The University of Osaka
Institutional Knowledge Archive

Title	大規模ソフトウェアリポジトリにおけるソフトウェア部品間の依存関係解析に関する研究
Author(s)	市井, 誠
Citation	大阪大学, 2009, 博士論文
Version Type	VoR
URL	https://hdl.handle.net/11094/1367
rights	
Note	

The University of Osaka Institutional Knowledge Archive : OUKA

https://ir.library.osaka-u.ac.jp/

The University of Osaka

大規模ソフトウェアリポジトリにおける
ソフトウェア部品間の依存関係解析に関する研究

2009 年 1 月

市井　誠

大規模ソフトウェアリポジトリにおける
ソフトウェア部品間の依存関係解析に関する研究

提出先　　大阪大学大学院情報科学研究科
提出年月　2009 年 1 月

市井　誠

内容梗概

近年ではソフトウェア開発の形態が多様化し，単一の開発現場ですべての開発を行うのではなく，開発現場やプロジェクトをまたがって資産を利用することが多い．最近活発化しているオープンソースソフトウェア開発では，公開されているソースコードやバイナリなどのソフトウェア部品（部品）を異なるプロジェクトへ再利用することで効率的な開発を実現している．また，ソフトウェアベンダで開発される商用ソフトウェアや，ハードウェア向けの組み込みソフトウェアの開発においても，類似したソフトウェアの開発コストを削減するため，部品を効率的に構築および再利用するプロダクトラインと呼ばれる手法が用いられる．再利用を行う開発者は，利用可能な部品を取得するために，ソフトウェア部品検索システム（部品検索システム）を用いることが多い．部品検索システムは，様々なソフトウェアから部品を収集し，ソフトウェアリポジトリとして蓄積する．また，開発者から問い合わせを受けると，ソフトウェアリポジトリから適切な部品を検索し，開発者に提供する．

本研究では，ソフトウェアリポジトリに含まれる部品間の依存関係に着目する．過去に行われた様々な研究において，ソフトウェアの構造解析や，部品の再利用支援のために，部品間の依存関係が用いられてきた．しかし，既存研究では単一のソフトウェア内で閉じた依存関係のみを解析しており，本研究で対象とする，多数のソフトウェアを含む大規模なソフトウェアリポジトリを用いた研究は行われてこなかった．そこで，本研究では，ソフトウェアリポジトリに蓄積された部品間の依存関係を解析し，その性質に関する調査を実施する．また，得られた知見を踏まえ，部品の理解支援手法の提案および実装を行う．具体的には，以下の点に着目した研究を行う．

1. 部品間の利用関係により形成されるネットワークの性質
2. 部品を再利用する際に必要な依存関係の理解

まず，1 について説明する．部品を頂点，部品間の利用関係を有向辺としたグラフはソフトウェア部品グラフ（部品グラフ）と呼ばれ，ソフトウェアの解析手法に広く用いられている．頂点の次数分布はグラフを特徴付ける要素であり，それがべき乗則に従うグラフは様々な分野で注目されている．WWW 上のページのリンク関係やソーシャルネットワークがその例であり，コミュニティ分析などの，その性質を利用した研究が行われている．部品グラフが持つ特徴が明らかになれば，ソフトウェア設計など，その性質が部品グラフに反映されることがらについて，新しい観点からの分析が可能になると考えられる．本研究では，大規模なソフトウェアおよび多数のソフトウェアの集合に基づく部品グラフの次

数分布がべき乗則に従うかどうかを調査した．その結果として，入次数の分布がべき乗則に従うのに対し，出次数の分布は従わないことが明らかになった．

続いて，2 について説明する．オープンソースソフトウェア開発および部品検索システムの普及により，大量の部品の中から再利用可能な部品を容易に取得できるようになってきた．取得した部品をソフトウェアへ組み込むためには，その部品が依存する部品もまた同時に組み込む必要がある．しかし，手作業で依存関係を調査し，必要な部品を取得することは以下の理由により容易ではない．まず，前述したように依存関係は複数のソフトウェアにまたがることがあるため，依存関係の調査範囲を絞ることが難しく，大量の部品の中から探す必要がある．さらに，ソフトウェアリポジトリの中には，バージョンアップや派生などで，同一または類似した部品を含むソフトウェアが複数存在するため，類似した部品の中から，組み合わせを考慮しながら必要な部品を選択しなければならない．そこで，再利用対象の部品が依存する部品集合の候補を抽出する依存関係解析手法を提案する．候補それぞれは代替可能な部品集合で構成され，開発者はその中から要求を満たすものを選択することで，対象の部品の依存関係を構成する部品集合を取得できる．また，Java ソフトウェア を解析対象とするシステム DACARA として実装し，オープンソースソフトウェア集合を対象にした適用実験を行った．結果として，対象に含まれる全ての部品について，提案手法は現実的な個数の再利用単位を抽出することができた．また，依存関係が複雑であり手作業での調査が困難である部品が多く存在することが明らかになり，再利用可能な部品を組み込む際の理解のために，提案手法による支援が有効であることを確認した．

論文一覧

主要論文

[1-1] 市井誠, 松下誠, 井上克郎, “Java ソフトウェアの部品グラフにおけるべき乗則の調査”, 電子情報通信学会論文誌 D-I, Vol.J90-D, No.7, pp.1733–1743, 2007 年 7 月（学術論文）

[1-2] Makoto Ichii, Reishi Yokomori, Katsuro Inoue, “Towards Effective Reference Analysis for Software Component Retrieval System”, In Proceedings of the Workshop on Accountability and Traceability in Global Software Engineering (ATGSE2007), pp.51–52, Dec. 2007 （国際会議録）

[1-3] Makoto Ichii, Makoto Matsushita, Katsuro Inoue, “An Exploration of Power-law in Use-relation of Java Software Systems”, In Proceedings of the 19th Australian Software Engineering Conference (ASWEC2008), pp.422–431, Mar. 2008 （国際会議録）

[1-4] Makoto Ichii, Takashi Ishio, Katsuro Inoue, “Cross-application Fan-in Analysis for Finding Application-specific Concerns”, In Proceedings of the 4th Asian Workshop on Aspect-Oriented Software Development (AOAsia4), `http://appsrv.cse.cuhk.edu.hk/~aoasia/workshop/APSEC08`, Dec. 2008 （国際会議録）

関連論文

[2-1] 早瀬康裕, 今枝誉明, 市井誠, 松下誠, 井上克郎, “潜在的意味解析手法を用いたソフトウェア変更情報のクラスタリング手法”, 情報処理学会論文誌, Vol.48, No.10, pp.3352–3356, 2007 年 10 月 (学術論文)

謝辞

本研究の全般に関し，常日頃より適切なご指導を賜わりました，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，心から深く感謝申し上げます.

本論文を執筆するにあたり，適切なご助言とご指導を頂きました，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 増澤 利光 教授，同 楠本 真二 教授に心から感謝致します.

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在籍中に，適切なご助言とご指導を頂きました，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 谷口 健一 名誉教授，同 萩原 兼一 教授，同 八木 康史 教授に感謝致します.

本研究を行うにあたり，直接具体的なご指導を頂きました，大阪大学大学院情報科学研究科 荻原 剛志 招聘教授，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 松下 誠 准教授，同 石尾 隆 助教，大阪大学大学院情報科学研究科 早瀬 康裕 特任助教に心より御礼申し上げます.

本研究を行うにあたり，ご助言やご指導を頂きました，立命館大学総合理工学院情報理工学部情報システム学科 山本 哲男 准教授，南山大学数理情報学部情報通信学科 横森 励士 講師，奈良先端科学技術大学院大学情報科学研究科情報システム学専攻 川口 真司 助教，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 肥後 芳樹 助教に心より御礼申し上げます.

最後に，井上研究室の皆様のご助言, ご協力に御礼申し上げます.

目 次

第 1 章　まえがき **1**
1.1　ソフトウェア再利用 2
1.1.1　再利用のアプローチ 2
1.1.2　再利用の手順 3
1.1.3　再利用のプロセス 3
1.1.4　ソフトウェア部品検索システム 4
1.1.5　部品の理解支援手法 7
1.2　依存関係に着目した研究 7
1.2.1　再利用 7
1.2.2　ソフトウェア理解 8
1.2.3　リファクタリング支援 9
1.2.4　部品グラフの特性 9
1.3　既存手法の問題点 10
1.4　本論文の概要 10

第 2 章　Java ソフトウェアの部品グラフにおけるべき乗則の調査 **13**
2.1　導入 13
2.2　関連研究 14
2.3　準備 16
2.3.1　ソフトウェア部品グラフ 16
本研究における定義 16
2.3.2　べき乗則 17
2.3.3　ソフトウェアメトリクス 19
2.3.4　部品集合 19
単一のソフトウェア 20
複数のソフトウェア 20
部分集合 21
2.3.5　解析手法 21
SPARS-J 21
解析の流れ 22
2.4　実験 22

2.4.1 実験内容 22
2.4.2 実験結果 22
実験 1 22
実験 2 26
実験 3 27
実験 4 27
2.4.3 実験結果のまとめ 31
2.5 考察 32
2.5.1 入次数 32
2.5.2 出次数 33
2.5.3 入次数と出次数の非対称性 33
2.5.4 部分集合におけるべき乗則 34
2.6 結論と課題 35

第 3 章 再利用支援のためのソフトウェア部品の依存関係解析手法 37
3.1 導入 37
3.2 ソフトウェア部品検索システムを用いた再利用 38
3.3 依存関係解析手法 40
3.3.1 参照の解析 41
解析手順 42
3.3.2 依存グラフの構築 44
3.3.3 再利用単位の抽出 45
3.3.4 ヒューリスティクスによるフィルタリング 46
3.4 依存関係解析システム 48
3.5 適用実験 49
3.5.1 実験対象 49
3.5.2 手順 50
3.5.3 実験結果 50
E1 に対する結果 52
E2 に対する結果 52
3.6 考察 53
3.6.1 API を用いた整合性の判定 53
3.6.2 スケーラビリティ 53
3.6.3 名前の重複する部品の存在 53
3.6.4 依存関係の理解 54
3.7 関連研究 54
3.7.1 ソフトウェア部品検索システム 54
3.7.2 依存関係の理解支援 55

3.8 結論と課題 55

第 4 章 むすび **59**

4.1 まとめ 59

4.2 今後の研究方針 59

図目次

1.1 SPARS-J 5

2.1 部品グラフの例 17
2.2 べき乗則のプロット 18
2.3 入次数の累積度数分布 – 単一のソフトウェア 24
2.4 出次数の累積度数分布 – 単一のソフトウェア 25
2.5 入次数の累積度数分布 – ソフトウェア集合 26
2.6 出次数の累積度数分布 – ソフトウェア集合 26
2.7 入次数の累積度数分布 – 部分集合 28
2.8 出次数の累積度数分布 – 部分集合 29
2.9 次数とメトリクスの散布図 31

3.1 ソースコード例 39
3.2 解析モデル 41
3.3 部品群の例 41
3.4 参照の解析アルゴリズム 43
3.5 依存グラフ構築の例 46
(a) Intermediate graph 46
(b) Dependency graph 46
3.6 DACARA のシステム構成 47
3.7 DACARA のスクリーンショット 56
(a) Component View 56
(b) Dependency View 56
3.8 再利用単位数の度数分布図 57
3.9 フィルタリングが再利用単位数に与える影響 58

表目次

2.1 実験で用いる部品集合 23
2.2 次数分布の特性量 23
2.3 入次数分布の特性量 – 部分集合 27
2.4 入次数上位 10 部品 30
2.5 出次数上位 10 部品 30
2.6 次数とメトリクスの相関 30

3.1 参照解析結果の例 44
3.2 実験対象パッケージ一覧 51
3.3 Classes と Names の要約 52

第1章　まえがき

近年の情報技術の発展により，生活の基礎となるインフラから身近な電化製品まで，我々の生活は大小様々な計算機やシステムにより支えられている．それらのシステムに要求される機能性や信頼性は年々高まってきており，システムに搭載されるソフトウェアの規模や複雑さもまた，それに応じて大きくなってきている．さらに，素早く市場のニーズに対応した製品を開発する必要性から，ソフトウェア開発に費やせる期間は短くなりつつあり，また，開発コストを低く抑えることへの要求も高い．ソフトウェア工学は，このような，大規模で複雑なソフトウェアを短期間かつ低コストで開発するという，相反する要求を満たすことを目標とした学問分野である．

既存のソフトウェアの一部を，新規に開発するソフトウェアへ流用する，ソフトウェアの再利用はソフトウェア開発のコスト削減のための手段として注目されてきた．再利用される単位は，ソフトウェアの開発過程で生成されるクラスや関数などのソフトウェアの構成要素であり，ソフトウェア部品，もしくは単に部品と呼ばれる．部品を再利用することで，同じ部品を重複して開発することを避けることができ，開発およびテストのコストが削減される．しかし，この自明とも言えるメリットを得ることは難しいと言われている．これは，再利用のための投資が無ければ再利用のメリットを得ることができないが，過剰に投資すると再利用により削減されるコストを上回ってしまうためである．例えば，様々な汎用性の高い部品を用意することで再利用の機会を向上させることができるが，その一部しか再利用されなければ部品の用意に費やしたコストが再利用により削減されるコストを上回ってしまう．そのため，効果的な再利用を実現することを目的とした様々なアプローチが提案されてきた．

ソフトウェア部品検索システム（部品検索システム）は，再利用の候補となる部品を効率的に管理するためのシステムである．部品検索システムは，様々なソフトウェア部品をソフトウェアリポジトリとして蓄積し，開発者から問い合わせを受けると，ソフトウェアリポジトリから適切な部品を検索して開発者に提供する．近年のオープンソースソフトウェア開発の活発化により，大量の部品が WWW を通じて利用可能となっている．オープンソースソフトウェアを対象とした部品検索システムは年々その検索対象を増大させており [28]，部品検索システムを用いた再利用の重要性を示唆している．

本研究では，再利用の対象となる部品集合における，部品間の依存関係に着目する．部品間には，関数の呼び出し関係などの依存関係（利用関係）が存在し，ソフトウェアの持つ機能は，部品同士の相互作用により実現される．部品間の依存関係はグラフとして表現することができ，部品の変更時の影響範囲の特定や，ソフトウェア設計の理解，ソフトウェ

アを変更する際の意志決定などに用いられる．

部品の再利用を行う上で，部品間の依存関係は適切に扱う必要がある．まず，再利用対象の，ある部品 A を開発中のソフトウェアに組み込む時には，その部品が依存する部品集合もまた同時に組み込まなければならない．何らかの理由により，依存する部品が利用できない場合には，利用できない部品に依存しないように部品 A に変更を加えるか，再利用そのものを諦める必要がある．また，部品 A の仕様などが記述された文書が利用できない場合には，部品の組み込み方を調査するため，部品 A に依存する部品，つまり部品 A の利用例となる部品を取得し，理解しなければならない．

このように，ソフトウェア部品間の依存関係はソフトウェア開発を行う上で重要な位置にあり，それを再利用を考慮しつつ適切に理解することが効率的な開発に繋がると言える．以降，ソフトウェア再利用および依存関係解析に関する研究についてそれぞれ述べた上で，既存手法では扱われていない，ソフトウェアをまたがった依存関係について説明し，本研究における調査および提案手法の導入をおこなう．

1.1 ソフトウェア再利用

ソフトウェア再利用とは既存のソフトウェア部品を同一システム内や他のシステムで利用することである [18]．ソフトウェア部品とは，クラスや関数，などのソフトウェアの開発過程で生成される成果物のことを指す．以降，ソフトウェア部品を単に部品と呼ぶ．部品の再利用により，ソフトウェア開発にかかるコストを削減することができると言われている．しかし，ある部品を再利用するために必要なコストが，その再利用により削減できるコストを上回ってはならない．

1.1.1 再利用のアプローチ

再利用に対するアプローチとして，組織的な取り組みにより効率的な再利用を目指すものと，軽量さを重視するものが存在する．

組織的な再利用 Jacobson らは，再利用を成功させるためには，計画的に組織的な投資が必要であると主張し，再利用を前提としたソフトウェア開発プロセスを提案している [30]．また，ソフトウェアプロダクトライン（SPL）という手法が提案されている [25]．SPL は，Jacobson らの方法論と比較して，モデル駆動開発などの比較的新しい概念を取り入れているが，組織的な取り組みや，開発対象の分析などの重要さを主張している点で共通している．

軽量な再利用 組織的に再利用を行うことで効率的な再利用を実現できるが，大規模な開発プロセスの変更が必要であるため，導入は容易ではない．特に，小規模な開発現場ではリスクが大きい．これに対して，Holmes らは組織的な投資を要求しない軽量な再利用を提案している [26]．Holmes らの手法では，事前に再利用可能な部品を開

発せず，ソフトウェア部品検索システムなどのツールの支援を受け，プログラマが必要に応じて部品を取得し，ソフトウェアに組み込む.

本研究で対象とする再利用は，ソフトウェア部品検索システムを用いた再利用に注目するため，後者の軽量な再利用に該当する．また，前者の再利用では，部品単位の再利用ではなく，アーキテクチャなどのソフトウェアの開発の枠組みのレベルでの再利用を視野に入れているため，本研究で注目する依存関係の問題は生じにくいと考えられる．

また，部品の組み込み方という視点から，再利用は 3 種類に分類することができる [59].

ブラックボックス再利用 開発者は，部品の実装や内部仕様を見ず，インターフェースや仕様書のみを見て部品を再利用する．部品に対する変更は行わない．例えば，部品ベンダから購入した Commertial off-the-shelf (COTS) 部品の再利用が挙げられる．

グラスボックス再利用 開発者は，ブラックボックス再利用と同様，部品を変更せずに再利用する．しかし，インターフェースに加え，部品の実装や内部仕様を知ることができ，部品に対してより深く理解することができる．例えば，Java SE [7] など実装の公開された標準 API や，自社で開発し，ソースコードが参照可能である再利用可能な部品の利用が挙げられる．

ホワイトボックス再利用 開発者は，部品の実装を知ることができ，必要に応じて実装やインターフェースを変更して再利用する．例えば，ソースコードの全文検索を行う部品検索システムを用いた軽量な再利用が分類される．

ソフトウェア部品検索システム，特にソースコードを検索するシステムを用いた再利用は，グラスボックス再利用もしくはホワイトボックス再利用に該当する．ソースコードが示されているため，再利用のために処理内容や依存関係を調査することは可能だが，開発者自らが行わなければならないために労力がかかる．

1.1.2 再利用の手順

1.1.3 再利用のプロセス

一般に，再利用のプロセスは以下の 3 段階により構成される [42].

1. 部品の取得: 開発者は，まず再利用可能な部品を取得する．

   部品は，ソフトウェア部品検索システムを用いて検索されることが多い．一般に，検索システムの問い合わせの形式で要求を表現することは難しく，必要な部品を得るためには試行錯誤が必要であると言われている [70].

   ソフトウェア部品検索システムの検索手段としては，一般の文書検索と同様に，キーワード検索や，カテゴリ階層による検索がされることが多い．一方で，部品の仕様を問い合わせとするシステムや，ソフトウェアの開発環境に組み込むことで開発者

が編集しているソースコードの情報を取得し，推薦という形で関連する部品を検索するシステムなど，ソフトウェア部品特有のものも提案および構築されている．具体的なソフトウェア部品検索システムは，1.1.4 節にて紹介する．

2. 部品の検証: 続いて，開発者は取得した部品が要求を満たすかどうか検証する．

   検証により部品の不足などが確認されれば，部品の取得の手順に戻る．この繰り返しは，開発者の要求を満たす全ての部品が揃うまで繰り返される．

   この手順においては，開発者による部品の理解が重要な要素となる．特に，ソフトウェア部品検索システムを用いた再利用の場合には，部品の単位が再利用を想定したものではないことが多く，部品単体では再利用できないことがある．この場合，依存する部品などの必要な部品を調査し，追加で取得しなければならない．また，理解するために必要な文書が得られない場合には，ソースコードなどから開発者自らが調査する労力が必要となる．

   また，この手順では，必要に応じて品質の検証が行われる．再利用対象の部品は，特別な契約が取り交わされている場合を除き，再利用する時の動作を保証していない場合が多い [18]．また，部品の利用条件（ライセンス）が，ソフトウェアへ組み込むのに適切かどうかを検証しなければならない．部品は知的財産物として保護されており，著作権者の許諾無しに利用することはできない．

3. 部品の適応: 最後に，必要に応じて部品を適応させる．

   取得された部品が開発者の要求に完全に合致しない場合，ソフトウェアに組み込む前に，目的に適応するように部品に変更を加える．

   ブラックボックス再利用およびグラスボックス再利用では，再利用する部品を組み合わせるための部品を新たに作成することで，部品を適応させる．ホワイトボックス再利用では，上記の方法に加え，部品そのものを変更することで適応させられる．部品を適応させることにより，より多くの部品が再利用可能となる．しかし，Selby [58] の調査によると，部品に対し大規模な修正を加えて再利用を行う場合，新規に作成する場合と比べ，欠陥を修正，もしくは分離するのにかかる労力が大きい．そのため，要求との差異が小さい部品を取得する必要がある．

以降，部品の取得に利用されるソフトウェア部品検索システム，および，検証および適応を支援する，再利用時の理解支援手法について述べる

1.1.4 ソフトウェア部品検索システム

再利用対象の部品の取得に用いられるソフトウェア部品検索システムおよび類する手法を，検索手段により分類して紹介する．

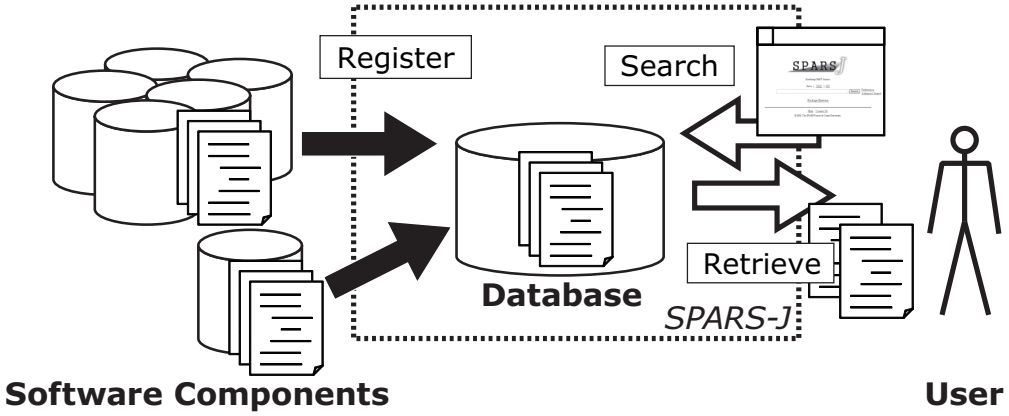


図 1.1: SPARS-J

キーワード検索（全文検索） 最近のオープンソースソフトウェア開発の活発化により，それらのソースコードを対象とした全文検索システムが活発に研究・開発されている．

SPARS-J [12, 72] は Java クラスを部品とし，ソースコードの全文検索を行うシステムである．SPARS-J は，あらかじめ部品をデータベースに登録しておき，開発者の問い合わせに対して，登録された部品の中から適切な部品を検索し，開発者に提供する．（図 1.1）．SPARS-J は，依存や類似といった部品間の関連を解析し，利用者に提供している．また，利用頻度に基づく部品の重要度評価手法であるコンポーネントランク法 [29, 71] により，重要な部品が優先的に検索可能である．Sourcerer [15] は，SPARS-J と同様に Java クラスを部品としてソースコードの全文検索を行うシステムであるが，Fingerprint と呼ばれる部品の特徴による検索が可能である．Google Code Search [5] は複数のプログラミング言語に対応した検索システムであり，大量のソフトウェア部品に対し，正規表現などによる柔軟な検索を行える．Koders [9] も同様に，複数のプログラミング言語に対応した検索システムである．Koders は，オープンソースソフトウェアの開発リポジトリとの連携を特徴としている．また，大須賀ら [52] は，SPARS-J, Google Code Search など，WWW を通じて利用可能なソフトウェア部品検索システムから，横断的に検索する手法を提案している．

カテゴリ検索（ディレクトリ検索） あらかじめ部品を階層化されたカテゴリに分類しておき，開発者は，カテゴリ階層を辿ることで要求を満たす部品を検索する．開発者は，問い合わせを考える必要が無く，提示されたものから選択するだけであるため容易に検索できる．

検索しやすいカテゴリ階層を維持するために，カテゴリ階層の構築や部品の分類は

手作業により行われることが多い．そのため，ComponentSource [1] といった商用の部品（COTS 部品）の検索システムなど，運用のための労力に見合う場面で用いられる．これに対して仁井谷らは，大規模なソフトウェアリポジトリに対してカテゴリ検索を適用することを目標とし，ソースコードに含まれる識別子に基づきカテゴリ階層の構築および部品の分類を自動的に行う手法を提案している [55].

また，カテゴリ検索に関する研究として，Li らは，カテゴリ階層を効率的に辿る手法を提案している [35].

仕様による検索 キーワード検索は WWW 検索などでなじみが深く手軽である一方，詳細な仕様を指定することが難しく，検索結果に無関係な物が含まれやすいという欠点がある．これに対して，必要な部品の入出力の仕様を入力とすることで，要求への適合性の高い部品を検索する手法がいくつか提案されている．

SPARTACAS [46] は必要な部品の仕様の形式的な記述をクエリとして部品を取得する．SPARTACAS では，クエリとして指定した仕様に完全に一致する部品に加え，部分的に一致する部品の組み合わせを検索する．鷲崎らは，クエリとして与えられたプロトタイプに類似した部品を検索する手法を提案している [67]．部品間の類似度は，構造面・振舞面・粒度面から評価され，検索結果として類似度で順位付けされた部品集合を得ることができる．Park の手法 [53] では，入出力の振る舞いを入力として部品を検索する．

必要な部品の仕様を直接入力できることから精度の高い検索が可能であるが，逆に，要求が「画像を表示する部品」の様に曖昧であり仕様を指定しにくい状況では検索することができない．そのため，キーワード検索に基づく手法と組みあわせて利用することが必要であると考えられる．

開発環境に統合した検索 独立したシステムではなく，開発環境に組み込むことで，開発者がキーワードとして表現することが難しい問い合わせを，自動的にエディタ上の情報などを利用して構築し，検索を行うシステムもいくつか提案されている．

Strathcona [27] は 統合開発環境 Eclipse のプラグインとして実装されたシステムであり，開発者が編集している Java クラスの継承関係や呼び出し関係に基づいて検索を行う．Ye らは，開発者が Emacs 上で編集しているソースコードからドキュメントコメントやメソッドに利用される文字列を取得することで，類似した部品を自動的に検索するシステム CodeBroker を開発している [70]．島田らは，開発者の編集しているソースコードに含まれる識別子やコメントに基づき，ソフトウェアリポジトリから部品を検索する Eclipse プラグイン A-SCORE を開発している [61]．Ye らの手法および島田らの手法では，部品再利用の障壁として知られている,「再利用可能な部品が存在しても，開発者が検索しようとしなければ利用されない」という問題を解決するために，部品の検索は開発者の明示的な指示なしに行われる．

[1] www.componentsource.com

1.1.5　部品の理解支援手法

取得した部品をソフトウェアへ組み込むために必要な理解を助ける手法について述べる．

依存する部品の取得や部品の適応を直接支援する手法として，Holmes らの手法 [27] が挙げられる．Holmes らの手法では，再利用対象の部品をツールに入力することで，その部品が依存する部品集合が可視化される．開発者は，示された部品それぞれに対し，取得して同時に再利用するか，依存しないようにもとの部品を書き換えるかを選択することで，依存関係の問題を解決することができる．しかし，この手法では再利用対象の部品が所属するソフトウェアを同時に入力として与える必要があるため，そのソフトウェアを入手できない場合や，ソフトウェアをまたがる依存関係が存在する場合には適用できない．

また，部品の利用方法を示すことで理解を支援する手法がいくつか提案されている．CodeWeb [41] は，ソフトウェアに含まれる部品の依存関係に対してアイテムセットマイニングと呼ばれる手法を適用することで,「クラス A が利用される時にはクラス B とクラス C もまた利用される」といった相関ルールを抽出する．Prospector [37] は，メソッドの引数や返値に基づき構築したグラフを探索することで，あるクラスのオブジェクトから別のクラスのオブジェクトを得るためのメソッド呼び出しの系列を出力する．PARSEWeb [63] も同様の系列を得ることを目的としたシステムであるが，単一のソフトウェアに対して解析を行う Prospector（および CodeWeb）と異なり，Google code search を用いて得たソースコードを用いることで，検索可能な任意の部品について利用方法を得ることができる．

1.2　依存関係に着目した研究

部品間の依存関係の分析は，ソフトウェア工学の様々な手法で行われている．分析には，部品を頂点，部品間の依存関係を辺とした，部品グラフ，もしくはモジュール依存グラフ（MDG）と呼ばれるグラフが用いられる．部品グラフは，ソースコードやバイナリを解析して得られる静的な部品グラフと，ソフトウェアの実行履歴を取得および解析して得られる動的な部品グラフに分類される．静的な部品グラフからは設計やアーキテクチャなどのソフトウェアの静的な側面を，動的な部品グラフは部品同士の協調動作などのソフトウェアの動的な側面を得ることができる．本研究では，ソフトウェアリポジトリから得られる静的な部品グラフに着目する．

1.2.1　再利用

まず，本研究において注目している再利用と関連した研究について述べる．

部品の再利用性評価手法　コンポーネントランク法 [29, 71] は，SPARS-J において，リポジトリに蓄積された部品の再利用性を評価するために用いられている．コンポーネントランク法は,「重要な部品とは，多くの重要な部品から利用されている部品である」という再

帰的な概念に基づく部品の評価手法であり，部品グラフ上の頂点に重みを与え，利用関係を表す有向辺を通じて重みを伝播させることにより，部品の評価値を計算する．また，再利用のためにコピーされた部品の存在を考慮し，類似した部品をグループ化して計算している．

Sourcerer [15] においても，部品の順位付けにコンポーネントランク法が用いられている．

その他の再利用支援手法　前節で紹介した，Strathcona [27]，CodeWeb [41] も，依存関係を用いてソフトウェア部品の再利用を支援するシステムである．Strathcona は，開発者が編集しているソースコードと，依存関係を含む構造上の特徴が類似した部品を検索する．また，CodeWeb は，クラス間の依存関係に対してデータマイニング手法を適用している．

1.2.2　ソフトウェア理解

開発者がソフトウェアに何らかの変更を加える際，まずはその変更対象のソフトウェアを理解することが必要である． Sillito の調査 [62] では，ソフトウェアの構成要素間の関連を発見し，その関連を理解することは，開発者にとって困難な作業であることが示されている．

デザインパターン検出　デザインパターンとは，典型的な問題に対応するためによく知られた解決策のことであり，ソフトウェアの中では定型的な設計やコード片として現れる．Gamma らによる 23 種類のパターンがもっとも有名であるが，他にも，マルチスレッドやエンタープライズシステムなど，特定の分野で利用可能なパターンも提案されている．また，特定のソフトウェア中に繰り返し現れるものもパターンと呼ぶことがある．ソフトウェア中で，デザインパターンが用いられている箇所を自動的に特定することで，開発者によるソフトウェア設計の理解を支援する．

あらかじめ定義したデザインパターンをソフトウェアの中から抽出する手法として，Antoniol らの手法 [14] および Niere らの手法 [51] がある．また，Sartipi らは，構成要素間に現れるパターンではなく，構成要素のグループ間に現れるパターンを記述し，抽出する手法を提案している [57]．これらの手法は抽出対象のパターンをあらかじめ定義しているが，これに対して，Tonella らは，あらかじめパターンを与えず，共通した関連をもつ部分をパターンとして抽出する手法を提案している [64]．

ソフトウェアクラスタリング　また，ソフトウェアをサブシステムに分割することで理解を支援する，ソフトウェアクラスタリングに関する研究が存在する．大きなシステムをより粒度の細かいサブシステムに分割することで，開発者がソフトウェアを理解する時に注目する範囲を限定することができ，理解の作業を効率化することができる．

Bunch [43] は，入力された部品グラフを分割することでクラスタリングを行うシステムである．Bunch はグラフの分割を探索問題として解き，クラスタ間の結合がなるべく少な

く，クラスタ内の頂点間の結合がなるべく多くなるようなグラフの分割を求める．

1.2.3　リファクタリング支援

リファクタリングとは，ソフトウェアの振る舞いを変更せず，ソフトウェアの保守性の向上を目的として実装に変更を加える作業のことを指す [24, 40]．部品グラフを用いて，リファクタリングすべき箇所を自動的に抽出する手法がいくつか提案されている．

アスペクトマイニング　横断的関心事とは，ソフトウェア中に横断的に現れる，モジュール化されていない機能（実装）のことであり，アスペクトなどへのリファクタリング候補となる [31]．横断的関心事を自動的に抽出する手法はアスペクトマイニング手法と呼ばれる．

Marin は，ソフトウェアに含まれるメソッドの呼び出し関係を解析することで，利用される頻度の高いメソッド，すなわち様々な箇所で利用されるメソッドを横断的関心事として抽出する手法を提案している [38]．

設計評価　Chatzigeorgiou らは，WWW 上のページ間の関連を分析する HITS アルゴリズム [33] を応用し，オブジェクト指向ソフトウェアの設計を評価する手法を提案している [20]．HITS アルゴリズムを用いることで，責任の集中しているクラスを特定し，その度合いを定量化している．オブジェクト指向ソフトウェアにおいて，責任の集中しているクラスは “God class” と呼ばれ，問題を起こしやすい設計であると言われている [36]．

Sarkar らは，非オブジェクト指向なプログラムについて，モジュール化の品質を計測するメトリクスを提案している [56]．提案されているメトリクスは，モジュール間の依存関係やモジュールの大きさに基づき計測される．

1.2.4　部品グラフの特性

ここまでに紹介した研究は，それぞれの提案する手法に依存関係を用いた研究であったが，部品間の依存関係そのものに着目した調査も行われている．

Myers は，部品グラフを複雑ネットワークとしてとらえ，構造の特徴などに関する調査を行った [47]．結果として，部品グラフがスケールフリーネットワークの特徴をもつことを示している．スケールフリーネットワークとは，次数分布にべき乗則が成り立つグラフのことであり，耐障害性などの性質と関連があることが知られている [16]．Concas らも同様に部品グラフを調査して次数分布にべき乗則が成立することを示している他，様々なメトリクスを調査して関連について考察している [22]．これらは静的な部品グラフに対する調査であるが，一方，Potanin らは，動的に解析した部品グラフに対して調査を行い，スケールフリーネットワークの特徴を持つことを示している [54]．

1.3 既存手法の問題点

本節では，ここまで部品の再利用に関する研究および部品間の依存関係に着目した研究について述べてきた．本研究では，以下の 2 点に着目する．

1. 複数のソフトウェアに基づく部品集合の部品グラフは調査されていない．

   ソフトウェア部品検索システムのデータベースであるソフトウェアリポジトリには，様々なソフトウェアから取得した様々な部品が含まれる．これらの中には，ライブラリとして多くのソフトウェアに依存されるソフトウェアもあれば，逆にライブラリを利用することで構築されたアプリケーションソフトウェアも存在する．このような，複数のソフトウェアを含み，さらにソフトウェアをまたがった依存関係が存在する部品集合に基づく部品グラフはこれまで調査されていない．

   ソフトウェアリポジトリにおける部品間の依存関係は，再利用支援などで用いられており，その性質を知ることは，より効率的・効果的な支援手法の構築に繋がると考えられる．また，様々な部品を解析することで，再利用に限らないソフトウェア一般に対する知見を得ることも期待できる．

2. 既存のソフトウェア部品検索システムは，検索された部品の再利用に必要な依存関係解析は行っていない

   ある部品を再利用するためには，開発者はその部品が依存する部品集合を調査して理解し，取得しなければならない．このとき，類似した部品が複数存在する場合にはそれらの中から適切なものを選択する必要がある．しかし，既存の部品検索システムは，そもそも依存関係を利用できないか，依存関係を利用できても，類似した部品を考慮していないため，開発者（利用者）が手作業で調査しなければならない．

   既存のソフトウェア部品検索システムは，それぞれ特徴的な検索手段により，部品を効率的に取得することを目指している．取得した部品の依存関係の理解を支援することができれば，ソフトウェア部品検索システムを用いた再利用がより容易になると考えられる．

1.4 本論文の概要

本研究では，ソフトウェアリポジトリに含まれる大量の部品集合を解析し，部品グラフに着目した調査，および，部品の再利用支援を目的とした手法の提案を行う．本論文では，これまでに行った調査および提案手法について述べる．

1. ソフトウェア部品グラフの次数分布におけるべき乗則の調査

   べき乗則とは，直感的には，ごく一部の要素は非常に多くの頻度で出現するのに対し，大多数の要素はほとんど出現しないような分布のことである．グラフを特徴付

ける要素として次数分布が挙げられる．次数がべき乗則に従うグラフは，一般のグラフには見られない特徴的な性質を持つことから，物理学や社会学，生物学，情報科学など様々な分野で注目されている．例えば，新たな辺が追加される際には既に大きな次数をもつ頂点ほど接続されやすい性質や，無作為に頂点を取り除いてもある程度は構造が維持されるが，ある点を境に急激に崩壊する性質などを持つ [13, 16, 49]．これまで，部品グラフ，特に大規模な部品集合に基づく部品グラフに対する次数分布の観点からの調査はあまり行われていなかった．

そこで，本研究では，部品グラフを，頂点の次数分布にべき乗則が成り立つかどうかという観点から調査する．部品グラフの次数にべき乗則が成り立つことが明らかになれば，これらのグラフ構造の成立や安定性に関わる性質に基づき，ソフトウェア設計などの部品グラフに反映されるソフトウェアの要素に対して新しい観点からの分析が可能であると考えられる．部品グラフは，単一のソフトウェアに含まれる部品集合の他に，互いに関連をもつ複数のソフトウェアに含まれる部品集合や，その部分集合など，ソフトウェアリポジトリから得ることができる部品集合を用いて構築する．

2. 再利用支援のためのソフトウェア部品の依存関係解析手法

   ソフトウェア部品検索システムを用いることで，大規模なソフトウェアリポジトリから容易に部品を検索できる一方，検索により取得した部品を実際にソフトウェアに組み込むためには手作業で依存関係の調査を行う必要がある．このとき，依存する部品の候補として，類似した API を持つ部品が存在する場合には，それらの中から適切に選択する必要がある．

   そこで，本研究では，再利用対象の部品が依存する部品集合の候補を，部品同士の関連を考慮しつつ，それぞれの API に基づいて抽出する．開発者は，既に利用している部品などを考慮して候補を選択することで，対象の部品を再利用するために必要な部品集合を取得できる．

   また，実際のオープンソースソフトウェア集合を用いた実験を行い，提案手法の有効性やスケーラビリティを評価する．

以下，第 2 節では　ソフトウェア部品グラフの次数分布におけるべき乗則の調査 について述べ，第 3 章で 再利用支援のためのソフトウェア部品の依存関係解析手法 について述べる．最後に第 4 章で本論文の研究についてまとめ，今後の研究方針を述べる．

第2章　Javaソフトウェアの部品グラフにおけるべき乗則の調査

2.1　導入

近年，短期間で大規模かつ高品質なソフトウェアを開発するための技術に対する需要が高まっている．その中でも，ソフトウェアの解析に基づく手法はソフトウェアの評価や理解支援，再利用支援など様々な分野で見られる．部品グラフは第 1 章で述べた様に様々な手法に用いられており，重要な概念である．

様々な分野において，グラフは研究対象の表現形式として用いられ，その性質が調査・研究されている．次数がべき乗則に従うグラフは，一般のグラフには見られない特徴的な性質を持つことから，物理学や社会学，生物学，情報科学など様々な分野で注目されている．例えば，新たな辺が追加される際には既に大きな次数をもつ頂点ほど接続されやすい性質や，無作為に頂点を取り除いてもある程度は構造が維持されるが，ある点を境に急激に崩壊する性質などを持つ [13, 16, 49]．これまで，部品グラフの性質に対する調査，特に次数分布の観点からの調査は少なかった．部品グラフの次数にべき乗則が成り立つことが明らかになれば，これらのグラフ構造の成立や安定性に関わる性質に基づき，ソフトウェア設計などの部品グラフに反映されるソフトウェアの要素に対して新しい観点からの分析が可能であると考えられる．

本研究では部品グラフの次数に関し，以下の 4 つの問題を調査する．

問題 1 単一のソフトウェアシステムに関して，様々な静的な利用関係を用いた部品グラフの次数分布はべき乗則に従うかどうか．

部品グラフの次数分布に現れるべき乗則に着目した既存研究はいくつか存在する [17, 22, 47, 65, 69] が，本研究と同一の定義の部品グラフに基づく研究は存在しないため，異なる結果が得られる可能性がある．特に，本研究では部品間の利用関係を詳細に解析しており，継承関係などの「代表的な」利用関係のみを用いている文献における結果とは差異が出ると考えられる．また，本研究では入次数および出次数を個別に調査する．

問題 2 複数のソフトウェアシステムからの部品集合を用いて構築した部品グラフの次数分布はべき乗則に従うかどうか．

本研究では，単一のソフトウェアだけでなく，含まれるソフトウェア同士が互いに利用関係を持つような，ソフトウェア集合に対する調査を行う．複数のソフトウェ

アから構築した部品グラフに対する調査を行った既存研究は存在しない．

問題 3 部品グラフの部分グラフの次数分布はべき乗則に従うかどうか．

ソフトウェア集合から構築した部品グラフの部分グラフの次数分布がべき乗則に従うかどうかを調査する．このような部分グラフは，依存関係解析や部品検索において用いられることがある．

問題 4 部品そのものの性質と次数との関連．

次数は部品を解析して得られる結果であり，何らかの部品の性質を反映した値であると考えられる．その関連を知ることは，以上の問題について考察するときの有効な手がかりとなる．

以降，2.2 節で関連研究について述べ，2.3 節でソフトウェア部品グラフなどの諸定義および解析手法などについて述べる．2.4 節で実験内容およびその結果について述べ，2.5 節で得られた結果に関して考察する．最後に 2.6 節で本章のまとめと今後の課題について述べる．

2.2 関連研究

Valverde らは，Java の基本ライブラリである JDK [7] や Java プログラムのクラス図を，クラスおよびインターフェースを部品，汎化や依存といった関連辺を利用関係とする部品グラフとみなしたとき，次数に関してべき乗則が成り立つことを示している [65]．また，グラフ上の辺が頂点数に対して少数であるにも関わらず頂点間の最短距離が小さいスモールワールド性 [68] を持つことも示している．さらに，これらの性質は再利用性や理解容易性などを考慮し，最適なソフトウェア設計を行った結果であると考察している．

Myers は C++プログラムのクラス図を部品グラフとみなしたとき，次数がべき乗則に従うこと，グラフがスモールワールド性をもつこと，および階層的な構造をもつことを示している [47]．Myers の実験では，次数は入力と出力に分けて調査され，入次数と出次数はいずれもべき乗則に従う点では共通しているが，それぞれ異なる性質を持つことが示されている．また，部品グラフと同様の性質をもつグラフの生成モデルを提案している．

Concas らは，Java プログラムおよび Smalltalk プログラムに対し，部品グラフの次数分布と CK メトリクス [21] を調査している [22]．結果として，べき乗則および対数正規分布が見られたことが報告されている．

Wheeldon らは，Java プログラムの部品グラフを，継承関係や変数宣言などの利用関係それぞれに対して構築して次数分布を調査している [69]．結果として，いくつかの種類の利用関係に対してはべき乗則が観測されている．

Baxter らは，Java プログラムのクラス間の関連を調査した結果として，べき乗則や対数正規分布が現れたことを示している [17]．

木下らは Java プログラムのクラス間の利用関係に対して，そのグラフ構造を測定している [73]．いくつかの Java プログラムに対し，入次数，出次数を個別に調査し，出次数は指数分布に近いことを示している．また，開発履歴情報を用いて入次数および出次数の最大値の変遷を調査し，出次数はある程度開発が進んだ時点で頭打ちになることを報告している．

以上の研究では，部品のソースコードを静的解析することにより得た部品間の利用関係から部品グラフを構築し，次数などを調査しており，本研究でのアプローチと一致している．しかし，以下の点で本研究の調査とは異なる．

- 本研究では，ソフトウェア集合に基づく部品グラフや，そのサブグラフに対する調査を行っている．

  既存研究は単一のソフトウェアに基づく部品グラフの調査のみであり，このような部品グラフに対する調査は行われていない．このような部品グラフは，ソフトウェア部品検索システム [29] のようないくつかのソフトウェア工学の手法にて利用されている．

  構築する部品グラフの詳細に付いては，2.3.1 節および 2.3.4 節で述べる．

- 上記と関連して，本研究では部品グラフの大局的な性質を調査することに焦点を当てている．

  本研究では，部品グラフの次数分布の他に，次数とソフトウェアメトリクスの相関について調査し（2.4.1 節），ソフトウェア開発の実践との関連についての考察を行っている（2.5 節）．これに対して，既存研究においては，主に次数分布にべき乗則が現れる背景に関する調査および考察に主眼が置かれている．

- 本研究における部品グラフの定義は，既存研究のいずれとも異なる．

  本研究の定義は，クラスを頂点とし，クラス間の利用関係を辺とする，という点では既存研究と共通しているが，より正確な部品間の依存関係を表現した部品グラフを構築するために，既存研究とは異なる利用関係の定義や，解析方法を用いている．そのため，得られる結果にも異なる傾向が見られると考えられる．まず，いくつかの既存研究では，辺として用いる利用関係は，いくつかの「代表的な」利用関係のみを用い，ローカル変数の宣言やメソッド呼び出しなど，利用関係の多数を占めることが多い種類を用いていない．また，いくつかの既存研究では，クラス間の利用関係を簡単な字句解析と名前のマッチングのみを用いて解析しているが，呼び出しに単純名が用いられている場合などに利用関係を取りこぼす可能性がある．

  一方，本研究では簡単な意味解析に基づき，静的に解析可能な全ての利用関係を用いて部品グラフを構築している．部品グラフの定義は 2.3.1 節，解析手法については 2.3.5 にて，それぞれ説明する．

以上で述べた研究は静的解析により得た部品グラフに基づいている．これに対し，Potaninらは動作するJavaプログラムを解析し，オブジェクト間のメッセージのやりとりの関係がべき乗則に従うことを示している [54].

2.3 準備

2.3.1 ソフトウェア部品グラフ

一般に，ソフトウェア部品は広義にはモジュールや関数，クラスなどのソフトウェアの構成要素のことを，狭義には再利用を目的として設計されたソフトウェアの実体のことを指す．本研究では広義のソフトウェア部品を用いる．以降，ソフトウェア部品を単に部品と呼ぶ.

ソフトウェアは構成要素の部品間で相互に属性や振る舞いを利用し合うことで一つの機能を提供する．いま，ある部品がある部品を利用するとき，この部品間に利用関係が存在すると言う.

部品を頂点，利用関係を有向辺としたグラフを部品グラフと呼ぶ．部品グラフの例を図2.1に示す．図2.1では，左側のソースコード片に定義されたクラス集合に基づいて右側の部品グラフが構築されている．ソースコード上では，`Warehouse`, `Shelf`, `Liquor` の3つのクラスが定義され，それぞれ頂点（部品）となる．また，`Warehouse` は `Liqur` 型の変数を宣言すると共にインスタンスを生成することから，頂点 `Warehouse` から頂点 `Liquor` への有向辺が作成される．同様に，メソッド呼び出しに基づいて `Warehouse` から `Shelf`，メソッド引数の型宣言に基づいて `Shelf` から `Liquor` への有向辺がそれぞれ作成される.

入次数は，ある頂点の入力辺の総数のことを指し，出次数は，ある頂点からの出力辺の総数のことを指す．例えば，図2.1の頂点 `Warehouse` の入次数および出次数はそれぞれ0および2であり，頂点 `Liquor` の入次数および出次数はそれぞれ2および0である.

本研究における定義

本研究では，以下に定義する部品を頂点，部品間の利用関係を有向辺として部品グラフを構成する.

部品 Javaプログラムのソースコードを解析して得られた，Java クラスおよびインターフェース．バイナリのライブラリ（“jar” ファイル）に含まれるクラスやインターフェースは部品として扱わない.

利用関係 以下の6通りの，Javaソースコードを静的に解析して得られる関係．ある部品間に，以下のいずれかの関係が存在する時，それらの間に利用関係が存在するとする.

- 部品がクラスもしくはインターフェースを継承

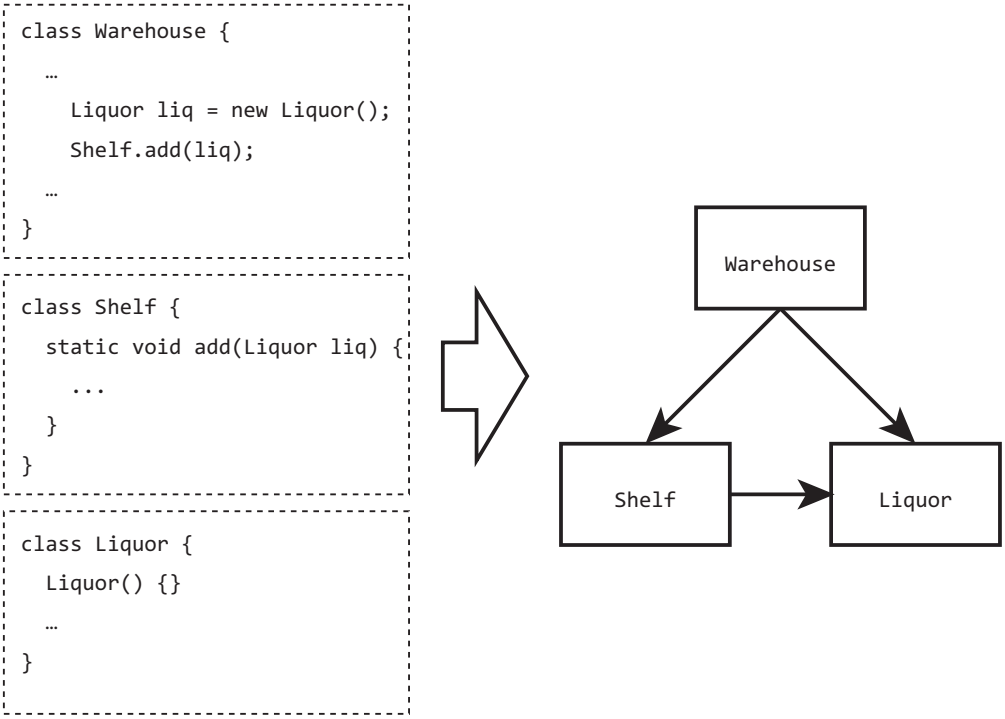


図 2.1: 部品グラフの例

- 部品がインターフェースを実装
- 部品がクラスまたはインターフェースを変数として宣言．フィールドの型, メソッドの返値および引数の型として宣言した場合を含む.
- 部品がクラスのインスタンスを生成
- 部品がクラスもしくはインターフェースのメソッドを呼出．呼び出されたメソッドが継承されたものである場合は，メソッド宣言の存在するクラスもしくはインターフェースに対しての利用関係となる．
- 部品がクラスもしくはインターフェースのフィールドを参照．参照されたフィールドが継承されたものである場合は，フィールド宣言の存在するクラスもしくはインターフェースに対しての利用関係となる．

2.3.2 べき乗則

本研究では，部品グラフの特徴として，入次数および出次数の分布がそれぞれべき乗則に従うかどうかに注目する．べき乗則は，Pareto の法則，Zipf の法則とも呼ばれ [50]，論文の共著関係 [49]，WWW 上のページのリンク関係 [13] など様々な分野において見られる．

ある分布がべき乗則に従うとき，要素 X が値 x をもつ確率 P は，x の $-\alpha$ 乗に比例する．これを確率分布関数で表すと式 (2.1) となる．

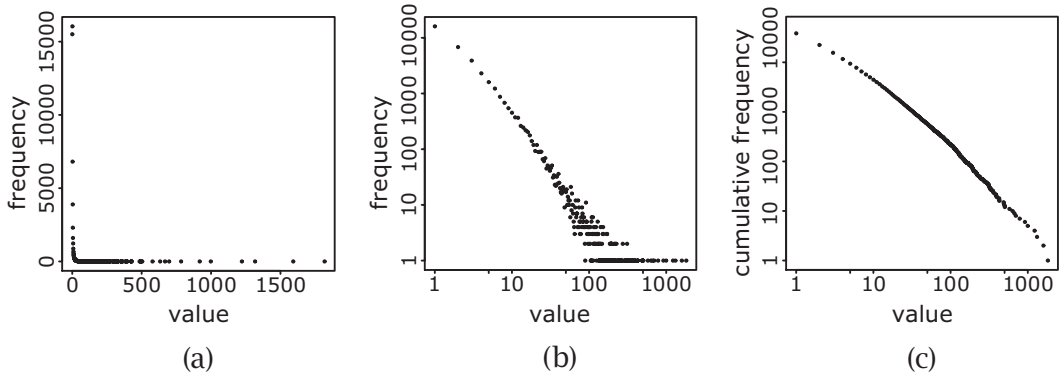


図 2.2: べき乗則のプロット

$$p(x) \sim x^{-\alpha} \tag{2.1}$$

この分布をプロットするときには一般に両対数軸が用いられる．通常の軸を用いてプロットすると図 2.2（a）の様になり，特徴を観測することが困難となるためである．

式 (2.1) の両辺の対数を取ると式 (2.2) のようになり，ある分布がべき乗則に従うときは，両対数軸でプロットしたときに点が傾き $-\alpha$ の直線上に並ぶことがわかる（図 2.2（b））

$$\log p(x) \sim -\alpha \log x \tag{2.2}$$

しかし，実際のデータの度数をプロットすると，値の大きな部分にばらつきが生じるため，大きな方から度数を累積させた値（累積度数）をプロットする．式 (2.1) を累積分布関数で表すと式 (2.3) のようになり，要素 X が x 以上の値をもつ確率は，x の $-(\alpha-1)$ 乗に比例することが分かる[1]．つまり，この場合も両対数軸を用いてプロットすると値が直線上に並ぶ（図 2.2（c））．

$$P(x) \sim x^{-(\alpha-1)} \tag{2.3}$$

式 (2.3) の（すなわち，式 (2.1)，(2.2) の）α の値は，累積度数を両対数軸上にプロットした結果を直線へ線形回帰したときの直線の傾きから求められる．また，分布がべき乗則に従う度合いとして，回帰時の自由度調整済み寄与率 R^{*2} を用いる [34]． R^{*2} は，回帰モデルがデータを説明できている割合を示す値であり，0 から 1 の値をとる．

[1]累積分布関数の導出に関しては文献 [50] 参照.

2.3.3 ソフトウェアメトリクス

本研究では，個々の部品に定義される特性値のことをソフトウェアメトリクス（メトリクス）と呼ぶ[2].

入次数および出次数が部品の表面的な特徴と関連があるかどうかを調査するため，対象の部品のソースコードのみを用いて計測可能なメトリクス，特にソフトウェア部品の品質と関連をもつメトリクスとの相関を求める．入次数は利用される頻度であることから複雑度などの品質と関連をもち，出次数は利用する部品数であることから規模と関連をもつことが考えられるためである．本研究では，メトリクスとして一般に規模の計測に用いられるソースコード行数 (LOC)，およびオブジェクト指向ソフトウェアの品質測定に用いられる CK メトリクス [21] のうち，対象の部品のソースコードのみを用いて計測可能な Weighted Methods per Class (WMC) および Lack of Cohesion of Methods (LCOM) を用いる．以下にそれぞれの詳細を示す.

LOC コメント行を除くソースコードの行数．部品の規模を表す.

WMC クラスに定義されたメソッドの重み付き和．この値が大きい程，複雑なクラスである．メソッドの重みとしては以下の 2 種類を用い，それぞれ WMC1，WMC2 とする:

- **WMC1**: すべて 1．この値はクラスに定義されるメソッド数そのものであり，実装された機能の大きさを表すと考えられる.
- **WMC2**: サイクロマチック数 [60]．分岐や繰り返しに基づき計測され，実装の複雑さを表す.

LCOM クラスの凝集度．一般に LCOM は複数の定義が存在するが，本研究ではクラス中のメソッドが属性を利用する割合を表す LCOM5 を用いる [19]．LCOM5 は，あるクラス中の全てのメソッドがそれぞれ全ての属性を利用するとき 0，全てのメソッドがそれぞれただ 1 個の属性のみを利用するとき 1 となる値である．この値が小さいほど凝集度が高い，すなわち実装された機能のまとまりが強い．なお，LCOM5 は実装を持つメソッドが 1 個以上であるクラスにのみ定義されるため，LCOM5 と次数との相関は LCOM5 が定義可能なクラスのみを用いて求める.

2.3.4 部品集合

実験では，単一のソフトウェアの部品集合として 4 種類，複数のソフトウェアの集合の部品集合として 2 種類，部分集合の部品集合として 8 種類を用いる．それぞれの部品数（頂点数），利用関係数（辺数），総行数を表 2.1 に，概要を以下に示す.

[2] 一般には，メトリクスはソフトウェアの開発活動中に計測された値から得られる特性値を意味する.

単一のソフトウェア

単一のソフトウェアの部品集合は，それぞれ，ソフトウェアの配布パッケージに含まれるソースコードを解析して得られる部品集合である．配布パッケージ中のバイナリファイルは，仮にソースコードがそれらに依存していたとしても，部品集合には含めない．以下の 4 種類の部品集合は，それぞれ，ドメインや規模が異なるものを準備した．

ANT Java のビルドツールである Apache Ant 1.6.2 [1] の配布パッケージに基づく部品集合．実験に用いる部品集合の中で最小規模である．

JBOSS Java のアプリケーションサーバーである JBoss Application Server 3.2.5 [8] の配布パッケージに基づく部品集合．

JDK Java の基本ライブラリである JDK1.4 に含まれる部品集合．基礎的な部品を中心とした利用関係が形成されていると考えられる．関連研究においても調査されている集合であり，結果を比較できる．

ECLIPSE Java の統合開発環境である Eclipse 3.0.1 [4] の配布パッケージに基づく部品集合．

複数のソフトウェア

複数のソフトウェアの部品集合は，複数の配布パッケージやソースコード管理システムから取得した部品で構成され，ソフトウェアをまたがる利用関係が存在する．

ASF Apache Software Foundation [2] の CVS リポジトリから 2005 年 6 月に取得した部品集合から構成される部品集合．それぞれのソフトウェアは比較的独立して開発されているが，共通ライブラリとしてのソフトウェアも開発されており，それらを中心とした利用関係が形成されていると考えられる．この部品集合に JDK は含まれない．

SPARS_DB 2005 年 6 月時点での，部品検索システム SPARS-J [12] の部品データベース（ソフトウェアリポジトリ）に含まれる部品集合．この部品集合は，JDK に加え，様々なオープンソースソフトウェアのソースコード管理システムから取得した部品を含む．例えば，Apache Software Foundation や SourceForge.net [11]，Eclipse，NetBeans [10] が含まれる．つまり，SPARS_DB は ASF を包含する．

JDK の様なライブラリ部品から，Eclipse のようなアプリケーション，小規模なサンプルコードまで様々な種類の部品が含まれる．ASF と比較して互いに関連性の低い多数のソフトウェアの集合である．含まれるソフトウェアの共通点はほとんど JDK を利用していることのみであり，JDK を中心とした利用関係が形成されると考えられる．

部分集合

部分集合 SPARS_DB より，以下の 3 通りの方針に基づき抽出した部品集合を用いる．それぞれ明示的にはソフトウェア単位やモジュール単位とならない．

無作為 無作為に取り出した部品集合

利用関係 ある部品を利用する部品集合．基準となる部品は，部品数が約 10,000, 約 1,000, 約 100 となるようなものからそれぞれ無作為に選ぶ．

単語 ある単語をソースコード中に含む部品集合．基準となる単語は，部品数が約 10,000, 約 1,000, 約 100 となるようなものからそれぞれ無作為に選ぶ．

以上の方針に基づき，以下の 8 通りの部品集合を調査に用いる．

RND10K，RND1K 方針:無作為による部品集合．

REL10K 方針:利用関係による部品集合．基準となる部品は `java.util.HashMap`.

REL1K 方針:利用関係による部品集合．基準となる部品は `java.io.OutputStreamWriter`.

REL1H 方針:利用関係による部品集合．基準となる部品は `java.awt.GraphicsEnvironment`.

KWD10K 方針:単語による部品集合．単語は `getstring`.

KWD1K 方針:単語による部品集合．単語は `labels`.

KWD1H 方針:単語による部品集合．単語は `getsummary`.

2.3.5 解析手法

SPARS-J

Java ソースコードの解析，および利用関係の解析にはソフトウェア部品検索システムである SPARS-J の解析部を用いる [29]．SPARS-J は Java ソースコードを解析し，クラスおよびインターフェースを部品として登録する．また，部品間の静的な利用関係を解析する．SPARS-J で解析される利用関係は 2.3.1 節で述べた本研究での定義と一致している．

解析の流れ

1. Java ソースコードの解析: SPARS-J 登録部が Java ソースコードを解析してデータベースに登録する．同時に利用関係の解析およびメトリクスの計算を行う．
2. 解析内容の出力: SPARS-J のデータベースから解析結果である部品情報，すなわちメトリクスおよび部品間の利用関係を出力する．
3. 出力の分析: 出力された情報に対して度数分布のプロットや相関の計算などを行う．

2.4 実験

2.4.1 実験内容

実験 1 問題 1 に対する実験として，単一のソフトウェアの部品集合を対象とし，本研究における定義の部品グラフに関して入次数および出次数がべき乗則に従うかどうかを確認する．入次数および出次数がべき乗則に従うかどうかを確認するために，両対数軸上にそれぞれの分布をプロットし，回帰直線を求める．

実験 2 問題 2 に対する実験として，複数のソフトウェアの部品グラフを対象として，実験 1 と同様の実験を行う．

実験 3 問題 3 に対する実験として，部分グラフを対象として，実験 1 と同様の実験を行う．

実験 4 問題 3 に対する実験として，入次数および出次数と，個々の部品との関連を調査する．部品集合 SPARS_DB に関して，入次数および出次数とメトリクスとの相関係数を求める．相関係数としては，分布の異なる値の集合同士の相関を求めることができるスピアマンの順位相関係数を用いる．入次数および出次数の関連する組み合わせについては，得られた相関係数が一般に弱い相関と解釈される値 (0.2 ∼ 0.4) の場合，散布図によりその関連を調査する．

また，本研究で用いたメトリクスで表現されない性質，例えば部品の役割などとの関連を調査するため，入次数および出次数の上位 10 部品の内容を確認する．

2.4.2 実験結果

実験 1

入次数の累積度数分布を図 2.3 に，出次数の累積度数分布を図 2.4 に示す．それぞれの回帰直線から求めた，式 (2.1) での α の値および R^{*2} の値を表 2.2 に示す．

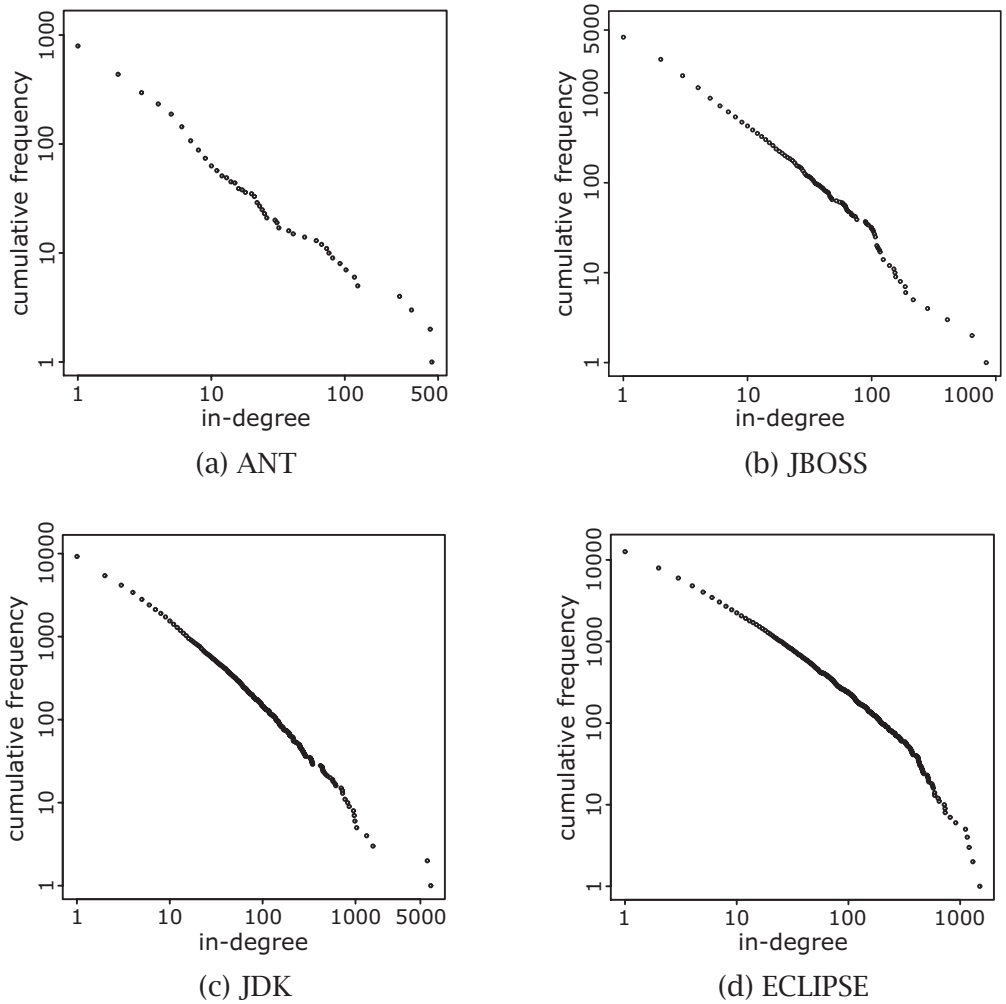
入次数 プロットがほぼ直線であり，回帰した結果の R^{*2} も非常に高い値であることから入次数はべき乗則に従うことが分かる．

表 2.1: 実験で用いる部品集合

	頂点数	辺数	LOC
ANT	1,260	4,995	95K
JBOSS	5,752	23,636	424K
JDK	11,556	107,198	1.1M
ECLIPSE	13,941	140,678	1.3M
ASF	59,486	303,755	4.5M
SPARS_DB	180,637	1,808,982	14M
RND10K	10,000	6,184	780K
RND1K	1,000	52	80K
REL10K	9,286	17,201	2.1M
REL1K	972	1,218	250K
REL1H	163	1,086	76M
KWD10K	8,938	24,317	1.6M
KWD1K	1,002	1,564	290K
KWD1H	129	124	28K

表 2.2: 次数分布の特性量

		α			R^{*2}
入次数	ANT	2.01	$\pm$	0.0195	0.984
	JBOSS	2.27	$\pm$	0.020	0.978
	JDK	2.11	$\pm$	0.00816	0.987
	ECLIPSE	2.19	$\pm$	0.0163	0.955
	ASF	2.37	$\pm$	0.0109	0.981
	SPARS_DB	2.02	$\pm$	0.00145	0.999
出次数	ANT	2.92	$\pm$	0.143	0.872
	JBOSS	3.17	$\pm$	0.104	0.905
	JDK	3.10	$\pm$	0.0818	0.876
	ECLIPSE	3.03	$\pm$	0.0770	0.856
	ASF	3.41	$\pm$	0.0637	0.942
	SPARS_DB	3.66	$\pm$	0.0693	0.903



(a) ANT (b) JBOSS

(c) JDK (d) ECLIPSE

図 2.3: 入次数の累積度数分布 – 単一のソフトウェア

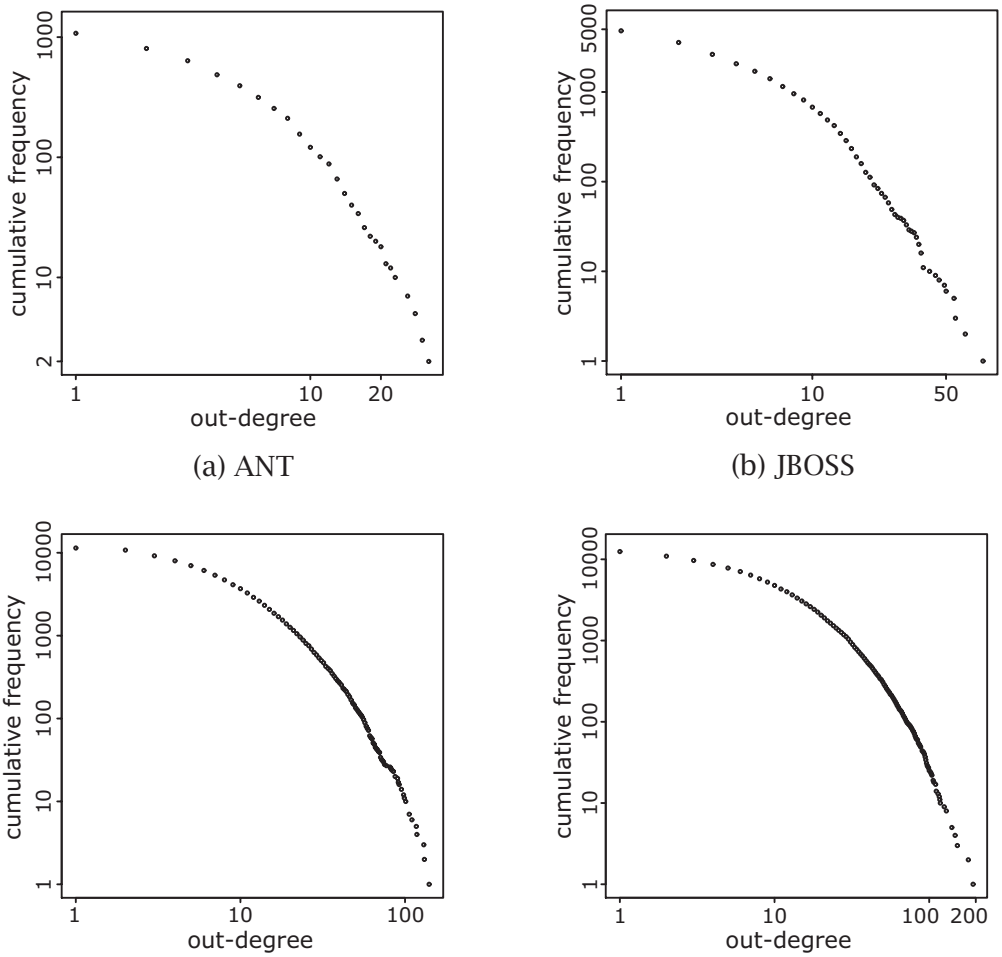
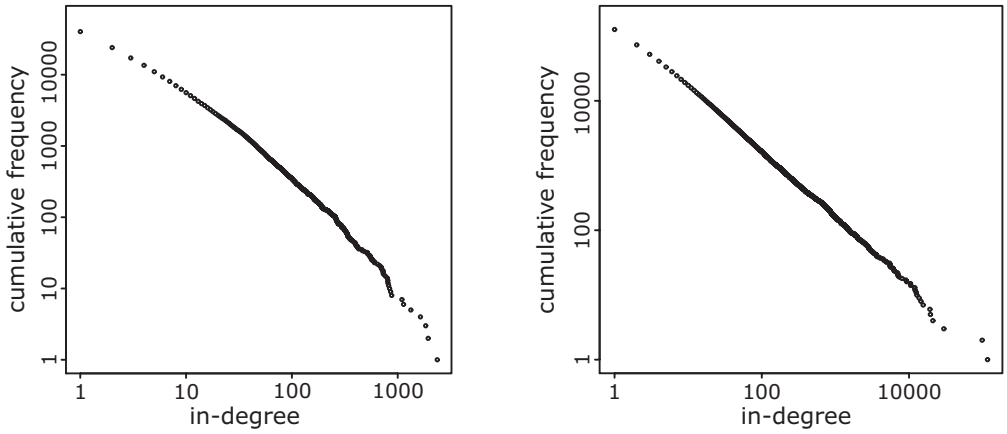
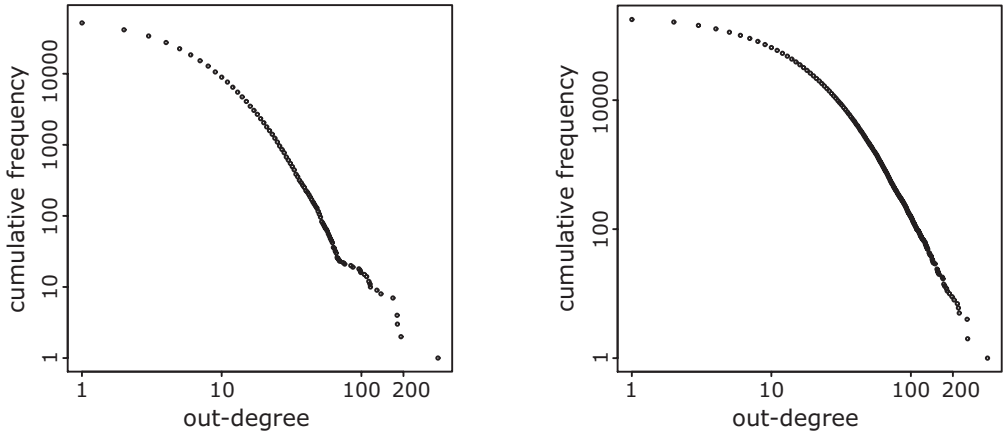


図 2.4: 出次数の累積度数分布 – 単一のソフトウェア



(a) ASF　　(b) SPARS_DB

図 2.5: 入次数の累積度数分布 – ソフトウェア集合



(a) ASF　　(b) SPARS_DB

図 2.6: 出次数の累積度数分布 – ソフトウェア集合

出次数 プロットは，次数の大きな部分については直線であるが，次数が小さな範囲で傾きに変化がみられ，完全にはべき乗則に従っていない．直線に回帰したときの R^{*2} もやや低い値である．Myers の調査 [47] でも入力と出力は個別に調査されているが，この傾向は観測されていない．

実験 2

複数のソフトウェア集合の部品集合である ASF および SPARS_DB の結果を示す．入次数および出次数のプロットを図 2.5 および図 2.6 に示す．それぞれの回帰直線から求めた α の値および R^{*2} の値は，表 2.2 に示されている．

図から，ASF，SPARS_DB 両方の部品集合に関して JDK と同様の傾向が見られることがわかる．すなわち，複数のソフトウェア集合に関しても以下が分かる．

入次数 べき乗則に従う．α の値は約 2 であり，JDK での値に近い．

出次数 値の大きな部分でのみべき乗則に従う．直線に回帰したときの R^{*2} の値も JDK と同様にやや低く，α の値は約 3 である．

実験 3

部分集合の部品グラフの入次数のプロットを図 2.7 に，直線に回帰して求めた α の値および R^{*2} の値を表 2.3 に示す．また出次数のプロットを図 2.8 に示す．これらについて，部品集合の構成方針ごとに特徴を挙げる．

無作為 RND1K は入次数・出次数ともにべき乗則には従っていない．RND10K は入次数がべき分布の特徴をもつ．

利用関係 部品数が 1000 部品未満である REL1H でもべき乗則に従う．ただし，全体的に α の値がもとの部品集合である SPARS_DB と比較して小さく，利用関係が一部の部品に集中していることが分かる．

単語 利用関係の部品集合と同様，部品数が 1000 未満の部品集合においても入次数にべき乗則が成り立つことが分かる．また，α の値も SPARS_DB に近い値となっている．

実験 4

部品集合 SPARS_DB について，入力および出次数上位 10 部品をそれぞれ表 2.4 および表 2.5 に，次数とメトリクスとの相関を表 2.6 に示す．また，表 2.6 にて弱い相関が見られる

表 2.3: 入次数分布の特性量 – 部分集合

	α			R^{*2}
RND10K	1.94	±	0.0210	0.979
RND1K	2.27	±	0.177	0.926
REL10K	2.30	±	0.0203	0.986
REL1K	1.63	±	0.0815	0.806
REL1H	1.83	±	0.0646	0.867
KWD10K	2.12	±	0.00927	0.993
KWD1K	2.21	±	0.0332	0.980
KWD1H	2.62	±	0.0805	0.983

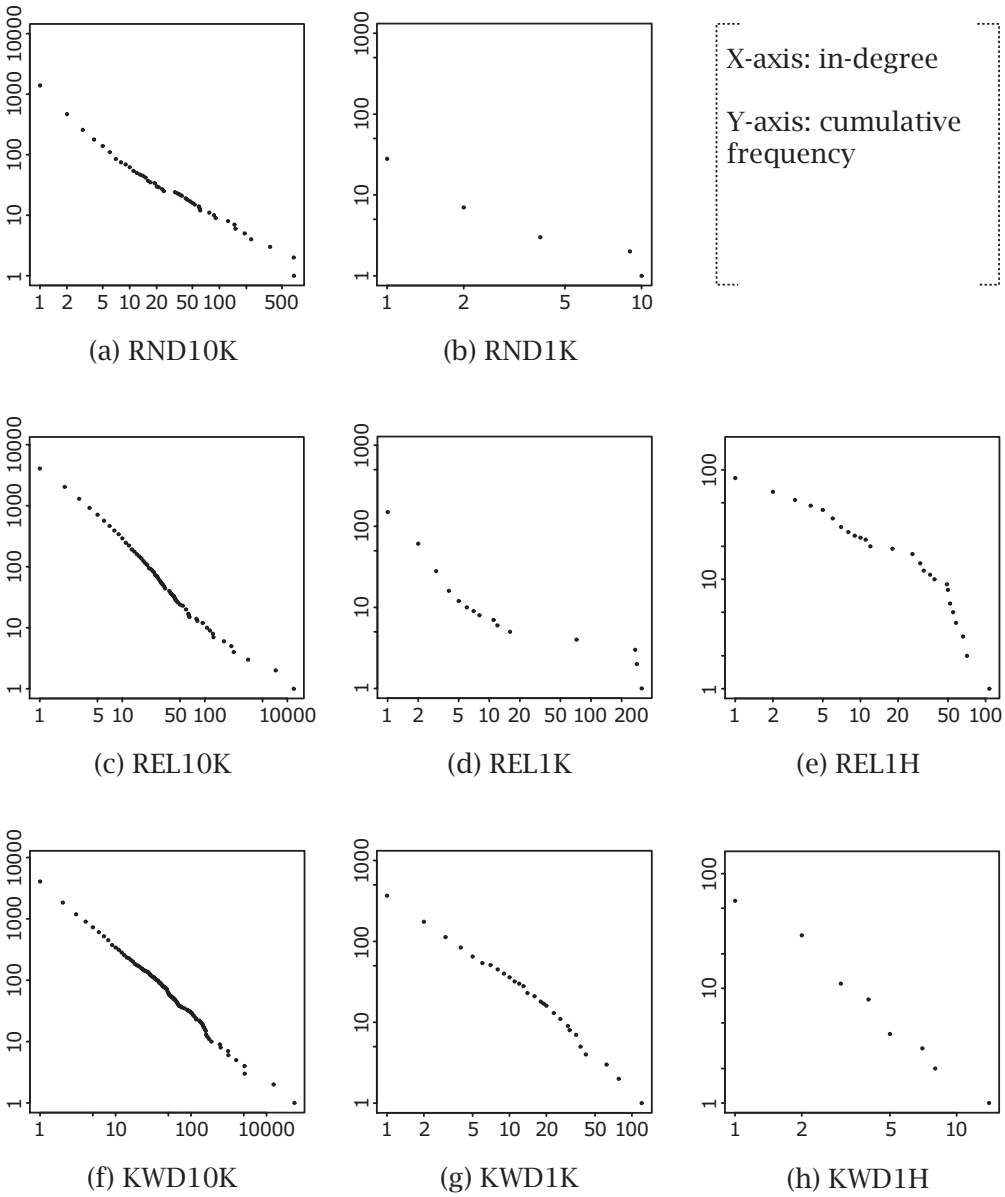


図 2.7: 入次数の累積度数分布 – 部分集合

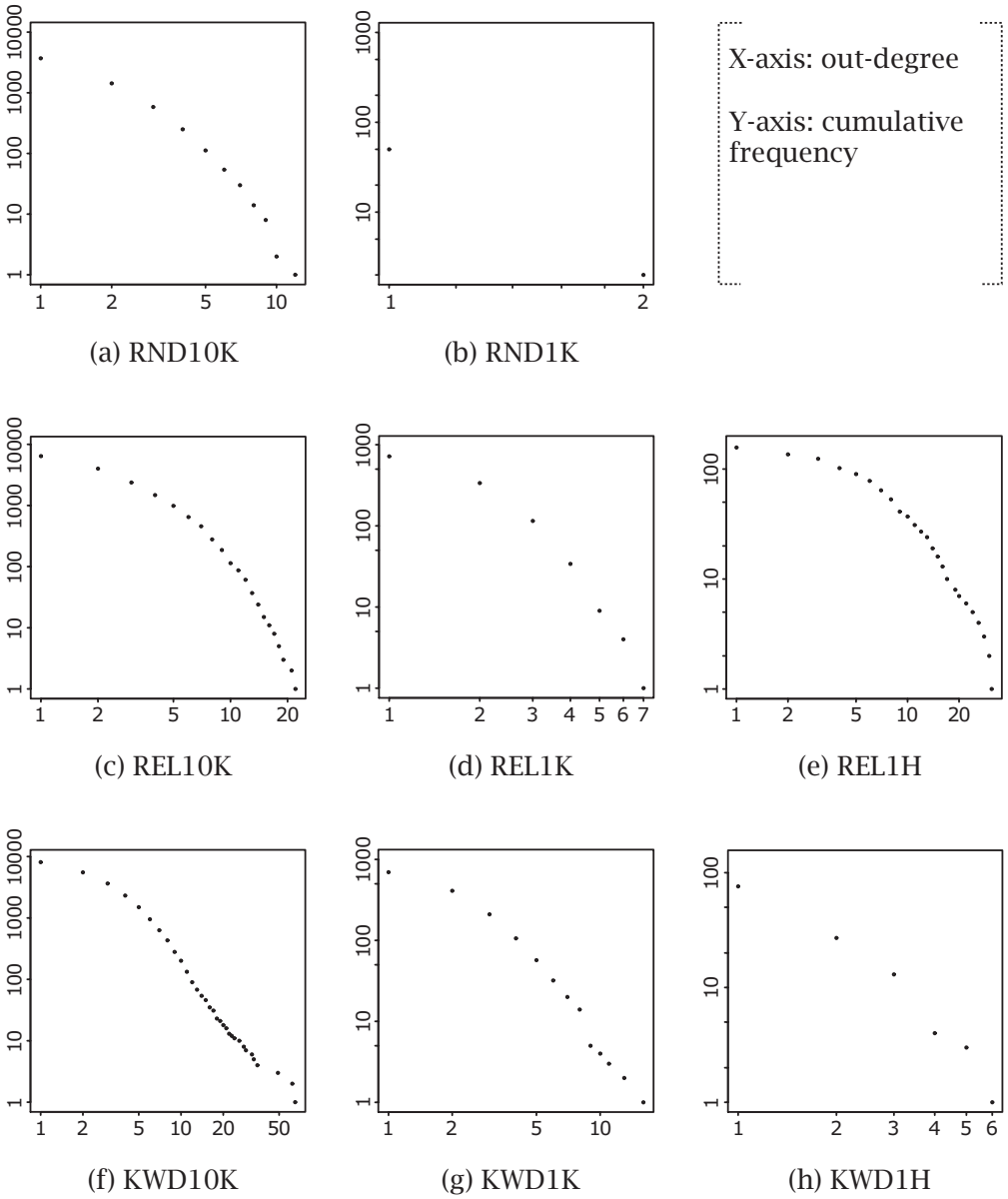


図 2.8: 出次数の累積度数分布 – 部分集合

表 2.4: 入次数上位 10 部品

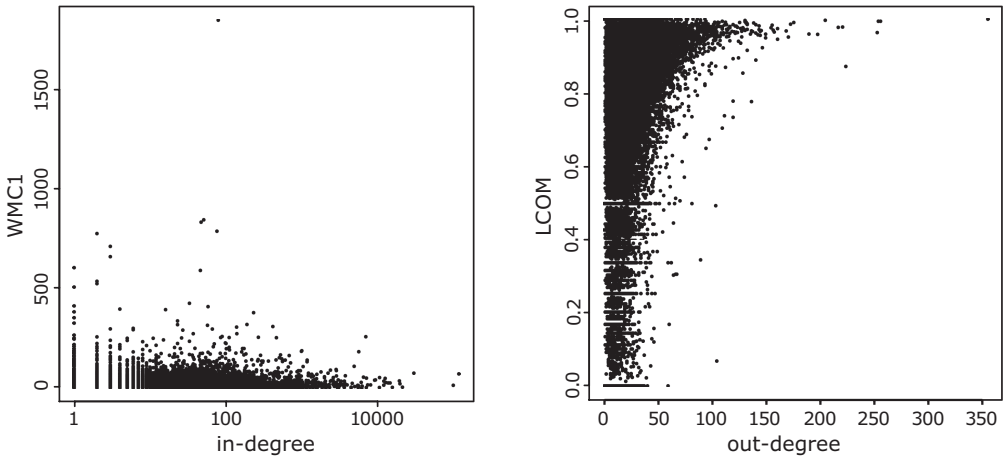
クラス名	LOC	入次数	出次数
java.lang.String	675	116,239	21
java.lang.Object	35	98,261	4
java.lang.Class	605	29,682	41
java.lang.Exception	15	21,046	2
java.lang.Throwable	136	19,519	12
java.lang.System	170	19,175	27
java.util.Iterator	5	15,522	1
java.util.List	27	14,462	4
java.util.ArrayList	200	13,656	19
java.lang.Integer	285	12,736	9

表 2.5: 出次数上位 10 部品

クラス名	LOC	入次数	出次数
org.apache...FunctionEval	364	1	354
org.jgraph.GPGraphpad	2,196	130	255
com.jgraph.GPGraphpad	2,200	131	253
org.jgraph.GPGraphpad	542	209	252
org.eclipse...ASTConverter	4,520	3	223
org.eclipse...JavaEditor	1,368	115	220
net.sourceforge...GanttProject	3,055	98	216
it.businesslogic...MainFrame	7,177	46	204
org...InstConstraintVisitor	1,626	3	197
org...ASTInstructionCompiler	2,449	1	189

表 2.6: 次数とメトリクスの相関

	出次数	LOC	WMC1	WMC2	LCOM
入次数	0.002	0.070	0.239	0.075	0.118
出次数	-	0.824	0.641	0.751	0.399
LOC	-	-	0.793	0.816	0.462
WMC1	-	-	-	0.567	0.494
WMC2	-	-	-	-	0.332



(a) WMC1 - in-degree　　(b) LCOM - out-degree

図 2.9: 次数とメトリクスの散布図

入次数と WMC1 および出次数と LCOM の散布図を図 2.9 に示す．なお，入次数と WMC1 の散布図は見やすさのために片対数軸を用いている．

入次数はどのメトリクスとも相関が低い．WMC1 とやや相関が見られるが，散布図から関連性は見られない．また，入次数が上位の部品は，JDK の基礎的な役割を持つ部品で占められている．例えば String は文字列を表現し，Object は全ての親クラスである．しかし，他の共通点は見られない．これより，入次数は，部品のソースコードのみの解析で得られた部品の規模や複雑度，凝集度との直接的な関連は低く，設計時に与えられた役割と関連が高いと考えられる． ただし，メトリクスとの相関については，本研究で用いたメトリクスは互いにある程度の相関があり，一般のメトリクスと相関がないとは言えない．例えば，今回用いなかった，部品の役割の基礎性や汎用性などの再利用に関わる値を計測できるメトリクスを用いることで，相関が得られる可能性がある．

出次数は LOC や WMC と相関が高い．部品グラフの出力辺は，ソースコード内で他の部品を利用する記述に基づくため，出次数と LOC の相関が見られることは自然である．出次数と WMC との相関は，LOC がそれぞれの WMC と相関が高いことが反映されていると考えられる．また，出次数は LCOM ともやや相関が見られ，散布図より，出次数の大きな部品は LCOM が高い傾向が分かる．LCOM は WMC や LOC とやや相関が見られることが反映されたものだと考えられる．LCOM が WMC 等と相関が見られる理由としては，一般にクラスの規模が大きくなるとクラスのもつ属性のうち一部のみを利用するメソッドが増え，凝集度が下がる傾向にあるためであると考えられる．これは，メソッドの抽出といったリファクタリングなどにより生じる．

2.4.3　実験結果のまとめ

2.1 節で述べた問題 1∼4 に関し，実験により明らかになったことを以下にまとめる．

問題 1 実験 1 にて，変数宣言やメソッド呼び出しなどを含む様々な静的な利用関係に基づく部品グラフについて，次数がべき乗則に従うかどうか調査した．結果として，入次数は非常に高い度合いでべき乗則に従うことが明らかになった． また，出次数は両対数軸プロットにおいて，値の大きな部分ではほぼ直線であるが値の小さな部分で曲線となり，べき乗則に従わないことが明らかになった．

問題 2 実験 2 にて，複数のソフトウェアから構成される部品グラフについて同様の調査を行った．結果として，複数のソフトウェア集合の部品グラフは，単一のソフトウェアの部品グラフと同様，入次数は高い度合いでべき乗則に従うが，出次数はべき乗則に従わないことが明らかになった．

問題 3 実験 3 にて，複数のソフトウェアから取得した部品集合の部分集合により構築した部品グラフについて，同様の調査を行った．無作為に部品を取り出した集合のうち，部品数の多いもの，利用関係に基づき取り出した部品集合，単語に基づき取り出した部品集合において，もとの部品集合と同様の結果，すなわち，入次数はべき乗則に従い，出次数はべき乗則に従わないという結果が得られた．

問題 4 実験 4 にて，部品そのものの性質と入次数および出次数との関連について，相関を求め，次数上位の部品の内容を分析して調査した．結果として，入次数は本研究で用いたメトリクスとは相関が無く，設計時に部品に与えられた役割や責任と関連し，出次数は部品そのものの規模や複雑さと関連することが明らかになった．

2.5 考察

本研究の実験により，2 つの興味深い結果が得られた．一つは，本研究で構築した部品グラフでは，入次数は高い度合いでべき乗則に従うが，出次数は従わないという非対称性がみられた点である．もう一つは，部品集合の部分集合から構成した部品グラフにも，全体集合の部品グラフと同様の性質がみられた点である．以下，入次数，出次数に関して得られた結果について個別に考察した上でそれらの非対称性について考察する．また，部分集合に見られた性質について考察する．

2.5.1 入次数

部品グラフの入次数がべき乗則に従うことの意味は，単にごく一部の部品が頻繁に利用されることのみではない．次数分布がべき乗則に従うグラフの生成モデルは複数提案されており，いずれも新しい辺は既に多くの次数をもつ頂点に接続されやすい点で共通している [39]．これは，表 2.4 に示されるような部品グラフ中で大きな入次数をもつ部品は，新たな部品が追加された時に，より大きな入次数をもつことを意味する．これより，部品グラフに新たな部品が追加された場合も，次数の大きな部品群の入れ替わりは生じにくいと考えられる．

この性質により，ソフトウェア開発の進行に伴うソフトウェアアーキテクチャの変化を観測できると考えられる．部品グラフは，ソフトウェア開発の進行に伴う部品や利用関係の追加や削除により変化するが，上述した性質より，入次数が上位の部品群には変化が生じにくい．しかし，ソフトウェアアーキテクチャなどの大きな枠組みでの変更が生じた場合には，それらの部品に入れ替わりが生じると考えられる．例えば，新たな共通部品を追加し，既存の多くの部品を，追加した部品を利用するように変更することが挙げられる．従って，入次数が上位となっている部品群の構成の移り変わりをみることで，アーキテクチャの変化の検出や，その安定度の測定ができると考えられる．

以上は次数がべき乗則に従うグラフに一般にみられる性質に基づいた考察であるが，部品グラフは入次数と出次数の非対称性など，一般のグラフとは異なる性質も観察されている．そのため，本考察の内容に対し実際に調査・検証を行うことが必要である．

2.5.2 出次数

実験により得られた出次数の分布には，対数正規分布の特徴である，値の小さな範囲に寄った「頂点」が見られた．しかし，対数正規分布と異なり，値の大きな範囲に，べき乗則のような長い裾野（long-tail）が見られた．この特徴はファイルサイズの分布として知られる Double-Pareto 分布に一致している [44]．出次数は LOC と相関が高いことを考慮すると，ファイルサイズと同様の分布に従うことは直感的に妥当であると考えられる．出次数が Dobule-Pareto 分布に従うかどうかに関する調査は本研究の範囲を超えるが，今後，この観点からの調査および検証を改めて行う必要がある．

出次数は，CK メトリクスでは Coupling Between Objects (CBO) として定義され，この値が大きいと複雑であり保守や理解が困難であるとされる [21]．実験では，出次数は値の大きな範囲のみに注目するとべき乗則に従うことが明らかになった．これは，高い出次数をもち保守性などに問題のある部品 (クラス) は一般的に存在することを意味し，以下の 2 通りの解釈が可能である．まず，保守性に問題がある部品も，多くはリファクタリングされず放置されることが考えられる．これは多くの開発者もしくは管理者が，設計品質について意識不足であることを意味する．これに対し，複雑になることを避けられない部品が一般に生じやすいことも考えられる．これはオブジェクト指向など，現在のソフトウェア開発パラダイムによる品質の限界を意味する．本研究で得られた結果からは一方に絞ることはできず，様々なソフトウェアに対して調査し，傾向を分析することが必要である．

2.5.3 入次数と出次数の非対称性

Myers らは入力と出力の非対称性，すなわち入次数と出次数はいずれもべき乗則に従うが，プロットの傾きが異なる点に注目していた [47]．本研究では，この非対称性は，入次数は非常に高い度合いでべき乗則に従うのに対し，出次数はべき乗則に従わないという，分布そのものが異なるという形で現れた．この理由として，本研究では辺としてより詳細な

利用関係を用い，部品間の静的な依存関係をより正確に表現した部品グラフを構築したためであると考えられる．出力に注目したとき，継承関係や属性はほとんど持たない部品が多いのに対し，メソッド呼び出しや変数宣言を含めた場合には，少数の利用関係をもつ部品がもっとも多くなるためにこのような結果が得られたと考えられる．また，ソフトウェアをまたがる利用関係を考慮したことにより，単一のソフトウェアの解析では得られないライブラリ部品に対する出力辺を含む部品グラフを構築できたことも，その理由の一つであると考えられる．

また，入次数と出次数の分布の違いとして，その値域に注目する．入次数はいずれの部品集合でも α の値はほぼ 2 であり，次数の値域は部品集合により大きく異なった．これに対し，出次数は部品数の多い部品集合ほど α の値も大きくなる傾向にあり，また，最大の部品集合である SPARS_DB でも最大値は 400 未満であった．これは出次数には事実上の最大値が存在することを示唆している．この要因として，入力辺と出力辺がそれぞれ異なる部品の性質に関連することが考えられる．入次数が大きな部品は基礎的な役割を持つ部品であり，部品集合が大きくなればよりいっそう利用され次数が大きくなるのに対して，出次数が大きな部品は規模が大きく複雑な部品であり，部品集合の大きさとは独立であることを反映していると考えられる．

2.5.4 部分集合におけるべき乗則

次数がべき乗則に従うグラフは，スケールフリーネットワークとも呼ばれ，無作為にノードを取り除いていっても，ある程度は構造が維持されるが，ある時点から急速に構造が崩壊する性質を持つ [16]．この性質は，無作為に無作為に部品を抽出することで構築した部品グラフの結果に現れた．すなわち，部品数が多いものは全体集合と類似した結果であったが，部品集合の小さなものは全くことなる結果となった．

これに対し，利用関係に基いて取り出した部品集合および単語に基づき取り出した部品集合は，部品数が少数であっても全体集合と同様の結果となった．利用関係に基づいて取り出した場合は，最低限，基準となる部品に対する利用関係が存在するため，無作為に取り出した場合と比較して少ない部品数でもべき乗則がみられた．しかし，逆にその部品に利用関係が集中する傾向がみられた．単語に基づく取り出し方は，明示的には利用関係とは無関係な取り出し方であるが，結果としてもっとも全体に近い性質が得られた．これは，同じモジュールに属する部品など，関連性のある部品群は共通する語を含むことが多いことが一つの要因であると考えられる．このような部品群は密な利用関係を持つことが多い．また，もうひとつの要因として，ある部品が対象の語を名前として含む場合にはその部品を利用する部品も変数宣言などの形で語を含むため，利用関係が維持されることが挙げられる．

スケールフリーネットワークには，全体と同様の特性がその一部にも現れる自己相似性と呼ばれる性質をもつものが存在することが知られており，部品グラフもそれらと似た性質をもつと考えられる．これより，単一のソフトウェアおよびソフトウェア集合の部品グ

ラフを対象とする手法が，それらの部分集合の部品グラフに対しても適用できる可能があることを示唆している．例えば，部品の再利用性の評価手法であるコンポーネントランク法 [29, 71] は単一のソフトウェアやソフトウェア集合を対象としているが，単語に基づく部品集合，つまり，ソフトウェア部品検索システムの検索結果として得られる部品集合に対して適用した場合も有効な結果が得られることが考えられる．ただし，実際の適用実験を行い有効性を評価することが必要である．

2.6 結論と課題

本章では，次数分布がべき乗則に従うかどうかという観点から様々な部品集合の部品グラフを調査した．

まず，既存研究よりも多くの利用関係に基づき構成した部品グラフの次数分布がべき乗則に従うかどうかを調査した．その結果，入次数はべき分布に従い，出次数はべき分布に似た異なる分布に従うことがあきらかになった．また，その性質は単一のソフトウェアだけではなく，複数のソフトウェア集合や，その部分集合についても成り立つことがあきらかになった．さらに，個々の部品の次数と関連する部品の性質に関して，入次数は部品の役割，出次数は部品の規模や複雑さと関連することがあきらかになった．

本研究では部品間の利用関係を区別せずに用いて部品グラフを構成したが，利用関係の種類ごとに個別の部品グラフを作成し，同様の調査を行うことが今後の課題である．

第3章　再利用支援のためのソフトウェア部品の依存関係解析手法

3.1　導入

近年，様々なオープンソースソフトウェアのプロジェクトが成功を収めている．オープンソースソフトウェアのソースコードはWWWを通じて入手でき，開発者はそれらを再利用することで効率的なソフトウェア開発を実現できる．SPARS-J [72]，Google code search [5] など，それらを対象としたソフトウェア部品検索システム（ソースコード検索システム）が一般に利用可能であり，これらを利用することで効率的に再利用可能なソフトウェア部品（部品）のソースコードを取得できる [28].

しかし，取得した部品は，それ単体では再利用できないことが多い．ライブラリを取得してシステムに組み込む必要がある場合や，一連の機能を持つ部品を全て取得する必要がある場合のように，ある部品Aを利用するために，別の部品Bを組み込むことが必要となる場合，部品Aは部品Bに依存していると考えることができる．取得した部品をソフトウェアに組み込むためにはその依存関係を理解し，必要に応じて依存する部品を取得することが必要である．ソースコードの再利用の場合には依存関係を開発者自らが調査する必要があるが，複雑な依存関係を手作業で解析し理解することは労力を要する．

さらに，複数のバージョンがリリースされていたり，ある部品を基にして作られた別の部品（派生部品）が存在するなどの理由で，1つの名前で参照される部品は1つに定まらない．バージョンの異なる部品には構造的な変更が存在することがあり [32]，ライブラリとして作成された部品であっても互換性の無い変更が加えられていることがあるため [23]，依存する部品として利用できるかどうかを，それぞれのAPIに基づいて判断しなければならない．このとき，依存する部品間に存在する依存関係もまた考慮する必要がある．

これらの問題を解決するために，本研究ではソフトウェア部品検索システムでの再利用支援を目的とした依存関係解析手法を提案する．提案手法は，再利用対象の部品が依存する部品集合の候補を，部品同士の関連を考慮しながらそれぞれのAPIに基づいて抽出する．開発者は，その中から開発環境特有の制約などの要求を満たす候補を選択することで，対象の部品を組み込むために必要な部品集合を取得できる．また，提案手法に基づき依存関係解析システムDACARAを構築する．DACARAは，ソフトウェア部品検索システムの連携を考慮し，事前に部品集合を解析してデータベースを構築しておくことで，大規模な部品集合に対する高速な依存関係の解析を実現する．

また，依存関係解析手法の有効性を評価するため，複数のバージョンを含むオープン

ソースソフトウェアの集合に対して適用実験を行う．全てのクラスに対して依存関係を解析し，その結果を分析することで，提案手法が依存関係を理解する作業の効率化に役立つことを示す．

以下，3.2 節ではソフトウェア部品検索システムを用いた再利用での依存関係の問題について述べ，3.3 節にて提案手法である依存関係解析手法について述べる．3.4 節では依存関係解析システム DACARA について述べ，そのシステムを用いた実験について 3.5 節で述べる．3.6 節で考察し，3.7 節で関連研究に触れる．最後に 3.8 節で本章をまとめる．

3.2 ソフトウェア部品検索システムを用いた再利用

本研究で対象とする，ソフトウェア部品検索システムを用いた部品の再利用と，それを実践する上での問題点について述べる．開発者が行う部品の再利用のための手順は，部品の取得と，開発対象への組み込みに大別される．このうち部品の取得に関しては，再利用性に基づく部品の順位付け [29, 71] や正規表現による検索 [5] など，様々な支援手法が存在する．

しかし，取得した部品を組み込むための情報はあまり提供されていない．現実にある部品を再利用する際には，その部品の記述に現れる，メソッド呼び出しや継承などの利用関係が存在する部品を抽出し，それらの部品をさらにソフトウェアに組み込む必要がある．このような，再利用対象の部品からの利用関係がある部品を，再利用の対象の部品が**依存する部品**とよび，再利用対象の部品から依存する部品に対して**依存関係**があると定義する．本研究では，利用関係が複雑になりやすく，理解が困難であると言われているオブジェクト指向プログラムの依存関係に注目する．

図 3.1 に示すコード片を用いて，Java の依存関係の理解について説明する．図 3.1 は，標準ライブラリ（Java SE）を用いてデータベースに接続するプログラムである．Java SE には複数のバージョンが存在するため，参照される名前それぞれに異なる API をもつ複数の部品が存在する．ここでは 1.3，1.4，1.5 の各バージョンが利用可能であるとする．

依存関係の理解は，ソースコード中の他の部品を参照する記述それぞれに対して，実際に指し示す部品を探して特定する作業となる．例えば，行 01 の Connection に対しては “Connection” で参照可能なクラスを探して特定し，行 02 の createConnection() に対しては，行 01 の Connection に対して特定したクラスから，その名前のメソッドを探す．ソフトウェア部品検索システムから取得した部品に対しては，この過程で以下のような問題が生じる．

1. 同じ名前で参照される部品が複数存在するため，その中から必要な API をもつ部品を選択しなければならない．

   例えば，Connection で参照されるクラスは全てが createStatement() メソッドおよび close() メソッドをもつが，ResultSet で参照されるクラスのうち，getURL() メソッドをもつものはバージョン 1.4 以降の 2 種類に限定される．

```
    ...
01: Connection conn = ConnectionManager.get();
02: Statement stmt = conn.createStatement();
03: ResultSet rs = stmt.executeQuery("SELECT URL...");
04: URI uri = null;
05: Timestamp time = null;
06: if (rs.next()) {
07:     uri = rs.getURL(1).toURI();
08:     time = rs.getTimestamp(); }
09: long ms = timestamp.getTime();
10: rs.close();
11: stmt.close();
12: conn.close();
    ...
```

図 3.1: ソースコード例

2. 依存する部品間にも依存関係が存在し，ソースコード上では直接名前が参照されない部品にも依存することがある．

   例では，行 07 で参照される toURI() メソッドをもつ URL クラスが getURL() の返り値の型として，また，Timestamp クラスの親クラスである Date クラスが行 09 で参照される getTime() メソッドの定義を持つクラスとして必要となる．これらは，親クラスやメソッド定義の返り値を確認して初めて判明し，それぞれに異なるクラスを選択した場合は，異なる結果になる可能性がある．

3. 依存する部品にもさらに依存する部品があり，同様の作業を再帰的に繰り返す必要がある．

   例では，Connection や ResultSet は Java SE のクラスであるため不要であるが，行 01 の ConnectionManager のみは Java SE のクラスでは無いため，クラスを取得した上で同様に依存関係を理解する必要がある．

本研究では，このような依存関係の理解にかかる労力を削減するための手法を提案する．提案手法は，対象部品の依存する部品の候補を，利用している API を考慮して解析し，複数の候補を提示することで，開発者が 1. や 2. を考慮しながら手作業で解析する必要をなくす．さらに，3. のために，複数の部品からなる集合を解析対象とする．

利用者（開発者）は，まずソフトウェア部品検索システムを用いた検索により，再利用対象の部品を取得する．提案手法は，再利用対象の部品集合を入力として受付け，それらが依存する部品集合の候補を返す．利用者は既に利用している部品などを考慮して候補を選択することで，対象の部品を再利用するために必要な部品集合を取得できる．

3.3 依存関係解析手法

本節では，提案手法である依存関係の解析手法について説明する．解析対象のモデルを図 3.2 に示し，主要な要素について説明する．

エンティティ（Entity） オブジェクト指向言語におけるクラスもしくはメンバ．

メンバは，メソッド（メンバ関数）およびフィールド（メンバ変数）であり，参照されるための名前をもつ．クラスは，ビルド単位において一意となる名前（完全限定名）のほか，その名前空間部分を省略した単純名で参照される．

参照（Reference） 部品であるクラスからエンティティを参照するためのソースコード上の記述．

ある参照の指し示す先を**参照先**と呼び，参照先をエンティティの集合から探して決定することを**特定**と呼ぶ．詳細については 3.3.1 節で述べる．

提案手法は，クラスの集合を入力として，それらが依存し，再利用するときに必要となるクラス集合の候補である**再利用単位**の集合を得る手法である．再利用単位は，クラス集合を要素とする集合である．要素は互いに代替可能なクラス集合であり，要素それぞれからクラスを一つずつ選ぶことで，入力の依存関係を満たすクラス集合を取得できる．

提案手法は，以下の 3 段階で構成される．

1. 参照の解析
   対象のクラス集合それぞれに含まれる参照を取得し，参照それぞれについて参照先の集合を求める．

2. 依存グラフの構築
   参照の解析により得られた，対象のクラス集合が依存するクラス間の関連を有向グラフで表現する．あるクラスは，その頂点へ向かう辺をもつクラスのいずれかが参照されるとき，参照される可能性があることを表す．

3. 再利用単位の抽出
   依存グラフに基づき，再利用単位の集合を得る．ある再利用単位は，対象のクラスに対応する頂点を始点として，同じ名前で参照されるクラスを複数含まないように依存グラフ上の頂点を訪問する，1 つの経路上の頂点集合として得られる．

以降の説明では，図 3.3 に示されるクラス c_1 を再利用する例を用いる．実線の枠で囲まれた範囲がクラス c_n を，点線の枠で囲まれた範囲がメンバ m_n を，破線のアンダーラインを引かれた文字列が参照 r_n を示す．例えば，クラス c_1 の名前は “Manager” であり，メソッド m_1 が “exec” という名前で宣言されている．m_1 には r_1, r_2, r_3 の 3 つの参照が含まれ，それぞれ「“Liquor” という名前の，ローカル変数の型」「“Liquor” という名前のコンストラクタ呼び出し」「“name” という名前のフィールド参照」を意味する．図中には，6

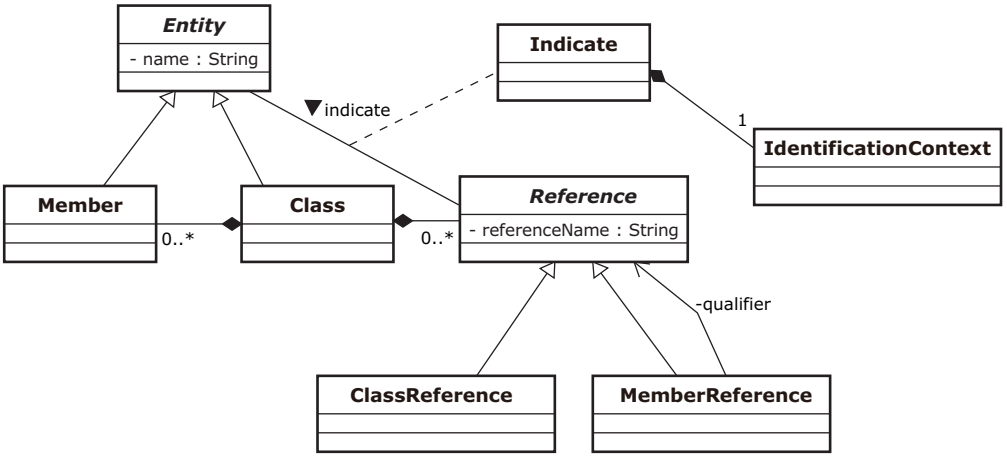


図 3.2: 解析モデル

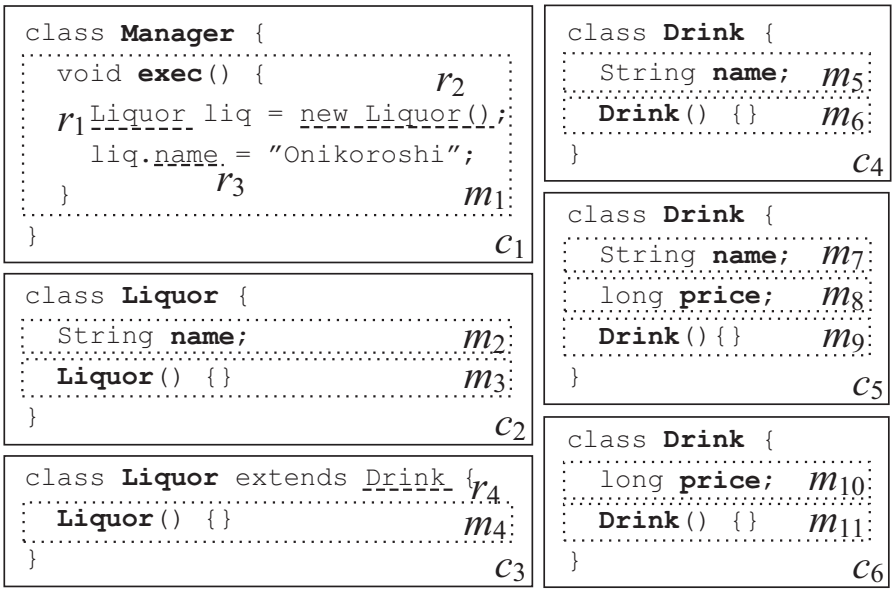


図 3.3: 部品群の例

個のクラスおよび 11 個のメンバが定義され，対象クラス c_1 中には 3 個の参照が存在する．なお，説明に用いない参照（c_2 中の “String” など）は省略している．

3.3.1 参照の解析

まず，参照について詳しく説明する．参照は**クラス参照**（ClassReference）と**メンバ参照**（MemberReference）に分類される．クラス参照は，継承の宣言，変数宣言，メンバの型（フィールドの型もしくはメソッドの返値の型）宣言の参照である．参照名に対応する

名前をもつクラスが参照先となる．メンバ参照はフィールドやメソッドなどクラスのメンバに対する参照である．図 3.3 では，r_1 と r_4 がクラス参照，r_2 と r_3 がメンバ参照となる．また，r_1 は r_3 を限定 (qualify) しており[1]，r_3 の参照先を特定するには r_1 の参照先が必要である．提案手法では複数のエンティティが同じ名前で参照されることを前提とするため，参照と参照先を 1 対多の関連で扱う．参照 r_1 に対しては，クラス c_2 および c_3 それぞれが参照先となる．

1 つの参照に対する複数の参照先の絞込みのために，**特定状況**（Identification Context）という概念を導入する．特定状況は,（参照，参照先クラス）の組の集合であり，直感的には，ある参照をある参照先に特定したとき，メンバの所有クラスなど，特定のために用いられたクラス集合を表す．メンバ参照 r_3 を，メンバ m_5 に特定する例を考えると，まず r_3 を限定するクラス参照 r_1 をクラス c_3 に特定し，続いて名前 name で参照されるメンバを親クラスから探すために継承のクラス参照 r_4 がクラス c_4 に特定される．これより，特定状況は $\{(r_1, c_3), (r_4, c_4)\}$ となる．

解析手順

参照の指し示す先を，特定状況とともに求めるアルゴリズムを図 3.4 に示す．なお，関数 name() は，引数で与えられた参照の参照名またはエンティティの名前を返し，関数 ownerClass() は引数で与えられた参照を含むクラスを返し，関数 indication() は引数で与えられた参照が指し示す，特定済みの参照先の集合を返す．また C は全てのクラス集合を示す．

関数 identify() は，対象の参照がクラス参照ならば参照名に一致する名前をもつクラスを参照先として特定する．クラス参照の参照先の特定参照は，参照そのものと参照先クラスの組のみを要素とする．対象がメンバ参照ならば，所有クラスを求めて関数 identifyMember() に処理を移す．

関数 identifyMember() は指定された所有クラス，もしくはその親クラスから対象の名前で参照されるメンバを参照先とする．メンバ参照とその参照先に対する特定状況は，所有クラスを決定するために用いられた，すべてのクラス参照とその参照先に対する特定状況の和集合である．

図 3.3 のクラス c_1 に対する解析結果を表 3.1 に示す．

なお，簡単のため，説明中では以下の要素は考慮していない．

- 単純名によるクラスの参照
  クラスは，完全限定名に加え，名前空間の省略された単純名で参照される．参照名が単純名であるときには，利用可能な名前空間との結合などにより完全限定名へ変換した上で参照先を特定しなければならない．なお，提案手法のそのほかの手順では，単純名でのアクセスを考慮している．

[1]文面上は r_3 を限定するのは変数 liq であるが，本手法では，ローカル変数はその型に置き換えて手順を進める．

```
001: identify(r:  Reference):
002:     if r is ClassReference:
003:         C_c := {c|c ∈ C, name(c) = name(r)}
004:         for c_c ∈ C_c:
005:             Add c_c to the result of r with context {(r, c_c)}
006:     else if r is MemberReference:
007:         if r has qualifier:
008:             r_q := the reference qualifying r
009:             for e_q ∈ indication(r_q):
010:                 x := the context where r_q is identified as e_q
011:                 if e_q is Class:
012:                     c_o := e_q
013:                     identifyMember(r, c_o, x)
014:                 else if e_q is Member:
015:                     r'_q := TYPE_DECRALATION class reference
                                 of e_q
016:                     for c_o ∈ indication(r'_q):
017:                         x' := x ∪ the context where
                                     r'_q is identified as c_o
018:                         identifyMember(r, c_o, x')
019:         else:  # i.e.  no reference qualifies r
020:             x := φ
021:             c_o := ownerClass(r)
022:             identifyMember(r, c_o, x)

101: identifyMember(r:  Reference, c_o:  Class, x:  Context):
102:     M_c := {m|m ∈ members of c_o, name(m) = name(r)}
103:     if M_c ≠ φ:
104:         for m_c ∈ M_c:
105:             Add m_c to the result of r with context x
106:     else:  # i.e.  not found in current owner class
107:         # search from parent class
108:         for r_p ∈ set of INHERITANCE class references of c_o:
109:             for c_p ∈ indication(r_p):
110:                 x' := x ∪ {(r_p, c_p)}
111:                 identifyMember(r, c_p, x')
```

図 3.4: 参照の解析アルゴリズム

- クラスおよびメンバのアクセス制御
  クラスやメンバはアクセス制御されることがある．参照名（もしくは参照名から得た完全限定名）に一致する部品それぞれに対し，参照を含むクラスからアクセス可能かどうか判定し，アクセス可能なもののみを参照先とする必要がある．

- メンバのオーバーロード
  多くのオブジェクト指向言語では，同じ名前で引数が異なるメソッドを複数定義することができる．そのため，メンバ参照の特定の際には，名前だけでなく，実引数に対応する参照および仮引数に対応する参照それぞれの代入可能性を考慮する必要がある．しかし，ビルド単位を仮定しない本手法においては極めて困難であるため，近似的な方法として引数の個数のみを考慮している．

- 入れ子クラス間でのメンバ参照
  Java など一部の言語では，入れ子に定義された内部のクラスが外部のクラスのメンバを参照することが可能である．そのため，メンバ参照を特定する時の所有クラスの候補として，継承関係にあるクラスだけでなく，入れ子関係にあるクラスも用いる必要がある．

- `("str"+"ing").getClass()` のような，演算による参照
  “ + ” 記号などでの演算も，その結果が何らかのクラスを示すならば参照として扱う必要がある．本手法の範囲では，メンバの所有クラスを求めるために必要となる場合が存在する．

3.3.2 依存グラフの構築

依存グラフの構築のため，まず，クラスを頂点とし，辺にラベルをもつ有向多重グラフである中間グラフ $G_i = (V_i, E_i)$ を構築する．中間グラフは再利用対象のクラスの参照解析の過程で得られた特定状況を用いて構築され，再利用対象のクラス集合を中心とした，

表 3.1: 参照解析結果の例

参照	参照先	特定状況
r_1	c_2	$\{(r_1, c_2)\}$
	c_3	$\{(r_1, c_3)\}$
r_2	m_3	$\{(r_1, c_2)\}$
	m_4	$\{(r_1, c_3)\}$
r_3	m_2	$\{(r_1, c_2)\}$
	m_5	$\{(r_1, c_3), (r_4, c_4)\}$
	m_7	$\{(r_1, c_3), (r_4, c_5)\}$

クラス間の参照関係を表現する．対象のクラス集合が含む参照の集合を R とし，特定状況の要素の集合 X を

$$X = \bigcup_{r \in R} \left(\bigcup_{i \in \text{indication}(r)} \text{context}(r, i) \right)$$

と定義すると，中間グラフの頂点 V_i および辺 E_i はそれぞれ以下のように定義される．

$$\begin{aligned} V_i &= \{\text{ownerClass}(r) | (r, c) \in X\} \cup \{c | (r, c) \in X\} \\ E_i &= \{(\text{ownerClass}(r), c, \text{name}(r)) | (r, c) \in X\} \end{aligned}$$

続いて，クラス集合を頂点とし，辺にラベルをもつ有向多重グラフである依存グラフ $G_d = (V_d, E_d)$ を構築する．依存グラフは，中間グラフの頂点をグループ化することで構築される．クラス c もしくはクラス c のメンバのいずれかを参照先に含む参照の集合を contribution(c) と表すと，以下に定義される equivalent(c_1, c_2) を満たす同値類によりグループが形成される．あるグループに含まれるクラス集合は，与えられたグラフの中で互いに「同じクラス」として代替可能である．

$$\begin{aligned} &\text{equivalent}(c_1, c_2) = \\ &\quad (\text{contribution}(c_1) = \text{contribution}(c_2)) \\ &\wedge \quad (\{(c_s, l) | (c_s, c_d, l) \in E_i, c_d = c_1\} = \{(c_s, l) | (c_s, c_d, l) \in E_i, c_d = c_2\}) \\ &\wedge \quad (\{(c_d, l) | (c_s, c_d, l) \in E_i, c_s = c_1\} = \{(c_d, l) | (c_s, c_d, l) \in E_i, c_s = c_2\}) \end{aligned}$$

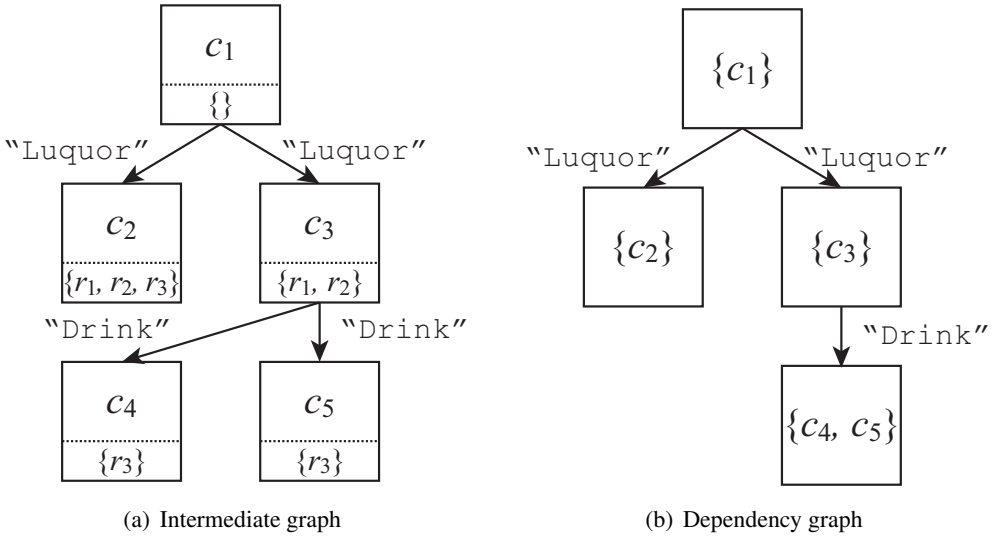
得られたグループの集合が依存グラフの頂点集合 V_d となる．また，グループに含まれるクラス間に，中間グラフにおいて辺が存在すれば，グループ間に辺を作成する．辺集合 E_d は中間グラフの辺集合 E_i に基づき以下の通りに定義される．

$$E_d = \{(v_s, v_d, l) | v_s, v_d \in V_d, (\exists c_s \in v_s)(\exists c_d \in v_d)(c_s, c_d, l) \in E_i\}$$

表 3.1 に基づいて構築される中間グラフを図 3.5（a）に示す．頂点 c_n は contribution(c_n) と共に示されている．図 3.5（b）がこの中間グラフに基づき得られる依存グラフである．c_4 および c_5 は，共通して c_3 からの同じラベル “Drink” をもつ入力辺を持ち，出力辺を持たず，$\{r_3\}$ の特定先のクラスであるため，同じ頂点にグループ化されている．

3.3.3 再利用単位の抽出

再利用単位は，依存グラフ上の頂点を，入力クラスに対応するクラスを含む頂点を起点として経由する経路上の頂点集合として得られる．1 つの再利用単位において，1 つのク



(a) Intermediate graph　　(b) Dependency graph

図 3.5: 依存グラフ構築の例

ラスから同一の名前で参照されるグループをたかだか 1 つに制限するため，経路上の辺の集合を $E_r(\subseteq E_d)$ としたときに経路は以下の条件を満たすものとする．

$$(\forall(v_{s1}, v_{d1}, l_1), \forall(v_{s2}, v_{d2}, l_2) \in E_r)(v_{s1} = v_{s2} \wedge v_{d1} \neq v_{d2}) \Rightarrow l_1 \neq l_2$$

全ての経路の組み合わせを求め，再利用単位の集合を得る．図 3.5（b）に対する結果としては，$\{\{c_1\}, \{c_2\}\}, \{\{c_1\}, \{c_3\}, \{c_4, c_5\}\}$ の 2 つの再利用単位が得られる．

再利用単位の完全性　得られた再利用単位の中には，メンバの所有クラスを適切に推定できないものなど，一部のメンバ参照を特定できないものが含まれることがある．参照先が 1 つ以上特定された参照の集合 R' を

$$R' = \{r | r \in R, \text{indication(r)} \neq \phi\}$$

としたとき，再利用単位に含まれるクラスを用いることで特定できる参照集合が R' に一致するものを**完全**な再利用単位，一致しないものを**不完全**な再利用単位と呼ぶ．不完全な再利用単位は対象の依存関係として不適切であるため，利用者に提示する結果から除外する．図 3.5（b）の例に対する結果として得られた再利用単位は，いずれも完全である．

3.3.4　ヒューリスティクスによるフィルタリング

得られた再利用単位には，現実にはほとんど利用されない組み合わせを含むものが存在しうる．たとえば，Java SE の異なるバージョンのクラスが混在するクラス集合は，依存関係としては現実的ではない．そこで，以下の 3 種類のヒューリスティクスを用い，得られる再利用単位を有用性の高いものへ絞り込む．

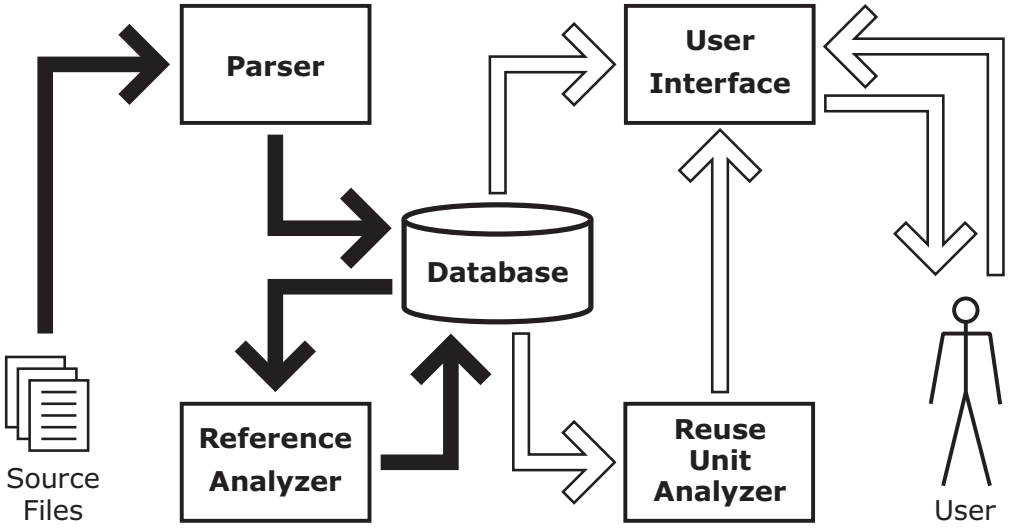


図 3.6: DACARA のシステム構成

F1（同一ファイルの優先） 参照解析の段階で，参照 r に対し，ある参照先が r と同じファイルに存在すれば，それ以外の参照先を除外する．

同一ファイル内のクラスは結びつきが強く，また，通常はファイル単位で再利用を行うことが多いためである．

F2（同一所属の優先） 依存グラフを構築するとき，中間グラフ上のあるクラス c が入力クラス集合のいずれかと同一の所属を持つ場合，c の親のいずれかから，c と同じラベルで参照される c 以外のクラス，すなわち c と衝突するクラスを全て除去する．ここで，所属とはソフトウェアの配布パッケージなど，クラスの取得元を指す．

再利用対象のクラスと同一の配布パッケージ内に候補が存在する場合，通常はまずその候補を選択することが多い．ただし，不具合の回避などのために，他のパッケージから候補を選択しなければならない場合を考慮し，再利用単位の解析時に利用者が有効/無効を切り替えられるようにする．

F3（所属混在の回避） 依存グラフを探索する際，できるだけ限られた所属で構成されるように経路を選択する．具体的には，選択した頂点と衝突する頂点に含まれるクラス集合の所属集合に，所属集合が包含されるクラス集合をもつ頂点は選ばない．

冒頭で述べたように，同一のライブラリの，異なるバージョンが混在するような再利用単位は現実的に利用できない．F2 と同様に，再利用単位の解析時に利用者が有効/無効を切り替えられるようにする．

3.4 依存関係解析システム

提案手法に基づき，依存関係解析システム **DACARA**[2]を構築した．DACARA は，図 3.6 に示される 5 個のサブシステムから構成される．実装には Java 言語を用い，規模は約 25,000 行である．DACARA はソフトウェア部品検索システムとの連携を想定しているため，大規模な部品集合の解析と，利用時のレスポンスを考慮する必要がある．そのため，部品情報の抽出および参照の解析はあらかじめ行い，結果はデータベースに格納した上で必要に応じて取り出す方式をとる．また，ユーザーインターフェースは Web インターフェースとして実装され，SPARS-J などの部品検索システムと同様に WWW を通じた利用が可能である．

以降，それぞれのサブシステムを処理の流れとともに説明する．

ソースコード解析部（Parser） まず，ソースコード解析部が，部品集合のソースコードを解析する．ソフトウェアメトリックスの計測ツールである MASU [45] の構文解析部を利用し，ファイルに含まれるクラス，メンバ，その構造，および参照を部品情報としてデータベースへ格納する．解析対象言語はバージョン 1.4 以前の Java である．MASU および本システムは複数のプログラミング言語に対応した設計であり，将来的な他の言語への拡張を比較的容易に行える．

データベース（Database） 解析結果が格納されるデータベースは，リレーショナルデータベースシステム（RDBMS）へアクセスし，部品情報や参照の解析結果の格納および取得を行う．

参照解析部（Reference Analyzer） 参照解析部は，データベースから部品情報を読み込み，3.3.1 節で述べた参照解析を行い，結果をデータベースに格納する．データ量の削減のため，特定状況はファイル単位の和集合の形で格納される．参照先それぞれに対する特定状況は，参照解析の手順を考慮してファイル単位で格納された特定状況の部分集合をとることで得られる．

再利用単位解析部（Reuse Unit Analyzer） データベースの構築の完了後，再利用単位解析部がユーザの問い合わせに応じて 3.3.2，3.3.3 節で述べた再利用単位の抽出を行う．解析結果を現実的な時間でユーザへ返すため，抽出される再利用単位の数には最大値を設ける．

ユーザインタフェース（User Interface） ユーザインタフェースは，ユーザからの部品情報および依存関係に対する問い合わせを受け付ける Web アプリケーションであり，GWT [6] を用いて実装した．ユーザは，依存関係を出力する部品を選択するために，直接部品を指定するほか，クラス名やメソッド名などに含まれる単語から部品を検索することができる．

[2]API-based Dependency Analyzer for Assisting Component Reuse の頭文字のアナグラム

図 3.7（a）はクラス情報の表示画面であり，クラスのソースコードに加え，定義されたメソッドやフィールドの一覧，Java のパッケージ階層，利用関係や類似クラスの一覧を得ることができる．再利用単位の表示画面（図 3.7（b））へは，画面上のリンクより遷移する．画面左側に入力クラス一覧，画面右側に解析結果である再利用単位が表示される．初期状態では，同一のファイルに含まれるクラス全てが入力クラスである．再利用単位の集合は右側上段テーブルに表示され，そのうちから一つを選択すると，再利用単位の要素（依存グラフにおける頂点）の集合が右側中断のテーブルに表示される．要素に含まれるクラス集合は，右側下段のテーブルに表示される．クラスを選択することで，入力クラスへの追加が可能である．また，入力クラス一覧から入力クラスを削除することができる．さらに，依存グラフの表示や絞り込みの指定も可能である．

3.5 適用実験

本項では，提案手法および DACARA の有効性を評価するための，オープンソースソフトウェアの集合を用いた適用実験について説明する．実験では以下の 2 項目を検証する．

E1 再利用単位は必要であるか．

仮に，列挙された依存クラスの候補集合から，名前が重複しないようにクラスを 1 つずつ選ぶことで，常に適切な依存関係を取得できるならば，依存単位の理解に再利用単位の抽出は不要である．そこで，上記の様な単純な方法では依存関係を正確に得られず，再利用単位が依存関係の理解に役立つ場合がどの程度存在するのか確認する．

E2 得られる再利用単位の数は現実的か．

提案手法により完全な再利用単位が容易に得られたとしても，その数が多ければ，その中からの選択は困難である．そこで，解析結果として得られる個数を，再利用単位解析部での 2 種類のフィルタリング F2，F3 の有効・無効を切り替えながら計測する．

3.5.1 実験対象

Java SE 1.2, 1.3, 1.4 の 3 バージョンそれぞれのソースコードおよび Apache commons のライブラリ 33 種類の異なるバージョンの，合計 127 パッケージ[3]を用いる．パッケージ一覧を表 3.2 に示す．これらを対象として選んだ理由は，互いに利用関係を持つことと，複

[3]それぞれ，`http://java.sun.com/products/archive/` より取得できる Linux 版 JDK 1.2.2, 1.3.1_20，1.4.2_16 および 2008 年 5 月の段階で，`http://commons.apache.org/` に “Proper” として列挙されているソフトウェアパッケージの，`http://archive.apache.org/dist/commons/`(パッケージ名)`/source/` から入手可能なバージョン．

数バージョンが入手可能であることである．なお，総ファイル数は 22,005，クラス数は 31,594，メンバ数は 368,089 である．

3.5.2 手順

まず，全てのクラスをデータベースへ登録し，参照解析を行う．このとき，フィルタリング F1 は有効とする．続いて，再利用単位を求める処理を全てのクラスに対して行う．入力の単位は，1 つのファイルに含まれるクラス集合とする．これは現実の再利用で用いやすいと考えられる単位である．再利用単位を求める処理は，フィルタリング無し，F2 のみ，F3 のみ，F2 および F3 の 4 通りを試行する．なお，フィルタリング F2, F3 に用いられる所属の同一性の判断は，部品の所属するアーカイブファイルが同一かどうかで行う．また，再利用単位の最大数は 1,000 とした．

実験では以下の 6 項目を計測する．

Classes 対象のファイル（クラス集合）が利用する全てのクラスの数．メンバ参照の特定のために経由した継承関係に含まれるクラスも含む

Names 上記のクラスの完全限定名の種類数

AllUnits 得られた全ての再利用単位の個数

Units 得られた完全な再利用単位の個数

MinElements 完全な再利用単位の要素（クラス集合）数の最小値

MaxElements 完全な再利用単位の要素数の最大値

また，項目 E1 のために，フィルタリング無しの結果に対し，以下の条件に合うファイル数を求める．

C1 AllUnits $\neq$ Units.
クラスの選び方によっては特定できない参照が生じることを意味する．

C2 Names $\neq$ MinElements $\vee$ Names $\neq$ MaxElements.
1 つの名前につき 1 クラスを選択する方法では依存関係として不要なクラスを含み，依存関係を正しく得るためには依存する部品同士の関係を考慮する必要があることを意味する．

3.5.3 実験結果

実験環境として，RAM を 16GB，CPU として Opteron252 を 2 個搭載したワークステーション上の Linux システムを用いた．RDBMS としては PostgreSQL 8.2.7 を，JRE のバージョンは 1.6.0 を用いた．

表 3.2: 実験対象パッケージ一覧

パッケージ名		バージョン
Java Development Kit		1.2.2, 1.3.1, 1,4.2
	attributes	2.1, 2.2
	beanutils	1.5, 1.6, 1.6.1,1.7.0, 1.8.0-BETA
	betwixt	0.5, 0.6, 0.7, 0.8, 1.0-alpha-1
	chain	1.0, 1.1
	cli	1.0, 1.1
	codec	1.1, 1.2, 1.3
	collections	1.0, 2.0, 2.1, 2.1.1, 3.0,3.1, 3.2, 3.2.1
	configuration	1.0, 1.1, 1.2, 1.3, 1.4,1.5
	dbcp	1.0, 1.1, 1.2, 1.2.1, 1.2.2
	dbutils	1.0,1.1
	digester	1.5, 1.6, 1.7, 1.8
commons	discovery	0.1, 0.2, 0.4
	el	1.0,
	email	1.0, 1.1
	fileupload	1.0, 1.0-beta-1, 1.0-rc1, 1.1, 1.1.1, 1.2, 1.2.1
	io	1.0, 1.1, 1.2, 1.3, 1.3.1, 1.3.2, 1.4
	jci	1.0,
	jelly	1.0, 1.0-RC1
	jexl	1.1,
	jxpath	1.0, 1.1, 1.2
	lang	1.0, 1.0.1, 2.0, 2.1, 2.2, 2.3, 2.4
	launcher	0.9, 1.1
	logging	1.0, 1.0.1, 1.0.2, 1.0.3, 1.0.4, 1.1, 1.1.1g

データベースへの部品登録に 34 分，参照の解析に 28 分を要した．また，1 つの再利用単位の解析は平均 0.1 秒未満であった．いずれも十分に現実的な処理時間であると考えられる．

Classes および Names の最小値，最大値，中央値，第 1・3 四分位点を表 3.3 に示す．なお，特定可能な参照を含まなかった 3 ファイルは除外して値を求めた．

E1 に対する結果

2,953 ファイルが条件 C1 を満たし，1,151 ファイルが条件 C2 を満たしていた．全体が 2 万であることを考慮すると，決して無視できないファイル数であり，提案手法，特に再利用単位の解析が要求される場面が多いことを示していると考えられる．さらに，Names がその中央値である 8 より大きいファイルに絞った場合も，2,483 ファイルが C1 を満たし，1,038 ファイルが C2 を満たしていた．つまり再利用単位は依存するクラス数が多い時ほど有用であるといえる．

C1 を満たす例として，commons-scxml-0.5 に含まれる SCInstance.java の結果を説明する．2 つの再利用単位が得られ，一方が不完全であった．それぞれ Exception の親クラスとして参照される Throwable の参照先が異なり，完全なものは Java SE 1.4 の Throwable クラスを，不完全なものは Java SE 1.3 以前の同名クラスを参照先としていた．また，C2 を満たす例として commons-fileupload-1.2.1 の DiskFileUpload.java を挙げる．2 通りの完全な再利用単位が得られ，それらの要素数は 7 個および 8 個であった．この違いは，DefaultFileItemFactory で参照されるクラスによって，DiskFileItemFactory クラスが必要な場合と必要でない場合があることに起因する．

E2 に対する結果

再利用単位数の度数分布図を図 3.8 に示す．全ての再利用単位に対する値は○印で，完全な再利用単位に対する値は×印で示されている．また，見やすさのために対数軸を用いている．これより，ほとんどのファイルに対して，少数の再利用単位を得られることが分かる．再利用単位の数が少ないことは，開発者にとって理解が容易であることを意味する．

フィルタリングの効果を調査するための散布図を図 3.9 に示す．図 3.9（a）は，点 1 つがファイルを示し，縦軸はファイルそれぞれのフィルタリング無しでの再利用単位の数を，横軸は F1 有効時の再利用単位の数を表す．見やすさのために両対数軸を用いている．他

表 3.3: Classes と Names の要約

	Min.	1st Qu.	Med.	3rd Qu.	Max.
Classes	1	10	22	41	371
Names	1	4	8	13	144

の図も同様である．図 3.9（a）から，F2 により，基本的に大きく個数を絞れるが，100 個以上のまま全く絞れないものも存在することが分かる．これは，同じ所属のクラスをあまり利用してない場合には効果が小さいためだと考えられる．一方，図 3.9（b）では，一部を除き 10 未満まで絞り込めている．これは所属の組み合わせはクラスの組み合わせより圧倒的に少ないことによると考えられる．そして，図 3.9（c）,（d）より，F2 と F3 は相補的に効果が現れるフィルタリングであり，組み合わせることで，大量に存在したものも現実的な範囲まで絞り込めることがわかる．

3.6 考察

3.6.1 API を用いた整合性の判定

本手法で抽出される再利用単位は，依存するクラス集合の整合性を参照の特定結果のみから判断しているため，他の制約を満たせていない可能性が存在する．例えば，図 3.3 のメンバ参照 r_3 の参照先は String 型で宣言されていなければならないが，本手法では参照間の代入可能性は考慮していないため，仮に m_2 が int 型で宣言されていたとしても，それを含む「完全」な再利用単位を出力してしまう．しかし，再利用単位には代替のクラスが含まれており，容易に入れ替えることができるため，実用上の問題にはならないと考えられる．また，この問題に本質的に対応するには，言語特有の要素を含め代入などにおける制約を抽出して全て検証する必要があり，現実的ではないと考えられる．

また，ある部品が依存する部品が部品データベース中に含まれない場合にも，依存部品を網羅した再利用単位を得ることができない．部品集合中から依存関係を抽出する本手法では本質的に対応できないが，参照先を特定できなかった参照を示すことにより，開発者は得られた再利用単位に欠けている依存関係を理解することができるため，実用上の問題とはならないと考えられる．

3.6.2 スケーラビリティ

実験で用いたデータセットは 2 万ファイル程度であり，現実のソフトウェア部品検索システムのデータベース，例えば SPARS-J[12] の 30 万ファイルと比較すると小規模である．しかし，提案手法の計算量は同一の名前で参照されるクラスの数に大きく影響を受ける．本研究の実験では 1 つのソフトウェアパッケージに対して複数のバージョンを対象としており，同じ名前で参照されるクラスが多く存在する状態を意図的に作り出している．このような環境でも実用的な時間で解析できており，十分な実用性があると考えられる．

3.6.3 名前の重複する部品の存在

提案手法の参照解析は，同じ名前で参照できる部品は潜在的に代替可能であるという前提のもと，候補の絞り込みはほとんど行っていない．しかし，同じ名前で参照される部品

には，偶然同じ名前を持つ無関係なものも存在する．このような部品が多い場合には，クラス参照の参照先が増えることにより処理時間に影響を与えるため，参照の特定におけるフィルタリングの改良が必要となる．ただし，DACARA が対象とする Java では，クラスの名前の重複を避けるために名前空間（パッケージ）の命名規約が設けられているため，問題は起きにくいと考えられる．実際に，適用実験においても問題になるほどの重複は起きていなかった．今後，C#などの，明確な基準が存在せず名前の重複が多く存在すると思われる言語へと拡張する際には，偶然同じ名前である部品を利用関係の候補から除去することを考慮する必要がある．

3.6.4 依存関係の理解

提案手法を用いて再利用単位の集合を得た利用者は，そこから部品を選択することで，目的の部品集合を得る．このとき，利用者はまず再利用単位を選択した上で，再利用単位の要素（依存グラフの頂点）に含まれる部品集合から部品をひとつえずつ選ぶ．適用実験でも示された通り，この手順を手作業で行うことが十分に現実的である個数にまで再利用単位は絞り込まれているが，結果の順位付けなどを行うことにより，効率的な作業が可能になると考えられる．順位付けの基準としては，部品がどの程度多くのソフトウェアや部品から利用されているかという一般性や，部品が最後に更新された日付が挙げられる．また，非推奨扱いの API を用いる部品の優先度を下げることも考えられる．このように適切に順位付けすることで，利用者は再利用に適した部品を効率的に選択できると考えられる．

また，ユーザインタフェースの改良が挙げられる．例えば，既に所持しているライブラリなど，利用するソフトウェアパッケージを指定することで，再利用単位の選択肢を絞るとともに，再帰的に適用する際に不要な解析を行わないことが考えられる．

3.7 関連研究

3.7.1 ソフトウェア部品検索システム

再利用はソフトウェア開発の効率化に有効であるが，そのためには，効率的に再利用可能な部品を発見し，ソフトウェアへ組み込むための情報を得なければならない [18]．計画的に再利用可能な部品を生産し，それらの部品を組織的に利用することで，効果的な再利用が実現できると言われている [25, 30]．ただし，これらの手法には組織としての投資が必要となる．一方，必要に応じて既存のソフトウェアから部品を取り出して再利用する，軽量な再利用というアプローチも存在する [26]．オープンソースによるソフトウェア開発の事例が増えるにつれ，再利用可能なソースコード（部品）も増えており，軽量な再利用を支援できるような検索システムが構築されるようになった．

SPARS-J [29, 72] はそのひとつであり，利用頻度に基づいて部品の再利用性を評価する Component Rank 法が特徴である．また，Google code search [5] は正規表現による柔軟な

検索を提供しており，Koders [9] はオープンソースソフトウェアの開発リポジトリの連携を特徴としている．

3.7.2 依存関係の理解支援

このように，部品を効率的に取得するための研究およびツールは複数存在するが，ソフトウェアへ組み込むための情報，特に依存関係に関する情報の取得に関しては十分な支援ができていない．SPARS-J は対象の部品が利用する部品集合を解析しているが，参照先は参照と 1 対 1 の関連で特定されているため，代替の依存関係を取得できない．また，Koders および Google code search では識別子の検索の形式で参照関係を利用できるが，単語による検索に基づいているため，結果には多くの無関係な候補を含む．

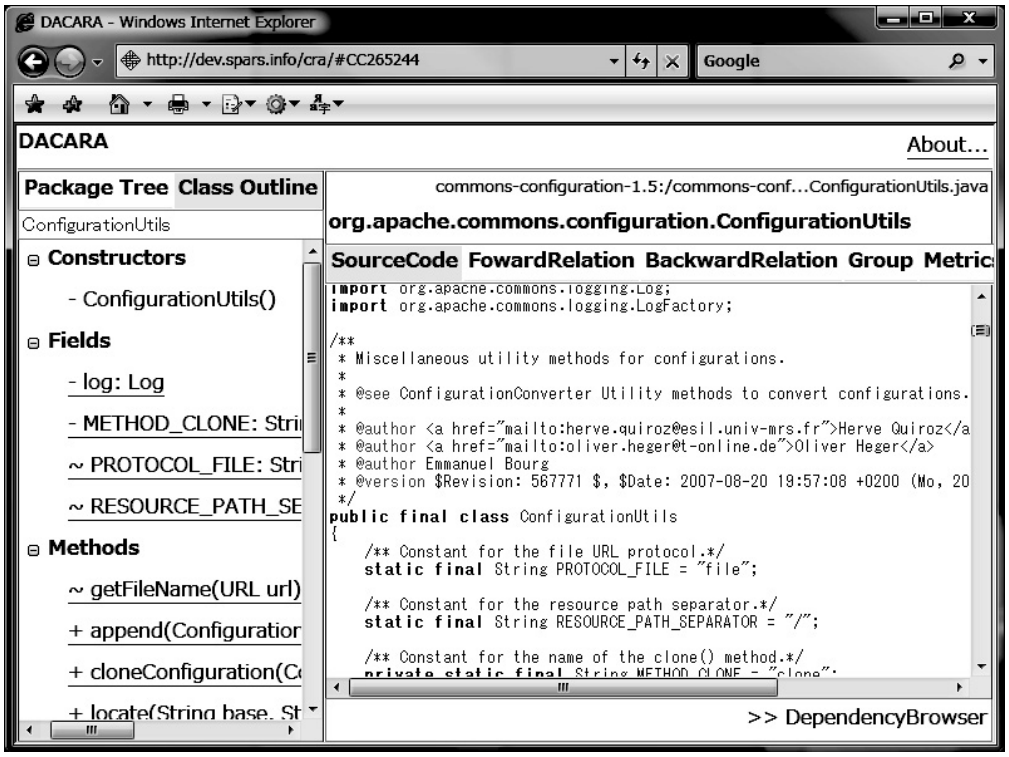
Ctags [3] は与えられたソースファイル集合に含まれる識別子の関係を解析するシステムであり，開発環境や検索システムに組み込んで利用される．また，CREBASS [48] は，版管理システムのリポジトリを解析し，時系列を考慮した参照関係を得ることができる．いずれも，ビルド単位を解析することを暗黙の前提としており，同名の部品の存在は考慮されていない．

Holmes らは，部品を再利用する際に必要となる依存関係を効率的に解決する手法を提案している [26]．この手法は，再利用対象の部品が依存する部品それぞれに対して，利用するか，利用せずに再利用対象の部品を書き換える手順を支援する手法である．再利用対象が依存する部品集合は，あらかじめ与えておく必要がある．また，鷲崎らは，ソフトウェアの中から再利用可能な部品集合を自動的に抽出する手法を提案している [66]．鷲崎らの手法では，再利用対象の部品を再利用するために必要な部品集合の特定および JavaBeans として利用するためのリファクタリングが行われる．これらの手法は，ともに，ソフトウェアをまたがる依存関係や，名前の重複する部品の存在は考慮されていない．一方，提案手法は，依存関係の修正の支援や変更を伴う部品集合の抽出は行わないが，ソフトウェアをまたがる依存関係を適切に処理することができる．提案手法は，これらの手法の適用に必要な依存関係を提供することができ，これらの手法と提案手法とは相補的な手法であるといえる．

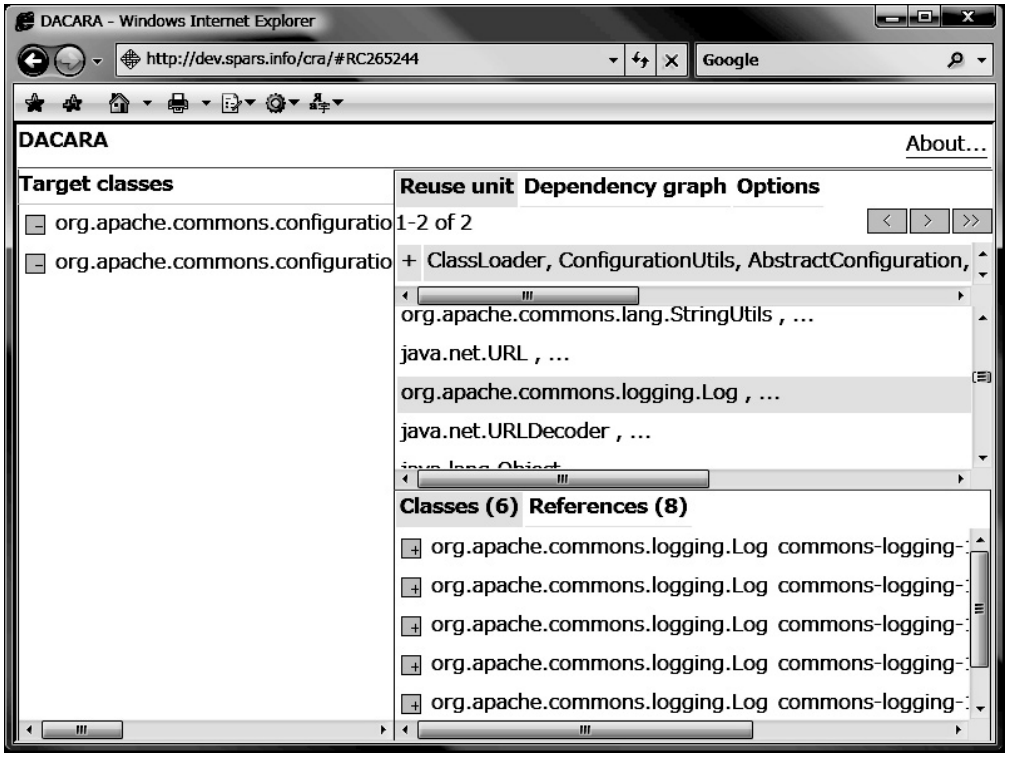
3.8 結論と課題

本章では，ソフトウェア部品検索システムを用いて部品を再利用する際に問題となる，依存関係の理解を支援するための手法を提案し，依存関係解析システム DACARA として実装した．また，提案手法を有効に生かせる場面が多く存在すること，および，フィルタリングが有効に働くことを，適用実験により示した．

今後の課題としては，利用者の利便性の向上のための，再利用単位の順位付けやユーザインタフェースの改良が挙げられる．また，Java 以外の言語への対応も課題である．最後に，既存のソフトウェア部品検索システムとの統合が挙げられる．

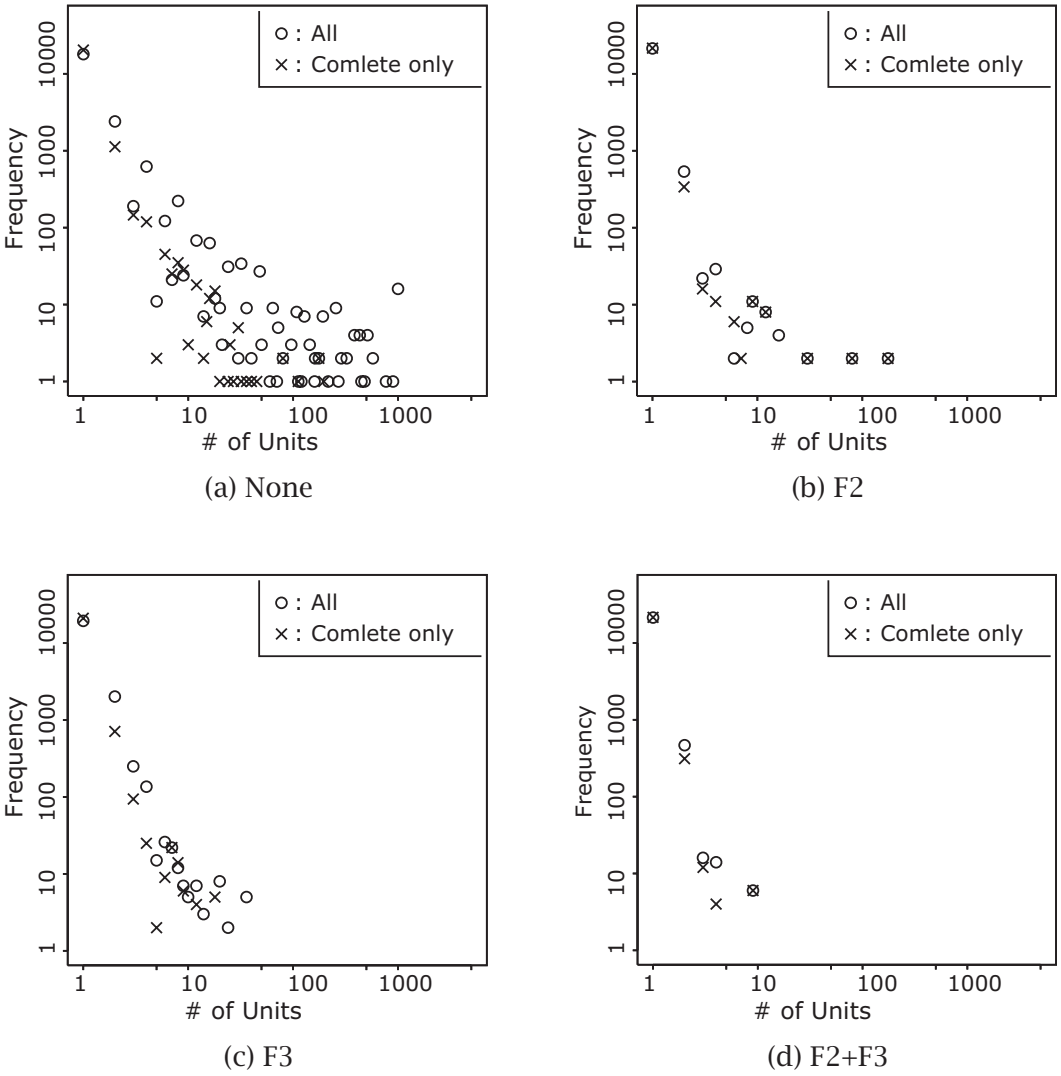


(a) Component View



(b) Dependency View

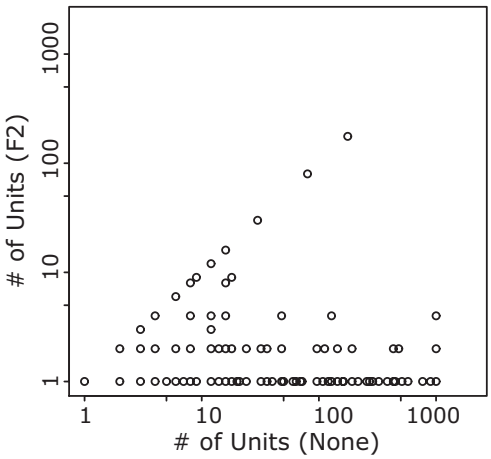
図 3.7: DACARA のスクリーンショット



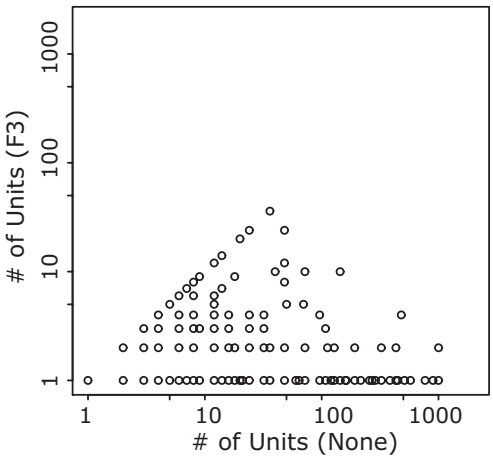
(a) None (b) F2

(c) F3 (d) F2+F3

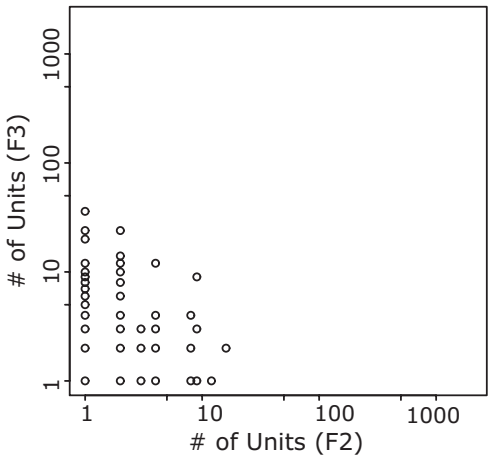
図 3.8: 再利用単位数の度数分布図



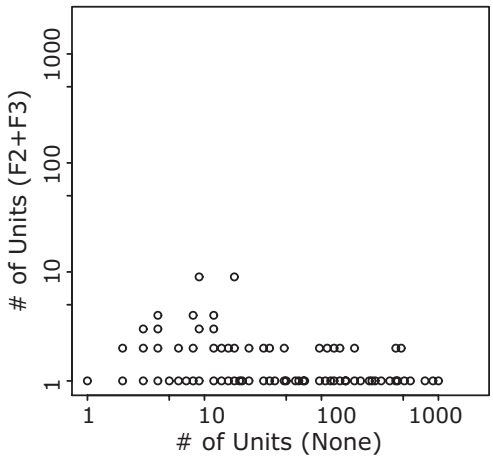
(a) None - F2



(b) None - F3



(c) F2 - F3



(d) None - F2+F3

図 3.9: フィルタリングが再利用単位数に与える影響

第4章　むすび

4.1　まとめ

本研究では，ソフトウェア部品検索システムのもつソフトウェアリポジトリに含まれる部品間の依存関係に着目した調査および再利用支援手法を提案した．

部品間の依存関係をグラフで表現した部品グラフの調査では，まず，単一のソフトウェアを解析して構築した部品グラフの入次数にはべき乗則が成り立ち，出次数には成り立たないことを示した．続いて，複数の互いに関連をもつソフトウェアを解析して構築した部品グラフも，同様に，入次数はべき乗則に従い，出次数は従わないことを示した．さらに，その部分集合により構築した部品グラフでは，構築方法によって結果が異なり，ソースコード上に含まれる単語に基づいて部品集合を構成したものに関しては，もとの集合の部品グラフと同様の性質が示された．また，これらの結果に関して考察を行った．

次に，再利用支援に関しては，ソフトウェアリポジトリ中にはソフトウェアをまたがった依存関係が存在すること，および，類似した API をもつ部品が存在することを考慮し，再利用対象の部品が依存する部品集合の候補を抽出する手法を提案した．また，ソフトウェア部品検索システムと連携するシステム DACARA として実装し，オープンソースソフトウェアを用いた実験により，効果的な理解支援を行えることを示した．

4.2　今後の研究方針

今後の研究として，部品の変更履歴を考慮したソフトウェア部品検索システムの構築を考えている．これは，2 通りのアプローチを考えている．

まず，本研究では類似した API を考慮しつつ再利用対象の部品が依存する部品集合の候補を示す手法を提案したが，候補の中で選択的な部品同士は同じ部品の版違いや派生した部品であることが多いと考えられる．それらの関係は利用者が部品を選択する上で有益な情報となり得る．そこで，同一のソフトウェアリポジトリに含まれる部品の中から，同じ部品の版違いを抽出し，変更の前後関係を解析することで，類似したもののなかでも新しいものや安定したものなど，開発者の要求に適合する部品を取得しやすくなると考えられる．

また，ソースコードの変更を管理する，版管理システムと連携し，過去の版を取得できるようにすることも考えている．直接的に版違いの部品を取得できることで，例えば取得した部品に不具合が含まれていた場合に，その不具合の修正された版，もしくは不具合が

発生する前の版を取得でき，より柔軟な再利用が実現できる．課題として，版管理システムに格納された大量の履歴をすべて取得すると膨大な量になってしまうため，変更内容などに応じたフィルタリングが必要であると考えられる．また，本研究で提案した依存関係解析手法では，依存関係の候補すべてをデータベースに格納していたが，版の組み合わせすべてについて格納すると膨大な量になるため，差異の小さな部品をグループ化するなどして，有効性を保ちつつスケーラビリティを向上させる工夫が必要となる．

参考文献

[1] Apache Ant. http://ant.apache.org.

[2] The Apache Software Foundation. http://www.apache.org.

[3] Ctags. http://ctags.sourceforge.net.

[4] Eclipse. http://www.eclipse.org.

[5] Google code search. http://www.google.com/codesearch.

[6] Google web toolkit. http://code.google.com/webtoolkit/.

[7] Java technology. http://java.sun.com.

[8] JBoss. http://www.jboss.org.

[9] Koders. http://www.koders.com.

[10] NetBeans. http://www.netbeans.org.

[11] SourceForge. http://sourceforge.net.

[12] SPARS-J. http://demo.spars.info.

[13] Réka Albert, Hawoong Jeong, and Albert-László Barabási. Diameter of the world-wide web. *Nature*, Vol. 401, No. 6749, pp. 130–131, September 1999.

[14] Giuliano Antoniol, Roberto Fiutem, and L. Cristoforetti. Design pattern recovery in object-oriented software. In *Proceedings of the 6th International Conference on Programming Comprehension (IWPC'98)*, pp. 153–160, June 1998.

[15] Sushil Bajracharya, Trung Ngo, Erik Linstead, Yimeng Dou, Paul Rigor, Pierre Baldi, and Cristina Lopes. Sourcerer: A search engine for open source code supporting structure-based search. In *Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'06)*, pp. 25–26, October 2006.

[16] Albert-László Barabási. *Linked: The New Science of Networks*. NHK 出版, 2002.

[17] Gareth Baxter, Marcus Frean, James Noble, Mark Rickerby, Hayden Smith, Matt Visser, Hayden Melton, and Ewan Tempero. Understanding the shape of Java software. In *Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'06)*, pp. 397–412, October 2006.

[18] Christine L. Braun. Reuse. In *in Encyclopedia of Software Engineering 2nd ed.*, pp. 1197–1214. John Wiley & Sons, 2002.

[19] Lionel C. Briand, John W. Daly, and Jurgen Wust. A unified framework for cohesion measurement in object-oriented systems. *Empirical Software Engineering*, Vol. 3, No. 1, pp. 650–117, 1998.

[20] Alexander Chatzigeorgiou, Spiros Xanthos, and George Stephanides. Evaluating object-oriented designs with link analysis. In *Proceedings of the 26th International Conference on Software Engineering (ICSE'04)*, pp. 533–542, 2004.

[21] Shyam R. Chidamber and Chris F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, Vol. 20, No. 6, pp. 476–493, June 1994.

[22] Giulio Concas, Michele Marchesi, Sandro Pinna, and Nicola Serra. Power-laws in a large object-oriented software system. *IEEE Transactions on Software Engineering*, Vol. 33, No. 10, pp. 687–708, October 2007.

[23] Danny Dig and Ralph Johnson. How do APIs evolve? a story of refactoring. *Journal of Software Maintenance and Evolution: Research and Practice*, Vol. 18, No. 2, pp. 83–107, April 2006.

[24] Martin Fowler. *Refactoring: Improving The Design of Existing Code*. ピアソン・エデュケーション, 2000.

[25] Jack Greenfield and Keith Short. *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools*. Addison Wesley, 2004.

[26] Reid Holmes and Robert J. Walker. Supporting the investigation and planning of pragmatic reuse tasks. In *Proceedings of the 29th International Conference on Software Engineering (ICSE'07)*, pp. 447–456, May 2007.

[27] Reid Holmes, Robert J. Walker, and Gail C. Murphy. Approximate structural context matching: An approach to recommend relevant examples. *IEEE Transactions on Software Engineering*, Vol. 32, No. 12, pp. 952–970, December 2006.

[28] Oliver Hummel and Colin Atkinson. Using the web as a reuse repository. In *Proceedings of the 9th International Conference on Software Reuse (ICSR'06)*, pp. 217–230, July 2006.

[29] Katsuro Inoue, Reishi Yokomori, Tetsuo Yamamoto, Makoto Matsushita, and Shinji Kusumoto. Ranking significance of software components based on use relations. *IEEE Transactions on Software Engineering*, Vol. 31, No. 3, pp. 213–225, March 2005.

[30] I. Jacobson, M. Griss, and P. Jonsson. *Software Reuse*. Addison Wesley, 1997.

[31] Stanley M. Sutton Jr. and Isabelle Rouvellou. Concern modeling for aspect-oriented software development. In Robert E. Filman, Tzilla Elrad, Siobhán Clarke, and Mehmet Ak sit, editors, *Aspect-Oriented Software Development*, chapter 21, pp. 479–505. Addison Wesley, 2005.

[32] Miryung Kim, David Notkin, and Dan Grossman. Automatic inference of structural changes for matching across program versions. In *Proceedings of the 29th International Conference on Software Engineering (ICSE'07)*, pp. 333–343, May 2007.

[33] J. Kleinberg. Authoritative sources in a hyperlinked environment. *Journal of the ACM*, Vol. 46, No. 5, pp. 604–632, September 1999.

[34] 久米均, 飯塚悦功. 入門 統計的方法 2 回帰分析. 岩波書店, 1987.

[35] Ge Li, Lu Zhang, Yan Li, Bing Xie, and Weizhong Shao. Shortening retrieval sequences in browsing-based component retrieval using information entropy. *Journal of Systems and Software*, Vol. 79, No. 2, pp. 216–230, February 2006.

[36] Raphael C. Malveau, III McCormick Hays W, Thomas J. Mowbray, and William J. Brown. *Antipatterns: Refactoring Software, Architectures, and Projects in Crisis*. John Wiley & Sons, 1998.

[37] David Mandelin, Lin Xu, Rastislav Bodik, and Doug Kimelman. Jungloid mining: Helping to navigate the API jungle. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation (PLDI'05)*, pp. 48–61, June 2005.

[38] Marius Marin, Arie Van Deursen, and Leon Moonen. Identifying crosscutting concerns using fan-in analysis. Vol. 17, No. 1, p. 3, December 2007.

[39] 増田直紀, 今野紀雄. 複雑ネットワークの科学. 産業図書, 2005.

[40] Tom Mens and Tom Tourwé. A survey of software refactoring. *IEEE Transactions on Software Engineering*, Vol. 30, No. 2, pp. 126–139, February 2004.

[41] Amir Michail. Data mining library reuse patterns using generalized association rules. In *Proceedings of the 22nd International Conference on Software Engineering (ICSE'00)*, pp. 167–176, June 2000.

[42] Hafedh Mili, Fatma Mili, and Ali Mili. Reusing software: Issues and research directions. *IEEE Transactions on Software Engineering*, Vol. 21, No. 6, pp. 528–561, June 1995.

[43] Brian S. Mitchell and Spiros Mancoridis. On the automatic modularization of software systems using the bunch tool. *IEEE Transactions on Software Engineering*, Vol. 32, No. 3, pp. 193–208, March 2006.

[44] Michael Mitzenmacher. Dynamic models for file sizes and double pareto distributions. *Internet Mathematics*, Vol. 1, No. 3, pp. 305–333, 2003.

[45] 三宅達也, 肥後芳樹, 井上克郎. メトリクス計測プラグインプラットフォーム MASU の開発. ソフトウェアエンジニアリングシンポジウム 2008 論文集, pp. 63–70, September 2008.

[46] Brandon Morel and Perry Alexander. Spartacas: Automating component reuse and adaptation. *IEEE Transactions on Software Engineering*, Vol. 30, No. 9, pp. 587–600, September 2004.

[47] Christopher R. Myers. Software systems as complex networks: Structure, function, and evolvability of software collaboration graphs. *Physical Review E*, Vol. 68, No. 4, p. 046116, 2003.

[48] 中山崇, 松下誠, 井上克郎. 関数の変更履歴と呼出し関係に基づいた開発履歴理解支援システム. 電子情報通信学会技術研究報告, Vol. 104, No. 47, pp. 7–12, March 2004.

[49] M. E. J. Newman. The structure of scientific collaboration networks. *Proceedings of the National Academy of Sciences of the United States of Americs*, Vol. 98, No. 2, pp. 409–415, 2001.

[50] M. E. J. Newman. Power laws, pareto distributions and Zipf's law. *Contemporary Physics*, Vol. 46, pp. 323–351, 2005.

[51] Jörg Niere, Wilhelm Schäfer, Jörg P. Wadsack, Lothar Wendehals, and Jim Welsh. Towards pattern-based design recovery. In *Proceedings of the 24th International Conference on Software Engineering (ICSE'02)*, pp. 338–348, May 2002.

[52] 大須賀俊憲, 金子伸幸, 山本晋一郎, 小林隆志, 阿草清滋. ソフトウェア統合検索を利用した再利用支援システム. ソフトウェア工学の基礎 XIV, pp. 203–208, November 2007.

[53] Young Park. Software retrieval by samples using concept analysis. *Journal of Systems and Software*, Vol. 54, No. 3, pp. 179–183, November 2000.

[54] Alex Potanin, James Noble, and Marcus Frean. Scale-free geometry in OO programs. *Communications of the ACM*, Vol. 48, No. 5, pp. 99–103, May 2005.

[55] 仁井谷竜介, 松下誠, 井上克郎. ソースコードの特徴語を用いた Java ソフトウェア部品の自動分類手法の提案. 情報処理学会研究報告, 2006-SE-149, pp. 49–56, July 2005.

[56] Santonu Sarkar, Girish Maskeri Rama, and Avinash C. Kak. API-based and information-theoretic metrics for measuring the quality of software modularization. *IEEE Transactions on Software Engineering*, Vol. 33, No. 1, pp. 14–32, January 2007.

[57] Kamran Sartipi and Kostas Kontogiannis. On modeling software architecture recovery as graph matching. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM'03)*, pp. 224–234, September 2003.

[58] Richard W. Selby. Enabling reuse-based software development of large-scale systems. *IEEE Transactions on Software Engineering*, Vol. 31, No. 6, pp. 495–510, June 2005.

[59] Arun Sharma, Rajesh Kumar, and P. S. Grover. A critical survey of reusability aspects for component-based systems. In *Proceedings of World Adacemy of Science, Engineering, and Technology Volume 21*, pp. 411–415, January 2007.

[60] 山田茂, 高橋宗雄. ソフトウェアマネジメントモデル入門. 共立出版, 1993.

[61] 島田隆次, 市井誠, 早瀬康裕, 松下誠, 井上克郎. ソースコードの編集内容を用いたソフトウェア部品の自動推薦手法. 情報処理学会研究報告, 2008-SE-162, pp. 31–38, November 2008.

[62] Jonathan Sillito, Gail C. Murphy, and Kris De Volder. Asking and answering questions during a programming change task. *IEEE Transactions on Software Engineering*, Vol. 34, No. 4, pp. 434–451, July 2008.

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

[64] Paolo Tonella and Giulio Antoniol. Object oriented design pattern inference. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM'99)*, pp. 230–238, September 1999.

[65] Sergi Valverde, Ramon Ferrer-Cancho, and Ricard V. Solé. Scale-free networks from optimal design. *Europhysics Letters*, Vol. 60, No. 4, pp. 512–517, 2002.

[66] Hironori Washizaki and Yoshiaki Fukazawa. A technique for automatic component extraction from object-oriented programs by refactoring. *Science of Computer Programming*, Vol. 56, No. 1-2, pp. 99–116, 2005.

[67] 鷲崎弘宜, 深澤良彰. 有向置換性類似度に基づくコンポーネント検索方式の実現と評価. 情報処理学会論文誌, Vol. 43, No. 16, pp. 1638–1652, June 2002.

[68] Duncan J. Watts and Steven H. Strogatz. Collective dynamics of 'small-world' networks. *Nature*, Vol. 393, No. 6684, pp. 440–442, June 1998.

[69] Richard Wheeldon and Steve Counsell. Power law distributions in class relationships. In *Proceedings of the Third IEEE International Workshop on Source Code Analysis and Manipulation (SCAM'03)*, pp. 45–57, 2003.

[70] Yunwen Ye and Gerhard Fischer. Reuse-conducive development environments. *Journal of Automated Software Engineering*, Vol. 12, No. 2, pp. 199–235, April 2005.

[71] 横森励士, 藤原晃, 山本哲男, 松下誠, 楠本真二, 井上克郎. 利用実績に基づくソフトウェア部品重要度評価システム. 電子情報通信学会論文誌 D-1, Vol. J86-D-I, No. 9, pp. 671–681, 2003.

[72] 横森励士, 梅森文彰, 西秀雄, 山本哲男, 松下誠, 楠本真二, 井上克郎. Java ソフトウェア部品検索システム SPARS-J. 電子情報通信学会論文誌 D-1, Vol. J87-D-I, No. 12, pp. 1060–1068, 2004.

[73] 木下喜幸, 玉井哲雄. グラフ手法による Java プログラムの構造と構造変化の分析. 情報処理学会研究報告, 2006-SE-152, pp. 41–48, May 2006.