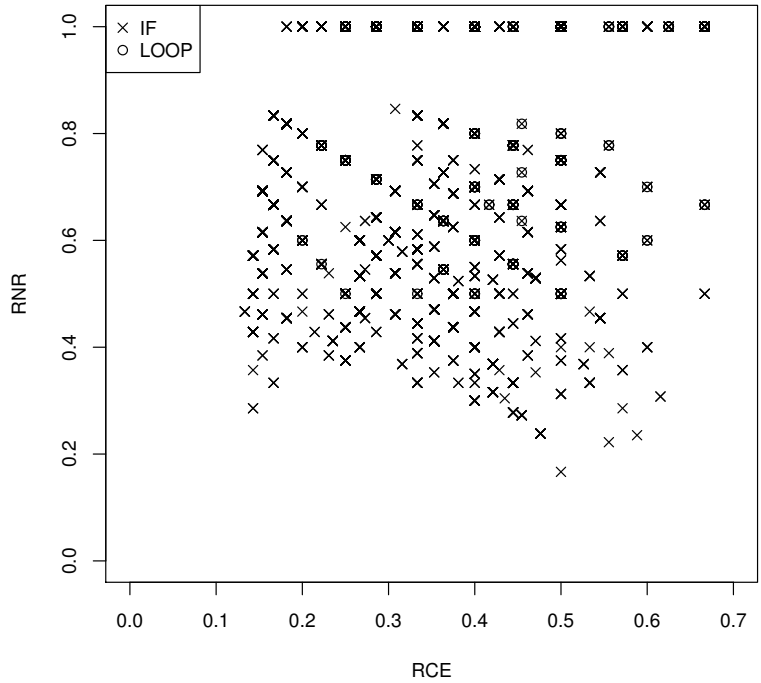


図 4.10: [IF，LOOP 分離版] 制御構造要素の割合と非繰り返し要素の割合 (Apache Tomcat)

- メソッド名に「hasNext」を含むメソッド呼び出し要素を持つ.
- 「LOOP」,「END-LOOP」で表現された，繰り返し処理を含む.

その結果，抽出されたパターン数とコーディングパターンの総数に対する割合を表 4.3 に示す.

また，コーディングパターンをイテレータの利用パターンと，イテレータの利用に関係がないその他のパターンに分類し，図 4.11〜図 4.14 にパターンの分散とインスタンス数の関連を示した.

図 4.11，4.12 に示すように，JHotDraw と jEdit に含まれる，イテレータを利用しているコーディングパターンは，パターンの分散が大きい．また，インスタンス数が多い，イテレータを利用したパターンも発見されている.

表 4.3: イテレータパターンの数

ソフトウェア名	パターン総数	イテレータ利用パターン数	割合
JHotDraw	375	8	2.1%
jEdit	2,902	28	0.9%
Apache Tomcat	8,782	434	4.9%
SableCC	450	132	29.3%

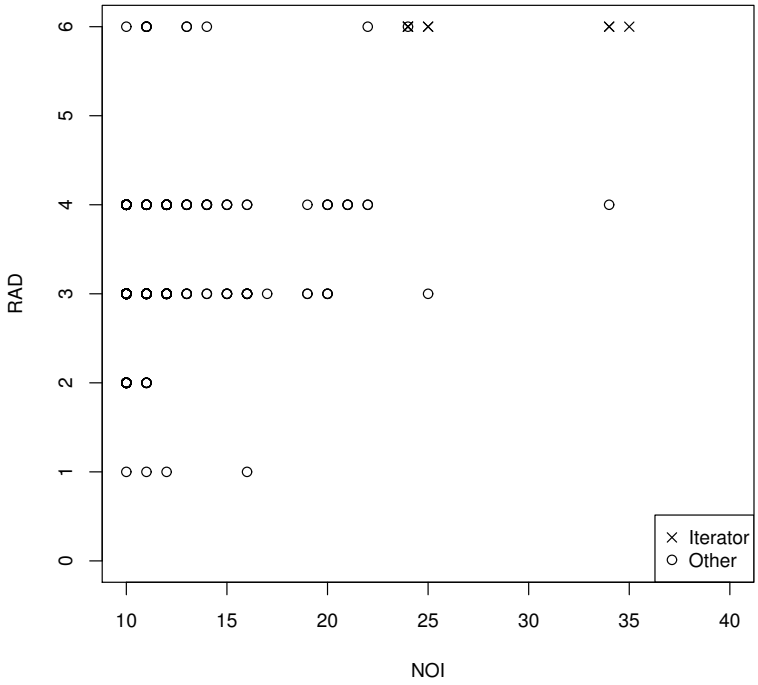


図 4.11: パターンの分散とインスタンス数の関連性 (JHotDraw)

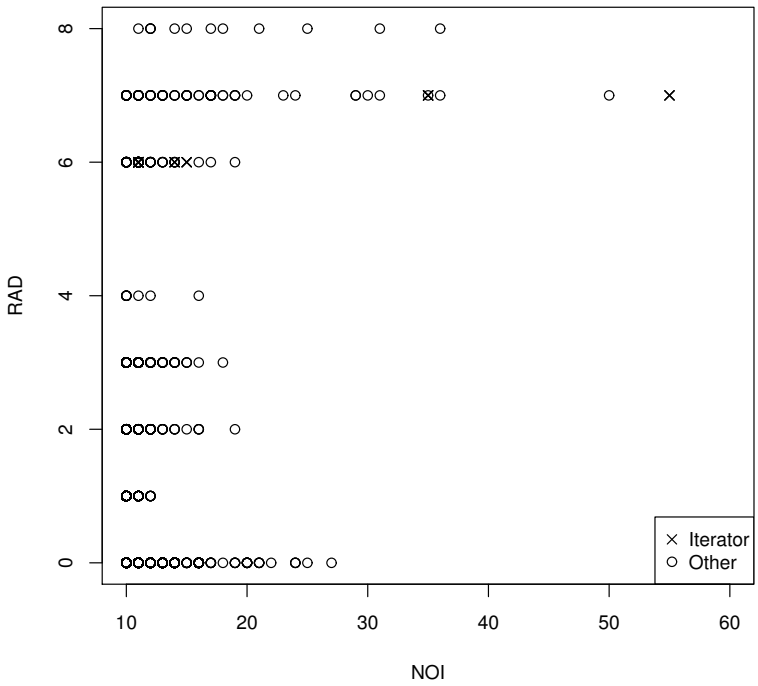


図 4.12: パターンの分散とインスタンス数の関連性 (jEdit)

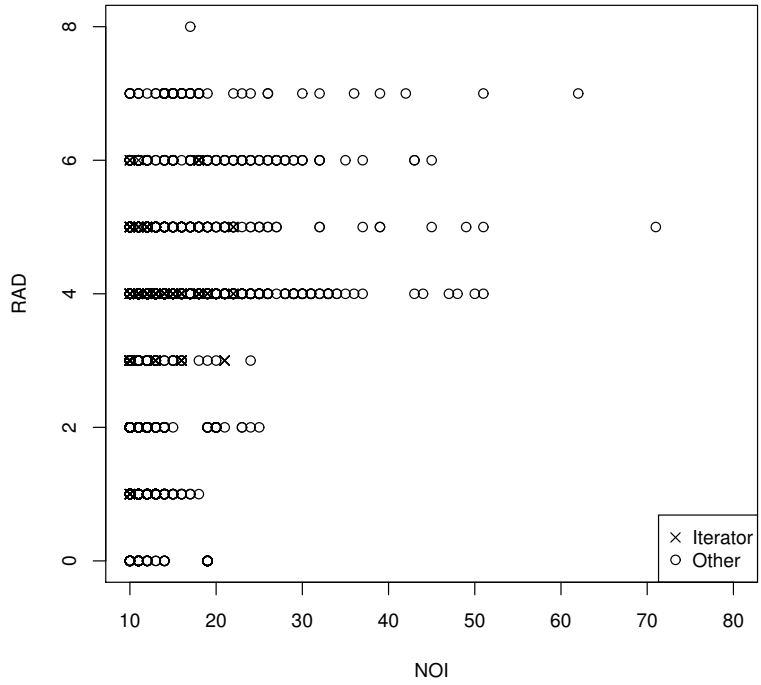


図 4.13: パターンの分散とインスタンス数の関連性 (Apache Tomcat)

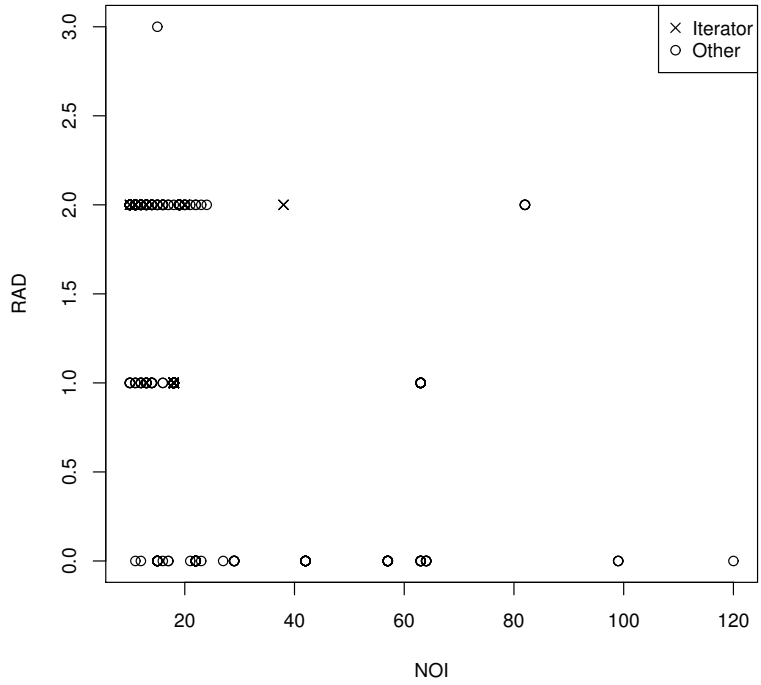


図 4.14: パターンの分散とインスタンス数の関連性 (SableCC)

図 4.14 に示した SableCC の場合では，SableCC のパッケージ階層が浅く作られ 1 つのパッケージに多数のソースファイルが格納される構造になっていた．そのため，パターンの分散のについては，パターンの種類間での差異が判断できなかった．

図 4.13 に示した Apache Tomcat では，パッケージ階層の深さは，分布を判断する上では十分であるが，傾向はみられなかった．これは，イテレータとプログラム固有の機能実装の両方を含んだパターンが，パターン階層中で局所的に現れていることが，原因として考えられる．

イテレータの利用は，Java のプログラマ間では既知の事項であり，パターンとしての重要度は低い．しかし，表 4.3 によると，SableCC から抽出パターンの中には，イテレータを含むパターンの割合は 29.3%に達している．

イテレータパターンを取り除く指標を作成することで，ユーザがプログラムを理解するために有用な，ソフトウェア固有の機能実装のような，パターンを発見しやすくできる．

コーディングパターンの従来の調査手法では，インスタンス数が多いものから順に調査を行っていた．しかし，JHotDraw や jEdit のように，イテレータを利用しているパターンが，インスタンス数の上位に登場していることから，有益なパターンが多数のコーディングパターン中に埋もれてしまうことも考えられる．この問題を回避するためには，パターンの分散が低いパターン，つまり，特定のパッケージやファイルに限定して登場しているパターンから順に調査するといった方法に切り替える必要がある．

また，ソフトウェア全体のパッケージ階層の浅いソフトウェアに関しては，傾向の判断が困難であるため，パターンの出現位置の情報として利用するメトリクスを改良する必要がある．

4.5 まとめ

コーディングパターンの検出結果は，膨大になり必要としているパターンを発見することが困難となっている．

これを解決するためには，コーディングパターンを種類別に分類し，コーディングパターン閲覧者が必要としているもののみを選び出し提示する必要がある．

そこで，本研究では，コーディングパターンの特徴を計測するためのメトリクスを提案し，その特徴間の関連と，コーディングパターンの種類との関係について分析を行った．その結果，パターンのインスタンス数，インスタンスの分布の広さ，パターンの要素中に含まれる繰り返し構造の割合といったメトリクスなどが，分析すべきパターンの選択にとって有用であることを確認した．

今後，本研究の分析結果をもとに，コーディングパターンのフィルタリング手法の実現を目指す．また，コーディングパターンを効果的に開発者に提示するために，コーディングパターン閲覧環境を統合開発環境上で提供することや，コーディングパターンからソースコードの利用方法を抽出し，ソフトウェア部品検索システム上でソフトウェア部品と共に提供することなどが，発展研究としてあげられる．

第5章　バージョンをまたがるコーディングパターンの分析

5.1　はじめに

コーディングパターンは，メソッド呼び出しやコントロール要素などの頻出列であり，ソフトウェア中でモジュール化されていない特定の種類の関心事を実装している [29]．コーディングパターンには，API の利用方法やアプリケーション固有の振る舞いも含まれる．たとえば，hasNext メソッドの次に next メソッドを呼び出すことは，Java の Iterator オブジェクトの典型的な使用方法である．このような API の利用方法を表現したパターン以外にもアプリケーション固有のパターンが存在する．アプリケーション固有のパターンとしては，Apache Tomcat 6.0.14 から検出された，デバッグのためのロギング関連のパターンがある．このパターンでは，isDebugEnabled と debug メソッドの組み合わせが 304 回登場する．また，Azureus 3.0.2.2 には，AEMonitor クラスを使用したマルチスレッドの同期化処理が 151 メソッドから検出された．さらに，jEdit 4.3 では，isEditable メソッドと if 文を使用した，読み込み専用ファイルへの変更防止処理のパターンが検出された．コーディングパターンは，プログラム中の暗黙のルールを反映しているため，開発者のソースコード理解を促進し，プログラム中の欠陥検出にも有効である [36, 44, 54].

我々の研究グループでは，コーディングパターンマイニングツールを開発し，これまでに，数種類のアプリケーションからコーディングパターンを検出し分析してきた [29]．表 5.1 は，JHotDraw 5.4b1 から検出されたコーディングパターンの一例である．このパターンは，Undo 処理に関連しており，⟨createUndoActivity(), setUndoActivity(), getUndoActivity(), setAffectedFigures()⟩ の 4 つの要素から成るパターンで，ソースコード中の 2 か所に出現している．

既存研究 [1, 53, 68] では，ソースコードから抽出されたパターンを再利用可能なコードとして使用している．しかし，一部のパターンは，特定のバージョンのソースコードのみから検出される．もしパターンが複数バージョンから検出されるのであれば，そのパターンは，再利用性が高いと考えられる．さらに，ソースコード上にパターンが存在し，そのパターンが多くのバージョンに登場しているというような知識は，ソースコードの理解作業に有益である．パターン長の長いパターンには，潜在的にパターン長が短いサブパターンが多数存在している．そのため，これらの類似した多数のパターンの中から，有益なパターンを選び出すことは難しい．

本研究では，clone genealogy の研究 [11, 40] で行われているのと同様に，調査対象のア

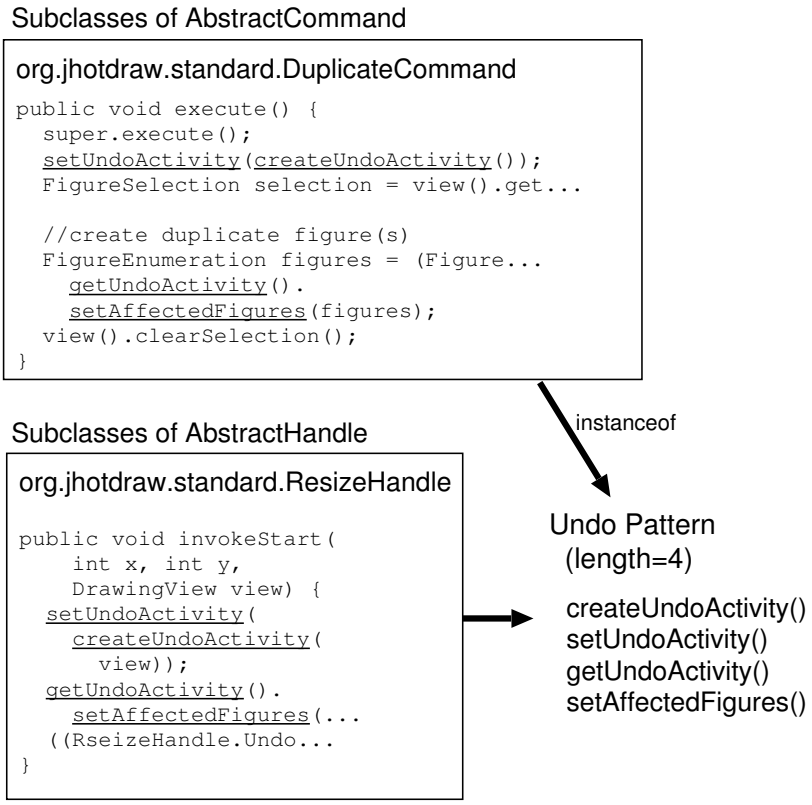


図 5.1: Undo パターン (JHotDraw 5.4b1)

プリケーションから検出されたコーディングパターンが，そのアプリケーションの何バージョンに含まれるかを調査した．コーディングパターン検出ツールは，シーケンシャルパターンマイニングのアルゴリズムとして，PrefixSpan[58] を使用している．それぞれのコーディングパターンは，メソッド呼び出し等の呼び出し要素や，`if`，`while`，`try-catch` 等の制御要素で構成される頻出列である．パターンは，構成要素が登場する順序が変更されない限り検出され続ける．

本研究では，表 5.2 に示す 10 個の Java アプリケーションを調査対象とした．2 つのインスタンスを持つ，2 つ以上の要素からなるパターンを全て取り出す．つまり，もし 2 つのメソッド中で，2 つのメソッド呼び出しが同じ順序で呼び出されている場合に，コーディングパターンとして検出している．これらのメソッド呼び出しのペアが，全てのバージョンで変わることなく登場していれば，このパターンは安定したパターンである．

5.2 コーディングパターンの出現するバージョン数の計算

本研究では，まず，実験対象アプリケーションのそれぞれのバージョンのソースコードから，個別にコーディングパターンを検出した．そして，バージョンごとのパターン検出

結果から，複数バージョンに共通して登場するパターンを検索した．

一度消失したパターンであっても，再度検出されるようになる可能性があるため，隣接した 2 バージョン間だけでなく，全てのバージョンのペアについて，共通パターンが存在しているかを調査する必要がある．

パターン p が検出されたバージョン数を返す関数 $NV(p)$ を，次のように定義した．

$$NV(p) = \left| \left\{ v_i | {}^{\exists} p_k \in P(v_i) : p \sqsubseteq p_k \right\} \right|$$

$P(v_i)$ は，ソフトウェアのバージョン v_i から検出された全種類のパターンを返す関数である．また，$p \sqsubseteq p_k$ は，p が p_k のサブシーケンスであることを意味する．2 つのパターンのシーケンス $S = \langle e_1, e_2, \cdots, e_l \rangle$ と $S' = \langle e'_1, e'_2, \cdots, e'_m \rangle$ が与えられたとき，$e_1 \subseteq e'_{I_1}, e_2 \subseteq e'_{I_2}, \cdots, e_l \subseteq e'_{I_l}$ を満たす整数 $1 \leq I_1 < I_2 < \cdots < I_l \leq m$ が存在するならば，S は S' のサブシーケンスである．また，S' は S のスーパーシーケンスである．たとえば，$\langle a, c, d \rangle$ は，$\langle a, b, c, d, e \rangle$ のサブシーケンスである．

パターン p_1 が，バージョン 1，2，3 の 3 つのバージョンから検出されているなら，$NV(p_1) = 3$ である．また，$NV(p)$ は，パターンが検出されるバージョンの連続性は考慮していないので，パターン p_2 がバージョン 1 と 3 の 2 つのバージョンから検出されているなら，$NV(p_2) = 2$ となる．

以前の研究 [19] では，パターンが検出されたバージョン数を計算するために，次の関数を用いていた．

$$NV_{old}(p) = |\{v_i | p_k \in P(v_i)\}|$$

$NV_{old}(p)$ は，サブシーケンスとスーパーシーケンスの関係を考慮せずに，要素が完全に一致するパターンのみが登場するバージョン数を計算する．

表 5.1 に示す例を用いて，$NV_{old}(p)$ と $NV(p)$ の違いを説明する．ソフトウェアのバージョン 1 でパターン $\langle a, c \rangle$ が検出され，バージョン 2 では，全ての出現箇所が $\langle a, b, c \rangle$ に変更される．そして，バージョン 3 になると，全出現箇所 4 個所のうち 2 か所が $\langle a, c \rangle$ へと変更されている．コーディングパターンの検出では，飽和系列を検出しているため，バージョン 2 では $\langle a, c \rangle$ が検出されない．しかし，バージョン 2 においても，$\langle a, c \rangle$ のスーパーシーケンスである $\langle a, b, c \rangle$ が検出されている．$\langle a, c \rangle$ に対する $NV_{old}(p)$ の値は，要素が完全一致したパターンが登場するバージョン数のみを扱うため 2 となる．一方で，$NV(p)$ の場合は，$\langle a, c \rangle \sqsubseteq \langle a, b, c \rangle$ であることを考慮するため，バージョン 2 においても $\langle a, c \rangle$ が検出されているとみなす．よって，$\langle a, c \rangle$ に対する $NV(p)$ の値は，3 となる．$NV(p)$ と $NV_{old}(p)$ の間には，常に $NV(p) \geq NV_{old}(p)$ の関係が成立する．

5.3 分析

5.3.1 分析手法

10 種類の Java を用いて開発されたオープンソースソフトウェア CAROL[16]，Cewolf[17]，dnsjava[20]，Jackcess[30]，JmDNS[34]，Joda-Time[35]，NatTable[51]，OntoCAT[55]，OVal [56]，transmorph[69] を実験対象とした．今回の実験では，それぞれのソフトウェアにつき，複数のバージョンのソースコードを用いている．これらのソフトウェアは，それぞれのプロジェクトの Web サイトからソースコードのアーカイブを取得して実験に用いた．実験に使用したソフトウェアのバージョン一覧を表 5.2 に示す．

まず，それぞれの対象ソフトウェアから検出したパターン数と，それぞれのパターンが検出されたバージョン数を調査した．パターン数は，それぞれのバージョンから検出されたパターン数の単純に合計したものではなく，バージョン間で重複したパターンを排除した，パターンの種類数に相当するものである．

次に，開発者は通常，最新バージョンのソースコードに対して変更を行うため，最新のバージョンに登場するパターンに対する関心が高い．そのため，最新バージョンに登場するパターンと最新バージョンに登場しないパターンの 2 種類に分類し，分類間での $NV(p)$ の差について評価した．

最後に，調査で見つかった，興味深いパターンを紹介する．

5.3.2 パターンの安定性

表 5.3 に，それぞれのソフトウェアから検出されたパターン数（#Pattern），全てのバージョンに登場するパターン数（#Stable Pattern）と全パターンの中で占める割合（#Stable Pattern / #Pattern），最初と最後のバージョンに登場するパターン数（#Common Pattern in first & last versions）とそのうち全バージョンに登場するパターンが占める割合（#Stable Pattern / #Common Pattern）を示した．

10 種類の実験対象ソフトウェアの全バージョンについて調査した結果，約 2,600 から 17,000 種類のパターンが検出された．これらの膨大な数のコーディングパターンを詳細に調査することは困難である．

ソフトウェア毎に，コーディングパターンの $NV(p)$ を調査し，その分布を箱髭図として図 5.2 の (a) に示した．多くのパターンの $NV(p)$ は，小さな値となっており，コーディングパターンは，ソフトウェアの開発が進む中で変化しやすく不安定である．また，表 5.3

表 5.1: $NV_{old}(p)$ と $NV(p)$ の違い

Pattern	1	2	3	$NV_{old}(p)$	$NV(p)$
$\langle a, c\rangle$	✓(4 instances)	- $NV_{old}(p)$ / ✓$NV(p)$	✓(2 instances)	2	3
$\langle a, b, c\rangle$	-	✓(4 instances)	✓(2 instances)	2	2

表 5.2: 実験対象の概要

Program	LOC Range	#Version	Version List
CAROL	7,546 to 25,944	12	1.0.1, 1.0.2, 1.1.0, 1.2.0, 1.3.0, 1.3.1, 1.3.2, 1.3.4, 1.4.0, 1.5.1, 2.0.0, 2.0.5
Cewolf	8,485 to 14,891	14	1.0, 1.1, 1.1.1, 1.1.2, 1.1.3, 1.1.4, 1.1.5, 1.1.6, 1.1.7, 1.1.8, 1.1.9, 1.1.10, 1.1.11, 1.1.12
dnsjava	5,084 to 33,330	51	0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.8.1, 0.8.2, 0.8.3, 0.9, 0.9.1, 0.9.2, 0.9.3, 0.9.4, 0.9.5, 1.0, 1.0.1, 1.0.2 ,1.1, 1.1.1, 1.1.2, 1.1.3, 1.1.4, 1.1.5, 1.1.6, 1.2.0, 1.2.1, 1.2.2, 1.2.3, 1.2.4, 1.3.0, 1.3.1, 1.3.2, 1.3.3, 1.4.0, 1.4.1, 1.4.2, 1.4.3, 1.5.0, 1.5.1, 1.5.2, 1.6.1, 1.6.2, 1.6.3, 1.6.4, 1.6.5, 1.6.6, 2.0.0, 2.0.1
Jackcess	4,483 to 29,016	32	1.0, 1.1, 1.1.1, 1.1.2, 1.1.3, 1.1.4, 1.1.5, 1.1.6, 1.1.7, 1.1.8, 1.1.9, 1.1.10, 1.1.11, 1.1.12, 1.1.13, 1.1.14, 1.1.15, 1.1.16, 1.1.17, 1.1.18, 1.1.19, 1.1.20, 1.1.21, 1.2.0, 1.2.1, 1.2.2, 1.2.3, 1.2.4, 1.2.5, 1.2.6, 1.2.7, 1.2.8
JmDNS	3,408 to 17,252	20	0.2, 1.0.RC1, 1.0.RC2, 1.0-Final, 2.0, 2.1, 3.0, 3.1, 3.1.2, 3.1.3, 3.1.4, 3.1.5, 3.1.6, 3.1.7, 3.1.8, 3.2.0, 3.2.1, 3.2.2, 3.4.0, 3.4.1
Joda-Time	40,311 to 138,710	19	0.9, 0.95, 0.98, 0.99, 1.0, 1.1, 1.2, 1.2.1, 1.3, 1.4, 1.5, 1.5.1, 1.5.2, 1.6, 1.6.1, 1.6.2, 2.0RC1, 2.0, 2.1
NatTable	5,520 to 42,377	20	alpha0.2, beta1.0, beta1.2, beta1.3, 1.5.0, 1.6.0beta1, 2.0RC2, 2.0, 2.1.0, 2.2.0RC1, 2.2.0, 2.2.1, 2.3.0beta1, 2.3.0beta1a, 2.3.0beta2, 2.3.0beta3, 2.3.0, 2.3.1, 2.3.1.1, 2.3.2
OntoCAT	6,226 to 13,605	19	0.9.4, 0.9.5, 0.9.5.1, 0.9.5.2, 0.9.5.3, 0.9.6, 0.9.6.1, 0.9.6.2, 0.9.6.3, 0.9.7, 0.9.7.1, 0.9.7.3, 0.9.7.4, 0.9.7.5, 0.9.7.6, 0.9.7.7, 0.9.8, 0.9.9, 0.9.9.1
OVal	3,249 to 25,235	19	0.1, 0.3, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0, 1.10, 1.20, 1.30, 1.31, 1.32, 1.40, 1.50, 1.60, 1.61, 1.70, 1.80
transmorph	6,612 to 19,090	13	1.0.0, 2.0.0, 2.1.0, 2.1.1, 2.1.2, 2.2.0, 2.2.1, 2.2.2, 3.0.0, 3.0.1, 3.0.2, 3.1.0, 3.1.1

によると，全バージョンに登場するパターン数は 55 から 567 であり，パターン全体に占める割合は 0.6%から 16.9%と少なくなっている．

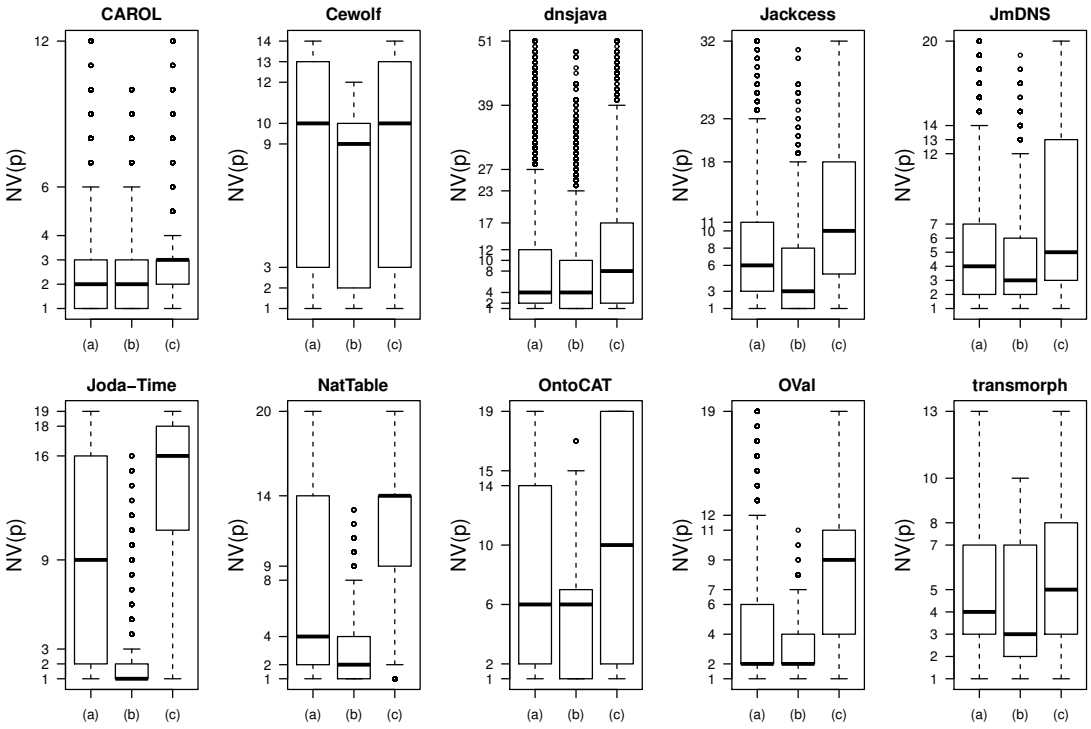
このコーディングパターンは，不安定であり検出されるバージョン数は少ないという結果は，Kim らによるコードクローンの安定性を調査した研究 [40] の結果と同様であった．多くのコードクローンは，数バージョン以内に消滅する．そして，メソッド呼び出しを含むコーディングパターンは，コーディングパターンと成り得るため，消滅したコーディングパターンの中には，開発者のコードクローニング活動の影響を受けるものがある．

表 5.3: 実験結果

Program	#Pattern	#Stable Pattern	#Stable Pattern / #Pattern (%)	#Common Pattern in first & last versions	#Stable Pattern / #Common Pattern (%)
CAROL	6,425	112	1.7%	146	76.7%
Cewolf	2,622	155	5.9%	157	98.7%
dnsjava	17,284	108	0.6%	287	37.6%
Jackcess	7,576	192	2.5%	291	66.0%
JmDNS	8,625	55	0.6%	93	59.1%
Joda-Time	6,663	524	7.9%	815	64.3%
NatTable	6,762	66	1.0%	152	43.4%
OntoCAT	3,348	567	16.9%	593	95.6%
OVal	6,275	57	0.9%	117	48.7%
transmorph	3,609	187	5.2%	256	73.1%

表 5.4: LOC と#Pattern 間の相関係数

Program	PCC
CAROL	0.641
Cewolf	0.988
dnsjava	0.883
Jackcess	0.995
JmDNS	0.734
Joda-Time	0.984
NatTable	0.900
OntoCAT	0.967
OVal	0.670
transmorph	0.940



(a) 全パターンの $NV(p)$, (b) 最終バージョンに登場しないパターンの $NV(p)$,
(c) 最終バージョンに登場するパターンの $NV(p)$

図 5.2: $\boldsymbol{NV(p)}$の分布

5.3.3 最終バージョンに登場するパターン

図 5.2(a) のコーディングパターンを，最終バージョンに登場するパターン (c) と最終バージョンに登場しないパターン (b) の 2 グループに分類し，それぞれのグループに含まれるコーディングパターンの $NV(p)$ の分布を図中に示した．

(b) に含まれるパターンは最終バージョンには登場しないので，$NV(p)$ の取り得る最大値は $max(b) = max(c) - 1$ となる．実際の (b)，(c) 間の分布を比較すると，全てのソフトウェアで $(b) < (c)$ の関係であり，CAROL と Cewolf の 2 つのソフトウェアでは中央値の差が 1 であるが，その他の 8 個のソフトウェアの中央値の差は 1 より大きく，最大で 15 の差となっていた．よって，最終バージョンに登場するパターンは，それ以外のパターンと比較して，安定であると考えられる．

5.3.4 検出パターン数の変化

図 5.3〜図 5.12 中の棒グラフは，直前のバージョン $v-1$ と比較して，バージョン v で新しく登場したパターンの数（正の方向）と，バージョン v で消滅したパターンの数（負の方向）を示している．また，それぞれのバージョンで検出されたパターン数とソースコードの行数（LOC）を折れ線グラフで示した．

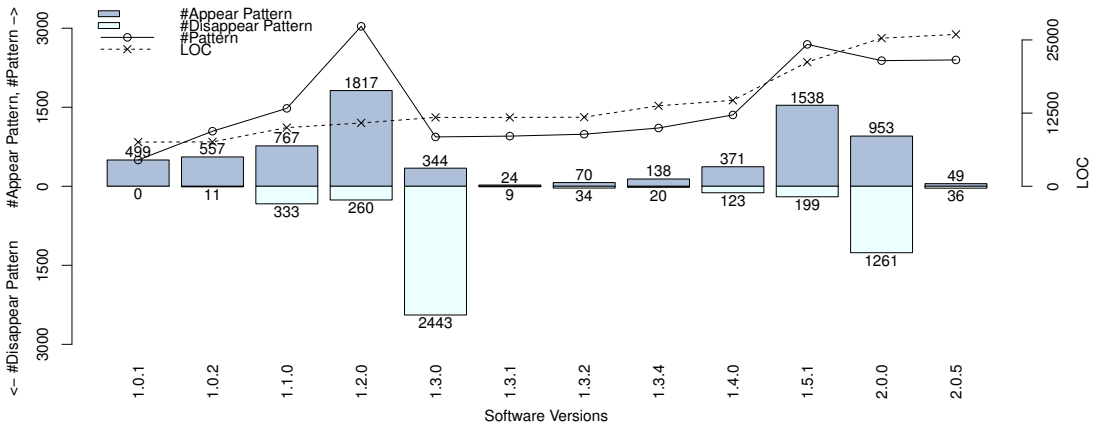


図 5.3: パターン数の変化 (CAROL)

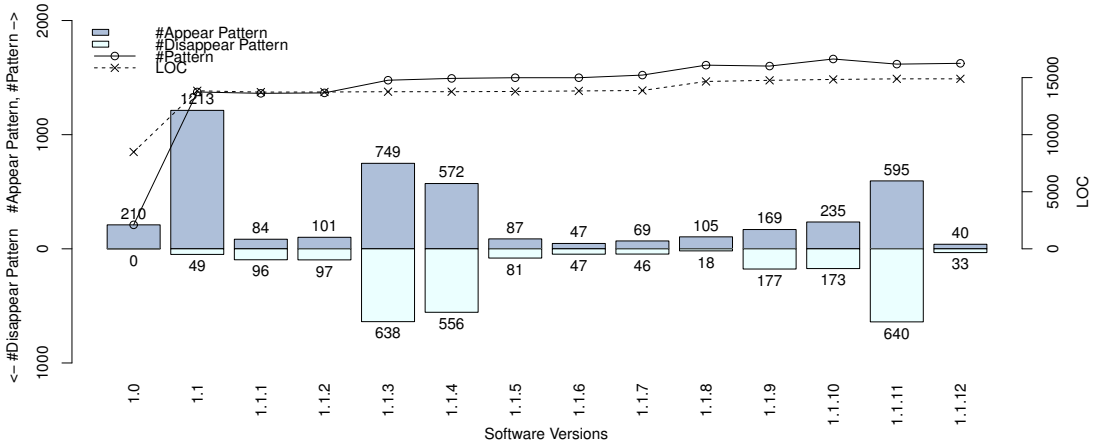


図 5.4: パターン数の変化 (Cewolf)

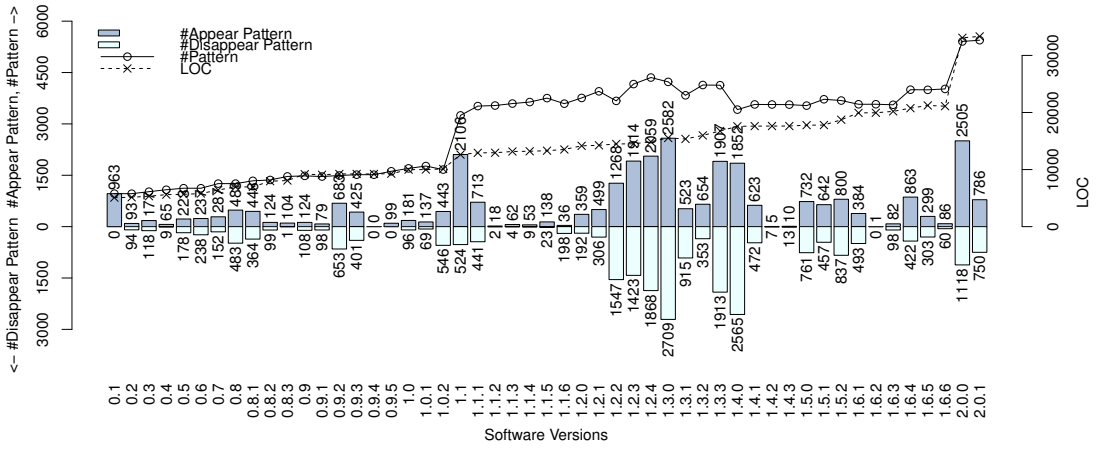


図 5.5: パターン数の変化 (dnsjava)

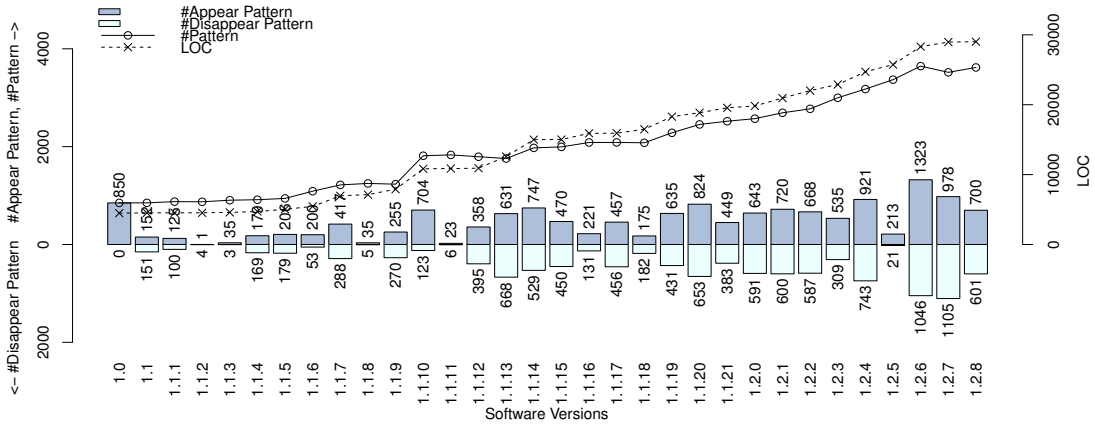


図 5.6: パターン数の変化 (Jackcess)

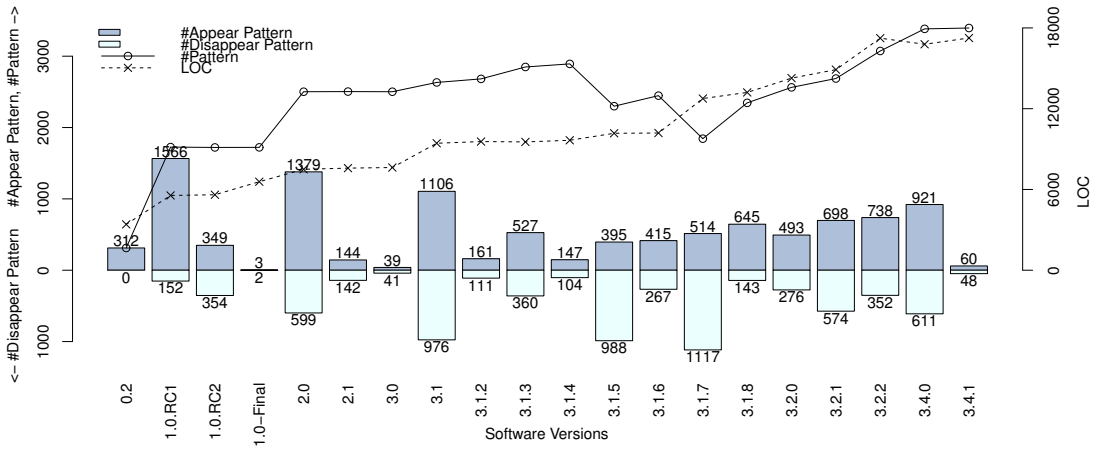


図 5.7: パターン数の変化 (JmDNS)

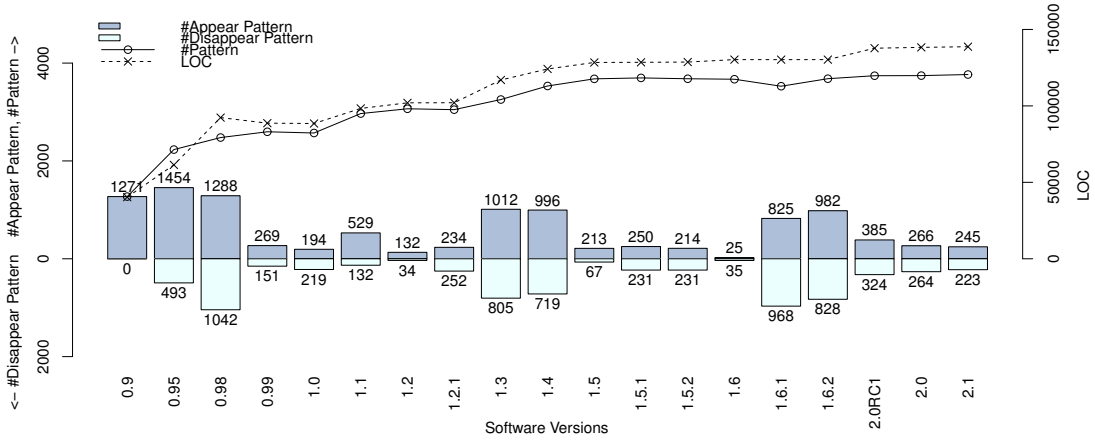


図 5.8: パターン数の変化 (Joda-Time)

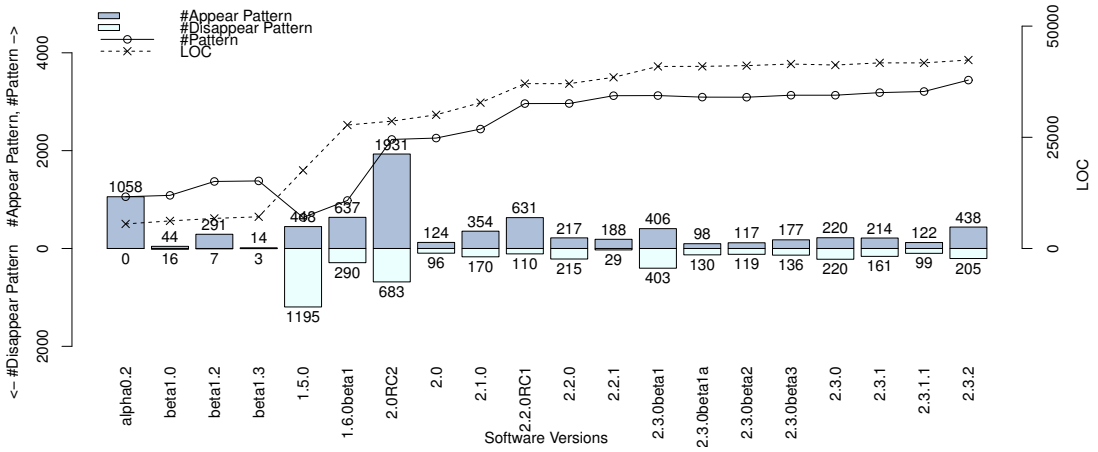


図 5.9: パターン数の変化 (NatTable)

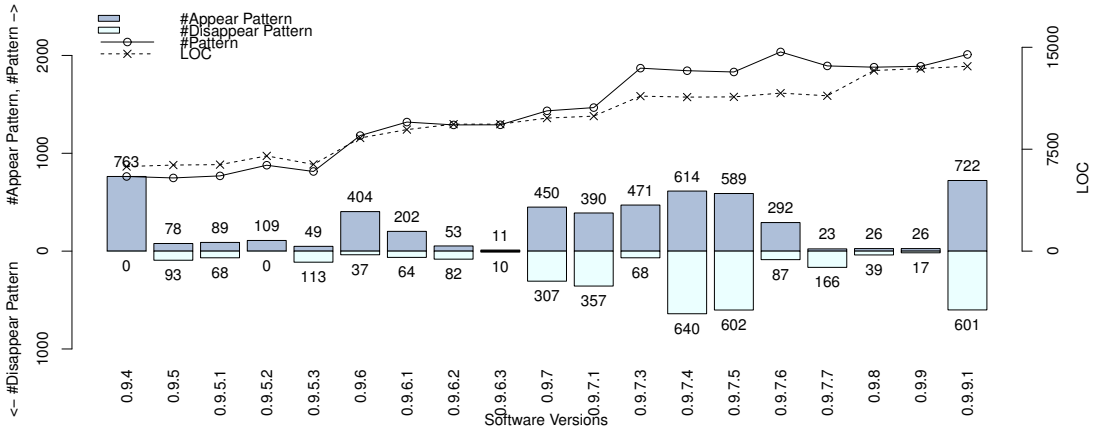


図 5.10: パターン数の変化 (OntoCAT)

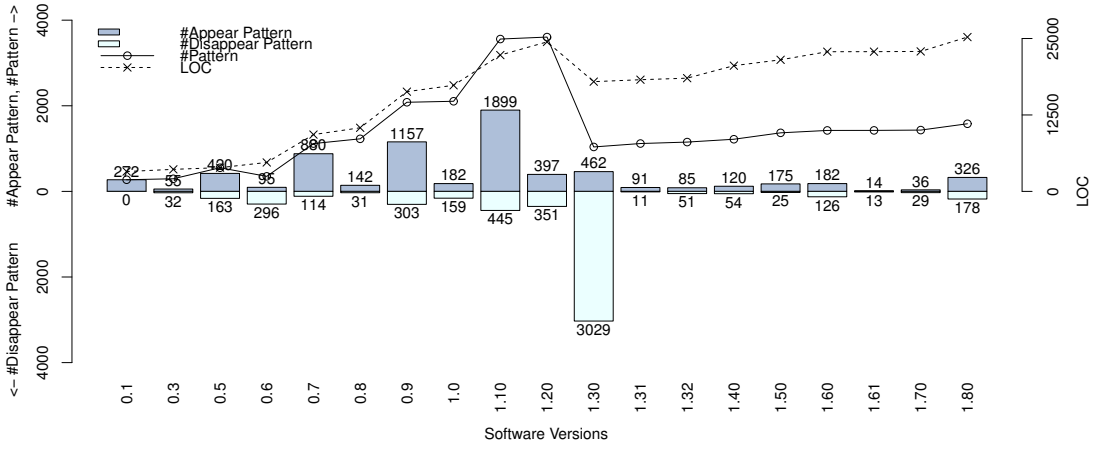


図 5.11: パターン数の変化 (Oval)

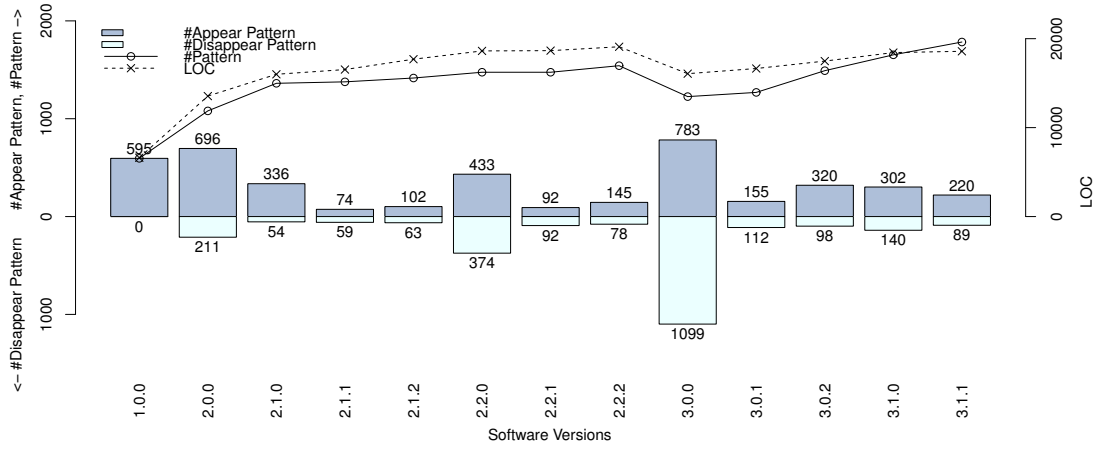


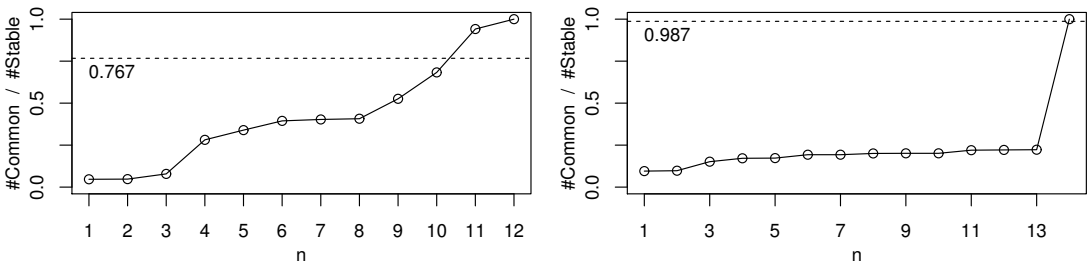
図 5.12: パターン数の変化 (transmorph)

1 つひとつのバージョンに注目すると，新しく登場したパターン数と，消滅したパターン数は，同程度である．これは，ソースコードが変更された結果，コーディングパターンが別のコーディングパターンへと変化しているためと考えられる．このことからも，コーディングパターン安定的なものでないと判断できる．

図 5.3〜図 5.12 中のソースコードの行数（LOC）と検出されたパターン数（#Pattern）は，全体として見ると，バージョンが進むにつれて徐々に増加している．それぞれの実験対象ソフトウェアの LOC と#Pattern について，ピアソンの積率相関係数（PCC）を算出し，その結果を表 5.2 に示した．PCC の値は，最小で 0.670，最大で 0.995 となった．10 個中 8 個のアプリケーションの PCC は 0.7 を超えているため，LOC と#Pattern の間には，強い正の相関がある．ゆえに，新しく追加されたコードにより，新たなパターンが生成されているといえる．

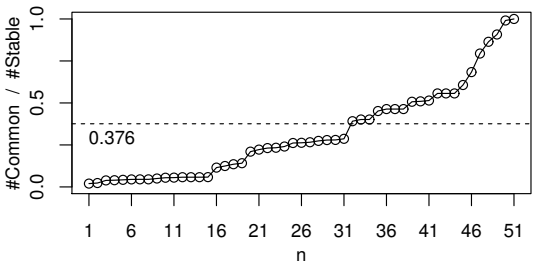
図 5.13 に，最終バージョンから数えて n バージョンに共通するパターン数が，全バージョンに共通するパターン中に占める割合の変化を示した．共通パターンの割合の増加は，緩やかであり，調査バージョン数を増やしていっても，なかなか共通バージョン数が増加しない．そのため，全バージョンに共通するような安定したパターンを抽出するために，最新バージョンから順番に調査していくことは効率的でない．

一方，最も古いバージョンと最も新しいバージョンの 2 バージョンに共通するパターンを抽出した場合の結果を表 5.2 の#Common Pattern in first & last versions に示した．また，これら 2 バージョンに共通するパターンのうち，全バージョンに共通するパターンの占める割合を，表 5.2 の#Stable Pattern / #Common Pattern (%) に示した．その結果，最も古いバージョンと最も新しいバージョンの 2 バージョンのみを調査し，その共通パターンを抽出すれば，そのうちの 37.6%から 98.7%が全バージョンに共通するパターンとなっていることがわかった．よって，全バージョンに共通するパターンを効率的に抽出するためには，なるべく離れたバージョンのコーディングパターンを調査すればよい．

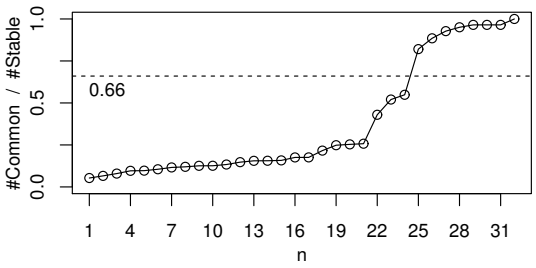


(a) CAROL

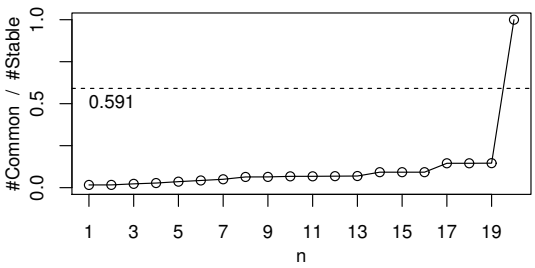
(b) Cewolf



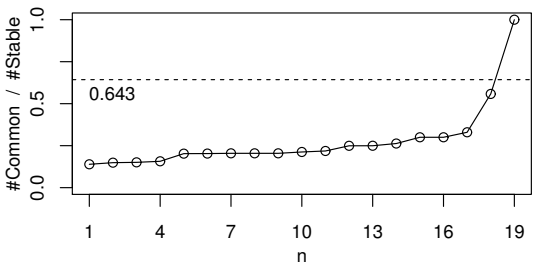
(c) dnsjava



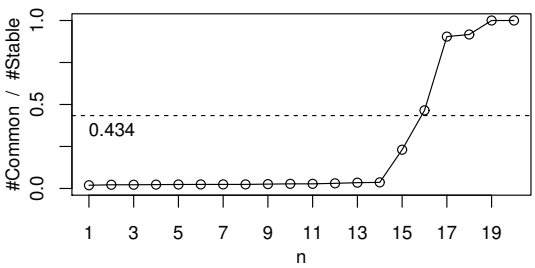
(d) Jackcess



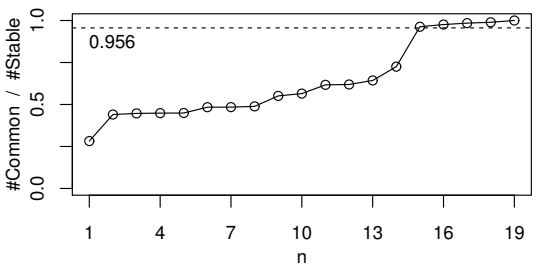
(e) JmDNS



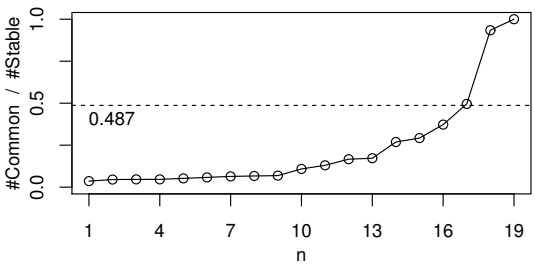
(f) Joda-Time



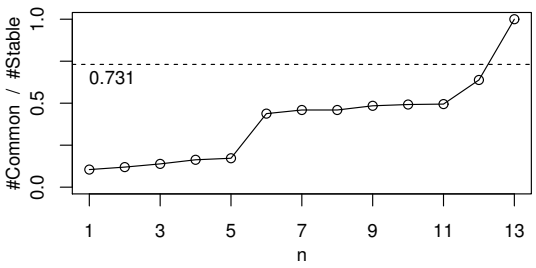
(g) NatTable



(h) OntoCAT



(i) OVal



(j) transmorph

図 5.13: 最終 n バージョンに共通するパターン

5.3.5 安定なパターンの例

本節では，今回の実験対象から検出されたコーディングパーンの中から特徴的なものを紹介する．全バージョンに登場するパターンのみを抽出し，その中から典型的と思われる6種類のパターンを選択した．

Pattern 1: *idiom*

次のような典型的なイテレータオブジェクトの使用例は，複数のアプリケーションから安定なパターンとして検出された．⟨iterator(), hasNext(), LOOP, next(), hasNext(), END-LOOP⟩ このパターンは，アプリケーションの種類を限定せず様々なアプリケーションから検出される最も有名なパターンの内のひとつであるため，開発者の関心度は低いと考えられる．しかし，このような基本的なパターンには，別要素が加わった多数の派生パターンが存在する．そのため，イテレータ関連のパターン全てを無視するのではなく，さらなる選別が必要である．

Pattern 2: *debugging*

⟨isDebugEnabled(), IF, debug(java.lang.String), END-IF⟩ は，Jackcess から検出されたパターンである．ソフトウェアがデバッグモードで実行されているときに，プログラムの振る舞いを切り替えるパターンである．この種のパターンは，プログラム中で一貫した記述を行う必要がある．

Pattern 3: *try-catch*

JmDNS から例外処理に関するパターン ⟨TRY, CATCH, printStackTrace(), END-TRY⟩ が発見された．このパターンは，処理の実行中に例外が発生した時，開発者に例外発生時の状況を伝えるために，printStackTrace() メソッドを呼び出すことでスタックメモリの状態を出力するパターンである．例外処理については，ソフトウェア固有の暗黙のルールが存在する場合もあり注意が必要である．

Pattern 4: *multi-threading*

このパターンは JmDNS の全バージョンから検出され，⟨SYNCHRONIZED, get(java.lang.Object), END-SYNCHRONIZED⟩ の 3 要素で構成されている．このパターンは，コレクションオブジェクトに対するアクセスの排他制御を表している．マルチスレッド関連のバグは，発見が難しいため，この種類のパターンに関連するコード，特にパターンに違反している部分のコードは，重点的な調査が必要である．

Pattern 5: *instantiation*

Joda-Tme からは，⟨getChronology()，org.joda.time.DateTime.<init>(long，org.joda.time.Chronology)⟩ というコーディングパターンが発見された．これは，アプリケーション固

有のパターンであり，Chronology オブジェクトから DateTime オブジェクトを作成するためのパターンである．このようなアプリケーション固有のパターンは，ソフトウェア開発プロジェクトに新たに参加した開発者が，ソフトウェアを理解するために役立つかもしれない．

Pattern 6: ***library***

NatTable は，GUI の作成に SWT[67] ライブラリを利用しており，⟨java.lang.Runnable.<init>(), asyncExec()⟩ という SWT の定型的な利用方法が検出された．このような特定のライブラリに関連するコーディングパターンを様々なアプリケーションから抽出すれば，実用的なサンプルコードを含む，ライブラリのマニュアルを作成できるかもしれない．

5.4 妥当性への脅威

バージョン間で同一のコーディングパターンを判別するために，メソッド名を基準としているため，メソッド名が変更された場合に同一パターンと判定できない．メソッド名の変更は，メソッドの振る舞いの変更を意味する場合があるため，本研究では，メソッド名の変更前後のパターンを同一パターンとして扱っていない．

5.5 まとめ

本研究では，バージョンをまたいで存在するコーディングパターンの安定性について調査を行った．コーディングパターン p を含むバージョンの数を $NV(p)$ として定義した．そして，12 以上のリリースバージョンを持つ 10 のソフトウェアからコーディングパターンを検出し $NV(p)$ を計測した．結果として，予想に反して大部分のコーディングパターンは，数バージョンのうちに消滅していることが判明した．全バージョンに登場するコーディングパターンの割合は，0.6%から 16.9%にすぎなかった．ゆえに，大部分のパターンは，早期に消滅してしまい，最終バージョンで検出されているパターンは，最終バージョンで検出されていないパターンと比較すると，安定しており，より多くのから検出される．また，最新バージョンと最古のバージョンに共通するコーディングパターンは，全バージョンに登場する安定したコーディングパターンである可能性が高いことがわかった．

ソフトウェア開発の初期段階においては，ソースコードが大幅に書き換えられることもあり，それに伴いパターンも変化する．さらに，異なるメジャーバージョン間では，パターンの共有が少ない．本研究では，メジャーバージョン，マイナーバージョンを区別せず，全てを含めて解析を行ったが，もし解析するバージョンの範囲を限定すれば，より興味にあったコーディングパターンを検出可能になるかもしれない．

第6章　むすび

本研究では，ソフトウェアの実装に関するコーディングパターンの特徴を分析し，コーディングパターンを利用する場合に参考となる基礎データを提供することを目指した．

まず，コーディングパターンの分析については，コーディングパターンが登場するソースコードを調査した．その結果，コーディングパターンには，アプリケーションの機能の実装に関連するパターン，一貫した例外処理の記述，偶然の一致など様々な種類があることが判明した．

次に，コーディングパターンのメトリクスの分析では，コーディングパターンを特徴付けるメトリクスを定義した．コーディングパターンのメトリクスを計測し分類することで，コーディングパターン利用者が，目的に応じて必要とするパターンを選び出すことができることを確認した．

そして，コーディングパターンとして検出された膨大なパターンの中から，再利用に適したパターンを選び出すための 1 つの方針として，コーディングパターンの存在するバージョン数の多いもの，つまり，ソフトウェアのライフサイクル中で長期間安定して存在するものを選び出す手法を提案した．ソフトウェア中に安定して存在するパターンは，抽出される全パターン中では，少数であることが判明した．

コーディングパターンの応用例としては，次のものが考えられる．

まず，ソースコードでコーディングパターンに該当する部分を編集する場合には，他のインスタンス部分についても，同様の編集を検討しなければならないため，エディタ上で，コーディングパターン部分をハイライト表示し開発者に注意を促すことが考えられる．

また，新規にメソッドを実装する場合においても，コーディングパターンからコードのひな形を作成することで，コーディングパターンの再利用を促進できる．

コーディングパターンから，コードを記述する上で守らなければならないルールを作成し既存のコードが，そのルールに従っているかどうかを検査するコードチェッカーへの応用が考えられる．

ライブラリの利用方法に関するパターンを複数のアプリケーションから抽出すれば，ライブラリの利用方法の学習に役立つ可能性がある．

参考文献

[1] Mithun Acharya, Tao Xie, Jian Pei, and Jun Xu. Mining API patterns as partial orders from source code: From usage scenarios to specifications. In *Proceedings of the Joint Meeting of the 11th European Software Engineering Conference and the 15th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pp. 25–34, 2007.

[2] Rakesh Agrawal and Ramakrishnan Srikant. Mining sequential patterns. In *Proceedings of the 11th International Conference on Data Engineering*, pp. 3–14, 1995.

[3] Christopher Alexander, Sara Ishikawa, and Murray Silverstein. *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press, 1977.

[4] ANTLR. `http://www.antlr.org/index.html`.

[5] Apache Struts. `http://struts.apache.org/`.

[6] Apache Tomcat. `http://tomcat.apache.org/`.

[7] AspectC++. `http://www.aspectc.org/`.

[8] AspectJ. `http://www.eclipse.org/aspectj/`.

[9] Azureus. `http://sourceforge.net/projects/azureus/`.

[10] Brenda S. Baker. A program for identifying duplicated code. *Computing Science and Statistics*, Vol. 6, pp. 49–57, 1992.

[11] Nicolas Bettenburg, Weyi Shang, Walid Ibrahim, Bram Adams, Ying Zou, and Ahmed E. Hassan. An empirical study on inconsistent changes to code clones at release level. In *Proceedings of the 16th Working Conference on Reverse Engineering*, pp. 85–94, 2009.

[12] David Binkley, Mariano Ceccato, Mark Harman, Filippo Ricca, and Paolo Tonella. Tool-supported refactoring of existing object-oriented code into aspects. *IEEE Transactions on Software Engineering*, Vol. 32, No. 9, pp. 698–717, 2006.

[13] Silvia Breu and Thomas Zimmermann. Mining aspects from version history. In *Proceedings of the 21st International Conference on Automated Software Engineering*, pp. 221–230, 2006.

[14] Magiel Bruntink, Arie van Deursen, Remco van Engelen, and Tom Tourwé. On the use of clone detection for identifying crosscutting concern code. *IEEE Transactions on Software Engineering*, Vol. 31, No. 10, pp. 804–818, 2005.

[15] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture: A System of Patterns*. John Wiley & Sons, 1996.

[16] CAROL. `http://carol.ow2.org/`.

[17] Cewolf. `http://cewolf.sourceforge.net/new/index.html`.

[18] James O. Coplien. *Advanced C++ Programming Styles and Idioms*. Addison-Wesley, 1991.

[19] Hironori Date, Takashi Ishio, and Katsuro Inoue. Investigation of coding patterns over version history. In *Proceedings of the 4th International Workshop on Empirical Software Engineering in Practice*, pp. 40–45, 2012.

[20] dnsjava. `http://www.dnsjava.org/`.

[21] Fernando Castor Filho, Alessandro Garcia, and Cecília Mary F. Rubira. Extracting error handling to aspects: A cookbook. In *Proceedings of the 23rd International Conference on Software Maintenance*, pp. 134–143, 2007.

[22] Martin Fowler. *Analysis Patterns: Reusable Object Models*. Addison-Wesley, 1996.

[23] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[24] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[25] Joseph Gil and Itay Maman. Micro patterns in java code. In *Proceedings of the 20th Conference on Object Oriented Programming, Systems, Languages, and Applications*, pp. 97–116, 2005.

[26] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification, Third Edition*. Addison-Wesley Longman, Amsterdam, 3 edition, June 2005.

[27] Jan Hannemann, Gail C. Murphy, and Gregor Kiczales. Role-based refactoring of crosscutting concerns. In *Proceedings of the 4th International Conference on Aspect-Oriented Software Development*, pp. 135–146, 2005.

[28] Terry Hon and Gregor Kiczales. Fluid AOP join point models. In *Proceedings of the 2nd Asian Workshop on Aspect-Oriented Software Development*, pp. 14–17, 2006.

[29] Takashi Ishio, Hironori Date, Tatsuya Miyake, and Katsuro Inoue. Mining coding patterns to detect crosscutting concerns in java programs. In *Proceedings of the 15th Working Conference on Reverse Engineering*, pp. 123–132, 2008.

[30] jackcess. `http://jackcess.sourceforge.net/`.

[31] jEdit. `http://www.jedit.org/`.

[32] JHotDraw. `http://www.jhotdraw.org/`.

[33] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *Proceedings of the 29th International Conference on Software Engineering*, pp. 96–105, 2007.

[34] JmDNS. `http://sourceforge.net/projects/jmdns/`.

[35] Joda-Time. `http://www.joda.org/joda-time/`.

[36] Huzefa Kagdi, Michael Collard, and Jonathan Maletic. An approach to mining call-usage patterns with syntactic context. In *Proceedings of the 22nd International Conference on Automated Software Engineering*, pp. 457–460, 2007.

[37] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. CCFinder: A multi-linguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, Vol. 28, No. 7, pp. 654–670, 2002.

[38] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *Proceedings of the 11th European Conference on Object-Oriented Programming*, pp. 220–242, 1997.

[39] Miryung Kim, Lawrence Bergman, Tessa Lau, and David Notkin. An ethnographic study of copy and paste programming practices in OOPL. In *Proceedings of the 2004 International Symposium on Empirical Software Engineering*, pp. 83–92, 2004.

[40] Miryung Kim, Vibha Sazawal, David Notkin, and Gail C. Murphy. An empirical study of code clone genealogies. In *Proceedings of the Joint Meeting of the 10th European Software Engineering Conference and the 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 187–196, 2005.

[41] Jens Krinke. Identifying similar code with program dependence graphs. In *Proceedings of the 8th Working Conference on Reverse Engineering*, pp. 301–309, 2001.

[42] Jens Krinke. Mining control flow graphs for crosscutting concerns. In *Proceedings of the 13th Working Conference on Reverse Engineering*, pp. 334–342, 2006.

[43] Charles W. Krueger. Easing the transition to software mass customization. In *Proceedings of the 4th International Workshop on Software Product-Family Engineering*, pp. 282–293, 2001.

[44] Zhenmin Li and Yuanyuan Zhou. PR-Miner: Automatically extracting implicit programming rules and detecting violations in large software code. In *Proceedings of the Joint Meeting of the 10th European Software Engineering Conference and the 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 306–315, 2005.

[45] Simone Livieri, Yoshiki Higo, Makoto Matushita, and Katsuro Inoue. Very-large scale code clone analysis and visualization of open source programs using distributed ccfinder: D-ccfinder. In *Proceedings of the 29th International Conference on Software Engineering*, pp. 106–115, 2007.

[46] Marius Marin. Reasoning about assessing and improving the seed quality of a generative aspect mining technique. In *Proceedings of the 2nd International Linking Aspect Technology and Evolution Workshop*, 2006.

[47] Marius Marin, Leon Moonen, and Arie van Deursen. Documenting typical crosscutting concerns. In *Proceedings of the 14th Working Conference on Reverse Engineering*, pp. 31–40, 2007.

[48] Marius Marin, Arie van Deursen, and Leon Moonen. Identifying aspects using fan-in analysis. In *Proceedings of the 11th Working Conference on Reverse Engineering*, pp. 132–141, 2004.

[49] Radu Marinescu. Detection strategies: Metrics-based rules for detecting design flaws. In *Proceedings of the 20th International Conference on Software Maintenance*, pp. 350–359, 2004.

[50] Tatsuya Miyake, Takashi Ishio, Koji Taniguchi, and Katsuro Inoue. Towards maintenance support for idiom-based code using sequential pattern mining. In *Proceedings of the 3rd Asian Workshop on Aspect-Oriented Software Development*, 2007.

[51] NatTable. `http://sourceforge.net/projects/nattable/`.

[52] Craig G. Nevill-Manning and Ian H. Witten. Identifying hierarchical structure in sequences: A linear-time algorith. *Journal of Artificial Intelligence Research*, Vol. 7, pp. 67–82, 1997.

[53] Anh Tuan Nguyen, Tung Thanh Nguyen, Hoan Anh Nguyen, Ahmed Tamrawi, Hung Viet Nguyen, Jafar Al-Kofahi, and Tien N. Nguyen. Graph-based pattern-oriented, context-sensitive source code completion. In *Proceedings of the 34th International Conference on Software Engineering*, pp. 69–79, 2012.

[54] Tung Thanh Nguyen, Hoan Anh Nguyen, Nam H. Pham, Jafar M. Al-Kofahi, and Tien N. Nguyen. Graph-based mining of multiple object usage patterns. In *Proceedings of the Joint Meeting of the 12th European Software Engineering Conference and the 17th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pp. 383–392, 2009.

[55] OntoCAT. `http://www.ontocat.org/`.

[56] OVal. `http://oval.sourceforge.net/`.

[57] Matthew M. Papi, Mahmood Ali, Telmo Luis Correa Jr., Jeff H. Perkins, and Michael D. Ernst. Practical pluggable types for java. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, pp. 201–211, 2008.

[58] Jian Pei, Jiawei Han, Behzad Mortazavi-Asl, Helen Pinto, Qiming Chen, Umeshwar Dayal, and Mei-Chun Hsu. Prefixspan: Mining sequential patterns efficiently by prefix-projected pattern growth. In *Proceedings of the 17th International Conference on Data Engineering*, pp. 215–224, 2001.

[59] Ruby on Rails. `http://rubyonrails.org/`.

[60] SableCC. `http://sablecc.org/`.

[61] Macneil Shonle, William G. Griswold, and Sorin Lerner. Beyond refactoring: A framework for modular maintenance of crosscutting design idioms. In *Proceedings of the Joint Meeting of the 11th European Software Engineering Conference and the 15th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pp. 175–184, 2007.

[62] Software Engineering Standards Committee of the IEEE Computer Society. *IEEE Standard 1517-1999(R2004): IEEE Standard for Information Technology – Software Life Cycle Processes – Reuse Processes*. IEEE, 1999.

[63] Spring Framework. `http://projects.spring.io/spring-framework/`.

[64] Jeffrey Stylos and Steven Clarke. Usability implications of requiring parameters in objects' constructors. In *Proceedings of the 29th international conference on Software Engineering*, pp. 529–539, 2007.

[65] Vijay Sundaresan, Laurie Hendren, Chrislain Razafimahefa, Raja Vallée-Rai, Patrick Lam, Etienne Gagnon, and Charles Godin. Practical virtual method call resolution for java. In

Proceedings of the 15th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, pp. 264–280, 2000.

[66] Toshihide Sutou, Keiichi Tamura, Yasuma Mori, and Hajime Kitakami. Design and implementation of parallel modified span method. In *Proceedings of the 5th International Symposium on High Performance Computing*, pp. 412–422, 2003.

[67] SWT. `http://www.eclipse.org/swt/`.

[68] Suresh Thummalapenta and Tao Xie. PARSEWeb: A programmer assistant for reusing open source code on the web. In *Proceedings of the 22nd International Conference on Automated Software Engineering*, pp. 204–213, 2007.

[69] transmorph. `https://github.com/cchabanois/transmorph`.

[70] Robin van der Rijst, Marius Marin, and Arie van Deursen. Sort-based refactoring of crosscutting concerns to aspects. In *Proceedings of the 4th International Linking Aspect Technology and Evolution Workshop*, 2008.

[71] Jianyong Wang, Jiawei Han, and Chun Li. Frequent closed sequence mining without candidate maintenance. *IEEE Transactions on Knowledge and Data Engineering*, Vol. 19, No. 8, pp. 1042–1056, 2007.

[72] Tao Xie and Jian Pei. Mapo: Mining api usages from open source repositories. In *Proceedings of the 2006 International Workshop on Mining Software Repositories*, pp. 54–57, 2006.

[73] Xifeng Yan, Jiawei Han, and Ramin Afshar. Clospan: Mining closed sequential patterns in large datasets. In *Proceedings of the SIAM International Conference on Data Mining*, pp. 166–177, 2003.

[74] Gang Zhang, Xin Peng, Zhenchang Xing, and Wenyun Zhao. Cloning practices: Why developers clone and what can be changed. In *Proceedings of the 28th IEEE International Conference on Software Maintenance*, pp. 285–294, 2012.

[75] 服部 剛之, 肥後 芳樹, 楠本 真二, 井上 克郎. コードクローンの分布情報を用いた特徴抽出手法の提案. ソフトウェア信頼性研究会 第 3 回ワークショップ論文集, pp. 9–17, 2006.

[76] 肥後 芳樹, 楠本 真二, 井上 克郎. コードクローン検出とその関連技術. 電子情報通信学会論文誌, Vol. J91-D, No. 6, pp. 1465–1481, 2008.

[77] 中山 崇, 松下 誠, 井上 克郎. ソースコードの差分を用いた関数呼び出しパターン抽出手法の提案. 情報処理学会研究報告, Vol. 2006-SE-151, No. 35, pp. 49–56, 2006.

[78] 結城 浩. Java 言語で学ぶデザインパターン入門 マルチスレッド編. ソフトバンククリエイティブ, 増補改訂版, 2006.