# 依存関係グラフを用いたソフトウェア保守の効率化に関する研究

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 井上研究室

小林 健一

Software Engineering Laboratory

*Department of Computer Science,*
*Graduate School of Information Science & Technology,*
*Osaka University*

# 略歴

**氏名：** 小林　健一

## 学歴

- 1992 東京大学 計数工学科 卒業 (指導教官:森下巖教授)
  - テーマ: マルチプロセッサ上の軽量コンテクストスイッチ機構
- 1994 東京大学大学院 情報工学専攻 修士課程 修了 (指導教官:田中英彦教授)
  - テーマ: 並列論理型言語の分散並列コンピュータ向け高効率実装
- 2021 大阪大学大学院 情報科学研究科 博士後期課程 (指導教官:井上克郎教授)
  - テーマ: 依存関係グラフを用いたソフトウェア保守の効率化に関する研究

## 職歴

- 1994～現在 富士通研究所＆富士通
  - HPC・ソフトウェア工学・機械学習の研究開発に従事
  - 現在，AI品質の研究に従事
- 2000 米国 HALコンピューターシステムズ 招聘エンジニア
- 2008 オーストラリア国立情報通信技術研究所 (NICTA) 客員研究員

# 関連研究

## 論文

- 小林健一, 松尾昭彦, 井上克郎, 早瀬康裕, 上村学, 吉野利明,
  **大規模ソフトウェア保守のための影響波及量尺度インパクトスケール**,
  情報処理学会論文誌, vol. 54, no. 2, pp. 870-882, 2013. 情報処理学会論文賞.
  2章
- 門田暁人, 小林健一,
  **線形重回帰モデルを用いたソフトウェア開発工数予測における対数変換の効果**,
  コンピュータソフトウェア誌, Vol.27, No.4, pp.234-239, 2010.
  3章
- 小林健一, 松尾昭彦, 松下誠, 井上克郎,
  **SArF: 依存関係に基づいてフィーチャーを集めるソフトウェアクラスタリング**,
  (情報処理学会論文誌に投稿中).
  4章

## 国際会議

- Kobayashi, Matsuo, Inoue, Hayase, Kamimura, and Yoshino,
  **ImpactScale: Quantifying change impact to predict faults in large software systems**,
  IEEE International Conference on Software Maintenance (ICSM), pp. 43-52, 2011.
  2章
- Kobayashi, Kamimura, Kato, Yano, and Matsuo,
  **Feature-gathering dependency-based software clustering using dedication and modularity**,
  IEEE International Conference on Software Maintenance (ICSM), pp.462-471, 2012.
  4章
- Kobayashi, Kamimura, Yano, Kato, and Matsuo,
  **SArF Map: Visualizing Software Architecture from Feature and Layer Viewpoints,**
  IEEE International Conference on Program Comprehension (ICPC), pp.43-52, 2013.
  5章

# ソフトウェア保守とその課題

**長期間**，社会を支えるソフトウェアシステム ⇒ 大規模化・複雑化


維持 & 改善

ソフトウェア保守

欠陥修正 | 予防 | 改善 | 適応

変更・障害管理 | 欠陥予測 | 工数予測 | リポジトリマイニング
理解 | コード解析 | フィーチャーロケーション | コード来歴 | テスト
進化 | 移行 & 更新 | リファクタリング | コードクローン
実証的研究 | 品質評価 | プロセス | CI & CD | 人的側面

**課題:**

- **知識の損失** (ドキュメントと実装との乖離，開発者の退職など)
- **コスト・工数の制約**

# 本研究の目的とアプローチ

- 目的
  - 継続的に知識の損失が起きるソフトウェア保守の環境下で，<u>適用可能性の高い</u>ソフトウェア保守の効率化技術を開発

- アプローチ
  - ソースコード内の依存関係(呼出やアクセス)を活用
    - 静的な依存関係はソフトウェア保守のほとんどの局面で抽出が可能
  - 課題「知識の損失」に対し
    - アーキテクチャレベルの知識を復元
  - 課題「コスト・工数の制約」に対し
    - 欠陥・工数予測によるリソース割当の最適化
    - 復元した知識による効率化

# 学位論文の構成

依存関係グラフを用いたソフトウェア保守の効率化に関する研究

1章 緒言
2章 依存関係グラフを用いた欠陥予測
3章 工数予測の精度向上のための対数変換と補正方法
4章 依存関係グラフを用いたソフトウェアアーキテクチャの復元
5章 依存関係グラフを用いたソフトウェアアーキテクチャの可視化
6章 結言

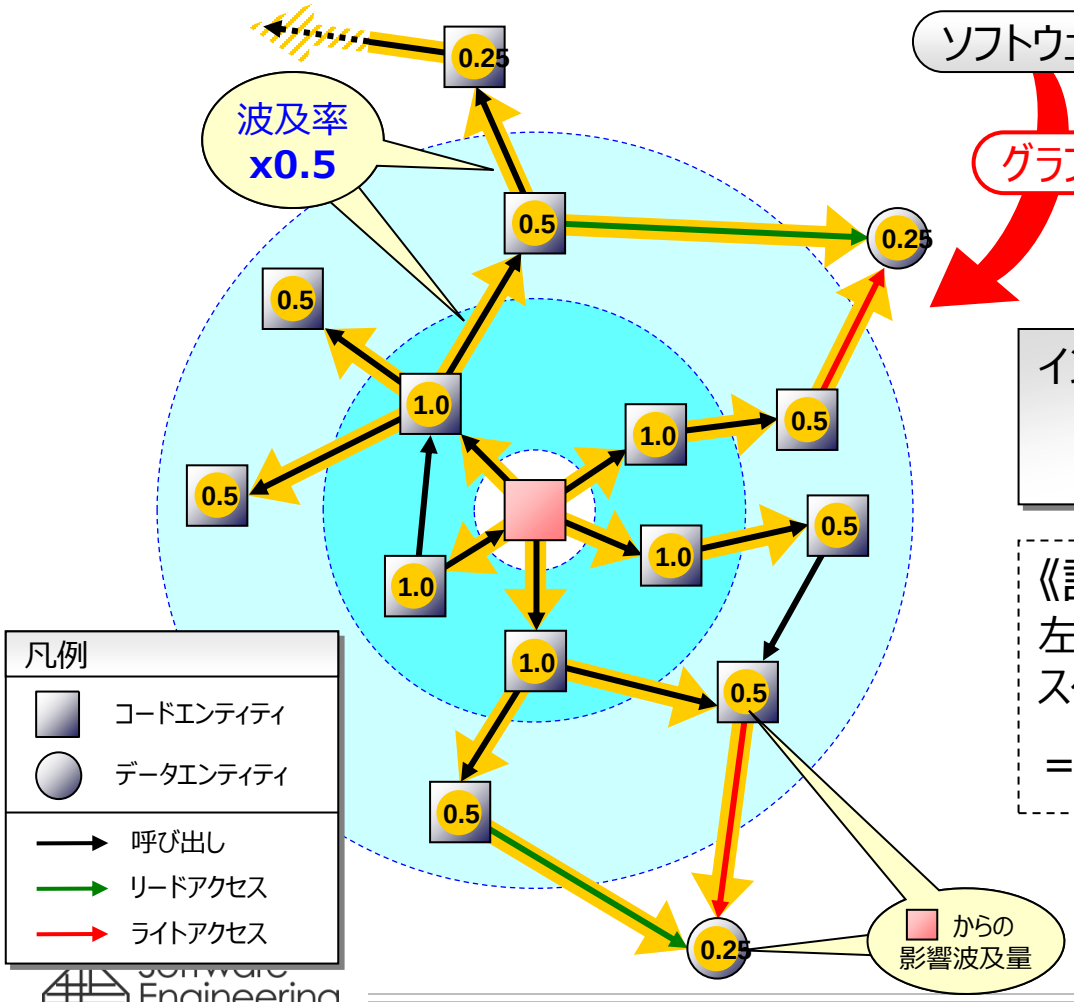
# 2章 依存関係グラフを用いた欠陥予測

# 背景と目的

- 背景
  - 保守の欠陥予測は難しい
    - 「リリース前欠陥と相関のあったメトリクスは，リリース後欠陥とは相関が無かった」[Fenton2000]
    - ソースコードのみから抽出可能なプロダクトメトリクスだけでは予測性能が不十分
  - しかし，実際の保守の現場では知識は常に失われ続けるため，他の情報が利用できないことが多い
- 目的
  - プロダクトメトリクスのみで精度の高い障害予測を実現したい

# インパクトスケール

ソフトウェアの一部（エンティティ）を変更した際に，その影響が他のどれだけの箇所に波及する可能性があるかを表す量（影響波及量）




ソフトウェアを解析して影響波及関係を抽出

グラフ構造で表現

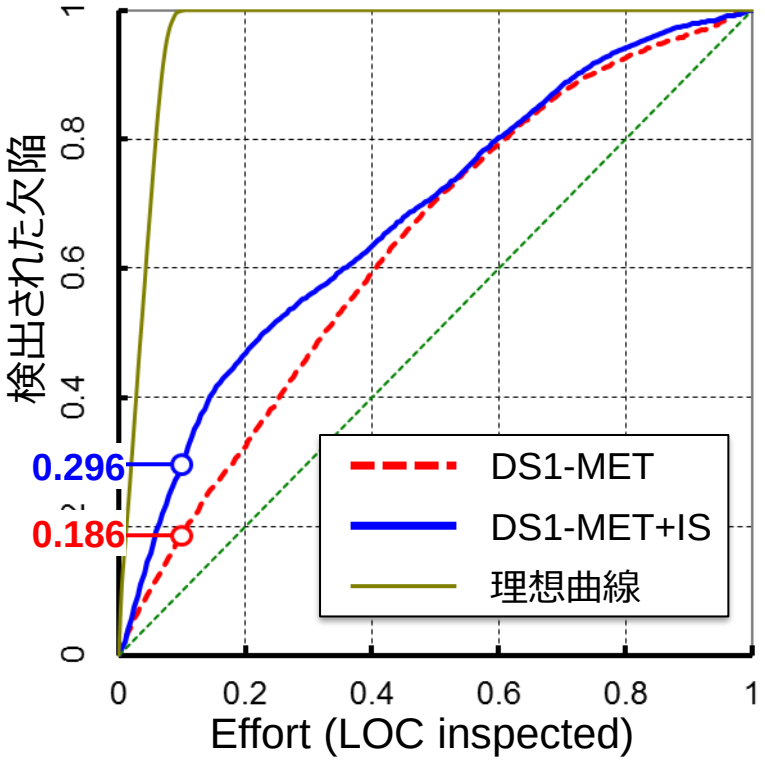
インパクトスケール
= 対象のエンティティから個々のエンティティへの影響波及量の合計

《計算例》
左図のコードエンティティ■のインパクトスケール

＝ 1.0×5個＋ 0.5×7個＋ 0.25×3個＋・・・

# 工数考慮モデルによる評価結果

《DS1の工数ベース累積リフトチャート》




| 性能尺度 | DS1-MET | DS1-MET+IS | ISによる向上 |
|---|---|---|---|
| $ddr_{10}$ | 0.186 | 0.296 | ×1.60 |

性能向上はウィルコクソンの符号付順位検定により統計的に有意

性能尺度 **$ddr_{10}$** は最初の工数10%における障害検出率．$ddr_{10}$ は限られた工数の中での予測性能を示す．高い $ddr_{10}$ は高い予測性能を意味する．

保守では工数，スケジュール，予算は常に限られている･･･

障害予測＋工数考慮モデル＋インパクトスケールを用いることで，工夫しない場合の **2.96**倍の効率化

インパクトスケールを導入すると，1.60倍の効率化

# まとめ

- ソースコードのみから抽出可能な影響波及量を定量化するメトリクス「インパクトスケール(IS)」を定義
- インパクトスケールはソフトウェア保守においても，欠陥と相関があることを確認
- 工数を考慮した欠陥予測モデルにて，効果を実証
  - 予測モデル無しに比べて，IS導入の工数効率は約3倍
  - 既存のメトリクスによる予測モデルに比べても，1.6倍

# 3章　工数予測の精度向上のための対数変換と補正方法

# 目的・分析

- 目的
  - ソフトウェア開発・保守において工数予測は役立つ
    - スケジュール管理や予算割り当て，優先順位付けなど
  - 本章では，工数予測の精度向上のための知見を提供

- 分析
  - 工数予測では指数曲線モデルが有効なことが多い
    - 工数実績は大プロジェクトほど分散が大きい傾向
    - プロジェクトの特性値は工数に対して積算的に働くことが多い
  - 指数曲線モデルは，応答変数や説明変数を対数変換すると，取り扱いやすい線形モデルとして扱うことができる．

# 知見・実験・結果

- 知見
  - 単に対数変換を施すだけでは，工数を過小評価するバイアスが発生することは，見過ごされがちである.
  - 対数変換により得られた線形モデルから得られた予測工数$\hat{Y}$に対して，バイアスを補正した予測工数$\hat{Y}^*$は以下となる．(SEEは推定値標準誤差)

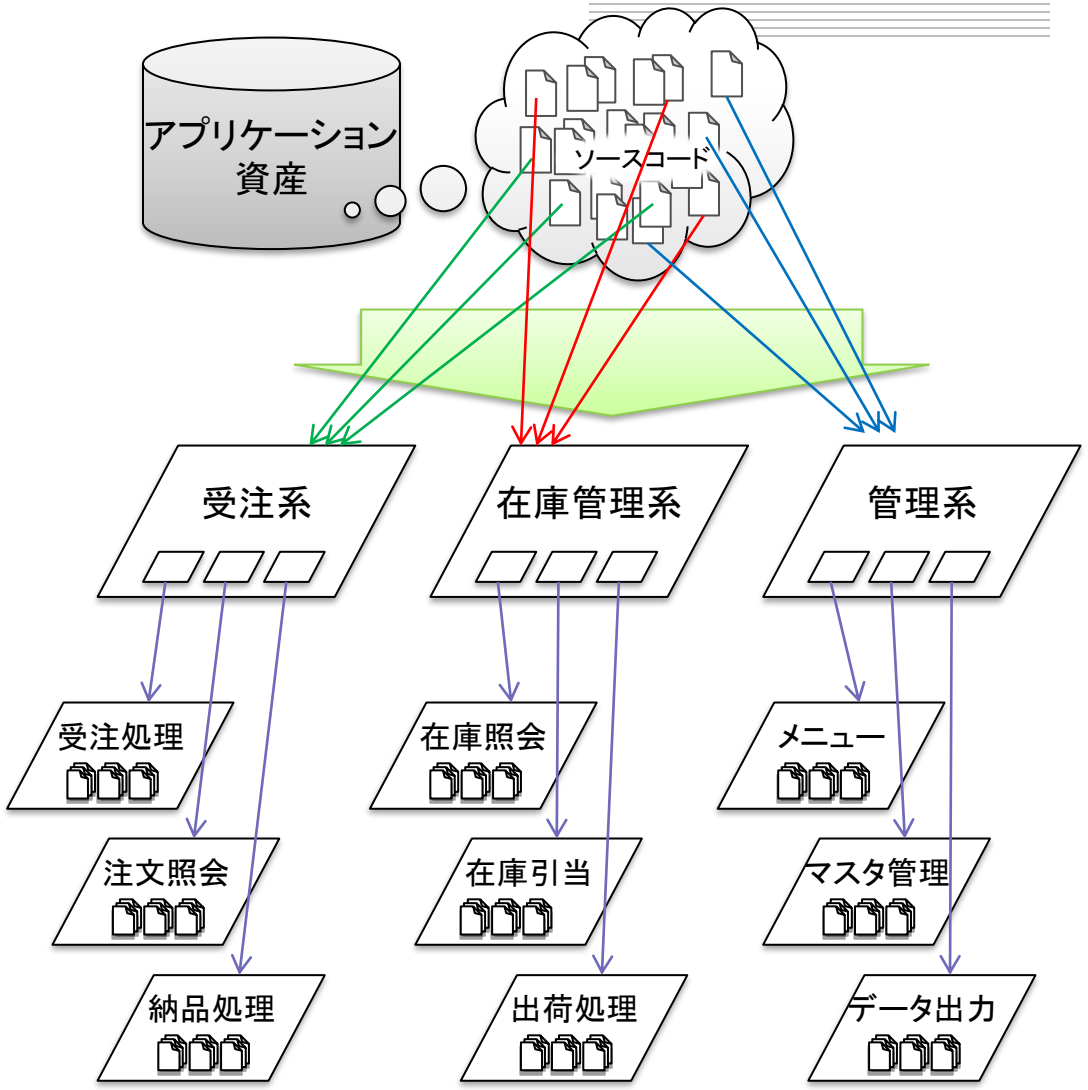
$$\hat{Y}^* = \exp\left(\frac{(\mathrm{SEE})^2}{2}\right) \cdot \hat{Y}$$

- 実験と結果
  - ソフトウエアプロジェクトの実績データであるDesharnaisデータセット用いて，工数予測を行った.
  - 線形モデルよりも，対数変換を施した指数曲線モデルの方が予測精度がよく，しかし，上記補正なしでは過小評価が生じていることを確認した.
  - また，上記補正により，バイアスが解消したことを確認した.

# 4章 依存関係グラフを用いた ソフトウェアアーキテクチャの復元

# 動機

- ソフトウェアシステムに対し，高いレベルの意思決定を行うには，どのような「機能」を，どこで実現しているか，を知る必要がある.
- 長期間運用している大規模ソフトウェアでは，複雑・多機能だが，ドキュメントは無いか古い.
- よって，これらをソースコードから自動で発見したい.



# 本章のゴール

以下の特徴を持つ，ソフトウェアクラスタリング手法 **SArF** を提案

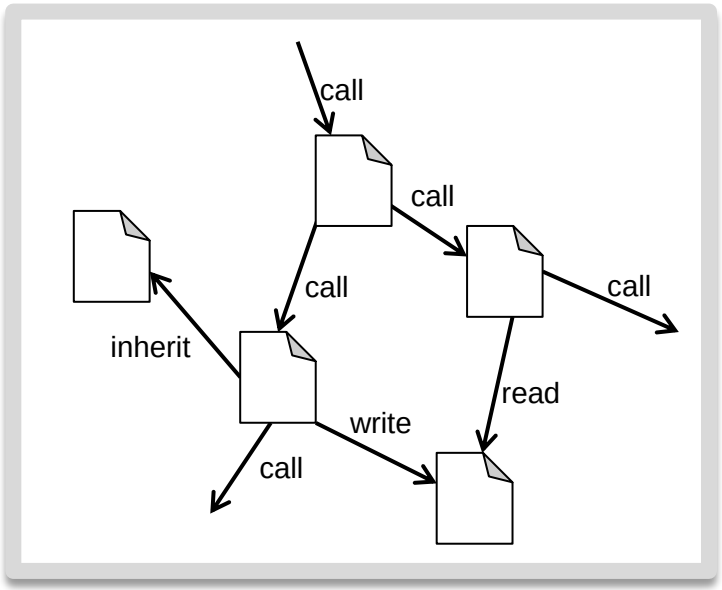
**S**oftware **Ar**chitecture **F**inder

- **フィーチャー***(機能)の観点でソフトウェアシステムを分割
- できるだけ人間の介入が不要
  - 従来は，**遍在モジュールの除去**を人間が行う必要があった
- ソースコードが入力情報
- 1000クラス以上でも動作

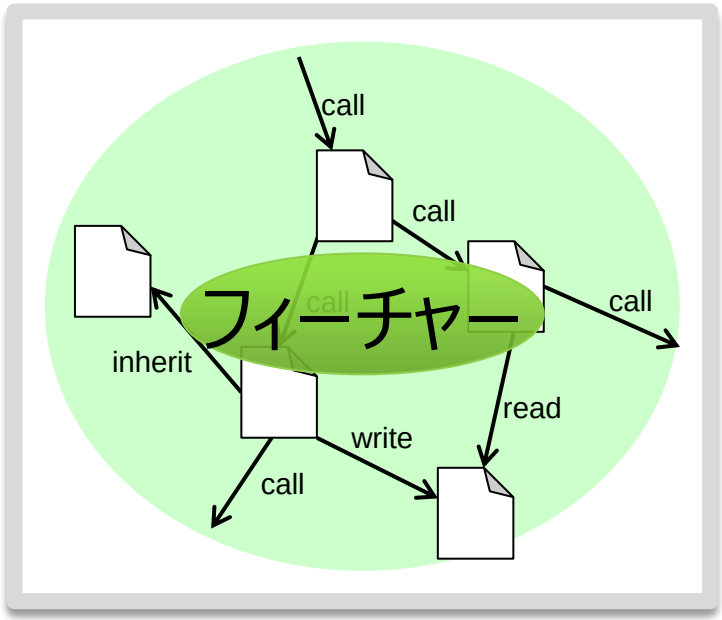
* 本研究での「フィーチャー」の定義は，フィーチャーロケーションの研究における「外部のユーザーから起動されるシステムの機能」[Eisenbarth+ 03]を用いる．

# ソフトウェアクラスタリングによるフィーチャーの発見

① ソフトウェアをグラフ構造で表現 (ソースコードから抽出した静的依存関係グラフ)



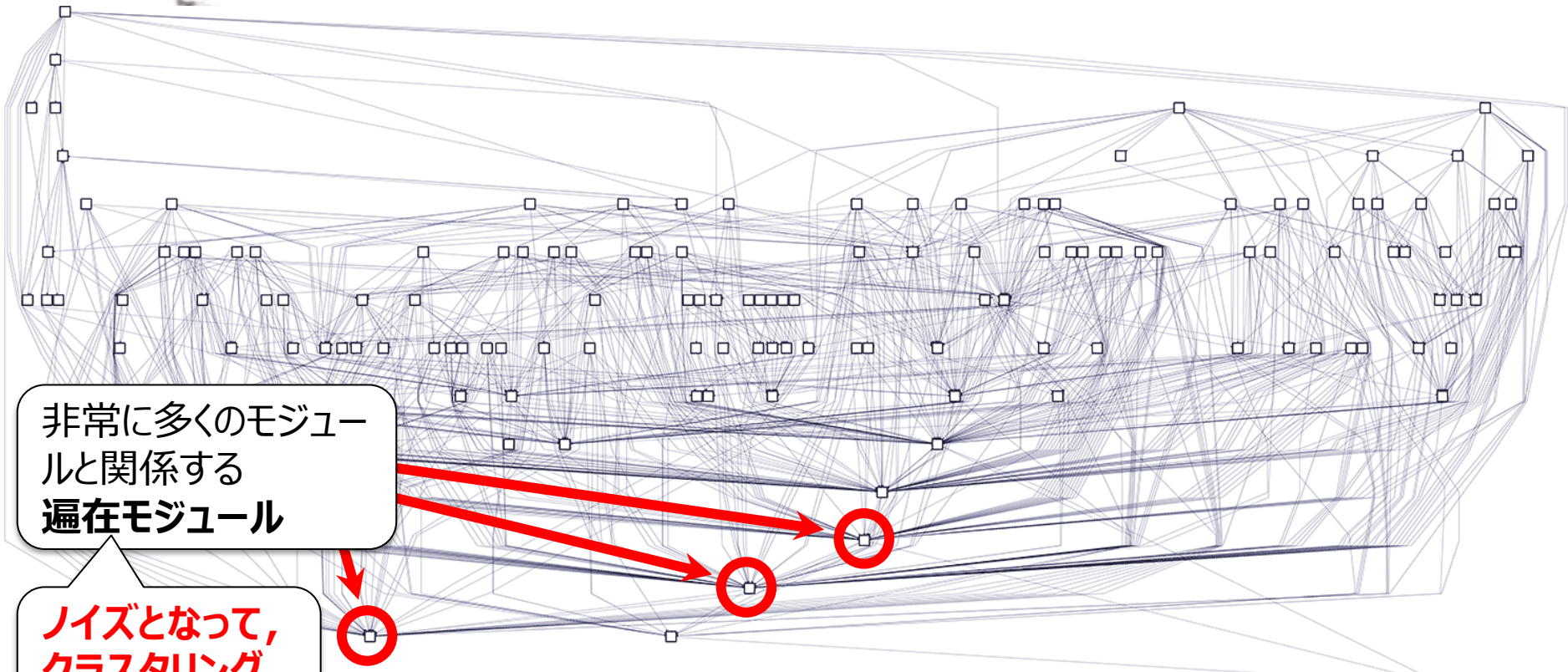

② フィーチャーは関連しているソースコード等の組み合わせで実現されていると仮定すると，




③ フィーチャーの発見は，グラフの分割問題(グラフクラスタリング)と考えることができる

# 依存関係グラフの実例と遍在モジュール


オープンソースのデータマイニングツール
- ver.3.0 (142クラス)

非常に多くのモジュールと関係する
**遍在モジュール**

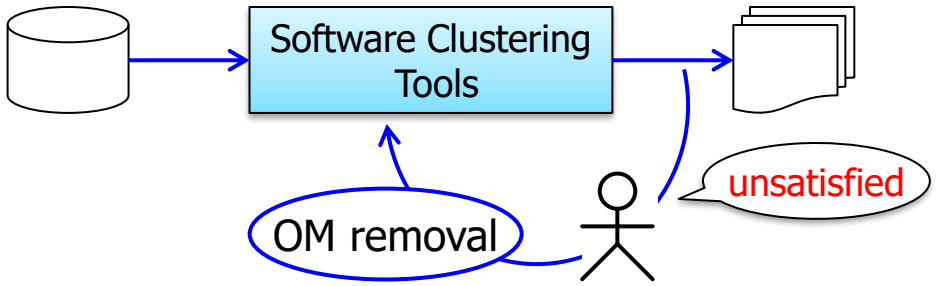
**ノイズとなって，クラスタリングの邪魔をする**

＊ 頂点＝モジュール (クラス, ソースファイル, データ), 有向辺＝依存関係

# 遍在モジュールと人間の介入

遍在モジュール(Omnipresent Modules; OM) [Muller93] はユーティリティや横断的関心事のように遍く広く使われるモジュール

- 遍在モジュールがノイズとなってソフトウェアクラスタリングが上手く動作しない
- 既存研究では，結果を精錬するために人間が取り除いていた

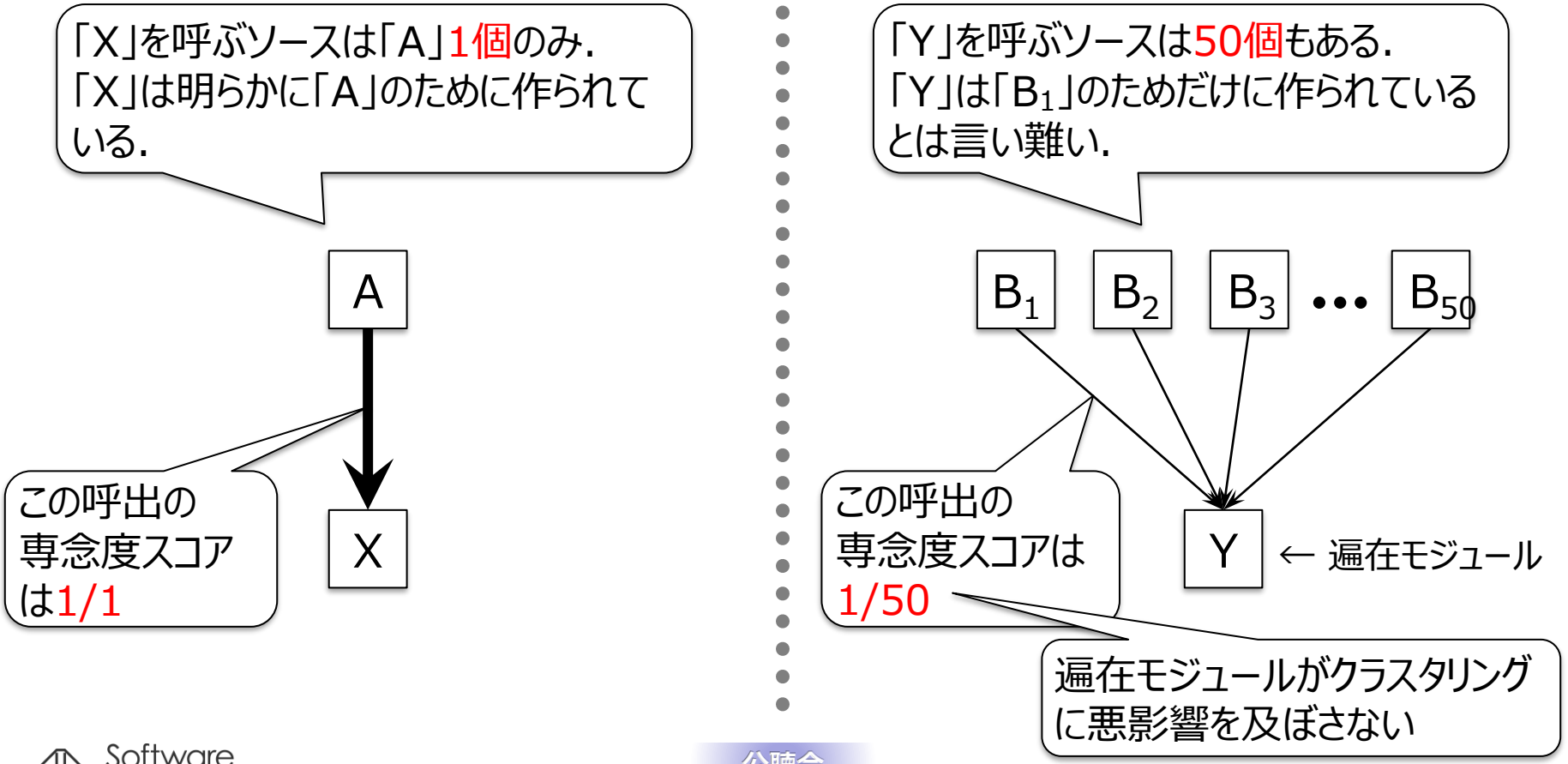



➡ 遍在モジュール除去が不要になれば，人間の介入が不要になる

# 専念度スコア (Dedication Score)

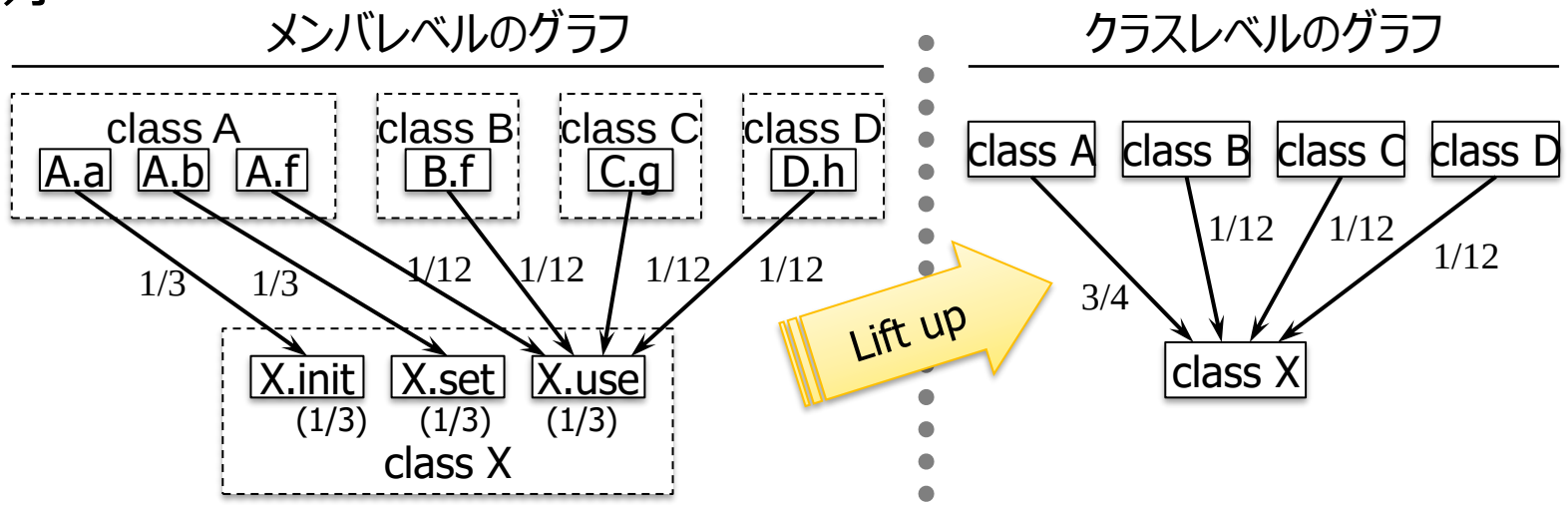
ログ関数などはフィーチャーにとって本質的ではない．本質的な関係のみを集めたい

➡「呼び元」にとって「呼び先」がどれだけ専念しているか?を数値化 (例．1/fanin)



# クラス-メンバ階層がある場合の専念度スコア

- 例




- 多階層の場合の定義

$$\mathrm{D_M}(A,B) = \sum_{m \in M_{B \leftarrow A}} \frac{1}{\mathrm{fanin}_X(m) \cdot \mathrm{m}_X(B)}$$

$A$, $B$はクラス，$M_{B \leftarrow A}$は$B$のメンバのうち$A$のメンバが依存するものの集合

$m$に依存する他クラスのメンバ数

$B$に属する他クラスから依存されるメンバ数

# モジュラリティ最大化法 [Newman04]

生物学(タンパク質ネットワーク)やソーシャルネットワーク解析でよく用いられる，コミュニティ発見(＝グラフクラスタリング)の一手法．

- アプローチ
  - 目的関数 **Modularity Q** を最大化するようグラフの頂点の併合を繰り返す
- アルゴリズム
  - 厳密解を求めるのはNP-hardだが，高速なヒューリスティクスあり
    - 典型的にO(|V| $\log^2$|V|)[Caluset04], O(|V| log|V|)[Blondel08]
    - 詳細は略

# Modularity $Q_D$ [Arenas07][Leicht08]

Modularity Qの重み付き有向き版を用いる

$$Q_D = \frac{1}{W}\sum_{i,j}\left[A_{ij} - \frac{k_i^{OUT} k_j^{IN}}{W}\right]\delta(c_i, c_j)$$

実際の重み

重みの期待値

クラスタ内の辺のみ考慮

依存関係の専念度 − 「専念していない」場合の専念度

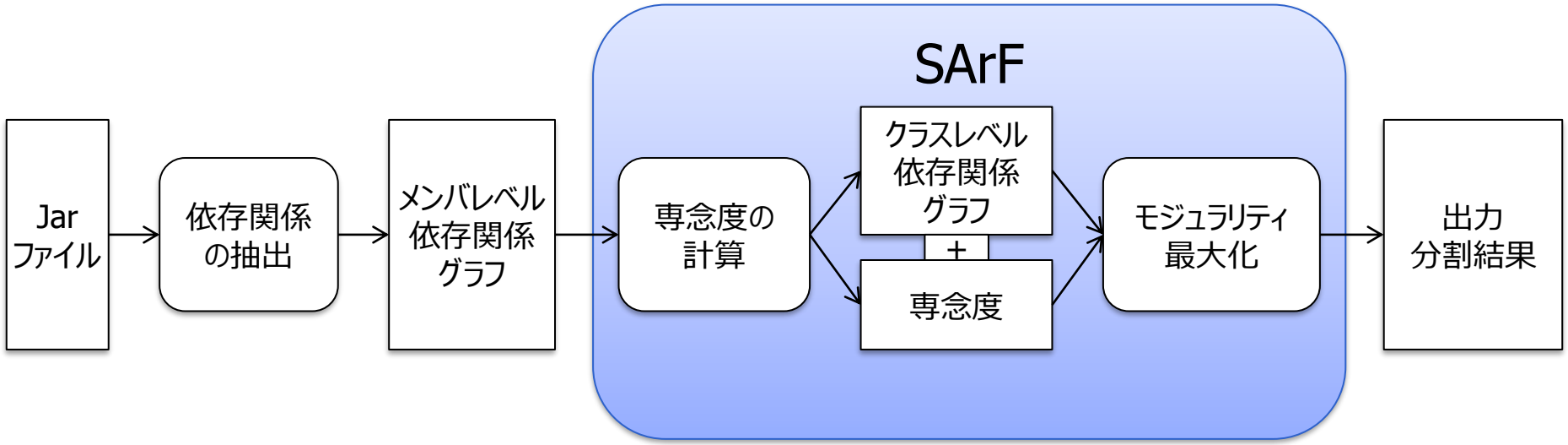
依存関係の有意さ

**専念度スコアの主旨に適う目的関数**

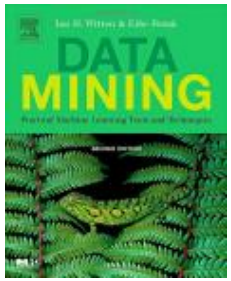
$A$ : adjacency matrix
$W=\Sigma A_{ij}$ : sum of weight
$k_i^{OUT}=\Sigma_j A_{ij}$ : weighted out degree
$k_j^{IN}=\Sigma_i A_{ij}$ : weighted in degree
$c_i$: $i$'s cluster
$\delta$: Kronecker's $\delta$

# SArFの処理フロー

Java言語の場合




# ケーススタディ
# Weka 3.0
# (オープンソースのデータマイニングツール)

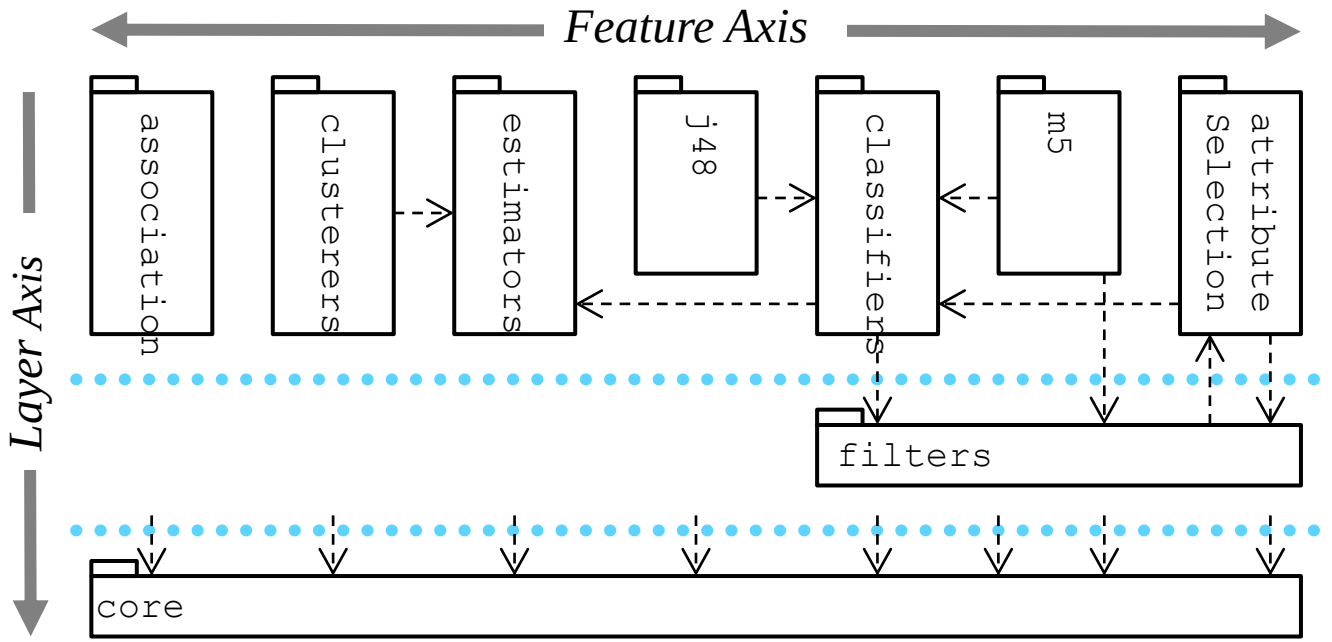


# Weka 3.0のアーキテクチャ

パッケージ図をオーソリティ分割結果として利用できる

- core以外の**パッケージはフィーチャーに一致**
- 過去の研究でもパッケージをオーソリティ分割結果として利用 [Patel08][Erdemir11]

#packages = 9

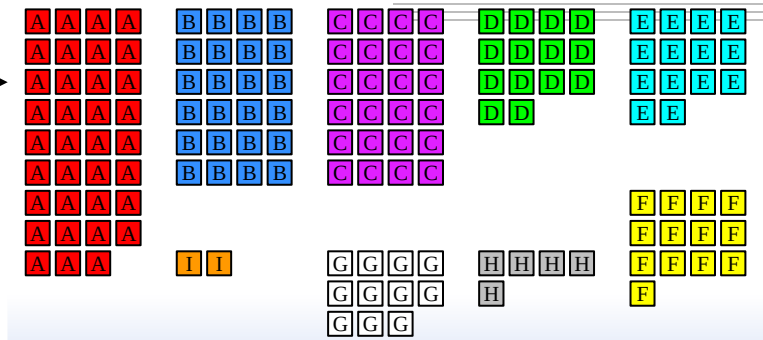



* All packages depend on package `core`.

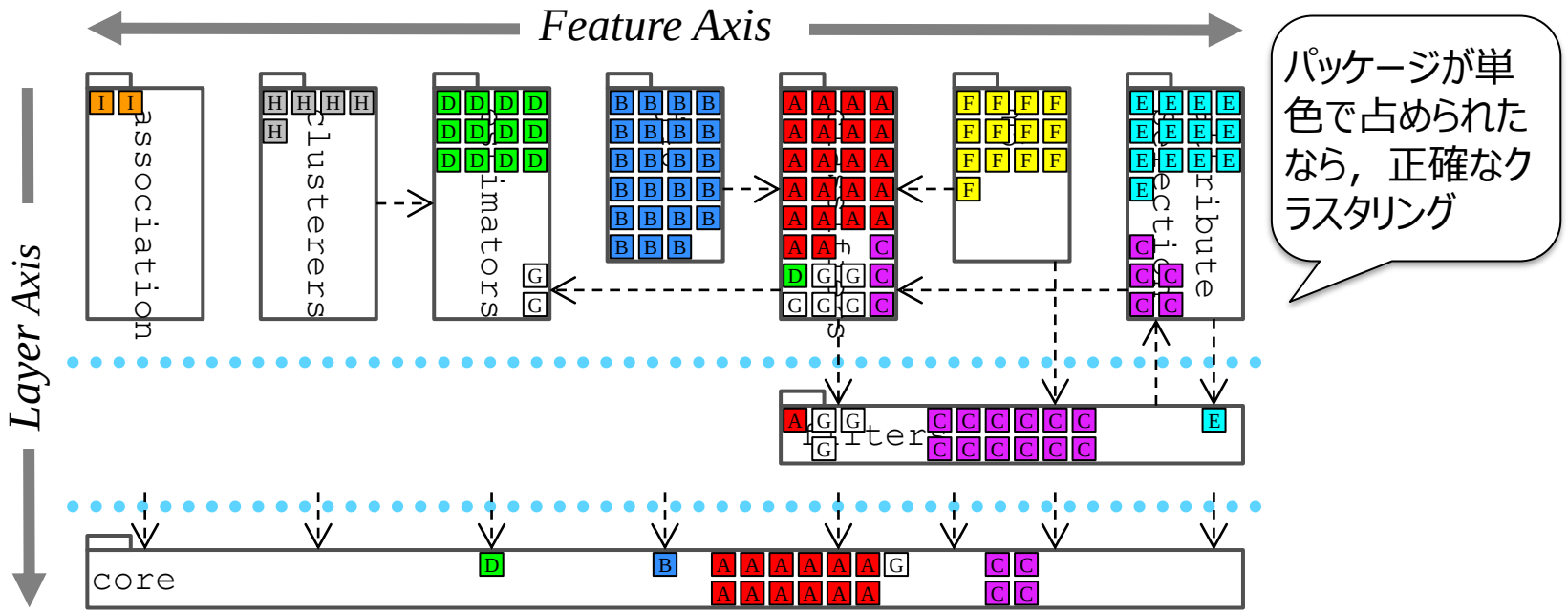
# SArFのクラスタリング結果 (Weka 3.0)

Weka jarファイル
142クラス

→ 依存関係グラフ → SArF →

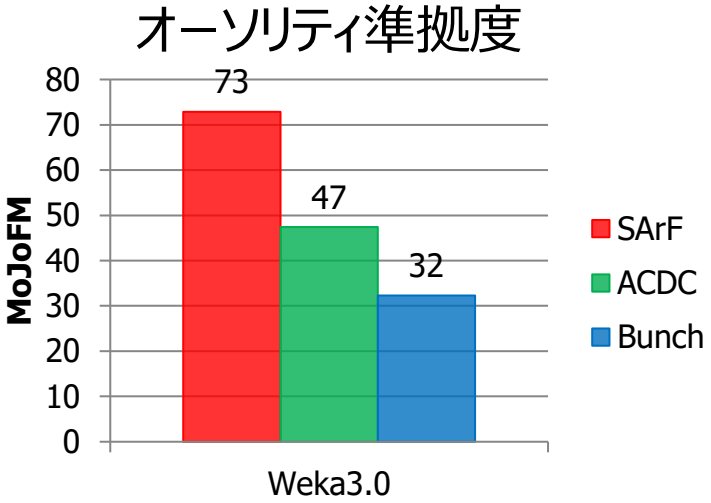



**フィーチャーを集めることに成功**



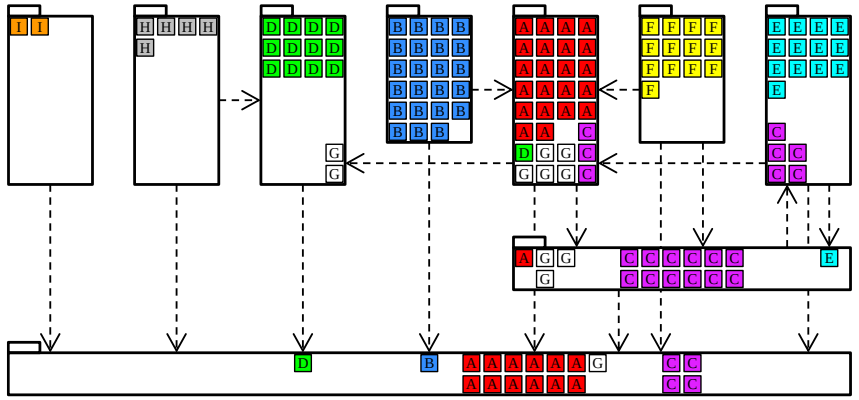

* All packages depend on package `core`.

# クラスタリング結果の比較



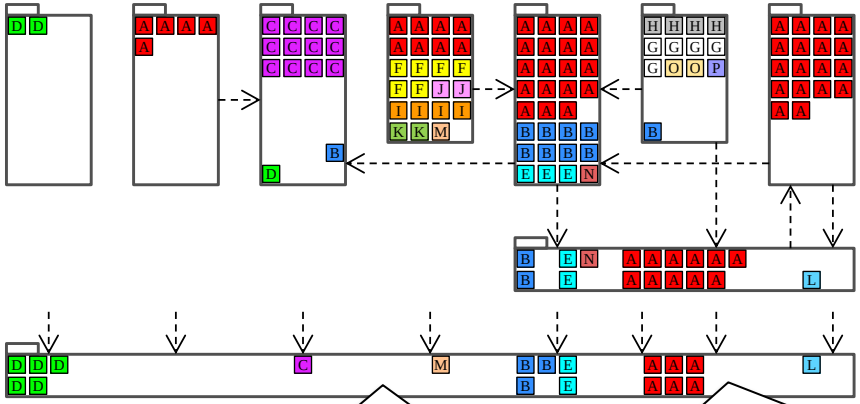

SArF

#clusters = 9



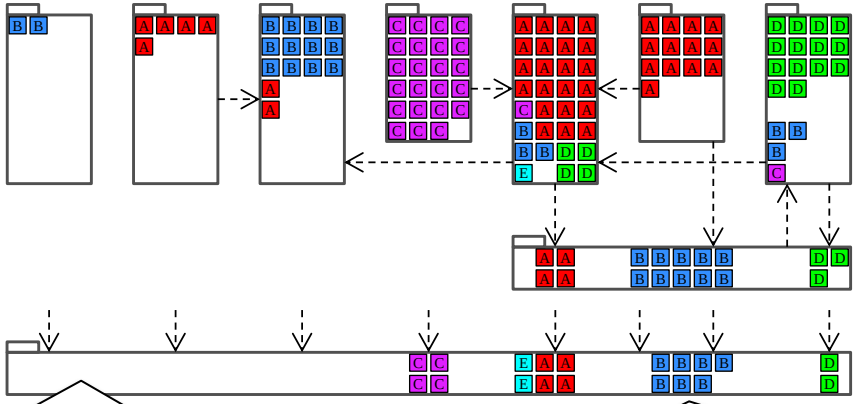
ACDC

#clusters = 16



Bunch

#clusters = 5



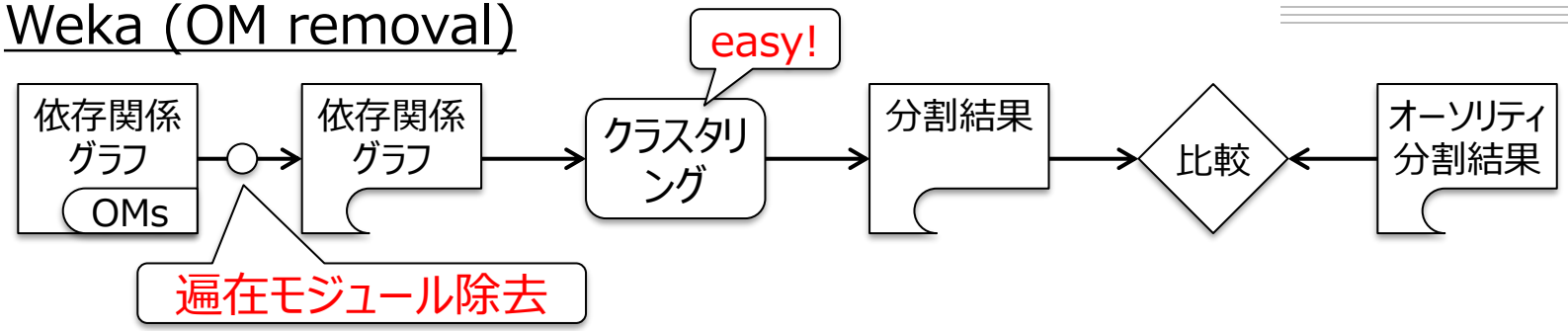
小さなクラスタが多数散乱

クラスタ A が多数のパッケージにまたがる

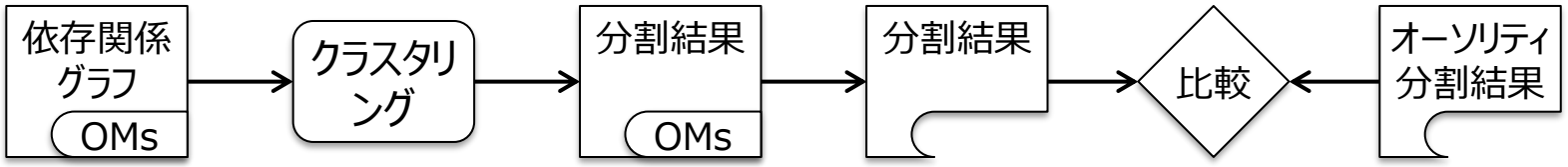
異なるパッケージを区別できてない

# 遍在モジュール除去は不要になったか?

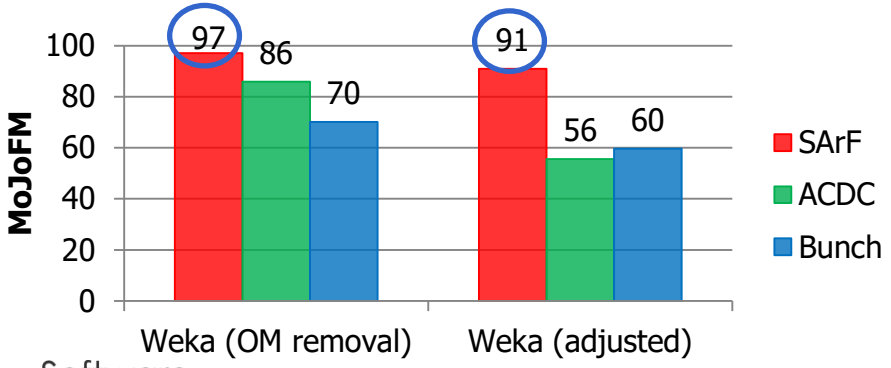
- Weka (OM removal)




- Weka (adjusted)




- 結果

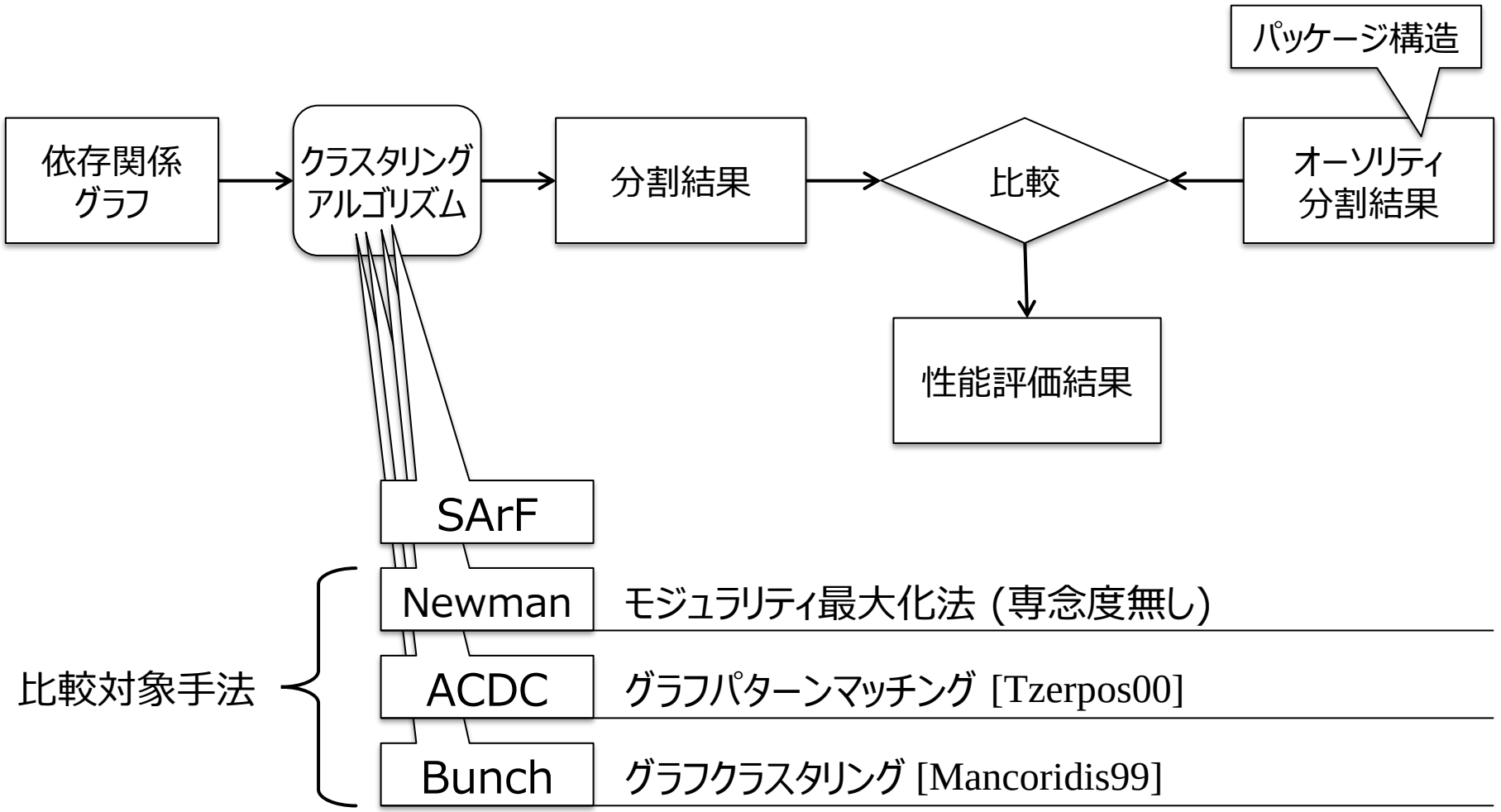



- 分かったこと
  - SArFは遍在モジュール除去なしでもありでも性能が高い
  - 他のアルゴリズムは遍在モジュールに脆弱

**結論: SArFは遍在モジュール除去が不要**

# 実験と評価

# 評価手順


パッケージ構造
依存関係
グラフ
クラスタリング
アルゴリズム
分割結果
比較
オーソリティ
分割結果
性能評価結果
SArF
Newman
モジュラリティ最大化法 (専念度無し)
比較対象手法
ACDC
グラフパターンマッチング [Tzerpos00]
Bunch
グラフクラスタリング [Mancoridis99]

# 評価データセット

mavenリポジトリとSourceForgeより利用数順に無作為抽出 (35種304バージョン)

| ソフトウェア名＋バージョン (数) | クラス数 | *Ka* |
|---|---|---|
| ant 1.4－1.9 (6) | 158-724 | 18.7 |
| aspectjweaver 1.5－1.8 (4) | 281-343 | 7.5 |
| avro 1.4－1.8 (5) | 72-185 | 8.8 |
| camel-core 2.8－2.17 (10) | 879-1428 | 41.9 |
| derby 10.1.1－10.12.1 (21) | 1172-1453 | 53.2 |
| easymock 2.0－3.4 (10) | 55-227 | 3.7 |
| elasticsearch 1.2－2.3 (10) | 2876-4582 | 220.2 |
| findbugs 1.0－3.0 (6) | 431-1102 | 28.7 |
| freemarker 1.5－2.3 (3) | 76-444 | 9.0 |
| geoserver 1.5－1.7 (3) | 104-217 | 11.7 |
| groovy 2.0－2.4 (5) | 706-1116 | 50.4 |
| gwt-servlet 1.4－2.7 (12) | 316-3536 | 54.2 |
| h2 1.2－1.4 (3) | 415-490 | 25.7 |
| hadoop-common 0.20－2.7 (11) | 567-1092 | 46.8 |
| hsqldb 1.6－2.3 (6) | 52-423 | 10.8 |
| httpclient 4.0－4.5 (6) | 218-393 | 19.8 |
| javassist 2.5－3.21 (22) | 123-211 | 12.4 |
| jmol 12.0－13.0 (3) | 466-540 | 32.7 |
| jsoup 0.2－1.10 (12) | 21-54 | 3.0 |
| junit 3.7－4.12 (15) | 45-183 | 12.7 |

| ソフトウェア名＋バージョン (数) | クラス数 | *Ka* |
|---|---|---|
| lucene-core 3.6－6.6 (25) | 506-796 | 20.6 |
| maven-core 2.0－3.3 (7) | 54-333 | 13.4 |
| netty 3.5－3.10 (6) | 515-593 | 32.8 |
| pmd 1.1－5.4 (22) | 247-1067 | 18.4 |
| proguard 3.4－4.4 (10) | 315-500 | 20.2 |
| saxon 6.5－8.7 (4) | 328-705 | 16.3 |
| snakeyaml 1.4－1.18 (15) | 92-110 | 6.0 |
| spring-web 3.1－4.3 (6) | 282-390 | 25.5 |
| squirrel-sql 3.0－3.5 (5) | 602-755 | 35.2 |
| sweethome3d 5.1－5.4. (3) | 218-225 | 7.7 |
| velocity 1.3－1.7 (5) | 181-235 | 14.2 |
| weka 3.4－3.8 (4) | 676-1615 | 54.3 |
| xalan 2.1－2.7 (6) | 157-459 | 12.3 |
| xercesImpl 2.0－2.11 (11) | 522-709 | 30.2 |
| zookeeper 3.3－3.4 (2) | 153-204 | 6.5 |

バイアスのない多様性のあるデータセット

- クラス数は数十から数千まで
- 10倍以上に成長したものあり

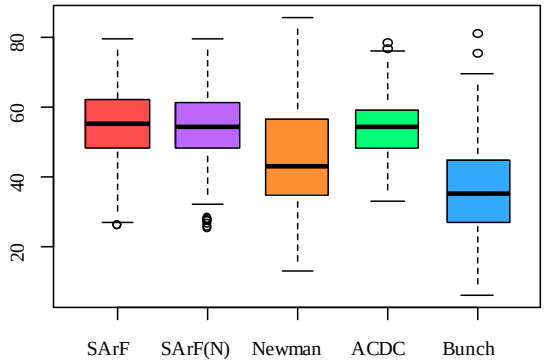
# 性能評価指標

| 評価軸 | 評価指標 |
|---|---|
| 品質 | オーソリティ準拠度 |
| | 適切なクラスタ数 |
| 安定度 | |
| 実行時間 | |

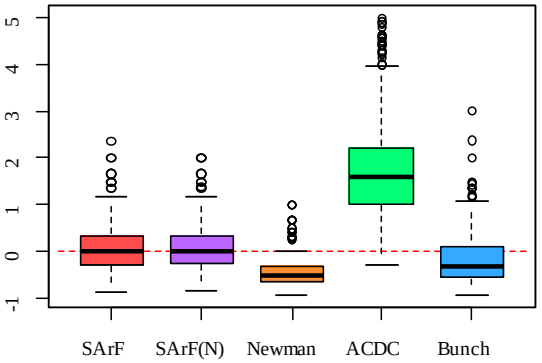
- オーソリティ準拠度
  - 出力した分割結果(CD)と，対象ソフトウェアの有知識者(オーソリティ)が用意した分割結果(オーソリティ分割結果; AD)との類似度．
  - 類似度にはMoJoFM [Tzerpos99]を使用． 大きいほど良い．
- 適切なクラスタ数
  - 出力した分割結果のクラスタ数($K$)と，オーソリティ分割結果のクラスタ数($Ka$)の相対誤差． 0に近いほど良い．
- 安定度
  - 些細な変更で出力が大きく変わることは望ましくない． バージョン間の類似度の平均を安定度とする． 大きいほど良い．

# 実験結果: 品質・安定度

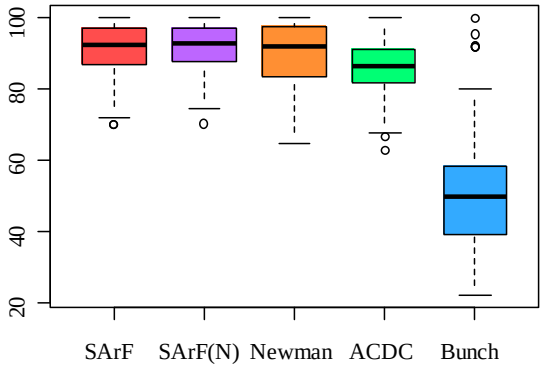
304バージョンについて各アルゴリズムを実行した結果:



(a) オーソリティ準拠度 (MoJoFM(%))



(b) クラスタ数の相対誤差 (RE)



(c) 安定度 (%)

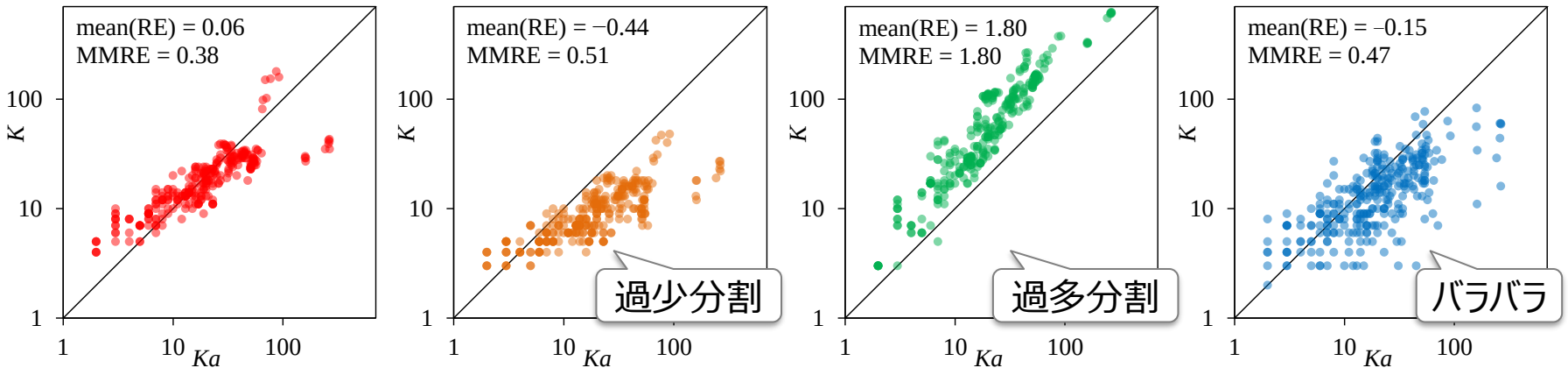
↑適切なクラスタ数
0(赤破線)に近いほど良い

- オーソリティ準拠度，適切なクラスタ数，安定度のすべての面で，SArFが統計的有意に優れる

※ SArF(N)はSArFの旧版 (ICSM2012発表)

# 実験結果: 適切なクラスタ数

304バージョンについて各アルゴリズムを実行した結果:



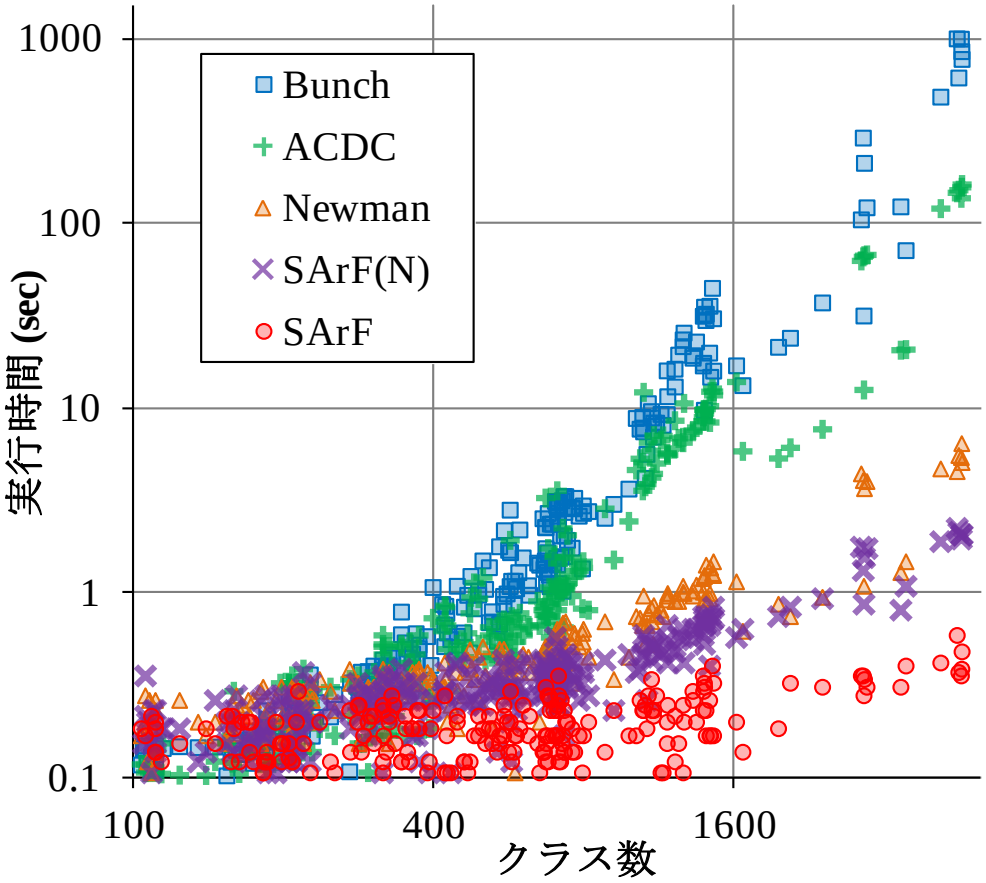

(a) SArF(L) (b) Newman (c) ACDC (d) Bunch

↑ $K$が出力分割結果のクラスタ数，$Ka$がオーソリティ分割結果のクラスタ数
$K$=$Ka$の斜線近くに分布するほど良い.

- SArFは目に見えて適切なクラスタ数を出力しており，mean(RE)，MMREの指標値も統計的有意に優れる

# 実行時間の比較

304バージョンについて各アルゴリズムを実行した結果:




- グラフは両対数軸なので，傾きが実行時間のオーダーを表す．SArFは明らかに他より実行時間のオーダーに優れる．
- SArFは10,000超クラスに対しても数秒で完了する．（ACDC, Bunchは実行不可）

| ソフトウェア＋バージョン | クラス数 | 実行時間 (sec) | |
|---|---|---|---|
| | | SArF | SArF(N) |
| JBoss (Wildfly) 10.1 | 10k | 1.28 | 3.62 |
| JRE 8 | 19k | 2.16 | 47.38 |
| Eclipse 4.7 | 41k | 3.16 | 81.54 |

# 4章まとめ

- 貢献
  - フィーチャーを集めるソフトウェアクラスタリングを提案
  - 遍在モジュール除去を不要にし，人間の介入を不要にした（自動化）
- 結果
  - ケーススタディにて以下を確認
    - フィーチャーが集まっている
    - 遍在モジュール除去が不要
  - 実験にて以下を確認
    - 品質(オーソリティ準拠度，適切なクラスタ数)，安定度，実行時間のすべての指標で，他のアルゴリズムに優れる
- 今後の課題
  - 実証実験 (オーソリティ分割結果の収集)
  - スケーラビリティ (10,000+クラス)

# 5章 依存関係グラフを用いた ソフトウェアアーキテクチャの可視化

# 本章のゴール

以下の特徴を持つ，ソフトウェア可視化手法 **SArF Map** を提案

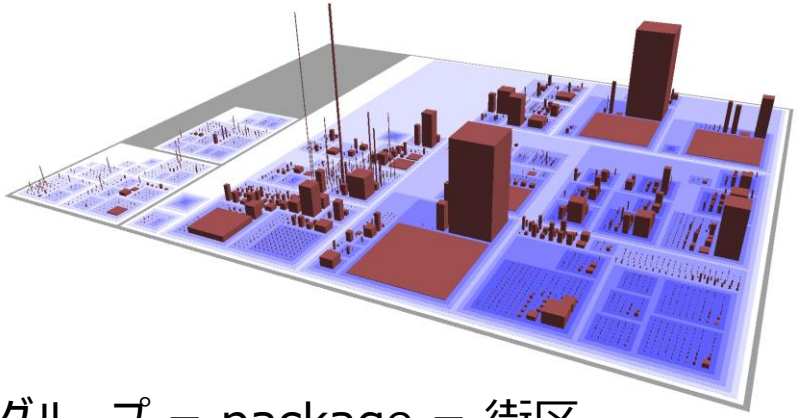
- 大規模ソフトウェアの高レベルの抽象化ビュー
  - **フィーチャー***でまとめるソフトウェア可視化
- 暗黙のアーキテクチャ知識を明らかにする
  - （ソフトウェアアーキテクチャの）**レイヤー**の可視化
  - アーキテクチャの設計品質を評価可能に
- 様々なステークホルダー間のコミュニケーションを促す
  - 意思決定者（経営者など）向け
  - 開発者向け

* 本研究での「フィーチャー」の定義は，フィーチャーロケーションの研究における「外部のユーザーから起動されるシステムの機能」[Eisenbarth+ 03]を用いる．

# 既存研究との対比

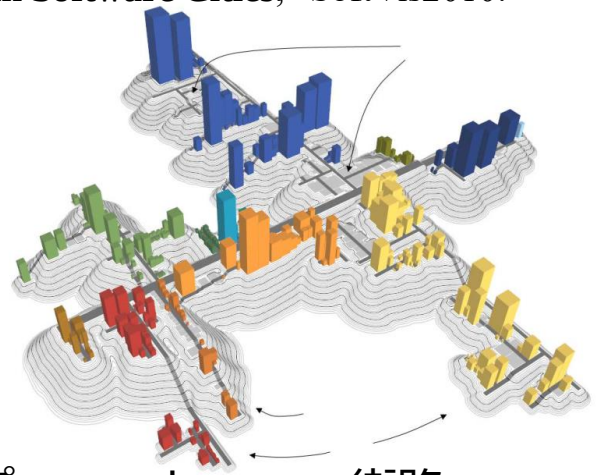
従来の都市メタファーソフトウェア可視化は，パッケージなど明示的なグループ分けを可視化

Wettel+ , “**Visualizing Software Systems as Cities**,” VISSOFT2007.



グループ = package = 街区
配置法: Rectangle-packing Tree Map Layout

Steinbrückner+, “**Representing Development History in Software Cities**,” SoftVis2010.



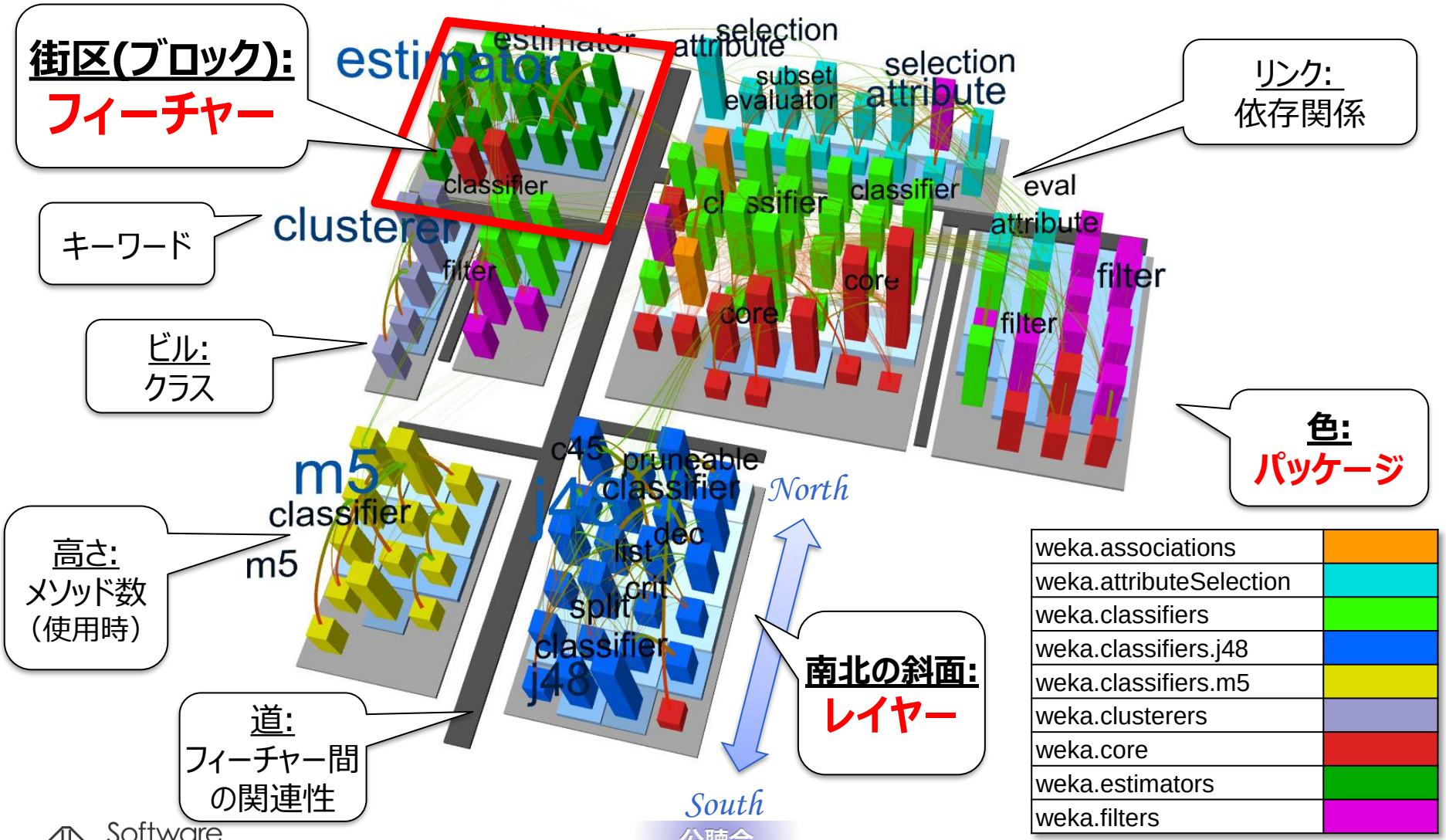
グループ = package = 街路
配置法: Hierarchical Street Layout

- SArF Mapは，隠れた「フィーチャー」を抽出して可視化するのが特徴
  - **グループ = フィーチャー = 街区**
  - **配置法: Street and Block Tree Layout**

# フィーチャーでグループ分けすることのメリット

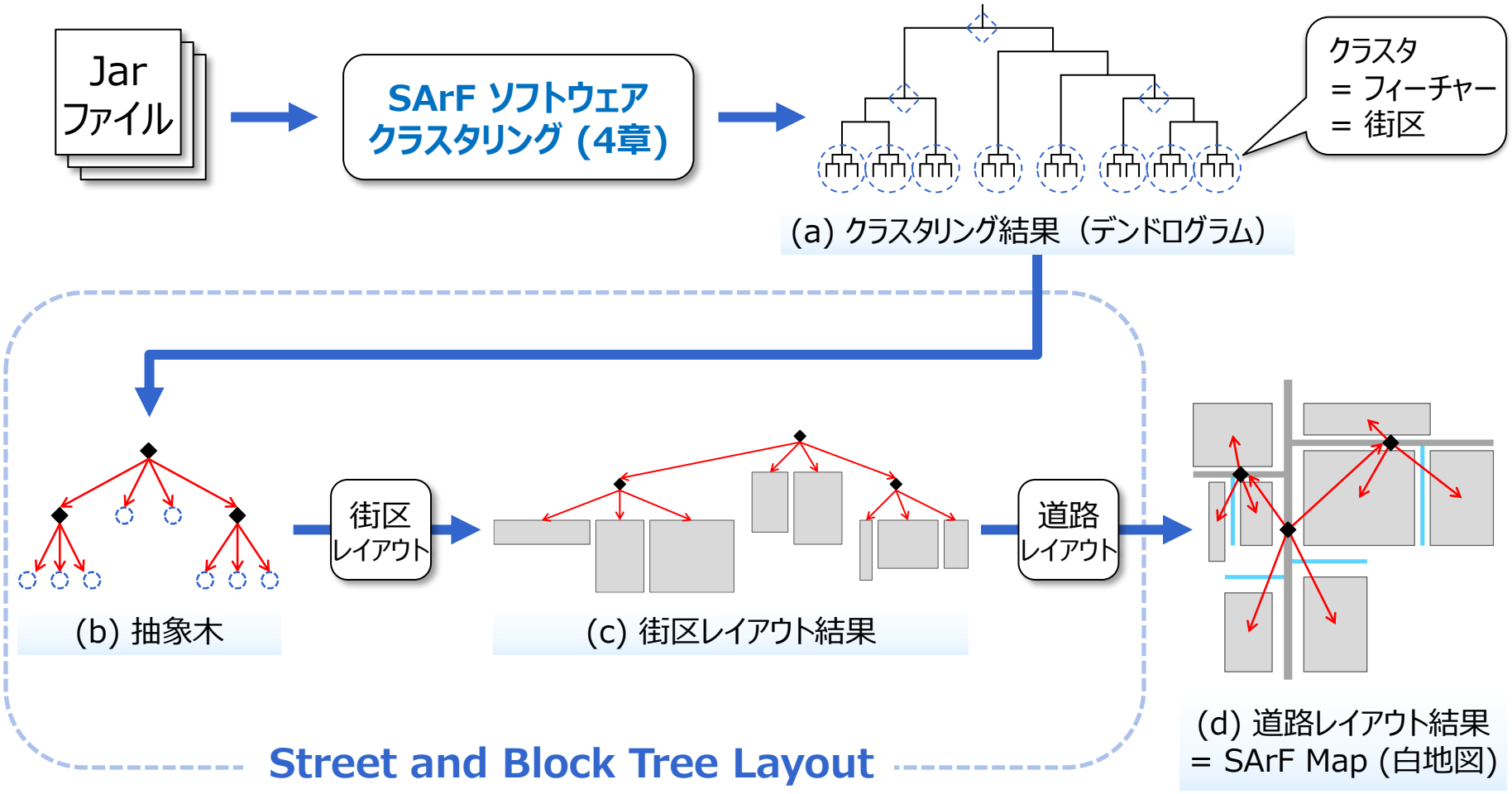
- 抽象度の高いアーキテクチャビュー
  - フィーチャーは高レベル意思決定者に有益
- レイヤー分解が可能
  - ソフトウェア全体のレイヤー分解は困難だが，
    フィーチャーのレイヤー分解は容易
- フィーチャーとレイヤーの2つの観点からのビューから，
  アーキテクチャの設計品質が分かる

# 例: Weka 3.0のSArF Map (142クラス)

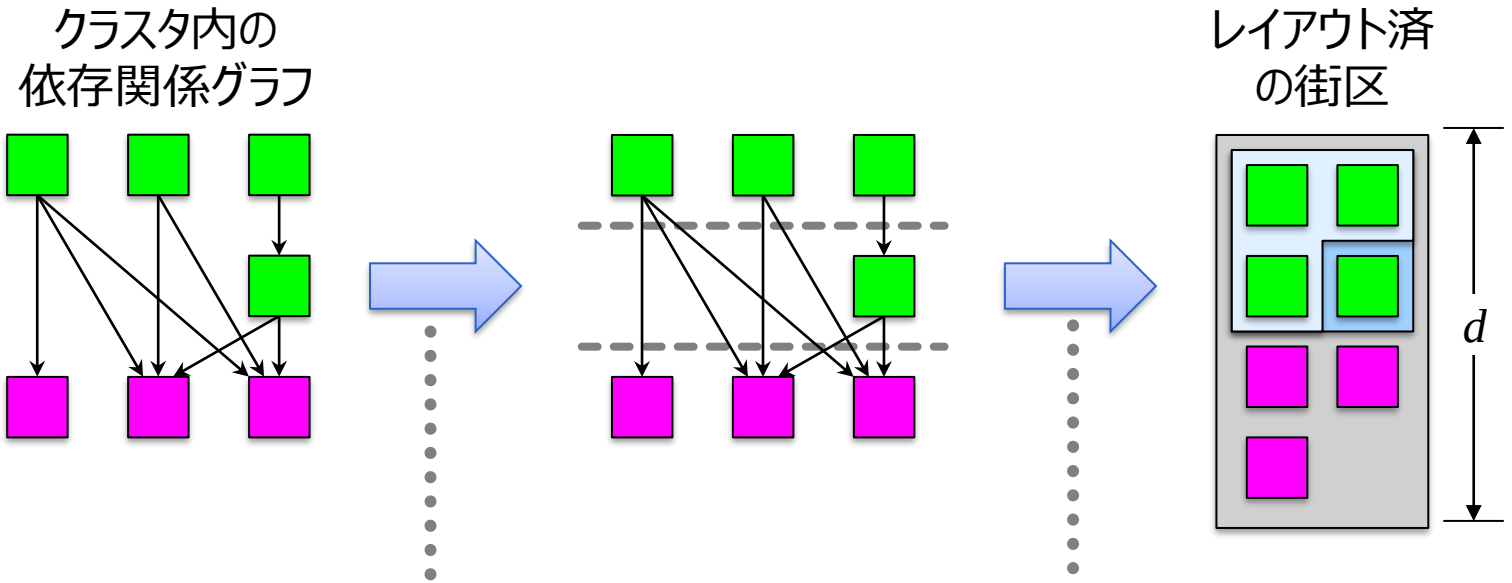



公聴会

# SArF Mapの手順


Jar
ファイル
SArF ソフトウェア
クラスタリング (4章)
クラスタ
= フィーチャー
= 街区
(a) クラスタリング結果 (デンドログラム)
街区
レイアウト
道路
レイアウト
(b) 抽象木
(c) 街区レイアウト結果
(d) 道路レイアウト結果
= SArF Map (白地図)
Street and Block Tree Layout

# レイヤーのための街区レイアウト




レイヤー可視化のために
依存関係グラフをレベル分解

以下の要請を表現するエネルギー関数を最小化

- 関係するビルは近く
- 依存関係は上から下に流れるように

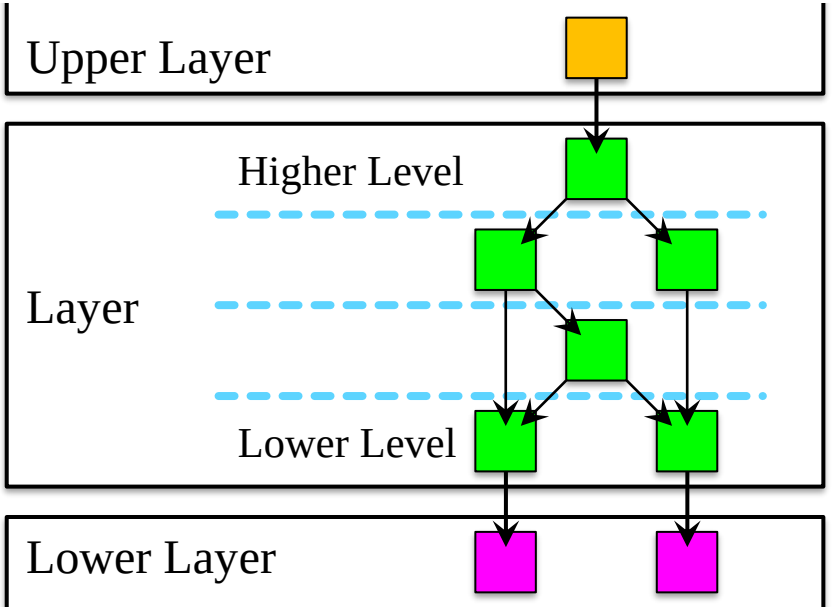
# レイヤー可視化のためのレベル分解

**小課題:** ソフトウェアアーキテクチャのレイヤーの抽出

- 自明ではない

**解決:** 依存関係の深さレベルに注目

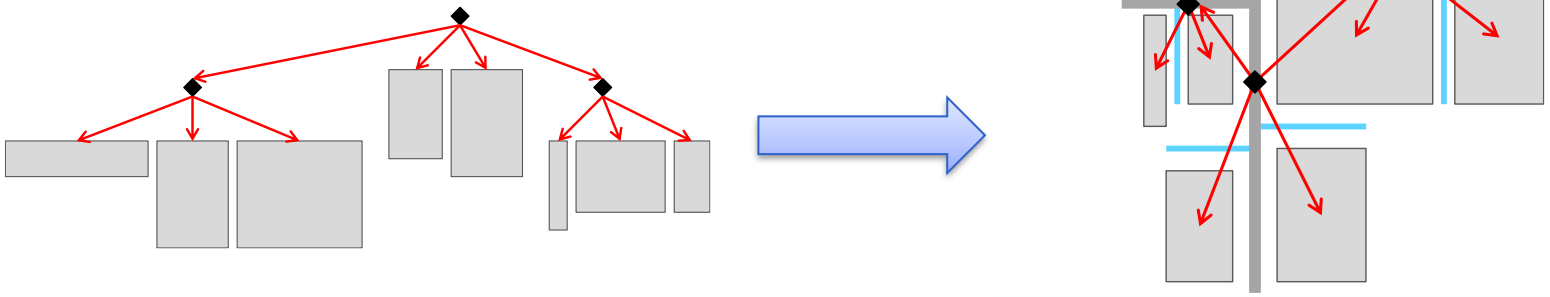
- レベル ⊂ レイヤー
- **レベル順で並べると，レイヤー順に並ぶ**
- 循環依存はヒューリスティクス (Greedy Cycle Removal [Tollis98]) で解消
  - フィーチャー単位なので解消可能



# フィーチャーのための道路レイアウト

手順:

1. デンドログラムを直交する道路としてレイアウト
2. 以下の要請を表現するエネルギー関数を最小化
   - 関係するフィーチャーは近く



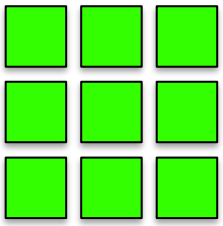
(c) Output of Block Layout (Layer Layout)

(d) Output of Street Layout (Feature Layout)

# ソースコード構成パターンによる設計評価

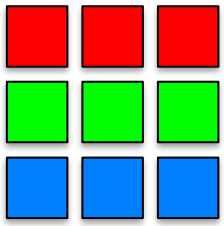
フィーチャー＝街区，パッケージ＝色，レイヤー＝南北による多元評価

(a) *Single-color*



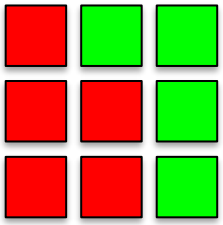
- このフィーチャーは単一パッケージからなる
- **よいデザイン**（このパッケージがここだけであれば）

(b) *Layered*



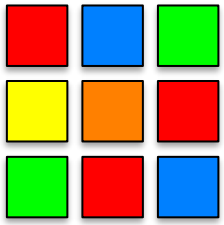
- このフィーチャーは複数のパッケージの積み重ね
- **よいデザイン**
- この場合，パッケージからはフィーチャーが読み取れないが，地図からは読み取れる

(c) *Subgroups*



- このフィーチャーは複数パッケージからなる，または，複数の小フィーチャーからなる
- **リファクタリングを検討**

(d) *Mixed-color*



- このフィーチャーは多くのパッケージに散らばっている
- **悪いデザイン**

# リサーチクエスチョン

**RQ1 フィーチャー** SArF Mapはフィーチャーを可視化できるか?

**RQ2 レイヤー** SArF Mapはレイヤーを可視化できるか?

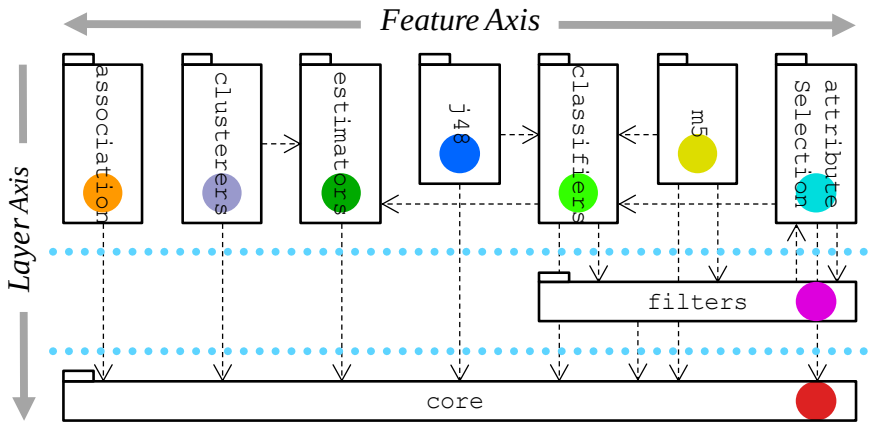
**RQ3 設計の良さ** SArF Mapはパッケージ設計の良さを評価するのに使えるか?

**RQ4 アーキテクチャ** SArF Mapアーキテクチャの知識を明らかにするか?

# Weka 3.0 (142クラス)

**事実:** Weka3.0のパッケージはcore以外はフィーチャーに一致[4-12,37]

- パッケージ図

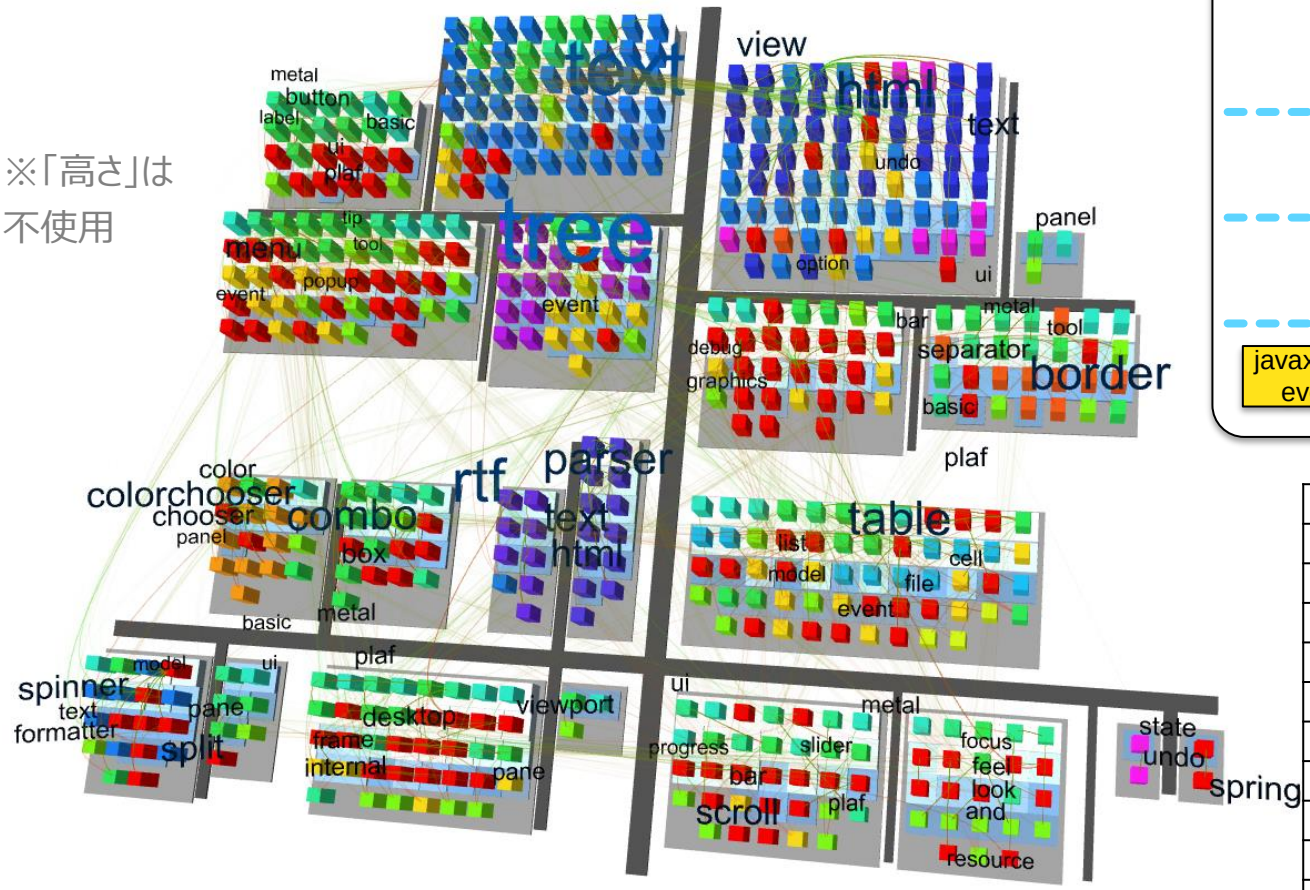


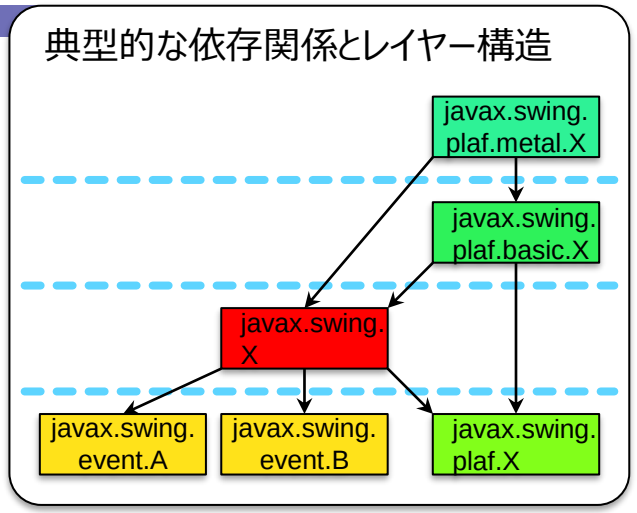

- SArF Map


- 観察
  - ***Single-color*** パターンが多い
  - ***Layered*** パターンが残りを占める．それらは実際のレイヤーと一致
  - すべて ***Single-color*** パターンか ***Layered*** パターンのため，パッケージ構造の設計は良い

# JDK Swing 1.4.0 (536クラス)

※「高さ」は
不使用




| | |
|---|---|
| javax.swing | |
| javax.swing.border | |
| javax.swing.colorchooser | |
| javax.swing.event | |
| javax.swing.filechooser | |
| javax.swing.plaf | |
| javax.swing.plaf.basic | |
| javax.swing.plaf.metal | |
| javax.swing.plaf.multi | |
| javax.swing.table | |
| javax.swing.text | |
| javax.swing.text.html | |
| javax.swing.text.html.parser | |
| javax.swing.text.rtf | |
| javax.swing.tree | |
| javax.swing.undo | |

- **観察:** ***Layered***, ***Subgroups***, ***Single-color*** パターンが見られる

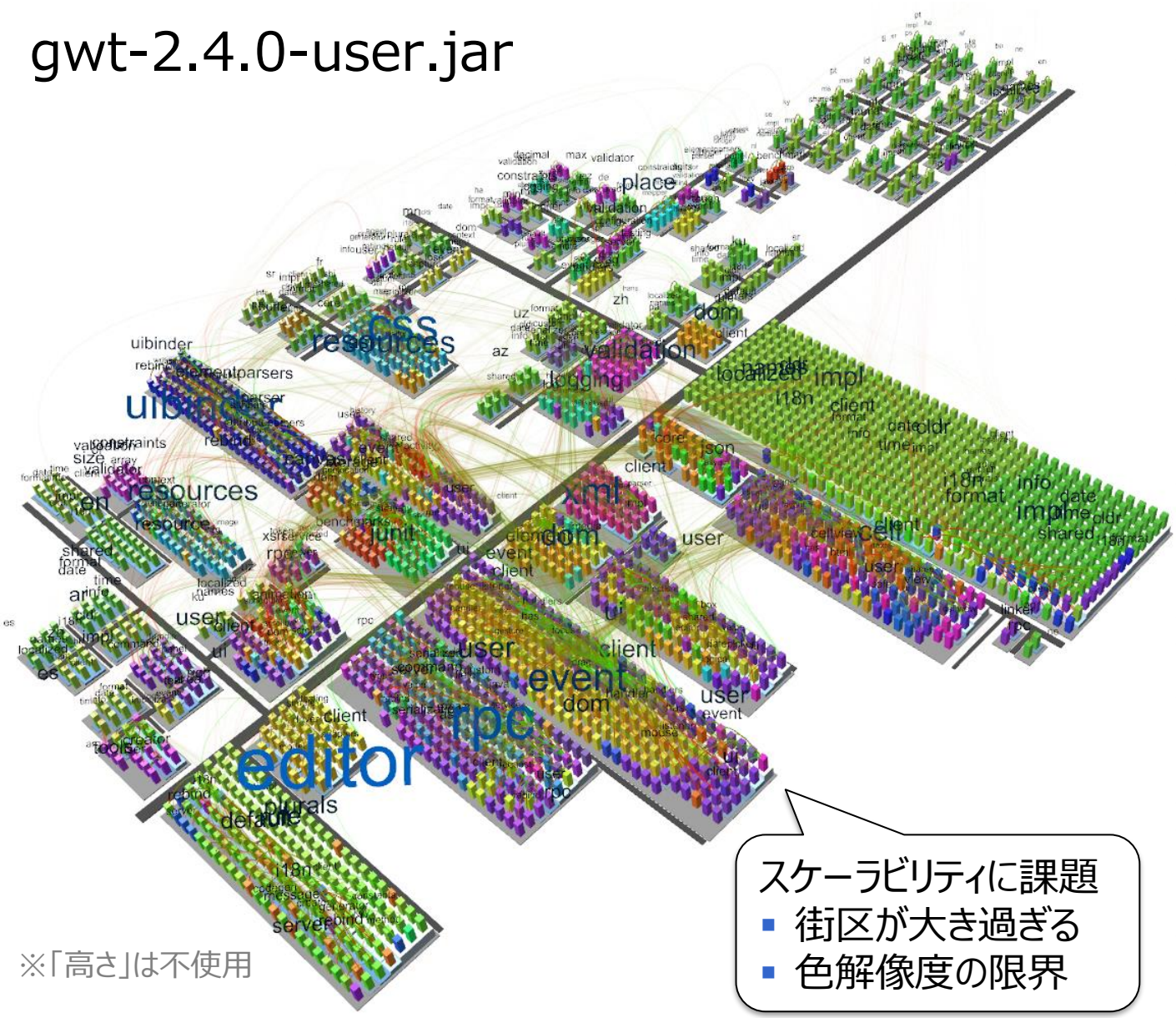
**RQ1 ✓ フィーチャー**

**RQ2 ✓ レイヤー**

**RQ4 ✓ アーキテクチャ**

# Google Web Toolkit (2567クラス)

gwt-2.4.0-user.jar



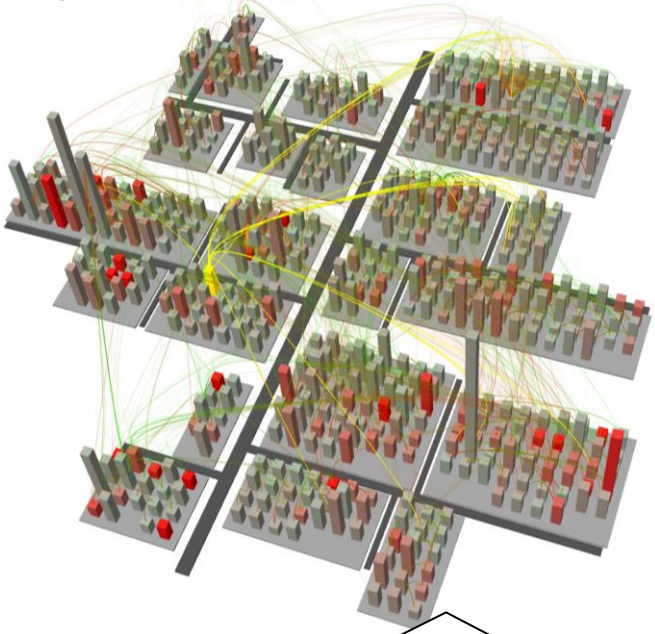

| | |
|---|---|
| com.google.gwt.activity | |
| com.google.gwt.animation | |
| com.google.gwt.benchmarks | |
| com.google.gwt.canvas | |
| com.google.gwt.cell | |
| com.google.gwt.codegen | |
| com.google.gwt.core | |
| com.google.gwt.debug | |
| com.google.gwt.dom | |
| com.google.gwt.editor | |
| com.google.gwt.event | |
| com.google.gwt.geolocation | |
| com.google.gwt.http | |
| com.google.gwt.i18n | |
| com.google.gwt.json | |
| com.google.gwt.jsonp | |
| com.google.gwt.junit | |
| com.google.gwt.layout | |
| com.google.gwt.logging | |
| com.google.gwt.media | |
| com.google.gwt.place | |
| com.google.gwt.precompress | |
| com.google.gwt.regexp | |
| com.google.gwt.resources | |
| com.google.gwt.rpc | |
| com.google.gwt.safecss | |
| com.google.gwt.safehtml | |
| com.google.gwt.storage | |
| com.google.gwt.text | |
| com.google.gwt.touch | |
| com.google.gwt.uibinder | |
| com.google.gwt.user | |
| com.google.gwt.util | |
| com.google.gwt.validation | |
| com.google.gwt.view | |
| com.google.gwt.widget | |
| com.google.gwt.xhr | |
| com.google.gwt.xml | |

スケーラビリティに課題
- 街区が大き過ぎる
- 色解像度の限界

※「高さ」は不使用

# 異なるステークホルダー向けの一貫性のあるビュー

■ 開発者向け地図



影響波及分析用の地図

赤色⇔灰色:
影響度
大⇔小

高さ:
メソッド数

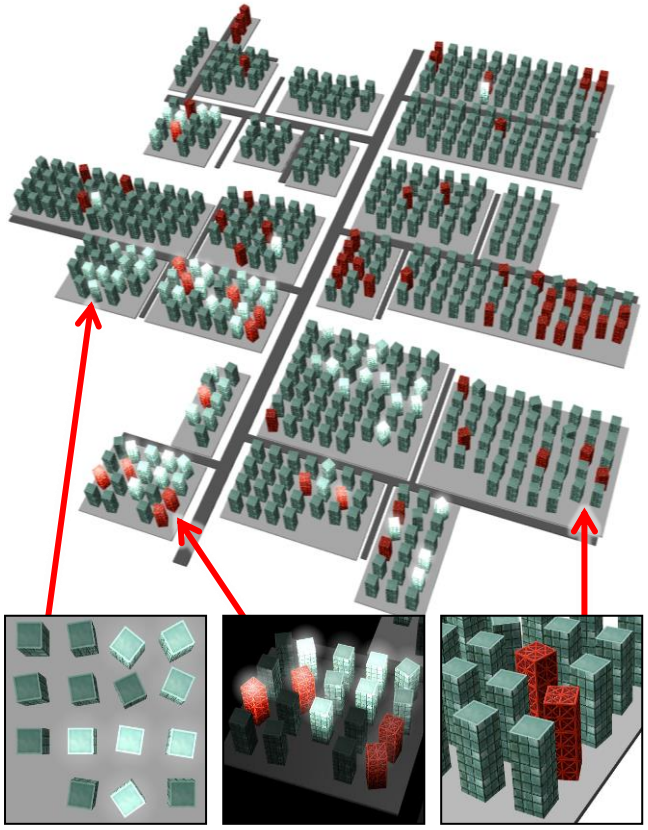
リンク:
依存関係

低レベルの詳細情報

従業員管理
システム (EMS)
(570クラス)

一貫性

■ 意思決定者向け地図



乱雑な街並:
低い保守性

明るさ:
利用頻度大

建築中:
最近の変更

高レベルのサマリ情報

# 5章まとめ

- 貢献
  - フィーチャーを集めるソフトウェアクラスタリングと都市メタファー可視化の組合せ
  - フィーチャーとレイヤーの可視化 (Street and Block Tree Layout)

- 結果：ケーススタディにて確認
  - フィーチャーとレイヤーの可視化 (RQ1, 2)
  - パッケージ設計の良さの評価 (RQ3)
  - アーキテクチャ知識の理解 (RQ4)
  - 異なるステークホルダー(意思決定者と開発者)向けに，一貫性のある異なるビューを提供

- 今後の課題
  - 実証実験
  - スケーラビリティ
  - ソフトウェア進化の可視化のためには，変更に対する頑健性が必要

# まとめ

# まとめ

- ソフトウェア保守では継続的な知識損失が課題

- 知識損失しにくいソースコードから抽出した依存関係グラフを用いる保守効率化技術を提案し，評価
  - 2章 依存関係グラフを用いた欠陥予測
  - 3章 工数予測の精度向上のため対数変換と補正方法
  - 4章 依存関係グラフを用いたソフトウェアアーキテクチャの復元
  - 5章 依存関係グラフを用いたソフトウェアアーキテクチャの可視化
    - 高レベルの意思決定が可能
    - アーキテクチャレベルの設計品質の評価が可能

# 今後の課題

- ソースコード以外から抽出した依存関係の活用
  - 実行時情報から抽出する動的依存関係
    - 実際の利用量を用いた応用
    - 使われない部分をスコープから外すことで，理解や操作が容易
  - ドキュメントから抽出する意味的依存関係
    - ソースコードや実行時情報からは抽出できない潜在的な関係が抽出可
    - 非開発者のステークホルダーにとってより解釈が容易

# Appendix

# ソフトウェア可視化の経験報告

小林健一, "**メタファーを用いた高抽象度ソフトウェア可視化実践の予備報告**," 情報処理学会 ウィンターワークショップ2014・イン・大洗, pp.13-14, 2014.

- SArF Mapを適用した実システムについて，カジュアルなユーザアンケートを実施
  - アンケート対象： 可視化された企業の実業務システムの担当SEまたは担当営業（N=数十）
  - 注意：対照実験のような統制された実験ではない

| 項番 | 分類 | 回答概要（複数回答あり） | 回答中の割合 |
|---|---|---|---|
| G1 | 利点 | 見える化による状況の把握 | 9割 |
| G2 | | 自動生成のため客観的 | 4割 |
| G3 | | 経営者でも現状把握が可能 | 1～2割 |
| G4 | | 各機能の関係性の理解に役立つ | 1～2割 |
| G5 | | 業務視点で優先順位付けが可能 | 1～2割 |
| G6 | | 問題個所ごとに個別に対処可能 | 1～2割 |
| G7 | | 属人知識の可視化による共有 | 1～2割 |
| B1 | 欠点 | 読み方や仕組みの理解に習得を要す | 3割 |
| B2 | | 線が多すぎて見づらい個所がある | 1～2割 |
| R1 | 要望 | 良否判定の結果を直ちに得たい | 1～2割 |
| R2 | | 問題への対策案も提示して欲しい | 1～2割 |

ほとんどの回答者が，まず可視化されたことの意義を認めている．

機能（フィーチャー）でまとめることで，**高位意思決定者**に役立つ．

高位意思決定者と開発者に共通のビュー提供という目的を果たしている．

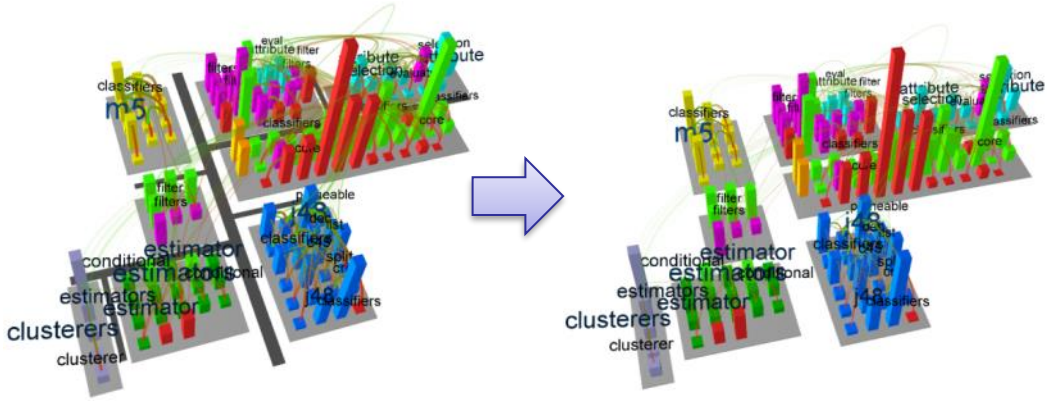
**開発者**にも有益なビューとなっている．

概念（クラスタリングや，依存関係グラフ）などは，直観的理解に及ばず，習熟や精査を要す．

# 適用を通じて観察された興味深い点

**メタファーは先入観による誤解を招き易い**
道はクラスタリングのデンドログラムの枝を意味する．これをデータフローと誤解したユーザがいたため，実適用では道は非表示とした．

**メタファーからの軽い逸脱は受容される**
カラフルなビルは現実的にはありえない
→ すべてのユーザが自然に受け入れた

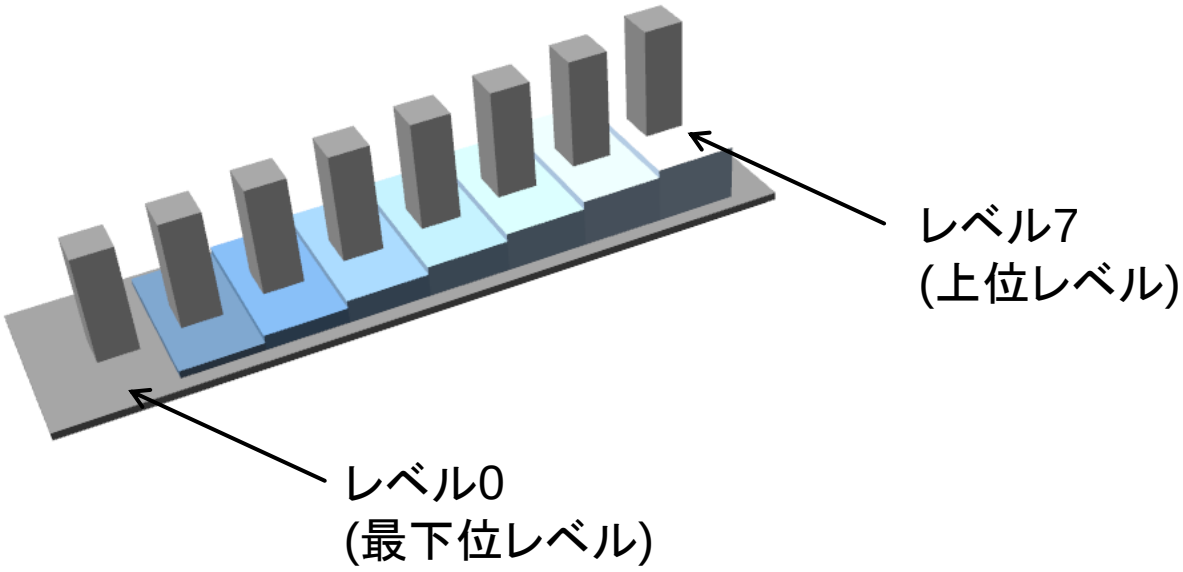



**ユーザに3D可視化に対する忌避は見られなかった**
概ね高い興味を持ち，肯定的に捉えていた．

- 経験報告まとめ
  - アンケート結果から，参考的ながらも，SArF Mapの狙いは受け入れられており，メタファーが大きな役割を果たしていると言える．
  - メタファーによりユーザの期待値が自然と高まっていた． 期待値に応えられているかどうかはこれから問われることになるが・・・

# 街区の中の斜面

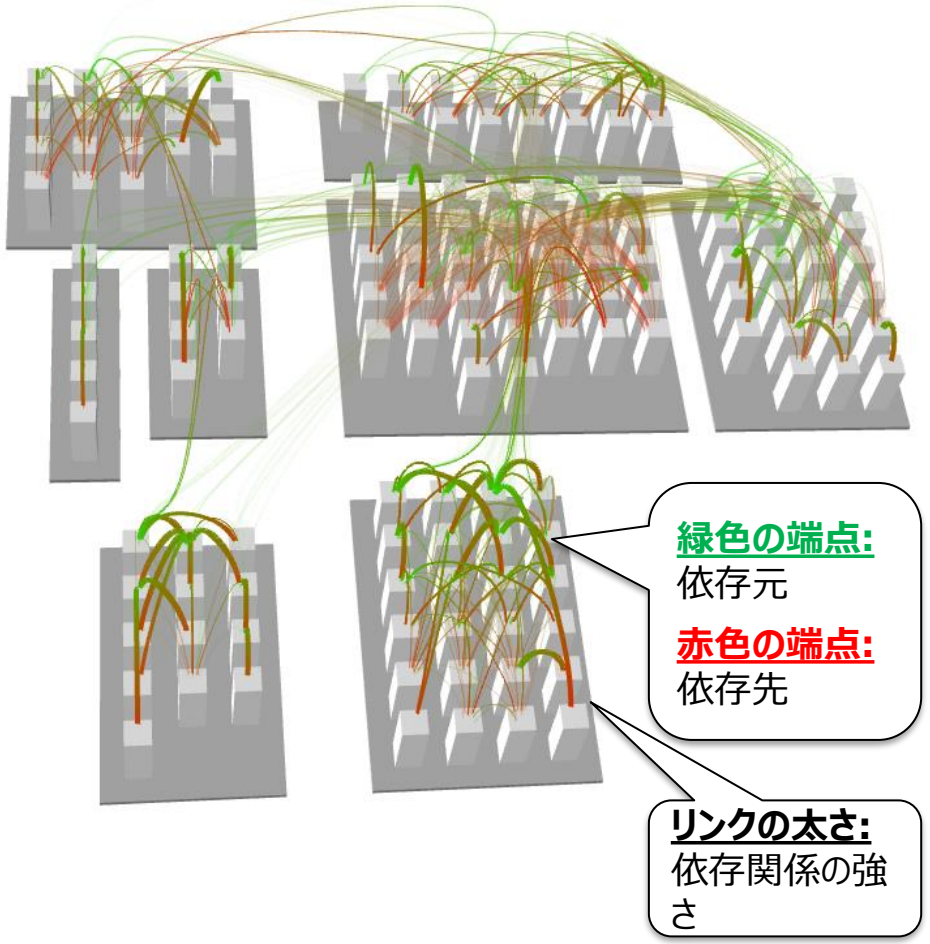
クラスごとに定まったレベルを，斜面上の高さで表現



# 依存関係の表現

依存関係を**3D Hierarchical Edge Bundle** [5-10,11,14]で表現

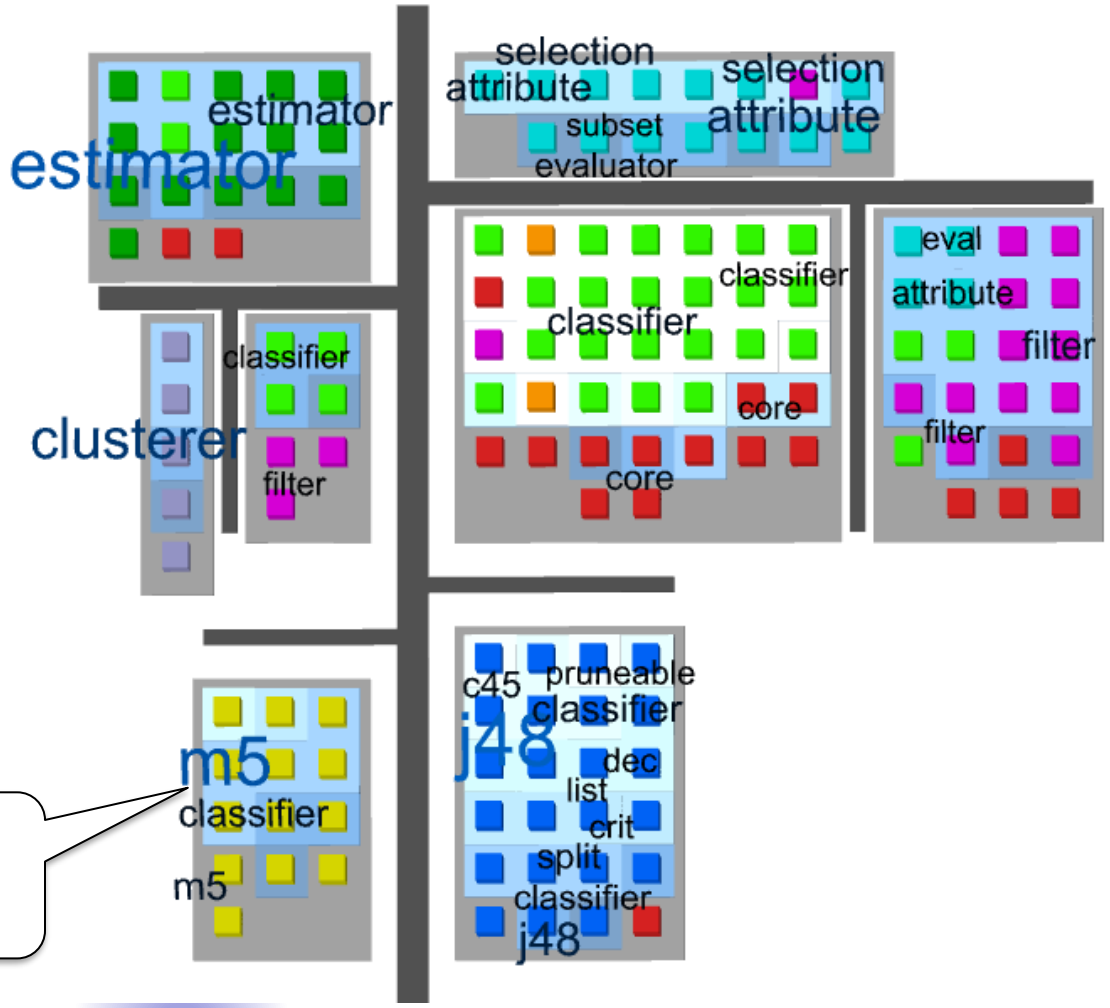
- ほとんどの依存関係はフィーチャー内に閉じる
- フィーチャー間の依存関係を容易に認識可能



# キーワード

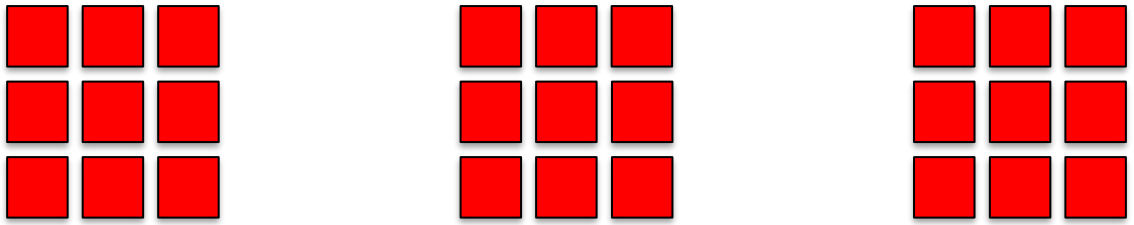
フィーチャーの理解支援のため，キーワードを抽出して表示

- クラスから候補語を抽出し，**局所的にtf-idf密度**が高い箇所に，そのワードを表示
- tf-idf = ワードの特徴量
  - ある文書で特に使われるワードはその文章を特徴づけるワードと言える



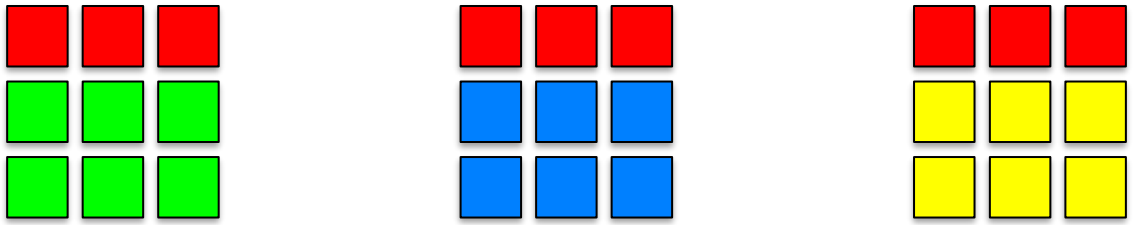
# 複数ブロックに渡るソースコード構成パターン

(a1) *Multi-feature Single-color*



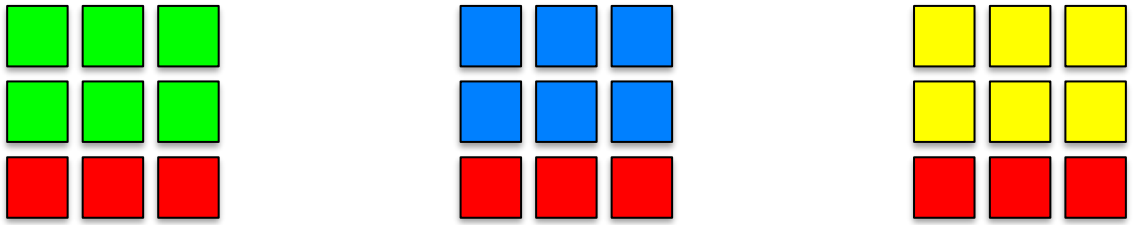
- **リファクタリングすべき**
- 地図から，パッケージ分割のヒントが読み取れる

(b1) *Multiple Layered - Dispatcher*



- 多くの場合，**よいデザイン**

(b2) *Multiple Layered - Utility / API*



- 多くの場合，**よいデザイン**

# まとめ

- ソフトウェア保守では継続的な知識損失が課題
- 知識損失しにくいソースコードから抽出した依存関係グラフを用いる保守効率化技術を提案し，評価
  - 2章 依存関係グラフを用いた欠陥予測
  - 3章 工数予測の精度向上のため対数変換と補正方法
  - 4章 依存関係グラフを用いたソフトウェアアーキテクチャの復元
  - 5章 依存関係グラフを用いたソフトウェアアーキテクチャの可視化