レガシーシステムマイグレーション支援のための
ソフトウェア分析・設計手法の研究

提出先　大阪大学大学院情報科学研究科

提出年月　2022 年 1 月

岡田　譲二

論文一覧

主要論文

1. 岡田譲二, 石尾隆, 坂田祐司, 井上克郎. “AutoSort: レガシーシステム分析のためのプログラミング言語の判定支援手法”, 情報処理学会論文誌, Vol.59, No.6, pp.1405–1414, 2018 年 6 月. (学術論文)
2. Joji Okada, Takashi Ishio, Yuji Sakata, Katsuro Inoue. “Towards Classification of Loop Idioms Automatically Extracted from Legacy Systems”, in Proceedings of the 13th International Workshop on Software Clones (IWSC 2019), pp.34–35, Hangzhou, China, February 2019.（国際会議録）
3. 岡田譲二, Abhay Parvate, 石尾隆, 坂田祐司, 井上克郎. “レガシーシステム移行時の性能劣化を改善するリファクタリング支援手法の提案”, デジタルプラクティス（採録決定）. (学術論文)

関連論文

1. 岡田譲二, 坂田祐司. “変更要件に関係するプログラム特定のための処理名抽出”, 情報処理学会研究報告, Vol.2012-SE-175, 2012 年 3 月.（国内会議録）
2. 岡田譲二, Abhay Parvate, 石尾隆, 坂田祐司, 井上克郎. “レガシーシステム分析のためのパターンマッチによるソースコードの自動抽象化”, ソフトウェアエンジニアリングシンポジウム 2017 論文集, Vol.2017, pp.280–281, 2017 年 8 月.（国内会議録）

内容梗概

企業においては，レガシーシステムと呼ばれるビジネス上は重要だが，保守の継続が困難な巨大なメインフレーム上の基幹システムが多数存在している．レガシーシステムは，ビジネス上重要な一方で，時代遅れの技術を用いて長年保守し続けられたため，その変更には多くの期間とコストが必要になるという問題がある．

この問題を解決するために，ソフトウェアのコンポーネントやプラットフォームを交換，再開発，移行することで，既存のソフトウェアシステムを進化させるレガシーシステムの現代化（モダナイゼーション）がしばしば行われ，そのアプローチとしてラッピング，マイグレーション，再構築といったアプローチが存在する．

中でも，異なるプログラミング言語に機械的に変換したり，全く新しいアーキテクチャや技術に置き換えたりするマイグレーションというアプローチは，実施のリスクも小さく得られる効果も大きい．

システムをマイグレーションする戦略としては Chicken Little 戦略と呼ばれる短期間のステップを少人数で繰り返してレガシーシステムを少しずつマイグレーションしていく戦略がよく知られている [1]．この戦略は，各ステップが失敗してもそのステップだけをやり直せばよいのでリスクが小さい．

本研究では，著者が所属する企業において実施したマイグレーションプロジェクトにおいて実際に直面した課題とその解決策について述べる．このマイグレーションプロジェクトにおいては，Chicken Little 戦略によるシステム再構築の 11 のステップの中の「レガシーシステムの分析」と「レガシーシステムの構造の分解」,「新アプリケーションの設計」について，それぞれ課題が発生していた．

「レガシーシステムの分析」では，分析の前提として対象のソースコードファイルのプログラミング言語が判明している必要があるが, 実際には拡張子が存在せずプログラミング言語が不明なソースコードファイルが多数存在しており, この判定を手作業で行うと多大な労力が必要となるという課題があった．本研究ではプログラミング言語の判定作業を支援するために，手作業で判定すべきファイルの代表をクラスタリングによって選出する手法を提案した．正確な判定を支援するため，提案手法はパターンマッチによる自動判定と，手作業での判定結果を用いた解析による判定誤りの補正をクラスタリングに組み合わせて用いる．提案手法の評価として，人手で正解のプログラミング言語を付与した 2 つの実際のレガシーシステムのファイル集合に対して本

手法を適用した．その結果，提案手法は 19 万ファイルのうち，99.49% のファイルを正しく分類できることを確認した．また，これらのファイルに対する人手での判定は 3.3 人月の工数が必要だったが，提案手法は 8 時間の計算時間と，人手による 15 分の確認だけで判定を完了した．

「レガシーシステムの構造の分解」では，有識者や既存の設計書がない状況で，システムの保守開発を行うためには，業務用語とシステムの用語の対応関係を把握することが重要である．本研究では，ソースコードを処理機能という粒度で分割し，入出力されるファイル名と処理機能から機能名を自動的に付与する手法を提案するとともに，この手法を実現するツールを試作した．また，COBOL 言語で記述されたプログラムに適用し，定性的な評価を行ったところ，本手法によって業務用語とシステムの用語の対応関係が把握しやすくなったことを示した．

また「新アプリケーションの設計」では，マイグレーション後の性能問題という課題があった．こういった実行時性能の劣化を防ぐため，プログラムを分散処理などに書き換えることが多いが，この書き換えは単純ではなく工数がかかる．本研究では，新しいプログラミング言語に書き換えられたプログラムを並列実行可能な形に書き換える作業を支援することで，移行後のプログラムの実行時性能を改善する手法について提案した．評価実験として，二つの実際のレガシーシステムのプログラム 7,565 本に対して提案手法を適用した．書き換え前後のプログラムに同一の入力データを与えることで振る舞いを保った書き換えができていることを確認するとともに，書き換え前後のプログラムの実行時間の評価を行うことで，提案手法の有効性を確認した．本手法で実際のレガシーシステムのソースコード 3,529 本の書き換えができ，書き換え前と比較して実行時性能を 2 倍から 50 倍改善させることができた．

これらの手法により，工数や予算が大きくかかりがちな企業におけるレガシーシステムのマイグレーションプロジェクトにおいて，一定の効率化を果たすことができたと考えられる．

謝辞

本研究を行うにあたり，日頃から様々なご指導を賜りました，大阪大学大学院情報科学研究科コンピュータサイエンス専攻井上克郎教授に，心から感謝申し上げます．

本論文を執筆するにあたり，様々なご指導ご助言を頂きました，大阪大学大学院情報科学研究科コンピュータサイエンス専攻楠本真二教授，ならびに八木康史教授に感謝申し上げます．

本研究を行うにあたり，研究方針など様々なご指導ご助言を頂きました，奈良先端科学技術大学院大学先端科学技術研究科情報科学領域石尾隆准教授に，心から感謝申し上げます．

また，本研究の実施にあたり，株式会社 NTT データの皆様には研究の機会や実践の場を与えていただきましたことを感謝いたします．その中でも直接研究を支援し，様々なアドバイスを頂いた，株式会社 NTT データ 坂田祐司氏にこの場でお礼を述べさせていただきます．研究についての支援やアドバイスを頂きました株式会社 NTT データ 竹之内啓太氏，同 横井一輝氏にはこの場で御礼申し上げます．本研究の実験においては，株式会社 NTT データ先端技術 Abhay Parvate 氏，同 Sarang Khairnar 氏，同 Biswajeet Rout 氏，同 羽野宇賢氏，株式会社 NTT データニューソン 永田宏樹氏，同 鈴木大地氏に多大なご協力をいただきました．篤く感謝を述べさせていただきます．

最後に，日々の研究生活の中でご助言，ご協力を頂いた，大阪大学大学院情報科学研究科コンピュータサイエンス専攻ソフトウェア工学講座の皆様に厚く御礼申し上げます．

目次

図目次

表目次

第 1 章

はじめに

1.1　レガシーシステム

レガシーシステムとは，ビジネス上は重要だが，保守の継続が困難な巨大なメインフレーム上の基幹システムのことである [2].

レガシーシステムは企業の中核をなすシステムのため，その企業に現在でも多くの収益をもたらしており，その障害は日常業務に深刻な影響を与える．また，長期間正常に動作し続けてきた信頼性の高いシステムでもある．

その一方で，レガシーシステムの多くは,20 〜 30 年以上前に作られた古いシステムであり，時代遅れの技術を用いて長年保守し続けられたため，その変更には多くの期間とコストが必要になるという問題がある [3]．レガシーシステムは，ビジネスのコアなシステムが故に, 新たなビジネス要求によって頻繁に変更され続けており，その結果，保守が困難な構造化されていないソースコードになっている．また，長年保守されているが故に，ともとのプログラマーが退職していたり，文書が古いままになっていたりする．これらの理由から，変更時にソースコードの調査や意図しない場所への影響調査が必要となり，その変更や修正には多くの期間とコストが必要になる．

現在の急速に変化するビジネス環境では，組織内外の変化，法律や規制の変化など，様々な変化に迅速に対応しなければならない．レガシーシステムも同様にそういったビジネス環境の変化に追随して変更を迅速に行われる必要がある．こういった状況において，レガシーシステムの変更や修正に多くの期間とコストが必要になるのは，企業において致命的な問題である．

1.2　レガシーシステムのマイグレーション

レガシーシステムの変更や修正に多くの期間とコストがかかる問題を解決するために，レガシーシステムの現代化（モダナイゼーション）がしばしば行われる．レガシーシステムの現代化とは，保守コストの削減と柔軟性の向上を目的に，ソフトウェアの

コンポーネントやプラットフォームを交換，再開発，移行することで，既存のソフトウェアシステムを進化させるプロセスである．

レガシーシステムがいまだに多くの企業で利用されているため，レガシーシステムの現代化は多くの企業の IT 戦略上重要な課題であり，レガシーシステムの現代化サービスを提供する IT 企業も多い．調査会社の MarketsandMarkets 社によると，レガシーシステムの現代化サービスの市場は 2020 年で 114 億米ドルで，2025 年には 248 億米ドルまで成長すると予測している [4].

レガシーシステムの現代化にはいくつかのアプローチが存在する [1].

- ラッピング
- マイグレーション
- 再構築

ラッピングは，既存のデータやプログラム，インタフェースを新しいインタフェースで囲むアプローチであり，古いコンポーネントに新しい操作性を与えたり, 新しい外観を提供する．ラッピングは既存のレガシーシステムに変更を加えないため，他のアプローチと比較して最も少ない変更量で済む．操作性や外観に対する柔軟性は向上する一方で，既存のレガシーシステムの内部ロジックの保守性などは向上しないため，保守コストの削減や柔軟性の向上の効果は限定的である．

マイグレーションは，実装，設計，仕様，要件など，レガシーシステムのできるだけ多くの部分を再利用することを目指すアプローチであり，異なるプログラミング言語に機械的に変換したり，全く新しいアーキテクチャや技術に置き換える．マイグレーションは先述したラッピングと後述する再構築の中間のアプローチであり，ラッピングよりも変更量が多いため，多くのコストがかかるものの得られる効果も大きい．また，再構築に比べて短期間で実施できるため，実施のリスクも小さい．

再構築は，レガシーシステムの仕様や要件を調査・理解した上で最新のプラットフォームやアーキテクチャ，ツール，データベースを使ったシステムに置き換えるアプローチである．レガシーシステムは，もともとの開発者が退職などでいなかったり，文書が十分に整備されていないことが多く，そのシステムについての知識が欠如しているため，仕様や要件を調査・理解するのは難しい．また，既存のレガシーシステムを捨てて新しく別のシステムを構築することとなるため，再構築は他のアプローチと比較して最も多くの変更が必要となり，実施期間やコストも最も大きくなる．

レガシーシステムの現代化は，その実施期間が長くなるにつれ，プロジェクト失敗のリスクが増大する [5]．現代化プロジェクトの期間が長くなると，その間のビジネス状況の変化も大きくなり，その変化に追随するためにシステムへの変更が入る．その変更の対応のために，さらに現代化プロジェクトの期間が伸びるという悪循環が起きる．特に，現代化プロジェクトの期間が長くなる再構築アプローチによるプロジェクトは失敗するリスクが高い．

再開発が許容できないほどリスクが高く，狙っている効果に対してラッピングが適さない場合は，マイグレーションが最良の選択となる．

システムをマイグレーションする戦略としては Chicken Little 戦略がよく知られている [5]．Chicken Little 戦略は，短期間のステップを少人数で繰り返してレガシーシステムを少しずつマイグレーションしていく戦略である．この戦略は，各ステップが失敗してもそのステップだけをやり直せばよいのでリスクが小さい．

Chicken Little 戦略によるシステム再構築のステップとしては，以下の 11 ステップがある [6]．

1. レガシーシステムの分析
2. レガシーシステムの構造の分解
3. 新インタフェースの設計
4. 新アプリケーションの設計
5. 新データベースの設計
6. 新環境の構築
7. 必要なゲートウェイの構築
8. レガシーデータベースのマイグレーション
9. レガシーアプリケーションのマイグレーション
10. レガシーインタフェースのマイグレーション
11. 新システムへの切り替え

レガシーシステムの分析は，レガシーシステムの仕様を理解し，新システムの要件を記述する作業である．レガシーシステムを理解するための情報源としては，ドキュメントやソースコード，開発者や利用者のノウハウなどがある．

レガシーシステムの構造の分解は，レガシーシステムに存在するプロシージャコールなどのモジュール間の依存関係を排除し，明確なインタフェースとして整理する検討作業である．

新インタフェースの設計と新アプリケーションの設計では，ステップ 2 の「レガシーシステムの構造の分解」で整理したインタフェースを新インタフェースとして設計し，そのインタフェースを境目として各アプリケーションモジュールの機能を設計する．

新データベースの設計では，レガシーシステムのデータ構造を基に新システムのデータ構造を設計する．レガシーシステムによってはデータとアプリケーションコードの区別が曖昧な場合が多く，その場合はアプリケーションコードの内容も新システムのデータ構造として設計する必要がある．

新環境の構築は，新システムが処理する処理量や UI などの非機能要件を考慮しながらハードウェア，ネットワークを構築する．

必要なゲートウェイの構築では，レガシーアプリケーションと新データベース，新アプリケーションとレガシーデータベースの間にそれぞれフォワードゲートウェイとリ

バースゲートウェイと呼ばれる二つのゲートウェイを構築する．このフォワードゲートウェイはレガシーアプリケーションから新データベースへの全ての呼び出しの間に挟まり，新データベースの結果をレガシーデータベースの結果に変換する．同様にリバースゲートウェイは，新アプリケーションからレガシーアプリケーションへの全ての呼び出しの間にも挟まり，レガシーデータベースの結果を新データベースの結果に変換する．

レガシーデータベース，レガシーアプリケーション，レガシーインタフェースのマイグレーションは並行して行い，徐々に新システムへの切り替えを行う．一度に全てのデータベース，アプリケーション，インタフェースをマイグレーションするのではなく，それぞれのサブセットを選び，それらに対応するフォワードゲートウェイとリバースゲートウェイを構築し，ゲートウェイを構築できたところから新システムを切り替える．これを反復的に繰り返すことで，徐々にレガシーシステムのデータベース，アプリケーション，インタフェースが小さくなり，新システムのデータベース，アプリケーション，インタフェースが大きくなる．最終的にはレガシーシステムがなくなり，新システムだけが残る．

1.3 本研究の概要

本研究では，著者が所属する企業において実施したマイグレーションプロジェクトにおいて実際に直面した課題とその解決策について述べる．このマイグレーションプロジェクトにおいては，レガシーシステムマイグレーションのステップ中の「レガシーシステムの分析」と「レガシーシステムの構造の分解」,「新アプリケーションの設計」についてそれぞれ課題が発生していた．

「レガシーシステムの分析」では，分析の前提として対象のソースコードファイルのプログラミング言語が判明している必要があるが, 実際には拡張子が存在せずプログラミング言語が不明なソースコードファイルが多数存在しており, この判定を手作業で行うと多大な労力が必要となるという課題があった．本研究ではプログラミング言語の判定作業を支援するために，手作業で判定すべきファイルの代表をクラスタリングによって選出する手法を提案する．正確な判定を支援するため，提案手法はパターンマッチによる自動判定と，手作業での判定結果を用いた解析による判定誤りの補正をクラスタリングに組み合わせて用いる．提案手法の評価として，人手で正解のプログラミング言語を付与した 2 つの実際のレガシーシステムのファイル集合に対して本手法を適用した．その結果，提案手法は 19 万ファイルのうち，99.49% のファイルを正しく分類できることを確認した．また，これらのファイルに対する人手での判定は 3.3 人月の工数が必要だったが，提案手法は 8 時間の計算時間と，人手による 15 分の確認だけで判定を完了した．

「レガシーシステムの構造の分解」では，有識者や既存の設計書がない状況下で業務

用語とシステムの用語の対応関係を把握することが重要である．本研究では，ソースコードを処理機能という粒度で分割し，入出力されるファイル名と処理機能から機能名を自動的に付与する手法を提案するとともに，この手法を実現するツールを試作した．また，COBOL 言語で記述されたプログラムに適用し，定性的な評価を行ったところ，本手法によって業務用語とシステムの用語の対応関係が把握しやすくなったことを示した．

また「新アプリケーションの設計」では，マイグレーション後の性能問題という課題があった．レガシーシステムでは，データベースから取得した結果に対して、ループや条件分岐などの制御構造 (Loop Idiom) を用いて加工を行う手続き型のプログラムが数多く存在する．これらのプログラムを新しいプログラミング言語にマイグレーションする際に実行時性能の劣化がしばしば起こる．こういった実行時性能の劣化を防ぐため，プログラムを分散処理などに書き換えることが多いが，この書き換えは単純ではなく工数がかかる．本研究では，新しいプログラミング言語に書き換えられたプログラムを並列実行可能な形に書き換える作業を支援することで，移行後のプログラムの実行時性能を改善する手法について提案する．評価実験として，二つの実際のレガシーシステムのプログラム 7,565 本に対して提案手法を適用した．書き換え前後のプログラムに同一の入力データを与えることで振る舞いを保った書き換えができていることを確認するとともに，書き換え前後のプログラムの実行時間の評価を行うことで，提案手法の有効性を確認した．本手法で実際のレガシーシステムのソースコード 3,529 本の書き換えができ，書き換え前と比較して実行時性能を 2 倍から 50 倍改善させることができた．

1.4 各章の構成

以降，第 2 章ではレガシーシステムを分析するための前提となるプログラミング言語の判定について詳細を述べる．第 3 章では業務用語とシステム用語の対応関係を把握する手法について述べる．第 4 章ではレガシーシステムを再構築後の性能劣化を改善するリファクタリングについて詳述する．最後に，第 5 章では本論文のまとめと将来の研究方針について述べる．

第 2 章

レガシーシステム分析のためのプログラミング言語の判定支援手法

レガシーなメインフレームシステムには，拡張子が存在せずプログラミング言語が不明なソースコードファイルが多数存在する．レガシーシステムを分析する際には各ソースコードファイルのプログラミング言語を判定する必要があるが，これを手作業で行うと多大な労力が必要となってしまう．本研究ではプログラミング言語の判定作業を支援するために，手作業で判定すべきファイルの代表をクラスタリングによって選出する手法を提案する．正確な判定を支援するため，提案手法はパターンマッチによる自動判定と，手作業での判定結果を用いた解析による判定誤りの補正をクラスタリングに組み合わせて用いる．提案手法の評価として，人手で正解のプログラミング言語を付与した 2 つの実際のレガシーシステムのファイル集合に対して本手法を適用した．その結果，提案手法は 19 万ファイルのうち，99.49% のファイルを正しく分類できることを確認した．また，これらのファイルに対する人手での判定は 3.3 人月の工数が必要だったが，提案手法は 8 時間の計算時間と，人手による 15 分の確認だけで判定を完了した．

2.1 はじめに

マイグレーションの最初のステップである「レガシーシステムの分析」は，レガシーシステムの仕様を理解し，新システムの要件を記述する作業である．

レガシーシステムの仕様を理解するための情報源としては，ドキュメントやソースコード，開発者や利用者のノウハウなどが用いられる．中でも現行システムのソースコードを静的解析し，理解するというアプローチが多く用いられる [7]．静的解析手法の多くはソースコードの構文解析を行っており，解析を行うための前提条件として，対象となるソースコードのプログラミング言語が何であるかをあらかじめ把握しておく必要がある．

本研究では，レガシーシステムを構成するソースコードファイルに使用されているプログラミング言語を判定する問題を取り扱う．一般的なソフトウェアシステムでは，ソースコードファイルの拡張子やディレクトリ構造からプログラミング言語を容易に判定することができるが，レガシーシステムではプログラミング言語の判定は容易ではない．最も大きな理由として，レガシーシステムでは拡張子やディレクトリが存在しないことが挙げられる．メインフレームは拡張子という概念が生まれる前から存在しているため，ファイルに拡張子をつけるという文化がなく，実際に多くのメインフレームのソースコードファイルには拡張子が付与されていない [8]．また，ディレクトリという概念のないフラットなファイルシステムが使われており，プログラミング言語ごとにディレクトリを分けるようなファイル管理が行われておらず，システムを構成する複数の言語のソースコードファイルやプログラムが利用するデータファイルが１つの場所に混在する状態となっている．

プログラミング言語判定の単純な自動化，たとえば既知の複数の構文解析器を適用して成功したものを選ぶといった方式は，レガシーシステムに多様な言語が用いられることと，プロジェクト独自の言語が含まれている可能性から適用が困難である．長年の保守開発の中では，一時的な利用や性能上の問題解決という名目によって，たとえばスクリプト系の言語やアセンブラなど，そのプロジェクトで主に採用されている言語とは異なる言語も使われている場合が多い．これらの少数の言語は，保守開発者の入れ替わりによって存在すること自体さえも引き継がれること無く今に至ってしまい，現在の保守開発者にとって未知の言語となっていることがしばしばある．また，規模の大きなプロジェクトでは，そのプロジェクト独自の拡張をプリプロセッサやコンパイラに施すことがある．新しいキーワードを追加したり，文法を追加したりするといった独自の拡張がなされたプログラミング言語は，一見すると元々の COBOL だったり，PL/I に見えたりしても，通常の構文解析器ではまったく解析できないソースコードとなっていることがある．

プログラミング言語の判定においては，ファイル種別の分類も必要である．ファイル種別とは，たとえば C 言語におけるソースファイルとヘッダファイル，COBOL におけるメインファイルとコピーファイルというように，言語としては同一であるが言語処理系にとっての役割が異なるようなファイルの分類である．本研究では，構文解析の起点となるファイルをメインファイル，構文解析の起点とならないファイル（ヘッダファイル，コピーファイル等）をインクルードファイルと呼んで区別する．インクルードファイルの多くは単独での構文解析が不可能であり，適切な解析を実行するには，これらのファイル種別まで含めたプログラミング言語の判定を行うことが必要である．プログラミング言語に固有のキーワードなどに基づくパターンマッチは，ある程度の自動的な判定は可能であるが，このようなファイル種別の判定には不十分である．

ソースコードファイルが与えられたとき，知識を持った人間なら，そのファイルの内容からプログラミング言語を推定することはそれほど難しくない．しかし，レガシー

システムではソースコードファイルが数万本あることは珍しくなく，全てのファイルのプログラミング言語を人手で判定するには多大な労力が必要である．第一著者が関わったプロジェクトでは，プログラミング言語が不明なソースコードファイルが 17 万ファイル存在しており，そのうち 3 万ファイルは判定開始時に想定されていなかった未知のプログラミング言語のものであった．このファイル集合に対して，ファイル名の命名規約の活用や，ファイル内のキーワードを grep で検索するといった方法で人手で判定を行ったところ，2 人で 1.5 ヶ月間の作業が必要であった．

本研究では，大規模なレガシーシステムに大量に存在する，プログラミング言語が分からないソースコードファイルについて，そのプログラミング言語とファイル種別を判定する作業を支援する手法を提案する．提案手法のアイディアは，クラスタリングによってファイルを分類し，クラスタの代表として選ばれたファイルだけに対して人間がプログラミング言語の判定を行うというものである．人間の労力を削減するために，パターンマッチによる判定が容易なプログラミング言語は事前に判定を行って人間の判定対象から除外する．また，人間が判定した結果に基づいて構文解析を実行し，得られた情報を用いて誤りを補正することで，分類精度を向上させる．

評価実験として，2 つの実際のレガシーシステムのファイル集合に対して提案手法を適用した．人手で判定した結果を正解として既存手法との正確性の比較を行うとともに，計算時間，作業時間の評価を行うことで，提案手法の有効性を確認した．

以降，2.2 節では既存手法について紹介し，2.3 節で提案手法について述べる．2.4 節で評価実験の方法と結果を説明し，2.5 節でまとめと今後の課題を述べる．

2.2 既存手法

プログラミング言語の判定問題は，レガシーシステムに固有の問題ではない．ドキュメントやメール等に含まれるソースコード断片に使われているプログラミング言語の識別や，プログラミング言語間で共通の拡張子が用いられている場合への対応といった観点から，プログラミング言語判定の研究が行われている [9]．具体的な判定手法としては，パターンマッチを利用した手法と，教師あり機械学習による判定手法が提案されている．

Linguist [10] は，プログラミング言語ごとに構文の強調表示等を適切に行うことを目的とした，パターンマッチを用いたプログラミング言語の判定ツールである．プログラミング言語の中に含まれる特徴的なキーワードやパターンがあらかじめ列挙されており，それらのキーワードを含む，もしくは決められたパターンにマッチするファイルを，当該のプログラミング言語として判定する．パターンマッチによる判定手法は，他の言語に比べて特徴が明確なプログラミング言語に関しては有効なものの，頻出する単語を共通して持つプログラミング言語を判定する場合，適切なキーワードを作成することが難しい．また，キーワードやパターンを定義していない未知のプログ

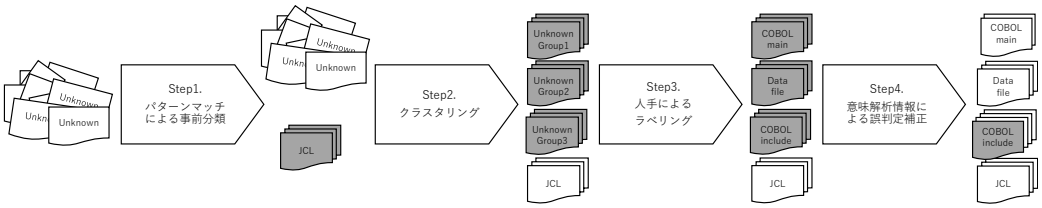

図 2.1　AutoSort の手順

ラミング言語に対しては，判定ができない，あるいは誤って既知のプログラミング言語として誤判定してしまうという問題もある．

パターンに頼らないプログラミング言語の判定手法として，教師あり機械学習を用いた手法が提案されている．Klein ら [11] は，ソースファイルの各行に出現する単語や末尾に出現した文字，記号などの特徴を用いた機械学習を提案している．van Dam ら [9] は，ソースコードを単語の列に分解し，n-gram などの機械学習モデルを適用した結果を報告している．これらの手法を適用するには，あらかじめプログラミング言語が判定されているソースコードファイルを多数用意しておく必要があり，拡張子のある一般的なプログラミング言語ではその準備が容易であるが，レガシーシステムでは教師データを準備することが困難である．また，プロジェクトごとに拡張された独自のプログラミング言語等への対応が困難である．

2.3　提案手法: AutoSort

本研究では，人手によるプログラミング言語判定を効果的に行う手法 AutoSort を提案する．提案手法はレガシーシステムのファイル集合を入力として，ファイル名や拡張子の情報を使うこと無く，ファイルごとにプログラミング言語とファイル種別のラベルを付与する．ファイル集合にはプログラムの実行時に利用されるデータファイルやパラメータファイルも含まれており，これらは厳密にはプログラミング言語ではないが，プログラミング言語の一種と見なしてラベルをつける．

AutoSort は図 2.1 に示すように 4 つのステップから構成される．

1. パターンマッチによる事前分類
2. クラスタリング
3. 人手によるラベリング
4. 意味解析情報による誤判定補正

提案手法は，まず，パターンマッチによる事前の分類を行う．このステップは，技術的には既存手法と同様である．次に，Klein ら [11] の手法と類似した字句的な特徴を用いたクラスタリングを適用し，人手で判定するべき代表的なファイルを選出する．そして，人手による判定を行ったのち，その結果を用いて呼出関係などの解析を

表 2.1　プログラミング言語判定パターン

プログラミング言語・ファイル種別	パターン
アセンブラ	1〜3 文字目がニーモニックキーワードである行数が全行数の 50% を超える
ADL	ダブルクォート内部以外で ”␣NAME␣IS”という文字列をファイル中に 1 つ以上含む
JCL メインファイル	”ˆ//.+␣+JOB” と ”ˆ//.+␣+EXEC” という文字列をファイル中にそれぞれ 1 つ以上含む
JCL インクルードファイル	”ˆ//.+␣+PROC” という文字列をファイル中に 1 つ以上含む, もしくは ”ˆ//.+␣+JOB” という文字列をファイル中に 1 つ以上含むが, ”ˆ//.+␣+EXEC” という文字列は 1 つも含まない
Telon	”TRANPORT␣MM/DD/YY␣HH:MM:SS” という文字列をファイル中に 1 つ以上含む

実行し，判定結果と解析結果の一貫性を確認することで誤判定の補正を行う．各手法を段階的に組み合わせることで各手法の弱点を補い，高い精度と少ない工数，未知のプログラミング言語が含まれていた時のロバストネスの実現を目指して設計した手順となっている．

2.3.1　ステップ 1: パターンマッチによる事前分類

ステップ 1 では，他のプログラミング言語に比べて特徴が明確で判定が容易なものを，パターンマッチによる判定を用いて分類する．特徴が明確で判定が容易なプログラミング言語とは，以下の様な特徴を持つプログラミング言語である．

- 当該プログラミング言語で記述されたソースコードであれば，必ず記述する予約語や記法が存在する．
- その予約語や記法は当該プログラム特有であり，他のプログラミング言語では (滅多に) 記述されない．

上記のような特徴を持つ言語同士の判別であれば，パターンマッチによる判定の問題点が起こらない．パターンマッチで判定する言語を限定的にすることで，世の中に無数に存在するプログラミング言語についての知識がなくとも適切なパターンを作成することができる．

第一著者が所属する企業が保守開発する 4 つのレガシーシステムを予備調査した結果，アセンブラ，ADL(Aim Description Language)， CA Easytrieve PLUS (以下 EASYPLUS と略す) [12]，JCL，Telon [13] が上記の特徴に当てはまると判断した．特に JCL については，構文解析の起点となる JCL メインファイルと，メインファイルから参照される JCL インクルードファイルという 2 つのファイル種別に分類する判定パターンを定義した．それぞれのプログラミング言語を判定するパターンを 2.1 に示す．表中の文字列は全て正規表現である．

2.1 の記述順でパターンマッチを適用していき，マッチしたファイルを当該のプログラミング言語として判定する．マッチしなかったファイルを，ステップ 2, 3 の判定処理の対象とする．

2.3.2 ステップ 2: クラスタリング

ステップ 2 のクラスタリングでは，プログラミング言語が持つ様々な特徴を用いて，ステップ 1 のパターンマッチに該当しなかったファイルを k 個のクラスタに分類する．クラスタリングは，未知の言語や独自拡張された言語のソースコードファイルが含まれていても，それらが他のプログラミング言語のファイルと書き方が類似していない限り，異なるクラスタとして認識することができる．なお，ここでの k は手法のパラメータであり，システムに使われているプログラミング言語の数よりも十分に大きい値を設定する．

ファイルの距離

本手法で想定しているファイル集合は，以下のようなものである．

- 単一のディレクトリに存在している
- ファイル名は連番で命名されている
- ファイル名に拡張子が付与されていない

このため本手法では．ソースコードの内容に起因する特徴量のみを選定することとし，名前や拡張子などを特徴量として使用しない．

レガシーシステムにおいては，アルファベットの大文字しか使えない，利用できる記号の種類が少ない，1 行あたりの桁数が固定であるなど，端末の環境の制限が強く，プログラミング言語にもそれらの特徴が反映されているため，これらの特徴に基づいた特徴量を選定することとした．

あるソースファイル f に対して使用する特徴量は以下の通りである．

- ファイルの行数（空行を含まない）
- 1 行あたりの平均文字数
- 空白を除く 1 ファイル内全ての文字に対する英字の割合
- 空白を除く 1 ファイル内全ての文字に対する数字の割合
- 1 ファイル内全ての英字に対する大文字の割合
- 括弧の 1 行あたりの平均出現数
- 特殊文字（ピリオドやカンマなど）の 1 行あたりの平均出現数
- 他のファイル名と同じ単語の出現数
- 様々な言語の予約語と考えられる単語（if や while など）230 種類それぞれについて，以下の 3 つの特徴量（合計 690 個）．
  - 1 文字目からその単語が出現する行の割合．
  - 各行の前半 1/3 でのその単語の平均出現数．
  - 各行の後半 1/3 でのその単語の平均出現数．

表 2.2　クラスタリングで用いる特徴量の選定理由

特徴量	選定理由
ファイルの行数	メインファイルは比較的長く，インクルードファイルは短いことが多い
1 行あたりの平均文字数	1 行あたりの文字数制限がある言語がある
英字の割合	コメントに英字しか記述できない言語と 2 byte 文字（日本語）を記述できる言語がある
数字の割合	データファイルは数字の割合が高い傾向がある
大文字の割合	COBOL などの言語は全て大文字で記述する傾向がある
括弧の平均出現数	ブロックを括弧で囲む言語と，括弧ではない予約語で囲む言語がある
特殊文字平均出現数	文の区切り文字が言語によって異なる
他のファイル名と同じ単語の出現数	他のファイルの呼び出しやインクルードなどを多数行う言語と，全く行わない言語がある
出現位置毎の予約語の平均出現数	予約語は必ず行頭に記述する言語と，どこに記述してもよい言語がある

これらの特徴量を選定した理由を 2.2 に示す．

ファイルからの単語の切り出しにおいては，空白，改行文字，引用符およびピリオドを区切り文字として，それ以外の連続する文字列を単語として切り出している．プログラミング言語ごとに引用符等の扱いのルールは異なるため，引用符で囲まれた範囲の内側，外側という区別は行わず，全ての区切り文字の出現を使用する．

特徴量は全部で 698 次元のベクトルとみなすことができるので，ファイル f_1, f_2 の距離 $d(f_1, f_2)$ は，それらに対応する特徴ベクトル $v(f_1), v(f_2)$ のコサイン類似度によって，以下のように定義する．

$$d(f_1, f_2) = 1 - cos(v(f_1), v(f_2))$$

K-means++ 法の適用

ファイル集合に対して，非階層型クラスタリング手法の 1 つである K-means++ 法 [14] を適用する．非階層型クラスタリングは，分割の良さを測る評価関数を定め，その評価関数を最適にするように分割する手法である．これに対して階層型クラスタリングは，要素数 1 の N 個のクラスタを初期状態として，最も距離が近い二つのクラスタを併合する作業を繰り返すことで階層的なクラスタ構造を得る手法である．非階層型クラスタリングは階層型クラスタリングに比べて計算量が小さく，レガシーシステムで分析すべきファイルは数十万を超えることがよくあることから，本手法では実用的な解析時間で終わることを重視し，非階層型クラスタリング手法を選定した．

K-means++ 法は，K-means 法 [15] の初期値設定を工夫したアルゴリズムである．K-means 法では任意の k 個のクラスタ中心を初期値として設定するが，K-means++ 法では以下の手順でクラスタ中心の選択を行う．

1. 1 つ目のクラスタ中心 c_1 となるファイルを，解析対象ファイル集合 F からランダムに選択する．
2. 各ファイル $f_j (j \in 1, 2, \cdots, |F|)$ に対して，最も近い選択済みのクラスタ中心との距離 $D(f_j) = \min_i d(c_i, f_j)$ を求める．

3. 確率分布 $\frac{D(f_j)^2}{\sum_{f \in F} D(f)^2}$ を用いてランダムに次のクラスタ中心 c_l となるファイルを選択する．つまり，どの選択済みクラスタ中心からも離れているようなファイルを高い確率でクラスタ中心として選択する．
4. クラスタ中心を k 個選ぶまで，2 から 3 を繰り返す．

クラスタ中心の選択以降は，K-means 法と同一で，以下の手順でクラスタリングを実行する．

1. 各ファイル $f_j (j \in \{1, 2, \cdots, |F|\})$ を，f_j と最も近いクラスタ中心 c_i が所属するクラスタ C_i に割り当てる．
2. クラスタに属するファイルの特徴ベクトルの平均 $c_i = \frac{1}{|C_i|} \sum_{f \in C_i} v(f)$ を，新しいクラスタ中心とする．
3. クラスタに変化がなくなるまで，あるいはパラメータとして指定する最大繰り返し数まで，1 と 2 を繰り返す．

K-means 法は計算コストが小さいという利点があるが，1 でランダムに選択するクラスタ中心の初期値によってクラスタリング結果が大きく異なってしまうという弱点がある．K-means 法よりも解析時間，クラスタリング精度の両面で K-means++ 法の方が有効であるため，本手法では K-means++ 法を採用した．

2.3.3 ステップ 3: 人手によるラベリング

ステップ 3 では，ステップ 2 で k 個に分類された各クラスタについて，1 つずつ代表となるファイルを人間に提示し，人間がどのプログラミング言語なのかという判定を行うことで，ラベリングを行う．

代表ファイルとして，本手法では各クラスタの中心に最も近いファイルを選択する．K-means 法は「各要素は群の平均から離れていない」ことを保証する方法であるため，クラスタに含まれるプログラムの「平均」に近いファイルを選ぶことで，出現する単語の数や種類の特徴を見落としにくいと期待している．K-means++ 法では，各クラスタについてクラスタ中心が得られているため，そのクラスタ中心からクラスタに所属する各ファイルまでの距離を容易に計算することができる．

こうして選ばれた代表ファイルを人間に提示し，提示された人間はそのファイルに対してプログラミング言語を判定する．判定された代表的なファイルのプログラミング言語を，そのクラスタに属する全てのファイルのプログラミング言語としてラベリングする．つまり，このステップが完了した時点で，全てのファイルに対してプログラミング言語のラベルが付与された状態となる．

ラベルとしてはプログラミング言語の名前を指定するだけであるが，プログラミング言語によってはファイル種別としてメインファイル，インクルードファイルの区別

表 2.3　ファイル間の関係に基づく制約

ファイル間の関係	ファイルの種別に関する制約
呼び出し関係 (A calls B)	B は何らかのプログラミング言語の メインファイルである.
インクルード関係 (A includes B)	B はインクルードファイルである. B の記述言語は A と同一である.
ファイル操作関係 (A reads/writes B)	B はデータファイルである.

も同時に指定する．また，データファイルに対しては，ファイル形式の名称とは別に，ファイル種別としてデータファイルであることを指定する．これらのファイル種別の指定は，次のステップ 4 で使用する情報となる．

2.3.4　ステップ 4: 意味解析情報による誤判定補正

ステップ 4 では，ステップ 3 までで判定したプログラミング言語が，既知の言語かつ，構文解析器や意味解析器を持っているプログラミング言語だった場合，そのプログラミング言語と判定されたファイルに対して構文解析および意味解析を実行し，その結果を用いて誤判定の補正を行う．このステップは必須ではなく，ステップ 3 で判定されたプログラミング言語に対応する構文解析器や意味解析器を持っている場合のみ実施し，対応する構文解析器や意味解析器を持っていない場合，単にこのステップを省略する．

このステップで補正が必要になる主な要因は，ステップ 2 におけるクラスタリングにある．同じ言語のメインファイルとインクルードファイルや，サイズが小さいファイル同士はそれぞれ特徴量の違いが少なく，異なる種類のファイルが偶然同一のクラスタに分類されることがある．中身に何が含まれていても良いデータファイルについても，偶然ソースコードファイルに似ている場合がありうる．

このようなファイル種別の補正に，ソースコードの意味解析から得られるファイル間の関係を利用する．対象のプログラミング言語の構文解析器や意味解析器を持っている必要があるが，意味解析を行うことでファイル内部に出現する他のファイル名の情報を取得し，ファイル間の関係に伴う制約に基づいて，ファイル種別の正確な判定を行う．

ファイル間の関係情報として，呼び出し関係 (A calls B)，インクルード関係 (A includes B)，ファイル操作関係 (A reads/writes B) の 3 つを用いる．呼び出し関係とは，プログラミング言語の種類に関係なく別のソースコードファイルを呼び出す命令による関係である．COBOL の CALL 文や JCL の EXEC PGM 文などが該当する。インクルード関係とは，ヘッダファイル，コピーファイルのようなインクルードファイルを埋め込む命令による関係である．COBOL の COPY 文や JCL の

INCLUDE 文などが該当する．ファイル操作関係とは，ファイルの入出力を行う命令である．COBOL の READ 文や WRITE 文が該当する．

これらの関係はファイルの構文解析および意味解析によってファイル名の出現位置を調べることによって得られるもので，2.3 に示すように，呼び出し，インクルード，ファイル操作の対象となるファイルは，それぞれメインファイル，A と同じプログラミング言語のインクルードファイル，データファイルに限られるという制約が存在する．ステップ 3 までの判定結果がこれらの制約を満たしていない場合，判定結果が誤っていると考えられるため，これらを用いて，以下のルールに従って補正を実行する．

- ファイル A から B への呼び出しが存在する場合，B は必ず何らかの言語のメインファイルである（A と B は異なる言語であることもありうる）．
  - B が何らかの言語のインクルードファイルと判定されているのなら，メインファイルとインクルードファイルの種別判定を誤ったものと見なし，B の言語のメインファイルと判定する．
  - B がデータファイルの場合は，何らかの言語のメインファイルであるとして，人間に再度提示し，プログラミング言語の判定のやり直しを行わせる．
- ファイル A が B をインクルードする場合，B を A と同じ言語のインクルードファイルであると判定する．
- ファイル A が B に対して何らかのファイル操作命令を実行する場合，B をデータファイルであると判定する．

これらの補正ルールは，あるファイル B が他のファイルから受ける操作に注目しており，通常はファイル B の種別に応じて受け付ける操作が異なることから，1 つのファイルに対しては 1 つのルールしか適用されない．1 つのファイルに対して複数のルールが適用される場合は，参照元もしくは参照先のファイルの判別や意味解析が失敗していると考えられるため，参照元と参照先両方のファイルを人間に提示して，プログラミング言語の判定のやり直しを行わせる．

このステップが完了すると，提案手法の出力として，プログラミング言語・ファイル種別によってラベル付けされたファイルの一覧が得られる．

2.4 評価実験

実際のレガシーシステムのファイル集合 2 つに対して，提案手法である AutoSort と既存手法との比較実験を行った．評価の観点は，提案手法の正確性と，適用に必要な工数および解析時間である．そのために，全てのファイルに対して従来のやり方であるファイル内のキーワード検索等に頼った人手でのプログラミング言語の判定を行い，それを正解とした．実行時間の評価の基準となる人手の判別工数は，このときの作業

時間を計測したものである．

正確性の評価指標としては，クラス分類器の評価で広く採用されている適合率，再現率を用いる．あるプログラミング言語 l について，

- l が正解であるファイルを l と判別した数を $TP(l)$
- l が正解以外のファイルを l 以外と判別した数を $TN(l)$
- l が正解であるファイルを l 以外と判別した数を $FN(l)$
- l が正解以外のファイルを l と判別した数を $FP(l)$

とすると，適合率は $\frac{TP(l)}{TP(l)+FP(l)}$，再現率は $\frac{TP(l)}{TP(l)+FN(l)}$ で表される．また，あるシステム x 全体のプログラミング言語種別の集合を $L(x)$ とし，システム全体のファイル総数を $File(x)$ とするとき，$\frac{\sum_{l \in L(x)} TP(l)}{File(x)}$ をシステム x の全体正解率と呼ぶこととする．

実行時間の評価としては，クラスタリングにおける人手でのラベリングの工数と解析時間を指標として用いる．

2.4.1 実験方法

実験に用いたファイル集合は，第一著者が所属する企業で実際に保守開発を行っているやや小規模なレガシーシステムと，大規模なレガシーシステムの 2 つである．2 つ のファイル集合の概要を 2.4 に示す.

判別するクラスとは，言語とファイル種別を足しあわせて重複を省いたものである．言語にメインファイルとインクルードファイルの種別があるものについては，表中では言語名の後ろにそれぞれ MAIN，INC と表記した．たとえばシステム A でいうと，アセンブラ， EASYPLUS，COBOL メインファイル (COBOL MAIN)，COBOL インクルードファイル (COBOL INC)，CICS MAP，Format，FORTRAN，SQL，DATA の 9 個である．なお，システム A の Format はこのプロジェクト独自の DSL であり，DATA はプログラムの実行時に利用されるデータファイルやパラメータファイルのことである．

提案手法の効果を評価するために，各ファイル集合に対して，以下の 3 つの手法を適用した．

AutoSort　提案手法．クラスタリングにおけるパラメータは，クラスタ数 k を 40，繰り返し数を 10 とした．この k の値は，過去の経験から，多くのシステムで用いられるプログラミング言語を上回ることが想定される値である．

Simon-Weber　機械学習によって言語判定を行う Klein らの手法 [11] に対応する実装 [16] である．この手法はあらかじめ言語の特徴を教師あり機械学習する必要があるため，$K = 10$ で K-分割交差検証を行い，10 回の検証結果の平均を評価

表 2.4　実験に用いたファイル集合

システム名	システム A	システム B
総ファイル数	18,399	174,735
言語の一覧	アセンブラ EASYPLUS COBOL CICS MAP Format FORTRAN SQL DATA	アセンブラ EASYPLUS COBOL JCL PL/I ADL DATA Telon
言語の総数	8	8
判別するクラスの一覧	アセンブラ EASYPLUS COBOL MAIN COBOL INC CICS MAP Format FORTRAN SQL DATA	アセンブラ EASYPLUS COBOL MAIN COBOL INC JCL MAIN JCL INC PL/I MAIN PL/I INC ADL DATA Telon
判別するクラス数	9	11
人手による判別工数 (人・時間)	56	485

した．このとき，ファイル集合は判別するクラスごとに 10 個ずつに分割しており，ファイルの割合は維持したものとなっている．

Clustering　提案手法のクラスタリングのみ，すなわちステップ 2 と 3 のみを単純に適用したもの．特徴量，距離，クラスタリングのパラメータは提案手法と同一である．

AutoSort のステップ 4 で利用する構文解析器･意味解析器としては, JCL, COBOL, PL/I のみを持っているとして，これらの構文解析器・意味解析器を用いる．

AutoSort と Simon-Weber の比較によって既存手法との違いを調査し，また，AutoSort と Clustering の比較によって本研究で導入しているパターンマッチや誤判定の補正の効果を調査する．

AutoSort および Clustering の処理は Python を用いて実装した．実行時間の計測環境は，CPU として Intel Xeon E7-2860 2.27 GHz を搭載し，256GB RAM，15000 rpm HDD を記憶領域として備えた計算機である．

表 2.5　正確性の比較実験結果 - システム A

		AutoSort		Simon-Weber		Clustering	
判別するクラス	Correct	適合率 (%)	再現率 (%)	適合率 (%)	再現率 (%)	適合率 (%)	再現率 (%)
アセンブラ	204	96.52	95.10	69.66	79.90	100.00	45.59
EASYPLUS	337	100.00	98.81	30.66	87.24	14.46	10.68
COBOL MAIN	8,858	99.92	100.00	92.31	27.51	99.95	97.53
COBOL INC	5,891	100.00	100.00	44.26	74.20	98.15	92.65
CICS MAP	2,769	100.00	99.53	85.66	97.91	99.70	96.86
Format	198	85.53	98.48	63.52	74.75	97.93	95.45
FORTRAN	18	100.00	100.00	1.86	72.22	77.78	77.78
SQL	23	100.00	100.00	37.93	95.65	100.00	95.65
DATA	101	92.86	77.23	15.12	79.21	10.75	99.01
全体正解率			99.71		55.65		93.66

表 2.6　正確性の比較実験結果 - システム B

		AutoSort		Simon-Weber		Clustering	
判別するクラス	Correct	適合率 (%)	再現率 (%)	適合率 (%)	再現率 (%)	適合率 (%)	再現率 (%)
ADL	3,704	100.00	100.00	0.00	0.00	98.43	96.44
アセンブラ	277	98.58	100.00	4.04	16.25	0.00	0.00
EASYPLUS	292	99.65	98.29	2.57	34.59	0.00	0.00
COBOL MAIN	20,960	100.00	99.99	60.54	9.95	100.00	100.00
COBOL INC	19,435	99.83	100.00	37.90	57.93	97.03	97.39
JCL MAIN	3,187	100.00	99.34	8.51	50.89	41.17	54.47
JCL INC	21,062	99.75	100.00	61.55	67.51	93.00	85.81
PL/I MAIN	22,312	99.38	98.24	58.34	63.83	96.69	93.96
PL/I INC	52,307	99.76	99.76	74.01	53.35	97.18	93.74
Telon	463	100.00	100.00	11.77	67.17	0.00	0.00
DATA	30,736	99.87	98.78	69.86	67.36	86.59	98.18
全体正解率			99.47		52.93		93.54

2.4.2　実験結果

正確性の評価

正確性の比較実験結果を 2.5 と 2.6 に示す．提案手法はほとんどのプログラミング言語・ファイル種別で適合率，再現率が 95% 以上となった．また，全体正解率は両システムで 99% 以上，合計で 99.49% となっており，従来手法や単純なクラスタリングの適用に比べて適合率，再現率はいずれも大きく向上していることがわかる．

提案手法の各ステップ完了時点での適合率の値を 2.7 に，再現率の値を 2.8 に示す．これらの表は紙面の都合上システム B についての結果のみを示すが，ステップ 3 までである程度高い適合率，再現率を達成していることが分かる．仮にステップ 4 で用いた JCL，COBOL，PL/I の構文解析器を持っていなかった場合は，このステップ 3 までの適合率，再現率となる．

提案手法のステップ 4 によって，クラスタリングの際に誤判定されたインクルードファイルが正しい分類に再分類されている．ステップ 4 は工数をかけて構文解析器を

表 2.7　各ステップでの適合率 - システム B

ステップ	1	2 , 3	4
ADL	100.00	100.00	100.00
アセンブラ	98.58	98.58	98.58
EASYPLUS	99.65	99.65	99.65
COBOL MAIN	0.00	100.00	100.00
COBOL INC	0.00	99.82	**99.83**
JCL MAIN	100.00	100.00	100.00
JCL INC	99.75	99.75	99.75
PL/I MAIN	0.00	85.45	**99.38**
PL/I INC	0.00	98.66	**98.76**
Telon	100.00	100.00	100.00
DATA	0.00	95.93	**99.87**

表 2.8　各ステップでの再現率 - システム B

ステップ	1	2, 3	4
ADL	100.00	100.00	100.00
アセンブラ	100.00	100.00	100.00
EASYPLUS	98.29	98.29	98.29
COBOL MAIN	0.00	99.99	99.99
COBOL INC	0.00	95.12	**100.00**
JCL MAIN	99.34	99.34	99.34
JCL INC	100.00	100.00	100.00
PL/I MAIN	0.00	98.24	98.24
PL/I INC	0.00	92.32	**99.76**
Telon	100.00	100.00	100.00
DATA	0.00	98.78	98.78

多種類用意できれば，より判定精度が上がることが予想される．ステップ 3 までの結果を基に，ファイル数が多そうなプログラミング言語から優先的に構文解析器を用意すると効率良く判定精度を向上させることができると考えられる．

適合率，再現率の高さは，第一著者が所属する企業では十分に実用的であると判断された．適合率，再現率が 100% にならないことを前提とした本手法の利用プロセスの確立が議論され，レガシーシステムの分析サービスを受ける顧客への事前説明を行ったうえで現場が利用するツールの 1 つとして採用された．

ただし，基幹システムの移行作業においては，システムの機能を見落とすことが多大な影響を及ぼす可能性があり，再現率について 100% を求められる場合も存在する．このような場合には人間が全てのファイルを目視する作業を無くすことができない．しかしその場合であっても，提案手法によってソースファイルにラベル付けされていれば，作業者はほとんどのソースファイルのラベルを確認する作業だけで済むため，一定の作業時間短縮の効果はあるものと思われる．判定結果として複数のプログラミング言語を候補として出力するなど，再現率を 100% とするような拡張を行うことが今後の課題である．

表 2.9　工数および解析時間の比較結果 - システム A

手法	AutoSort	S.-Weber	Clustering	人手
人手時間	7m41s	0m00s	7m37s	6 人日
解析時間	46m49s	11m42s	21m03s	-

表 2.10　工数および解析時間の比較結果 - システム B

手法	AutoSort	S.-Weber	Clustering	人手
人手時間	7m00s	0m00 s	7m12s	3 人月
解析時間	7h10m40s	2h32m14s	3h16m48s	-

適用に必要な工数および解析時間の評価

提案手法の実行および人手での分析に要する工数の比較実験結果を 2.9 と 2.10 に示す．人手による分析作業の場合，1 ファイルあたりの作業時間は平均で 10 秒程度であるが，ファイル数に比例した工数が必要であった．提案手法ではファイル数とは関係なく，クラスタ数に比例する工数で実施できる．提案手法における分析でも 1 ファイルあたり 10 秒程度という工数は変わらず，$k = 40$ とした本実験ではシステム A，システム B ともに約 7 分であった．全自動で動作する機械学習手法に比べると作業時間は必要であるが，十分に実用的として現場でも受け入れられた．

なお今回の実験では，手作業での実験開始時に分析者にとって未知の言語であったもの (Format, FORTRAN）が含まれていたため，実作業時にはこの言語が何であるかを有識者に数時間インタビューしているが，本実験の工数にこの時間は含めていない．この工数はどの手法を採用するにしても必要な工数であり，ファイル数とは関係なく一定の工数が必要であると考える．

本手法の適用範囲

本手法で選定した特徴量は，レガシーシステムの端末環境に由来するプログラミング言語の特徴を反映したものであるため，レガシーシステムのプログラミング言語であれば精度良く分類ができると考えている．

一方で，レガシーシステム以外でよく用いられるプログラミング言語（C 言語や Java など）に関しては，本手法で提案した特徴量以外に，新しい特徴量を導入する必要があるかもしれない．しかし，クラスタリングによって代表ファイルを選出するというアイディア自体はレガシーシステムに固有というわけではないため，文書中に埋め込まれたソースコード断片等のプログラミング言語判定に応用することも考えられる．

プログラミング言語によっては，複数のバージョン，派生言語，方言などが存在する．そのようなプログラミング言語が複数使用されているようなシステムを対象として，それらの違いを判定しなければいけない場合，ファイル間での特徴量の差異が小

さくなり，分類精度が低下すると考えられる．

2.5 本章のまとめ

本研究では，レガシーシステム内に雑多に混在する複数のプログラミング言語で記述されたファイル集合から，それらのプログラミング言語およびファイル種別を判定する手法 AutoSort を提案した．提案手法は，クラスタリングによって特徴が類似するファイルをまとめ，そこから人間が判定すべき代表ファイルを選出することで，効果的な判定を可能とした．また，作業者の労力を減らすために，パターンマッチによる一部の言語の分類と，事後的な解析による誤判定の補正を組み込んだ．

提案手法と既存手法との比較実験では，クラスタリングが機械学習手法よりも正確な結果となること，また，パターンマッチおよび誤判定の補正が提案手法の正確さに貢献していることを確認した．適用する場合の作業工数も十分に現実的なものであり，現場で使用するツールとして受け入れられた．

今後の課題としては，再現率について 100% を求められるような状況に対応するための再現率を重視した提案手法の拡張が挙げられる．また，クラスタリングによって代表ファイルを選出するというアイディア自体はレガシーシステムに固有というわけではないため，文書中に埋め込まれたソースコード断片等のプログラミング言語判定に応用することも考えられる．

第 3 章

変更要件に関係するプログラム特定のための処理名抽出手法

有識者や既存の設計書がない状況で，システムの保守開発を行うためには，業務用語とシステムの用語の対応関係を把握することが重要である．本稿では，ソースコードを処理機能という粒度で分割し，入出力されるファイル名と処理機能から機能名を自動的に付与する手法を提案するとともに，この手法を実現するツールを試作した．また，COBOL 言語で記述されたプログラムに適用し，定性的な評価を行ったところ，本手法によって業務用語とシステムの用語の対応関係が把握しやすくなったことを示した．

3.1　はじめに

マイグレーションの 2 つ目のステップ「レガシーシステムの構造の分解」は，レガシーシステムに存在するプロシージャコールなどのモジュール間の依存関係を排除し，明確なインタフェースとして整理する検討作業である．

「レガシーシステムの構造の分解」で行うレガシーシステムのインタフェース整理では，業務の区切り目をインタフェースにすることが多い．しかし有識者や既存の設計書ない場合には，業務とシステムの対応関係が分からず，業務の区切り目がシステムではどこに対応していて，インタフェースとしてどこを抽出すべきか分からないことが多い．このため，有識者や既存の設計書がない状況下で業務用語とシステムの用語の対応関係を把握することが重要である．

業務用語とシステムの用語の対応関係の把握は，処理方式によって難易度が変わる．企業の業務システムは，その処理方式から対話処理とバッチ処理に大別され，バッチ処理のほうが「変更依頼が分析され，システムの用語に翻訳される」は難しい．

対話処理とは，画面を介してユーザとシステムがやり取りする処理方式である．対話処理では，変更依頼に記載されている業務用語（申込書チェック，引当処理など）

からソースコードに記載されているシステム用語（ファイル名，機能名，項目名など）の翻訳は比較的容易である．その理由としては，対話処理では業務用語を画面内の項目名などに使っている場合が多く，業務用語と画面の対応関係をとるのは容易であり，画面から利用されているソースコード数やファイル数は少ないためである．

一方で，バッチ処理とは，一定量のデータなどを一括して処理する方式 [17] であり，COBOL 言語などで実装されたジョブと呼ばれる処理を，JCL などの言語で順次実行する処理方式のことである．バッチ処理は，あらかじめ定められた複数の処理が一度に行われるため，調査すべきソースコード数やファイル数は多く，対応関係の調査には非常に時間がかかる．特にソースコードはファイルよりも数が多く，調査に手間がかかる．

このため，バッチ処理について業務用語と機能名の対応関係を理解する作業を支援する技術が必要である．

そこで本研究では，バッチ処理を対象として，ソースコードを処理機能という粒度に分割し，入出力されるファイル名と処理機能から機能名を自動的に付与する手法を提案する．

本研究の構成は以下のとおりである．まず，3.2 節では業務用語とシステム用語の対応関係の理解が必要な状況について例を交えながら詳細を述べる．3.3 節では，本研究で提案する処理機能について述べ，3.4 節でソースコードから処理機能を抽出し，機能名を自動的に付与する手法について述べる．3.5 節で本手法を実装したツールにおける適用実験の結果を示す．3.6 節で関連研究を述べ，最後に 3.7 節でまとめと今後の課題について述べる．

3.2　業務用語とシステム用語の対応関係

本節では，簡単な想定在庫数計算機能を考え，これを具体例として業務用語とシステム用語の対応関係を分析する既存の手法と，既存手法での問題点を述べる．また，既存手法での問題点を解決する方向性についても述べる．

3.2.1　想定する保守開発担当者の知識レベル

本研究で想定している状況は，保守開発担当者が既存システムについての十分な知識を持っていない状況である．具体的には，業務要件に記載されている業務用語とシステム用語の対応関係についての知識は持っておらず，各ソースコードがどのような内容か把握していない知識レベルであることを想定している．他社が開発したシステムの保守開発を受注した場合などは，このような知識レベルの場合が多い．

3.2.2 既存システムの想定在庫計算機能

本節で考える想定在庫数計算機能は，1) 生産計画ファイルに記載された「本日生産されて入荷できる予定の在庫数」と，2) 想定在庫ファイルに記載された「倉庫に昨日まであった在庫数」の 2 つの合計を本日の想定在庫数とし，想定在庫ファイルとして出力する機能である．この機能の詳細は，以下の通りである．

- 生産計画ファイルは,「商品コード」,「生産予定日」,「生産予定数」,「生産工場」という 4 つの項目を持つ．
- 想定在庫ファイルは,「商品コード」,「想定在庫数」という 2 つの項目を持つ．
- 生産工場には，α 工場と β 工場がある．α 工場は倉庫に近いが，β 工場は倉庫からやや離れた場所に位置する．
- 現行機能では，β 工場で生産される商品は本日中に倉庫に入荷されないため，α 工場で本日生産される予定の商品だけを，想定在庫数としている．

この想定在庫計算機能のデータフロー図を図 3.1 に示す．この図は，四角のオブジェクトがプログラムを，丸みを帯びたオブジェクトがファイルを，その間を繋ぐ矢印がプログラムとファイルがどのような関係にあるかをそれぞれ表している．例えば，左上の FILE-A，PGM-1，FILE-B は,『プログラム「PGM-1 生産計画ファイル抽出」はファイル「FILE-A 生産計画ファイル」を入力とし，ファイル「FILE-B 生産計画ファイル」を出力する』ことを意味している．

また，同じ名前が付いているファイル名同士は，同じレコードレイアウトのファイルであることを意味する．さらに，各プログラム名・ファイル名は，既存の設計書の情報を元に作られているとする．

このようなデータフロー図は，ソースコードと機能一覧などの設計書があれば既存のツールで作成することができるが，既存のツールで作成されたデータフロー図は，プログラム数が数百，数千個になることも珍しくない．本研究ではスペースの都合上，関連部分のみを抜粋した．

3.2.3 変更依頼の内容

このような既存システムに対して，以下のような変更依頼に基づいて保守開発を行うことを考える．

- β 工場から倉庫までの配送が改善され，β 工場で生産される商品も当日中に倉庫に入荷されるようになった．このため，想定在庫数も β 工場の当日の生産数を考慮して算出するように変更する．

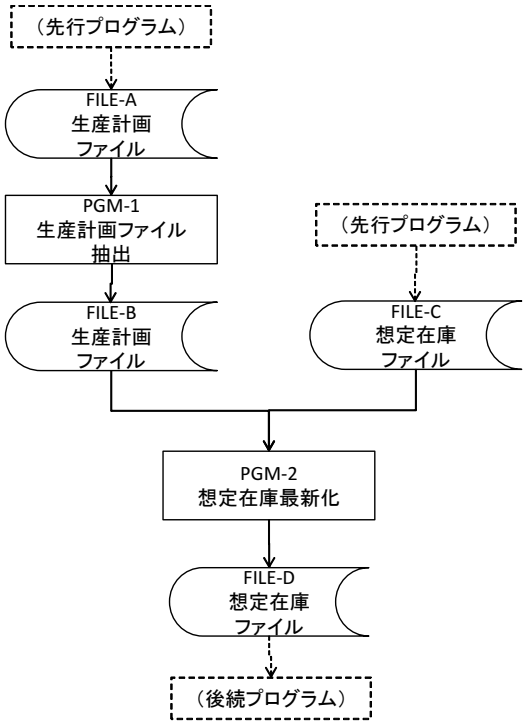

図 3.1　既存手法によるデータフロー図の例

3.2.4　既存手法による業務用語とシステム用語の対応関係の付け方

業務用語とシステム用語を対応付ける方法としては，変更依頼からキーワードを抽出し，それに該当するファイル名や機能名をデータフロー図から検索する方法がある．

この方法では最初に，変更依頼からキーワードを抽出する．今回の例では，キーワードとして「β工場」,「想定在庫数」,「生産数」といったファイル名に関連するキーワードや,「想定在庫数算出」といった機能名に関連するキーワードを抽出することが考えられる．

次に，これらのキーワードをもとに，データフロー図を検索し，変更依頼に関係しそうなファイル名や機能名を抽出する．具体例では．ファイル名に関連しそうなキーワードとして「β工場」,「想定在庫数」,「生産数」が抽出できる．また，これらのキーワードを項目として持つファイルを検索し,「生産計画ファイル」と「想定在庫ファイル」が抽出できる．一方で，機能名に関連するキーワードとして「想定在庫数算出」を挙げたが，これらに合致する機能名がデータフロー図内には存在せず，変更依頼から関連する機能名を把握することができない．

このため，機能名では絞り込みが出来ず,「生産計画ファイル」と「想定在庫ファイル」が関連するプログラムを全て抽出し，それらのソースコードの中身を全て読み，今

回の変更依頼と関係があるかどうかを調査しなければいけない.

3.2.5 既存手法の問題点

前節で述べたような手法には，既存システムについての知識がない保守開発担当者は変更依頼から既存システムの機能名に合致するキーワードを選定することができないという問題がある.

既存の機能名に合致するキーワードを選定できない原因は，データフロー図に記載されている機能名として，過去のシステム開発担当者や保守開発担当者によって，まちまちな粒度で，まちまちな名前を付けられた機能名を利用しているためである.

3.2.6 問題点解決の方向性

前述の問題は，以下の 2 つの条件が満たされていれば解決すると考えられる.

- 決まった粒度，決まった命名規約で機能名が付与されたデータフロー図がある
- その命名規約を保守開発担当者が把握している

以上の 2 つの条件があれば，既存システムの知識がない保守開発担当者であっても，命名規約を参考にして，変更依頼からデータフロー図の機能名に合致するキーワードを作成することができるようになる．適切なキーワードを作成することが出来れば，該当する機能やプログラムを検索することもでき，結果として調査しなければならないソースコードを減らすことができると考えられる.

以降の節では，統一された粒度と名前を持つ機能として処理機能と呼ばれる単位を提案する．また，この処理機能を用いてデータフロー図を作成する手法についても述べる.

3.3 処理機能とは

本研究では，ソースコードから処理機能を抽出し，機能名を自動付与する手法について述べるが，本節ではまず，本手法で抽出される処理機能について定義する.

本手法で提案する処理機能とは，複数のレコードからなるファイルを一括処理する決まった粒度の処理のことであり，表 3.1 で示した 7 種類を定義した.

この処理機能は，データベースなどで用いられる集合を操作する関係代数の演算子を参考に作成・定義した．関係代数の演算子としては，「和」，「差」，「交わり」，「直積」，「制限」，「射影」，「結合」，「商」，「属性名変更」，「拡張」，「要約」などが知られている [18].

関係代数と処理機能の対応関係を表 3.2 に示す．処理機能は，関係代数の名前を保守開発担当者にも馴染み深い名前に変更したり，いくつかの関係代数の演算子をまと

表 3.1　処理機能の定義

処理機能	定義
抽出	入力ファイルの一部のレコードだけを出力ファイルとして出力する
振分	入力ファイルのレコードを条件に従って，複数の出力ファイルに振り分けて出力する
マッチング	2 つの入力ファイルを特定の項目同士で比較し， (1) 項目の値が等しいレコードが両方の入力ファイルに存在する， (2) 項目の値が等しいレコードが片方にしか存在しない， の 2 種類に振り分けて出力する
和	複数の同じファイルフォーマットの入力ファイルのレコードを，順に出力ファイルとして出力する
作成	1 個以上の入力ファイルの値を利用して、入力とは異なるファイルフォーマットの出力ファイルを出力する
編集	入力ファイルの一部のレコードの値を変更して，出力ファイルとして出力する
集計	入力ファイルを集計し，集計キーと集計値（合計値、平均値、最大値、最小値）からなる出力ファイルを出力する

表 3.2　関係台数と処理機能の対応

関係代数	処理機能
和	和
差	マッチング
交わり	
商	（「マッチング」と「作成」の組み合わせで表現）
制限	抽出/振分
射影	作成/編集
直積	
結合	
属性名変更	
拡張	
要約	集計

めて一つの処理機能としたり，逆に一つの関係代数の演算子を 2 つの処理機能として分割するなどして，定義した．

関係代数を参考にした理由は，本手法で対象としているバッチ処理が複数のレコードをまとめて処理する方式であり [17]，この複数のレコードをひとまとまりの集合として考えた場合，バッチ処理は関係代数の演算子に相当すると考えられるためである．

一方，関係代数の演算子その物を利用しなかった理由は，関係代数の演算子名と，一般的な業務システムで用いられる機能名が大きく乖離しており，一般的な保守開発担当者がキーワードとする機能名に相応しく無いと考えたためである．例えば，「ある指定した条件に合致するレコードのみを出力する」という操作は，関係代数の演算子では「制限」であるが，これを処理機能では「抽出」と表現している．

処理機能を用いたデータフロー図の具体例として，3.2 節で説明した在庫集計機能に対して処理機能を抽出したデータフロー図を図 3.2 に示す．

処理機能を抽出した結果，既存手法では「想定在庫最新化」だけだった PGM2 が，「生産計画ファイル抽出」，「生産計画ファイル・想定在庫ファイルマッチング」，「想定在庫ファイル編集」，「想定在庫ファイル和」の 4 つとして表現されている．

このため 3.4 節の例では，まずキーワード選定の際に，変更依頼に記載された「想定在庫数も…考慮して算出」から，処理機能を意識して，「算出」などではなく，「編集」

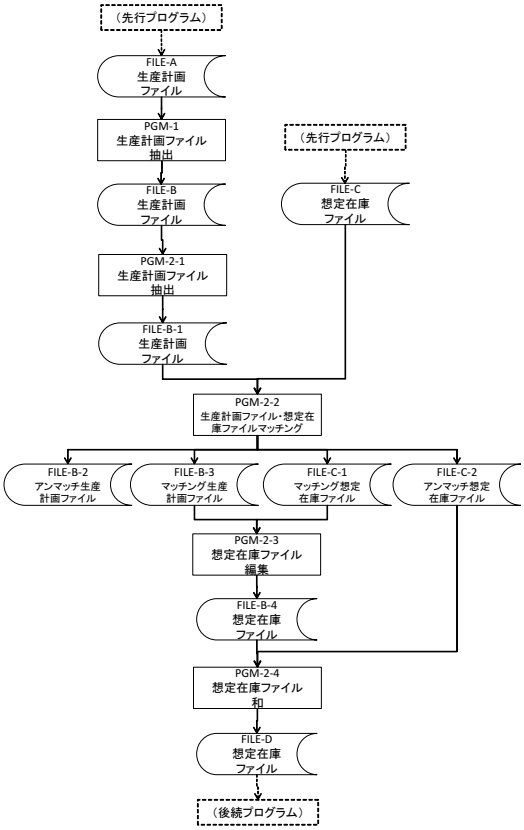

図 3.2　処理機能を抽出したデータフロー図の例

という動詞を考える．また，ファイル名は既存手法でも分かっていた通り，「想定在庫ファイル」を考え，先ほどの処理機能「編集」と組み合わせて，機能名「想定在庫ファイル編集」というキーワードを選定する．このようにして作成した「想定在庫ファイル編集」という機能名であれば，提案手法で作成するデータフロー図から検索することができ，今回の変更依頼に対応するプログラムは「PGM-2」であることが分かる．

3.4　処理機能の抽出手法

本節では，前節で述べた処理機能をソースコードから抽出し，抽出された処理機能を用いたデータフロー図を作成する手法について述べる．

3.4.1　処理機能の抽出手法の概略

本手法の概略は，図 3.3 に示した通りであり，大きく分けて 4 つの手順からなる．まず，ソースコードを構文解析し，抽象構文木を得る．次に，この抽象構文木を単純な要素だけからなる抽象構文木に変換する．続いて，この変換された抽象構文木に対

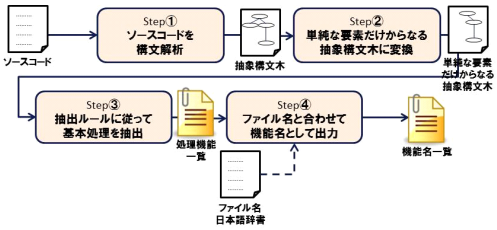

図 3.3　提案手法の概略

表 3.3　Step2 の抽象構文木に用いられる要素

単純な要素	抽象構文木での表現
順構造	Block 他の要素
条件分岐	If 条件式 真文となる Block 偽文となる Block
反復	Loop 反復条件式 反復される Block
外部入力	Input 外部入力が格納される変数 外部入力元
外部出力	Output 外部出力する変数 外部出力先
代入	= 代入先の変数 代入元の式

して，予め定義しておいたマッチングルールを適用し，処理機能を抽出する．最後に，抽出された処理機能名と，処理機能の入出力ファイル名から機能名を作成する．以降ではこの 4 つの手順の詳細を述べる．

3.4.2　Step1: ソースコードの構文解析

Step1 では，ソースコードを構文解析し，抽象構文木を作成する．Step1 で作成される抽象構文木は，ソースコードが記述されているプログラミング言語独自の要素からなる．

例えば，COBOL 言語の「MOVE 1 TO VAR」という文は，(MOVE, VAR, 1) といった要素で抽象構文木を作成されるし，C 言語の「var = 1」という文は (=, var, 1) といった要素で抽象構文木を作成される．

3.4.3　Step2: 単純な要素だけからなる抽象構文木に変換

Step2 では，抽象構文木を単純な要素だけからなる抽象構文木に変換する．単純な要素としては，「順構造」，「条件分岐」，「反復」，「外部入力」，「外部出力」，「代入」の 6 種類を考える．この変換のために，Step1 で作成したプログラミング言語独自の要素が，表 3.3 に記載された単純な要素のどれに該当するかというマッピングルールを作成しておき，変換の際には，このマッピングルールを用いて，単純な要素だけからなる抽象構文木に変換する．

例えば，COBOL 言語の抽象構文木 (MOVE, 1, VAR) は，(=, VAR, 1) となる．

表 3.4　マッチングルールの一例

抽出する処理機能	マッチングルール
抽出	「変数と定数を比較する条件式を持つ条件分岐の外側の順構造に，外部入力が存在する」 かつ 「条件分岐の真文 Block もしくは偽文 Block に，外部出力が存在する」

3.4.4　Step3: 抽象構文木に対するマッチング

Step3 では，変換された抽象構文木に対して，予め定義しておいたマッチングルールを適用し，処理機能を抽出する．

マッチングルールの一例を表 3.4 に示す．

3.4.5　Step4: 機能名の作成

Step4 では，抽出された処理機能名と，処理機能の入出力ファイル名から機能名を作成する．「作成」以外の 6 つの処理機能（「抽出」,「振分」,「マッチング」,「和」,「編集」,「集計」）は，「入力ファイル名　処理機能名」という命名規約で命名する．また，「作成」は「出力ファイル名　処理機能名」という命名規約で命名する．

例えば，処理機能が「抽出」で，入力ファイル名が「想定在庫ファイル」の場合，機能名として「想定在庫ファイル抽出」と命名する．また，処理機能が「作成」で，出力ファイル名が「想定在庫ファイル」の場合，　機能名として「想定在庫ファイル作成」と命名する．

3.5　適用実験

本節では，3.4 節で述べた処理機能の抽出手法を実装したツールについて述べ，COBOL 言語のソースコードから，業務用とシステム用語の対応関係を理解するために有用な処理機能を抽出することができることを示す．

3.5.1　処理機能抽出ツール

試作した処理機能抽出ツールの概要を図 3.4 に示す．

Step1 を実現するために Strafunski [19] を利用した．Strafunski は Haskell で実装された言語処理モジュールであり，SDF [20] で記述した言語仕様を利用して，対象言語のソースコードを抽象構文木（実態は Haskell のツリー構造）に変換する．Strafunski を利用した理由としては，将来的に解析対象の言語を COBOL 言語だけではなく，C 言語や Java などの多言語に対応することを考慮した際に，自ら改変可能な言語パーサを利用することが望ましいと考えたためである．

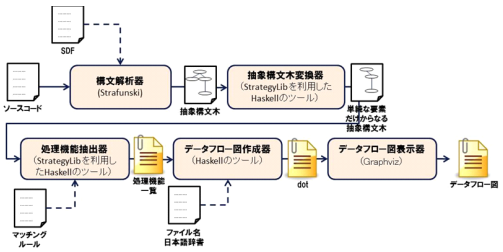

図 3.4　試作した処理機能抽出ツールの概要

また，Step2 で利用する抽象構文木の走査・変換モジュールとしては，StrategyLib [21] と呼ばれる Haskell のツリー構造を操作することができるライブラリを利用した．このライブラリを利用すると，ツリー構造の走査・変換の条件を簡単に記述することができる．

Step3 で利用する抽象構文木のマッチングルールには，前述の StrategyLib を利用して処理機能毎に Haskell でマッチングルールを作成した．

Step4 として，マッチングにより抽出された処理機能と入出力ファイルの関係を表示するために，Graphviz [22] を利用した．

3.5.2　抽出対象のプログラム

適用実験において，処理機能を抽出する対象プログラムとして，JCL と COBOL 言語で書かれた申込書チェック機能を用いた．この機能は，申込書ファイルや，それに付随する補助情報ファイル（漢字氏名ファイルなど）を入力として，申込書の各項目間の必須チェックや関連チェック，補助情報との関連チェックなどを行い，なんらかのチェックが通らずエラーとなる申込書レコードを，エラー理由と共にエラーリストとして出力する機能である．

図 3.5 にこの機能について既存手法で作成したデータフロー図を示す．

3.5.3　処理機能の抽出結果

試作したツールによって作成されたデータフロー図を図 6 に示す．

Graphviz で出力しやすいように，3.4 節でのサンプルと比べてノードの形を変更している．楕円形のノードがファイルを表しており，長方形のノードがプログラムを表している．また，ファイルのノードの 1 行目はファイルの日本語名を，2 行目はファイルの物理名を表している．逆にプログラムのノードの 1 行目はプログラムの物理名を，2 行目はプログラムの日本語名を表している．

処理機能を抽出した結果，表 3.5 に示すように既存手法で作成したデータフロー図に比べて機能が細分化され，機能名が変化した．

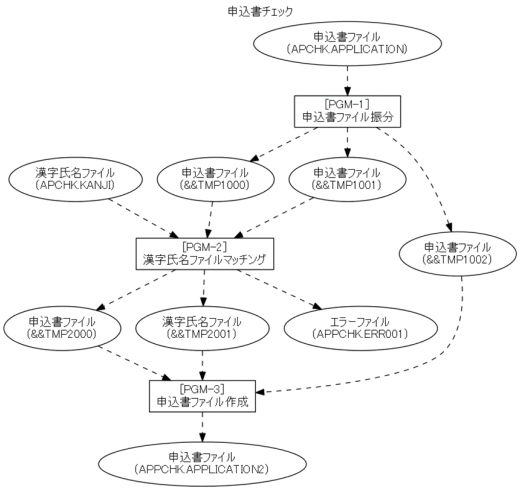

図 3.5　既存手法で作成した申込書チェック機能のデータフロー図

表 3.5　処理機能の抽出による機能名の変化

ID	既存手法での機能名	提案手法での機能名
PGM-1	申込書ファイル振分	申込書ファイル振分
PGM-2	漢字指名ファイルマッチング	申込書ファイル抽出
		申込書ファイル和
		申込書ファイル・漢字氏名ファイルマッチング
PGM-3	申込書ファイル作成	申込書ファイル編集
		申込書ファイル和

3.5.4　評価

図 3.6 をシステムの保守開発担当者 3 人に見せ，業務用語とシステム用語の対応関係の理解に役立つかという観点でコメントを求めた．

この結果，対応関係の理解に役立つという以下のようなコメントを得た．

- 漢字氏名ファイルの内容を申込書ファイルに転記している箇所を調べたい場合，既存手法では，漢字氏名ファイルと申込書ファイルを入力にしている PGM-2 と PGM-3 両方を調査しなければいけなかった．本手法では，「申込書ファイル編集」が PGM-3 だけであるため，PGM-2 は調査しなくてもよいことが分かる．

また一方で，影響調査の範囲も小さくできるという以下のようなコメントも得た．

- 例えば，漢字氏名ファイルの一部の項目を変更する変更依頼があった場合，既存手法では，PGM-3 において&&TMP2001 の漢字氏名ファイルと，&&TMP2000 と&&TMP1002 の申込書ファイルの 3 つが入力となって，一つの申込書ファ

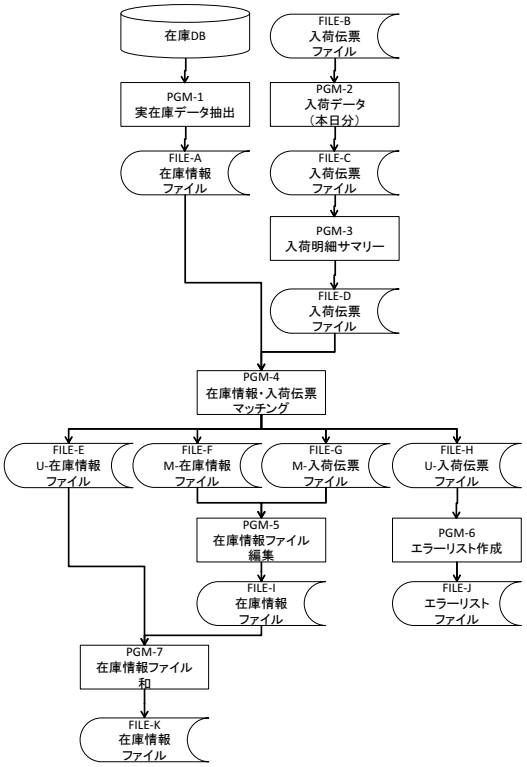

図 3.6　本手法で作成した申込書チェック機能のデータフロー図

イルを出力しているため，&&TMP2000 と&&TMP1002 の申込書ファイル両方に影響がないかを確認しなければいけなかった．一方で本手法では，&&TMP1002 の申込書ファイルは漢字氏名ファイルとは無関係なことが分かる．

3.6　関連研究

業務用語からその実装箇所を特定する研究は、実装箇所特定（Feature Location）と呼ばれ，これまで多くの研究がされてきた．Bogdan ら [23] は，実装箇所特定の研究・技術を特定技術ごとに，動的解析，静的解析，テキスト解析，それらの組み合わせに大別した．本研究の提案手法は，静的解析の一つであると考えられる．

実装箇所特定の静的解析手法として，Matin ら [24] は，オブジェクト指向言語を対象に，特徴とソースコードの対応関係を取る手法として，関心事グラフ（Concern Graph）を利用する手法を提案している．この手法では，まずプログラムを解析し，プログラムモデルと呼ばれるグラフを作成すする．続いて，プログラムモデルから、利用者が必要ないと考えるノードやエッジを取り除いて関心事グラフを作成する．本研究の提案手法とは，適用対象とグラフの粒度の 2 点で異なる．Martin らの手法はオブ

ジェクト指向言語のプログラム一般が対象であるが，我々の手法は手続き型言語一般のバッチ処理が対象である．また，Martin らの提案するプログラムモデル及び関心事グラフのノードは，クラス，フィールド，メソッドのいずれかで，エッジは，呼び出し，読み込み，書き込み，チェック，作成，宣言，スーパークラスの 7 種類の関係のいずれかである．一方で，本研究の提案手法のデータフロー図のノードは，ファイル，処理機能のどちらかであり，エッジは，入力，出力のいずれかである．また Martin らの手法では，データと機能の関係を読み込みエッジ，書き込みエッジのどちらかで表現するが，本研究の手法では，この関係を処理機能 7 種類として表現している．

静的解析手法の別のアプローチとして，Zhao ら [25] が，ソースコードの分岐を保存したコールグラフと機能名との対応関係を利用する SNIAFL というアプローチを提案している．この手法の前提には，過去の設計者やプログラマが機能名などの識別子に意味のある名前をつけていることが必要となる．本研究の手法は，識別子の名前に意味が無い場合，あるいはプログラムの内容を表していない場合でも適用できることが異なっている．

3.7 本章のまとめ

本研究では，ソースコードを入力として業務用語とシステムの用語の対応関係の理解を支援するために，関係代数を応用した処理機能を提案し，ソースコードから処理機能を抽出する手法について提案した．提案手法では，ソースコードを構文解析して得られた抽象構文木に対して，予め定義しておいた処理機能との対応関係を表すルールによって処理機能を抽出し，抽出された処理機能を元に作成した機能名を利用したデータフロー図を作成する．また，処理機能をソースコードから抽出し，自動付与された機能名によるデータフロー図を作成するツールを試作した．このツールは，COBOL 言語のソースコードから処理機能を抽出し，自動付与された機能名によるデータフロー図を自動生成することができた．さらに本研究では，試作したツールを用いて評価を行い，COBOL 言語のソースコードから業務用語とシステム用語の対応関係を理解するために有用なデータフロー図を得ることができることを示した．

今後の課題としては，本手法の有効性の検証があげられる．具体的には，被験者や対象機能を増やした本手法の定性的評価の実施と，定量的な評価の実施である．今回行った定性的な評価では，処理機能とファイル名から作成した機能名はシステムの理解に役立つという結果であったが，評価者や評価の対象機能が少ない．また実際にどの程度保守における理解工数削減に役だっているかという定量的な評価は行えていない．今後は，被験者や対象機能を増やした定性的評価を実施すると共に，今回提案した機能名一覧を利用した場合と利用しなかった場合で，どの程度の工数削減が見込めるのかを計測し，定量的な評価を行う．

第 4 章

レガシーシステム移行時の性能劣化を改善するリファクタリング支援手法

長年保守し続けられてきたメインフレーム上の基幹システム (以降はレガシーシステムと呼ぶ) では，データベースから取得した結果に対して、ループや条件分岐などの制御構造 (Loop Idiom) を用いて加工を行う手続き型のプログラムが数多く存在する．これらのプログラムを新しいプログラミング言語に移行する際に実行時性能の劣化がしばしば起こる．こういった実行時性能の劣化を防ぐため，プログラムを分散処理などに書き換えることが多いが，この書き換えは単純ではなく工数がかかる．本研究では，新しいプログラミング言語に書き換えられたプログラムを並列実行可能な形に書き換える作業を支援することで，移行後のプログラムの実行時性能を改善する手法について提案する．評価実験として，2 つの実際のレガシーシステムのプログラム 7,565 本に対して提案手法を適用した．書き換え前後のプログラムに同一の入力データを与えることで振る舞いを保った書き換えができていることを確認するとともに，書き換え前後のプログラムの実行時間の評価を行うことで，提案手法の有効性を確認した．本手法で実際のレガシーシステムのソースコード 3,529 本の書き換えができ，書き換え前と比較して実行時性能を 2 倍から 50 倍改善させることができた．

4.1 はじめに

マイグレーションの「新アプリケーションの設計」ステップでは，ステップ 2 の「レガシーシステムの構造の分解」で整理したインタフェースを境目として各アプリケーションモジュールの機能を設計する．

新アプリケーションの設計の方法として，既存のレガシーシステムの設計をそのまま流用して別のプログラミング言語で書き直すことや，レガシーシステムのプログラ

ムを機械的に新しいプログラミング言語に書き換えることが多く行われる．こういったレガシーシステムの設計流用や機械的な書き換えを行ったアプリケーションでは，メインフレームとサーバのハードウェアアーキテクチャの違いなどから，実行時性能の劣化がしばしば起こる [26] [27].

実行時性能を改善する方法として，移行後のハードウェアスペックの増強（スケールアップ）やハードウェア台数の増強（スケールアウト）がある．スケールアップはハードウェアの調達コストやランニングコストの増大に繋がることと，増強できるスペックにも限界があるため，スケールアウトを選択することが多い．しかしスケールアウトはプログラム自体が並列実行に適している必要がある．

レガシーシステムの移行において単純なスケールアウトで解決できない性能劣化が起こることがある．レガシーシステムでは複数のプログラムを並列に実行することができるため，それをそのまま移行することで移行後システムでも複数プログラムの並列実行ができ，このような部分に関してはスケールアウトで性能改善を行うことができる．一方で，単一のプログラム内の処理の中身までを単純に並列実行することはできない．例えば，データベースやファイルから取得したひとまとまりのデータを一括して処理するバッチ処理と呼ばれる処理方式では，一つのプログラム内で大量のデータを処理するが，各データについての処理を単純に並列実行することができない．単一のプログラムにおいて実行時間が非常に長くかかる場合，そのプログラムがボトルネックとなり，複数プログラムの並列実行では全体の実行時間を短くできなくなる．

単一のプログラムの性能劣化を改善するためには，そのプログラム内の処理の中身を並列実行できるようにプログラム自体を書き換える必要があるが，この書き換えは単純ではなく工数がかかる．

本研究では，レガシーシステムを新しいプログラミング言語に移行した際の性能劣化（特に単一プログラムによる性能劣化）を解決するため，プログラム内の処理を並列実行可能な形に書き換える作業を支援する手法について提案する．本手法では，レガシーシステムのプログラムの中からパターンマッチによって並列実行可能な処理を抜粋し，書き換え方法を示す．書き換え作業者はその内容を基に抜粋された処理を並列実行可能なプログラムに書き換える．

評価実験として，2 つの実際のレガシーシステムのプログラム集合に対して提案手法を適用した．書き換え前後のプログラムに同一の入力データを与えることで振る舞いを保った書き換えができていることを確認するとともに，書き換え前後のプログラムの実行時間の評価を行うことで，提案手法の有効性を確認した．

以降，4.2 節では移行に伴う性能劣化問題の詳細とその解決のために行われる一般的な手法について紹介し，4.3 節で提案手法について述べる．4.4 節で評価実験の方法と結果を説明し，4.5 節で実験結果について考察する．4.6 節では関連研究につて紹介し，4.7 節でまとめと今後の課題を述べる．

売上データ

商品ID	売上日	数量
10001	2021/01/02	200
10002	2021/01/02	50
10002	2021/01/05	500
10004	2021/01/01	300

処理結果

商品ID	売上日	数量	大量flg
10001	2021/01/02	200	False
10002	2021/01/05	500	True
10004	2021/01/01	300	False

図 4.1　バッチ処理の入出力例 (Filter)

4.2　背景

4.2.1　レガシーシステムにおけるバッチ処理プログラム

基幹システムでは，データベースやファイルから取得したひとまとまりのデータを一括して処理するバッチ処理と呼ばれる処理方式が数多く存在する．レガシーシステムにおけるバッチ処理の入出力結果の例を図 4.1 に，そのバッチ処理プログラムの例を図 4.2 に示す．この処理は「入力の売上データの中から数量が 100 以上のレコードだけを抽出し，数量が 400 以上のレコードの場合は大量 flg に True を，そうでない場合は False を設定した結果を出力する」処理であり，このプログラムでは「売上データから 1 レコードずつ読み込み，読み込んだレコードの『数量』の値が 100 以上なら，その内容を抽出結果として出力する」ことによって実現している．

レガシーシステムのバッチ処理プログラムにおいて「ループ内でファイルを 1 レコード分読み込み，そのレコードに対する処理を行い，結果を出力する」といった実装は一般的である．このようなループを使った手続き的な実装では，レコード数だけ繰り返される各レコードに対する処理は直列に処理されることとなる．また全てのレコードの処理を完了させるためには，各レコードの処理に掛かった時間の総和分だけの時間が必要となる．

このような各レコードを直列に処理するプログラムには潜在的な性能問題がある．メインフレームはオープン系サーバに比べて処理性能が高く，各レコードの処理が非常に高速に行われるため，各レコードの処理を直列処理してもその総和の時間も小さく，性能問題として顕在化しない．一方でこういったプログラムをメインフレームに

数量が 100 以上の売上レコードを抽出し，
数量が 400 以上の場合「大量 flg」を True にする

```
1:  READ 売上データ
2:  PERFORM UNTIL 売上データEOF
3:    IF 売上データ.数量 >= 100 THEN
4:      MOVE 売上データ TO 抽出結果
5:      IF 売上データ.数量 >= 400 THEN
6:        MOVE True  TO 大量flg
7:      ELSE
8:        MOVE False TO 大量flg
9:      END-IF
10:     WRITE 抽出結果
11:   END-IF
12:   READ 売上データ
13: END-PERFORM.
```

図 4.2　バッチ処理プログラムの例 (Filter)

比べて相対的に処理性能が劣るオープン系サーバで実行すると，各レコードの処理で時間がかかり，その総和の時間も大きくなるため性能問題として顕在化する．このような性能問題は，メインフレーム上で動くレガシーシステムのプログラムをそのままオープン系サーバで実行できるようにするリホストと呼ばれる再構築や，レガシーシステムのプログラムを機械的に別の言語に書き換えてオープン系サーバで実行するリライトと呼ばれる再構築を行った際に起こる [26] [27] [28].

性能問題が起こった時に一般的に行われる性能改善方法として，処理の並列化がある．各レコードの処理を別個に並列実行することで，並列数に応じて処理にかかる時間を短くすることができる．

4.2.2　バッチフレームワーク

バッチ処理プログラムの並列化を簡易に行える方法として，バッチフレームワークを利用することが挙げられる．バッチフレームワークが備えるべき仕様は JSR-352 [29] で提案されており，その実装として SpringBatch [*1] や JBeret [*2] などがある．バッチフレームワークはバッチ処理に必要な基本的な機能や並列実行のための仕組みを有しており，バッチフレームワークの利用者は実現したいバッチ処理プログラム固有の処理ロジックだけを実装することで並列実行可能なバッチ処理プログラムを実現することができる．

代表的なバッチフレームワークである SpringBatch では，以下の 3 つの処理について利用者が個別に実装することでバッチ処理プログラムを実現することができる．

- Reader: 加工対象のデータを読み込む処理

[*1] https://spring.io/projects/spring-batch
[*2] https://github.com/jberet/jsr352

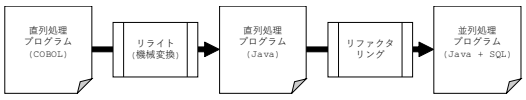

図 4.3　レガシーシステムプログラムの移行からバッチフレームワークの利用までの流れ

■Reader

```
SELECT 商品ID, 売上日, 数量 FROM 売上データ WHERE 数量 >= 100
```

■Processor

```
1: public Sale2 process(Sale item) throws Exception {
2:   Sale2 item2 = new Sale2(item);
3:   if(item.数量 => 400){
4:     item2.大量flg = True;
5:   else{
6:     item2.大量flg = False;
7:   }
8:   return item2;
9: }
```

■Writer

```
INSERT 抽出結果 SET 商品ID = #{売上データ2.商品ID}
      売上日 = #{売上データ2.売上日}
      数量 = #{売上データ2.数量}
      大量flg = #{売上データ2.大量flg}
```

図 4.4　バッチフレームワークによる実装例 (Filter)

- Processor: 読み込まれたデータ 1 レコード分を加工する処理
- Writer: 加工後のデータを書き出す処理

多くの場合，Reader や Writer は SQL で記述し，Processor は Java などの手続き型プログラミング言語で記述する．

リホストやリライトにおいて性能問題が顕在化した場合，このようなバッチフレームワークを利用し，既存の直列に処理するバッチ処理プログラムを Reader/Processor/Writer の各要素に分割して実装しなおすことで，並列化実装を実現し，性能問題を解決する．図 4.3 にレガシーシステムプログラムの移行からバッチフレームワークの利用までの一連の流れを示す．

図 4.1 を実現するバッチフレームワークでの並列実装の例を図 4.4 に示す．Reader では加工対象のデータを読み込む処理として「売上データから数量が 100 以上のデータだけを抽出する」ことを意味する SQL の SELECT 文を記述している．Processor では読み込まれたデータ 1 レコード分を加工する処理を記述する．今回の例では入力の「数量」が 400 以上のレコードの場合は「大量 flg」に「True」を，そうでない場合は「False」を設定する処理を記述している．Writer では加工後のデータを書き出す処理として「Processor で加工後の売上データを抽出結果テーブルに書き込む」ことを意味する SQL の INSERT 文を記述している．バッチフレームワークを用いてこのように実装すると，Reader でデータが読み込まれ，Processor 部分が並列実行され，Writer でその結果が出力されるバッチ処理プログラムを実現することができる．

4.2.3 バッチフレームワークへの変換

このように既存の直列処理のプログラムをバッチフレームワークを使った並列化実装に書き換えること（以降はバッチ処理プログラムのリファクタリングと呼ぶ）はバッチ処理プログラムの性能問題解決に有用なものの，実際にリファクタリングを行うことは容易ではない．容易ではない理由として処理の混在の問題が挙げられる．既存の直列処理のプログラムではバッチフレームワークの Reader/Processor/Writer に相当する処理が混在して書かれており，既存のプログラムのどこがそれぞれと対応する部分なのか，対応する部分をバッチフレームワークでどのように記述すべきかを判断することが難しいためである．

図 4.2 のような COBOL のバッチ処理プログラムでは，バッチフレームワークの Reader に記述する「数量が 100 以上」というロジックも，Processor に記述する「数量が 400 以上」というロジックも一つのプログラム内に混在して記述されており，それぞれがバッチフレームのどの要素と対応しているのかはプログラムの詳細を理解しないと判断することができない．実際のレガシーシステムのバッチ処理プログラムにおいては一つの COBOL プログラムが数千行から数万行に及ぶこともあり，その内容を理解しながらバッチ処理プログラムのリファクタリングを行うことは非常に困難である．

4.3 提案手法

本研究では，性能改善を目的としたレガシーシステムのバッチ処理プログラムからバッチフレームワークの実装へのリファクタリングを支援する手法を提案する．提案手法ではレガシーシステムのバッチ処理プログラムのリファクタリングを以下のステップで実施する．

1. Reader 部分の抽出
2. Reader 部分の SQL への変換
3. Processor の書き換え
4. Writer 部分の SQL への変換

以降の節ではまずシンプルなバッチ処理プログラムに対する各ステップの詳細を紹介し，節 4.3.5 では発展的な内容として実際のレガシーシステムに存在する複雑なバッチ処理プログラムに対する考え方を述べる．

```
PERFORM UNTIL 売上データEOF
  IF 売上データ.数量 >= 100 THEN
    WRITE 抽出結果
  END-IF
  READ 売上データ
END-PERFORM.
```

図 4.5　図 4.2 の Reader 部分に該当するロジック

4.3.1　Reader 部分の抽出

まず機械変換されたレガシーシステムのバッチ処理プログラムのソースコードから Reader に該当する部分を抽出する．

Reader は読み込むテーブルの全レコードの中から Processor や Writer が処理する対象レコードのみを抽出する処理である．レガシーシステムのバッチ処理プログラムの実装では，ループを使って 1 行ずつ読み込み，そのレコードを出力するかどうかを分岐文を使って手続き的に記述する．このため，各レコードを処理するループ中の READ/WRITE に相当する文と，その READ/WRITE に相当する文の実行有無を決定する制御構造に着目することで Reader に該当する部分を抽出できる．

例えば図 4.2 の例では，10 行目の WRITE 文によって処理されたレコードが出力されるが，この WRITE 文は 3 行目の IF 文によって実行されるかどうかが決まる．このため，3 行目の IF 文はバッチフレームワークの実装の Reader に関係するロジックであることが分かる．一方で，5 行目の IF 文は WRITE 文の実行有無には関係しておらず，Reader に関係するロジックではない．

提案手法では以下の手順で Reader 部分に該当するロジックを抽出する．

- プログラム呼び出し文を呼び出し先の内容でインライン展開する
- 各レコードを処理するループに含まれない文を削除する
- READ/WRITE 文及び，その READ/WRITE 文の実行有無を決定する制御構造以外の文を削除する
- 分岐文のネストは条件式を AND で繋げて 1 つの分岐文にまとめる

図 4.5 に 図 4.2 の Reader 部分に該当するロジックを示す．

4.3.2　Reader 部分の SQL への変換

次に抽出した Reader 部分に該当するロジックを SQL に変換する．

一般的には Reader 部分に該当するロジックは無数にあり，それに対応する SQL も無数にある．しかし我々の予備調査の結果，実システムに存在する多くのバッチ処

表 4.1　Loop Idiom の一覧

Loop Idiom	意味
Edit	全ての行を出力する.
Filter	抽出条件にマッチした行だけを出力する.
Grouping	行をグループごとに分類しそれぞれの値を計算する. 例: min/max
Join	2 つの入力ファイルでキーが同じ行を結合して出力する.
Difference	片方のファイルから他方のファイルに属する行を取り除いた行を出力する.
Split	条件に従って入力ファイルの行を複数の出力ファイルに分割する.
Union	2 つの入力ファイルの全ての行を 1 つのファイルとして出力する.

```
PERFORM UNTIL 売上データEOF
  IF 売上データ.商品ID == 商品データ.ID THEN
    WRITE 売上詳細2
    READ 売上データ
  ELSE IF 売上データ.商品ID > 商品データ.ID THEN
    READ 商品データ
  ELSE
    READ 売上データ
  END-IF.
END-PERFORM.
```

図 4.6　Loop idiom (Join)

理プログラムの Reader 部分に該当するロジックは少数のパターンでカバーできることが分かったため，Reader 部分に該当するロジックのパターンごとに対応する SQL を予め用意しておき，その SQL に変換する.

表 4.1 に我々が見つけた Reader 部分に該当するロジックのパターン (以降，Loop Idiom と呼ぶ) の一覧を示す.

図 4.5 は Filter の Loop Idiom の例だが，別の例として図 4.6 に Join の Loop Idiom の例を示す．この例では売上データに対して，売上データの商品 ID と同じ ID の商品データの情報を結合して出力する.

また各 Loop Idiom と対応する SQL への変換方法は表 4.2 の通りである.

また実際のソースコードでは，条件式に中間変数が使われている場合があるが，その場合は中間変数についてデータフロー解析を行い，関連する文を抽出し，その内容を基に SQL に変換する.

4.3.3　Processor の書き換え

次に Processor 部分の書き換えを行う．本提案手法のステップ (1)「Reader 部分の抽出」では READ/WRITE 文の実行有無を決定する制御構造に着目したが，このステップでは逆にその際に無視した分岐文や代入文に着目する．例えば図 4.2 の例では,

表 4.2　Loop Idiom と対応する SQL

Loop Idiom	**SQL**
Edit	SELECT * FROM [入力ファイル]
Filter	SELECT * FROM [入力ファイル] WHERE [WRITE 文を内包する分岐文の条件式]
Grouping	SELECT * FROM [入力ファイル] GROUP BY [WRITE 文を内包する分岐文の 条件式中の属性]
Join	SELECT * FROM [入力ファイル 1] JOIN [入力ファイル 2] ON [WRITE 文を内包する分岐文の 条件式中の属性]
Difference	SELECT FROM [入力ファイル 1] EXCEPT SELECT FROM [入力ファイル 2]
Split	複数の SELECT * FROM [入力ファイル] WHERE [WRITE 文を内包する分岐文の条件式]
Union	SELECT * FROM [入力ファイル 1] UNION SELECT * FROM [入力ファイル 2]

4 行目の MOVE 文（代入文）や 5 行目から 9 行目までの IF ブロックは処理対象となった行の編集処理であり，Processor に記述する内容である．

基本的には，ステップ (1) で無視した分岐文や代入文をそのまま Processor に記述するだけだが，ステップ (2)「Reader 部分の SQL への変換」で抽出した Loop Idiom が Grouping だった場合は以下のような Reader 部への記述を追加する．

I. 単純な代入文の場合，Reader でその変数の最終レコードの値を取得する SQL を書く
II. 変数に 1 を加算している場合（例： $num = num + 1$），Reader の SQL で属性名に COUNT を用いる
III. 変数に別の変数の値を加算している場合（例： $total = total + num$），Reader の SQL で属性名に SUM を用いる
IV. 前回の値よりも大きい（もしくは小さい）場合のみ値を更新ししている場合（例： $if(prev > now)prev = now$），Reader の SQL で属性名に MAX (もしくは MIN) を用いる

4.3.4　Writer 部分の SQL への変換

最後に Writer 部分を SQL に変換する．WRITE 文で出力されるファイルに相当するテーブルへの INSERT 文を記述する．

表 4.3　対象のレガシーシステムのプロフィール

システム	業種	ファイル数	LOC
A	金融	6,550	14.8M
B	保険	1,015	2.3M

4.3.5　Loop Idiom の組み合わせ

本節では 1 つのバッチ処理プログラムに Loop Idiom が 1 つだけ含まれる場合について説明したが，実際のレガシーシステムでは 1 つのバッチ処理プログラムに複数の Loop Idiom が含まれることも多い．このような場合，表 4.1 で示した Loop Idiom の組み合わせとしてみなすことで，Reader 部分の SQL に変換することができる．

例えば，1 つのバッチ処理プログラム中に Filter と Join の両方が含まれていた場合，Filter に由来する WHERE 句と，Join に由来する JOIN ON 句の両方を持つ SQL とする．ただし，本プロジェクトでは予算の関係で変換は行わなかった．

4.4　評価実験

実際のレガシーシステムのシステム再構築プロジェクトの一環として，レガシーシステムのプログラムに対して，提案手法を用いて Loop Idiom の抽出と対応するリファクタリング方法の提示を行い，その結果に従って人手でリファクタリングを行った．また，それぞれの種類の Loop Idiom を含むプログラムを 1 つずつ選択し，手法の適用有無による工数及び性能についての比較実験を行った．

4.4.1　実験方法

実験に用いたソースコードは，第一著者が所属する企業で実際に保守開発を行っているレガシーシステム 2 つのものである．各システムのプロフィールを表 4.3 に示す．

レガシーシステムのプログラムは全て COBOL で記述されていたが，実験を行う前に機械変換ツールを用いて Java に変換しており，実験には Java に変換後のプログラムを利用している．

まずこれらのレガシーシステムのバッチ処理プログラムにどのような Loop Idiom がどれだけあるのかを作成したツールを用いて調査する．

次に Loop Idiom を 1 つだけ含むプログラムについては，全て人手でリファクタリングを行う．この際，リファクタリング実施者には，リファクタリング対象のプログラムにどの Loop Idiom が存在するのかという情報と共に，図 4.7 のような SQL に書き直す際に必要な情報が，プログラムのどの文に対応するのかといった情報も与え

#	ソースコード	Edit
1	Statement st = connection.createStatement();	
2	String sql = “SELECT 商品ID, 売上日, 数量 FROM 売上データ”;	○
3	ResultSet rs = st.executeQuery(sql);	○
4	rs.next();	○
5	while(!rs.isLast()){	○
6	Sale item = toSale(rs);	編集内容
7	Sale2 item2 = new Sale2(item);	編集内容
8	if(item.数量 => 400){	編集内容
9	item2.大量flg = True;	編集内容
10	else{	編集内容
11	item2.大量flg = False;	編集内容
12	}	編集内容
13	sql = “INSERT 抽出結果 SET 商品ID = ” ++ item2.商品ID	○
14	++ “ 売上日 = ” ++ item2.売上日	○
15	++ “ 数量 = ” ++ item2.数量	○
16	++ “ 大量flg = ” ++ item2.大量flg;	○
17	st.executeUpdate(sql);	○
18	rs.next();	○
19	}	○

図 4.7　リファクタリング実施者に提示する情報例

る．SQL に書き直す際に必要な情報としては，Edit であれば編集内容，Filter であれば Where 句となる WRITE 文を内包する分岐文の条件式などである．実プロジェクトの予算及び期間的な制約のため，Loop Idiom の組み合わせとなるプログラムに関しては，Loop Idiom の調査までは行うもののリファクタリングを行わない．

リファクタリング後には，リファクタリング前の振る舞いを保持できているかを確認するため，書き換え前後のプログラムに同一の入力データを与えて同じ出力がされるかを確認するテストを行う．実際のレガシーシステムの再構築プロジェクトでは，ここまで実施したものを運用する．

その後，実プロジェクトとは別に，手法の適用有無による工数及び性能についての比較実験を行う．これらの比較実験では，各 Loop Idiom を含むプログラムを 1 つずつランダムに選択し，そのプログラムだけを対象とする．

工数についての比較実験では，提案手法を用いてリファクタリング方法を示した場合と提案手法無しでリファクタリングを行う場合の工数を比較し，提案手法の工数削減効果について確認する．工数についてはリファクタリングによる書き換えの時間だけでなく，上述の振る舞いを保持できているかを確認するテストの時間も含む．実施者による影響を小さくするため，Loop Idiom 毎に実施者への提案手法の有無の割り当てを入れ換えて実施した．例えば，Loop Idiom が Edit の際にリファクタリング実施者 A が提案手法有りで実施し，B が提案手法無しで実施した場合，Loop Idiom が Filter の際には A が提案手法無しで実施し，B が提案手法有りで実施するといった割り当てを行った．

性能についての比較実験では，同一のデータを用いてリファクタリング前後のプログラムを実行し，リファクタリング前後で実行時性能が変化するか確認する．リファクタリング後の実行時性能については，並列度を 1 と 8 に変えて実行時間を計測する

表 4.4　Loop Idiom の調査結果

Loop Idiom	A システム	B システム
Edit	1,093 (16.7%)	123 (12.1%)
Filter	809 (12.4%)	144 (14.2%)
Grouping	226 (3.5%)	9 (0.9%)
Join	473 (7.2%)	56 (5.5%)
Difference	49 (0.7%)	2 (0.2%)
Split	369 (5.6%)	119 (11.7%)
Union	57 (0.9%)	0 (0%)
組み合せ	3,474 (53.0%)	562 (55.4%)

ことで並列度の違いによる実行時性能についても比較する．

使用したマシンのスペックとバッチフレームワークは以下の通りである．

- CPU: Intel Core i5 1.4GHz
- 仮想 CPU コア数: 8
- Memory: 16 GB
- OS: RedHat Enterprise Linux 7.5
- バッチフレームワーク: Spring Batch 4.3.3
- DBMS: MySQL 8.0.24

リファクタリング前後の両方で，DB に格納されている各テーブルの主キーとなるカラムにインデックスを付与した．それ以外の DB チューニングはリファクタリング前後ともに実施していない．

またプログラム実行時に必要な入力データは各プログラムが必要なデータ形式に合わせて作成したダミーデータを用いた．Edit/Filter/Grouping/Split はそれぞれ 124 万レコードのデータ，Join は 124 万レコードのデータと 400 レコードからなるデータ，Difference は 124 万レコードのデータと 123 万レコードからなるデータ，Union は 124 万レコードのデータと 124 万レコードからなるデータをそれぞれ用いた．

4.4.2　Loop Idiom の調査結果

レガシーシステム内に含まれる Loop Idiom の調査結果は表 4.4 の通りである．「組み合わせ」は複数の Loop Idiom が 1 つのファイルに存在していたことを意味している．

実験に用いた実システムにおいては，組み合わせでない Loop Idiom が半数近くを占めている．また Loop Idiom の種類によって出現数にばらつきがあることが分かり，どちらのシステムも Edit/Filter/Join/Split の出現数が多く，Difference/Union は殆ど出現しないことも分かった．また，全てのプログラムは 7 種類の Loop Idiom またはその組み合わせで表現できた．

実プロジェクトでは 1 つの Loop Idiom だけが含まれるプログラムの全て 3,529

表 4.5　提案手法の有無によるリファクタリング工数の差

Loop Idiom	手法無し (人日)	手法有り (人日)
Edit	5	3 (60%)
Filter	10	5 (50%)
Grouping	15	5 (33%)
Join	15	8 (53%)
Difference	12	5 (42%)
Split	5	5 (100%)
Union	3	3 (100%)

表 4.6　性能改善結果

Loop Idiom	実施前 (s)	1 並列 (s)	8 並列 (s)
Edit	51.9	42.6 (1.2 倍)	10.8 (4.8 倍)
Filter	57.6	41.2 (1.4 倍)	8.9 (6.5 倍)
Grouping	46.3	2.9 (16.0 倍)	0.7 (66.1 倍)
Join	54.1	40.4 (1.3 倍)	27.3 (2.0 倍)
Difference	97.7	4.6 (21.2 倍)	1.9 (51.4 倍)
Split	59.6	45.8 (1.3 倍)	10.1 (5.9 倍)
Union	102.1	86.7 (1.2 倍)	14.6 (7.0 倍)

本について，リファクタリング及びテストを行った．一方で，複数の Loop Idiom が含まれる 4,036 本のプログラムについてはリファクタリングを行っていない．

4.4.3　提案手法によるリファクタリング工数の差

提案手法の有無によるリファクタリング工数の違いは表 4.5 の通りである．

Split や Union では提案手法の有無で工数の差は無いものの，それ以外の Loop Idiom においてはいずれも本提案手法による手法有りの方が工数が小さくなっている．特に Filter/Grouping/Difference については工数が 50% から 66% 程度まで削減されており，工数削減の効果が大きい．

4.4.4　リファクタリングによる性能改善結果

リファクタリングによる性能改善の結果は表 4.6 の通りである．「実施前」は Java への機械変換がなされただけのプログラムの実行結果であり，「1 並列」及び「8 並列」は提案手法を用いてリファクタリングを行ったプログラムの実行結果である．

いずれの Loop Idiom においてもリファクタリング実施後のプログラムの方がリファクタリング前よりも実行時性能が改善されることが確認された．またいずれも並列度を 1 から 8 に上げると実行時性能が改善されることから，並列実行が効果的に働いていることが分かる．

特に Grouping と Difference については，他の Loop Idiom と比較して並列度 1 においても顕著に性能が改善されている．

4.5 考察

Loop Idiom の出現数のばらつきや提案手法による工数，作業品質，性能への影響について，それぞれ考察を行った．

4.5.1 Loop Idiom の出現数

Loop Idiom の種類ごとに出現数にばらつきがあることについて，それぞれのシステムの保守を担当する有識者にヒアリングを行ったところ，「連続するバッチ処理では，その前半に Join を行い，その後 Edit/Filter といった処理を多数行い，最後に必要に応じて Grouping を行うといった業務が多いため，Join/Edit/Filter/Grouping が多いのは感覚と一致する」という回答を得た．いくつかのバッチ処理についてそのプログラム間の実行順を調査したところ，有識者の回答通りの傾向があることを確認した．

4.5.2 提案手法による工数への影響

提案手法の有無による工数の差について，その理由をリファクタリング実施者にヒアリングを行ったところ，「本手法の適用の有無で工数が変わらなかった Split/Union は元々のプログラムの構造が単純で，プログラムの規模自体が大きくても構造把握自体は難しくなかった．一方で適用無しで工数が相対的に大きかった Grouping/Join/Difference は，構造が難しく内容を把握するのに時間がかかった．また Edit/Filter は構造の把握自体は難しくないものの，条件分岐が多い場合に出力条件と編集条件を見分けることに工数がかかるため，機械的にその見分けをしてくれる本手法は工数削減に有効だった」という回答を得た．

4.5.3 提案手法による作業品質への影響

本手法の有無によるリファクタリングの作業品質についても確認したところ，提案手法の適用があった場合はリファクタリング時のバグの作り込みは 1 件だけであったが，提案手法の適用が無かった場合には 8 件のバグの作り込みがあったことが分かった．この差異についてリファクタリング実施者にヒアリングを行ったところ，「本手法の適用がある場合には抽出箇所などが明確になるため，作業内容が難しくなくバグを作り込みにくい．一方で適用無しの場合はどのような Reader としてどのような SQL を選ぶべきかを考える必要があり，その選択ミスや，出力条件と編集条件の見極めの誤りなどバグを作りこむ要素が多い」という回答を得た．

4.5.4 提案手法による性能への影響

本手法の適用によりどの Loop Idiom においても性能が改善されたが，Loop Idiom の種類によって異なる 3 種類のボトルネックがそれぞれ改善されたためと考えられる．

1 つ目は CPU がボトルネックとなるもので，複雑な編集や抽出条件を持つ場合の Edit や Filter が該当する．図 4.2 の例は非常に簡単な抽出条件だが，実際のソースコードでは数千行に及ぶ条件式や計算ロジックを持っていることが多く，CPU がボトルネックになっている．このような場合は，バッチフレームワークによる Processor の並列化が効果的となる．

2 つ目のボトルネックとしては I/O ネックが挙げられる．これは Split や Union，単純な Edit や Filter が該当する．これらの Loop Idiom は処理が単純であり，CPU 部分ではなくデータの入出力を行う I/O がネックになる．このような場合は，バッチフレームワークによるチャンク化（複数件をまとめて入出力する方式）が性能改善に貢献する．

3 つ目の改善ポイントしては インデックスを用いた計算量の変更があり，Grouping や Difference が該当する．これらの Loop Idiom は，Reader の SQL においてインデックスを張ったカラムについて集約計算や差集合を求める内容であり，MySQL の最適化により通常であれば $O(n)$ 必要な計算も，インデックスを使用することで $O(logn)$ となったためと考えられる．

4.6 関連研究

Allamanis ら [30] は，ソースコード内のループに着目してパターンマイニングを行う手法を提案している．この手法では変数の読み書きを対象にしているが提案手法はファイルの読み書きを対象にしている点で異なる．また，この手法では 11 プロジェクト，577 KLOC のソースコードに対して評価を行っているが，我々の提案手法は 17.1 MLOC とより大規模なソースコードに適用しており，また実システムのプロジェクトとして実施した点も異なる．

提案手法と同様の問題を対象にしている研究として，Wiedermann ら [31] [32] の研究がある．彼らによると ORM (Object-relational Mapping) を用いたシステムでも性能問題は起きており，その原因としてデータベースの内部で処理したほうが良いとも思われる Filter や Join などの操作をプログラマーが手続き型プログラミング言語で記述していることを指摘している．提案手法と同じく，こういったプログラムを SQL クエリに書き換えることで性能問題を解決しようとしている．Wiedermann らの手法では，提案手法における Filter や Join に関しては書き換えることができるが，提案手法でカバーしている Grouping や Split，Union など他の Loop Idiom に

関しては書き換えることができない．また，手法の評価は 900 LOC と小規模なソースコードに留まっている．

同様の問題を解決する別のアプローチとして Cheung ら [33] は，基となるプログラムコードから後条件と不変条件を生成し，それらの条件を満たす SQL クエリ を合成するという手法を提案している．これらの手法では，Filter や Join だけでなく，Grouping についても書き換えることができるが，基となるプログラムは ORM のようなテーブル構造を変換する操作だけである．一方で提案手法がターゲットにしているバッチ処理では，テーブル構造を変換する操作に加えて，データの値を編集する操作も一つのプログラムに混在して書かれている．データの値を編集する操作を後条件と不変条件だけで表現するのは不十分であり，Cheung らの手法でバッチ処理を合成することはできない．この手法は 123 KLOC のソースコードに対して評価しているが，提案手法の方がより大規模に適用している点も異なる．

4.7 本章のまとめ

本研究では，レガシーシステムのプログラムが新しいプログラミング言語に移行された時に起こる性能劣化を改善するリファクタリング手法について提案した．また，2 つの企業のレガシーステムのソースコードを人手で調査し，どのような Loop Idiom がどれくらい存在するかを調査し，各 Loop Idiom の種類について提案手法による支援を用いてリファクタリングを行った．

提案手法による支援を行うことでリファクタリングに必要な工数を最大 66% 削減でき，リファクタリング前後で実行時性能が最大 62 倍まで改善されることを示した．

本手法は実プロジェクトで採用されており，本手法による支援によってリファクタリングされたプログラムは実案件で運用されている．これらのことから本手法によるリファクタリング支援およびそのリファクタリングは有効であったと言える．

第 5 章

おわりに

5.1 まとめ

本論文では，レガシーシステムのマイグレーションの 3 つのステップ「レガシーシステムの分析」と「レガシーシステムの構造の分解」,「新アプリケーションの設計」における課題と解決方法について述べた.

レガシーシステムの分析においては, その前提となるプログラミング言語の判定という課題について述べ，その解決策としてパターンマッチによる自動判定と手作業で判定すべきファイルの代表をクラスタリングによって選出する手法について提案した. この手法により従来 3.3 人月かかっていたプログラミング言語の判定作業を 15 分で完了させることができるようになった.

レガシーシステムの構造の分解においては，アプリケーションの分解の単位として処理機能を定義し，その単位で機能を分割する手法を提案した．提案手法を実装したツールによって分解された処理機能については，有識者から業務を把握しやすい分解であるという評価を得ることができた.

新アプリケーションの設計においては, マイグレーション後に従来よりも性能が劣化するという課題について述べ, ループや条件分岐の制御構造 (Loop Idiom) を用いて結果加工を行う手続き型のプログラムがその原因の一つであることを示した．典型的な Loop Idiom とその並列実行可能な形への書き換え作業を支援する手法を提案することで, 実行時性能を 2 倍から 50 倍改善させることができた.

5.2 今後の研究方針

プログラミング言語の判定では, レガシーシステムのプログラミング言語の特徴を反映した特徴量を用いているため, それ以外のシステムでもよく用いられるプログラミング言語（C 言語や Java など）に関しても対応していくことが挙げられる．この場合, 新しいプログラミング言語に対応した新しい特徴量を導入する必要があると考

えられる．また, 複数のバージョン, 派生言語, 方言などが存在するプログラミング言語について, それらの違いまで正確に判定しなければいけない場合, ファイル間での特徴量の差異が小さくなり, 分類精度が低下すると考えられる．この場合には, プログラミング言語の判定を行った後に, バージョンや方言特有のキーワードなどを用いてバージョンや方言を特定するステップを追加する必要があるかもしれない．

性能劣化の改善では, リファクタリングの自動化があげられる．提案手法では, リファクタリング方針の提示だけであり, リファクタリング後のソースコードの提示は行っていない．リファクタリング後のソースコードを自動的に作成し, リファクタリング前と同じ値を返すことを確認するテストまで自動実施することができれば, より多くの工数が削減されることが見込まれる．今後はこれらの自動化を実施する予定である．

またマイグレーションにおいては今回取り上げた 3 つ以外のステップにも多くの課題は存在する．それらの内いくつかは一般的なソフトウェア工学でも共通的な課題であるが, データ移行や段階的なカットオーバーの仕方などはマイグレーション特有の問題である．今後は, 一般的なソフトウェア工学での成果を活用するとともに, マイグレーション特有の課題解決について研究を進めていきたい．

参考文献

[1] J. Bisbal, D. Lawless, B. Wu, and J. Grimson, “Legacy information systems: Issues and directions,” IEEE Software, vol.16, no.5, pp.103–111, 1999.

[2] K. Bennett, “Legacy systems: Coping with success,” IEEE Software, vol.12, no.1, pp.19–23, 1995.

[3] R. Khadka, B.V. Batlajery, A.M. Saeidi, S. Jansen, and J. Hage, “How do professionals perceive legacy systems and software modernization?,” Proceedings of the 36th International Conference on Software Engineering, pp.36–47, Hyderabad, India, 2014.

[4] MarketsandMarkets, “Application modernization services market by service (application portfolio assessment, cloud application migration, application replatforming), cloud deployment mode, organization size, vertical, and region - global forecast to 2025,” 2020.

[5] M.L. Brodie and M. Stonebraker, Migrating legacy systems: gateways, interfaces & the incremental approach, Morgan Kaufmann Pub, 1995.

[6] M.L. Brodie and M. Stonebraker, “Darwin: On the incremental migration of legacy information systems,” Technical report, GTE Laboratories, 1993.

[7] J. Bisbal, D. Lawless, B. Wu, J. Grimson, V. Wade, R. Richardson, DO’ Sullivan, “A survey of research into legacy system migration,” Technical report, Trinity College Dublin, 1997.

[8] 神居俊哉，高尾　司，メインフレーム実践ハンドブック. z/OS (MVS), MSP, VOS3 のしくみと使い方，リックテレコム，2009.

[9] J.K. van Dam and V. Zaytsev, “Software language identification with natural language classifiers,” Proceedings of the 23rd International Conference on Software Analysis, Evolution, and Reengineering, pp.624–628, Osaka, Japan, 2016.

[10] Github, “Linguist”. `https://github.com/github/linguist`.

[11] D. Klein, K. Murray, and S. Weber, “Algorithmic programming language identification,” arXiv preprint arXiv:1106.4064, pp.1–11, 2011.

[12] C. Technologies, “Ca easytrieve plus,” `http://www.ca.com/jp/devcenter/`

ca-easytrieve.aspx.

[13] C. Technologies, “Ca telon,” http://www.ca.com/us/products/detail/ca-telon.aspx.

[14] D. Arthur and S. Vassilvitskii, “K-means++: The advantages of careful seeding,” Proceedings of the 18th ACM-SIAM Symposium on Discrete Algorithms, pp.1027–1035, 2007.

[15] J. MacQueen, “Some methods for classification and analysis of multivariate observations,” Proceedings of the 5th Berkeley Symposium on Mathematical Statistics and Probability, pp.281–297, California, USA., 1967.

[16] S. Weber, “Programming-language-identification,” https://github.com/simon-weber/Programming-Language-Identification.

[17] 独立行政法人情報処理推進機構, “機能要件の合意形成ガイド,” 2010.

[18] C. Date, データベース実践講義: エンジニアのためのリレーショナル理論, O’Reilly Japan, 2006.

[19] “Strafunski”. http://www.haskell.org/haskellwiki/Applications_and_libraries/Generic_programming/Strafunski.

[20] “Sdf”. http://www.syntax-definition.org/.

[21] “strategylib”. http://hackage.haskell.org/package/StrategyLib.

[22] “graphviz”. http://graphviz.org/.

[23] B. Dit, M. Revelle, M. Gethers, and D. Poshyvanyk, “Feature location in source code: a taxonomy and survey,” Journal of software: Evolution and Process, vol.25, no.1, pp.53–95, 2013.

[24] M.P. Robillard and G.C. Murphy, “Concern graphs: finding and describing concerns using structural program dependencies,” Proceedings of the 24th international conference on Software engineering, pp.406–416, 2002.

[25] W. Zhao, L. Zhang, Y. Liu, J. Sun, and F. Yang, “Sniafl: Towards a static noninteractive approach to feature location,” ACM Transactions on Software Engineering and Methodology, vol.15, no.2, pp.195–226, 2006.

[26] 独立行政法人情報処理推進機構, “システム再構築を成功に導くユーザガイド 第2版〜ユーザとベンダで共有する再構築のリスクと対策〜,” 2018.

[27] 独立行政法人情報処理推進機構, “デジタル変革に向けた it モダナイゼーション企画のポイント集〜注意すべき 7 つの落とし穴とその対策〜,” 2018.

[28] T. Suganuma, T. Yasue, T. Onodera, and T. Nakatani, “Performance pitfalls in large-scale java applications translated from cobol,” Companion to the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications, pp.685–696, Nashville, USA, 2008.

[29] S. Kurz, “Batch applications for the java platform version 1.0.,” 2014.

[30] M. Allamanis, E.T. Barr, C. Bird, P. Devanbu, M. Marron, and C. Sutton, "Mining semantic loop idioms," IEEE Transactions on Software Engineering, vol.44, no.7, pp.651–668, 2018.

[31] B. Wiedermann and W.R. Cook, "Extracting queries by static analysis of transparent persistence," Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pp.199–210, 2007.

[32] B. Wiedermann, A. Ibrahim, and W.R. Cook, "Interprocedural query extraction for transparent persistence," ACM Sigplan Notices, vol.43, no.10, pp.19–36, 2008.

[33] A. Cheung, A. Solar-Lezama, and S. Madden, "Optimizing database-backed applications with query synthesis," ACM SIGPLAN Notices, vol.48, no.6, pp.3–14, 2013.