Geminiを用いた効果的なコードクローン分析方法

肥後 芳樹，吉田 則裕，楠本 真二，井上 克郎

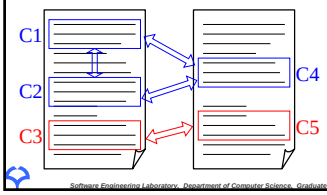
大阪大学 大学院情報科学研究科
{y-higo, n-yosida, kusumoto, inoue}@ist.osaka-u.ac.jp

はじめに

- 本発表では，より効率的にコードクローン分析を行うためのハウツーを紹介する
  - 紹介するハウツーはこれまでの経験から得られたものであり，特に理論的な根拠があるわけではない
- コードクローン情報だけでは，それらをどう扱うかの決定は難しい
  - 他の資産（ドキュメント，プロセス，開発者の知識など）とつき合わせて考えることが重要

コードクローン

- コードクローンとは
  - ソースコード中に存在する一致または類似したコード片
  - コピーアンドペーストなどのさまざまな理由により生成される
- ソフトウェアの保守を困難にする
  - あるコード片にバグがあると，そのコードクローン全てについて修正の検討を行う必要がある
- クローンペアとクローンセット



クローンペア	クローンセット
(C1, C2)	{C1, C2, C4}
(C1, C4)	{C3, C5}
(C2, C4)	
(C3, C5)	

コードクローン解析ツール

- コードクローン検出ツール: CCFinder[1]
  - 与えられたソースコード内に存在するコードクローンを検出
  - さまざまな言語に対応，C/C++, Java, COBOL, ...
  - 高いスケーラビリティ
  - CCFinderX
- コードクローン分析ツール: Gemini[2]
  - ICCAのサブシステムの一つ
    - Aries: リファクタリング支援
    - Libra: 修正支援
  - CCFinderの検出したコードクローンを視覚的に表示
  - メトリクスを用いたコードクローンの特徴づけ

[1] T. Kamiya, S. Kusumoto, and K. Inoue, "CCFinder: A multi-linguistic token-based code clone detection system for large scale source code", *IEEE Transactions on Software Engineering*, 28(7):654-670, 2002.
[2] Y. Ueda, T. Kamiya, S. Kusumoto and K. Inoue, "Gemini: Maintenance Support Environment Based on Code Clone Analysis", *Proc. Of the 8th IEEE International Symposium on Software Metrics*, 67-76, 2002.

利用実績

- 研究機関での利用
  - コードクローン情報を必要とする研究で使用
  - 多数の論文参照
- 産業界での利用
  - EASE，SEC関連プロジェクトでの利用
  - 試用・商用ソフトウェア開発プロセスへの導入
  - 国内外100社以上で利用
- その他
  - プログラム著作権関係の裁判証拠
  - 大学の演習

目次

1. 検出オプション 最小一致トークン数

- 万能な値は存在しない
  - プログラミング言語，ソフトウェアの規模，ドメインに応じて検出されるコードクローンの量は異なる
  - これまでの傾向としては，
    - 同規模（総行数がほぼ同じ）のソフトウェアの場合，C言語の（手続き型）プログラムの方がJava言語の（オブジェクト指向）プログラムよりも多くクローンを含む傾向がある
    - GUIのプログラムの方が，CUIのプログラムのよりもクローンを多く含む傾向がある
- 新規でコードクローン分析を行う場合は 30トークンで
  - あまりクローンが検出されないようであれば，値を下げて再検出
  - あまりに多くのクローンが検出されるのであれば，値を上げて再検出

1. 検出オプション トークンの正規化

- CCFinderはデフォルト設定では，ユーザ定義名や型名などを表すトークンを特別なトークンに置き換えた後に，クローン検出を行う
  - 変数名などが異なるコード片をクローンとして検出できる
  - 偶然の一致により，クローンとして検出されてしまうコード片がある
- 新規でクローン分析を行う場合は，デフォルト設定で
  - 偶然の一致により，あまりにも多くのクローンが検出されているようであれば，特定の正規化オプションを切る，などの対象が必要
    - 例：キャスト名を正規化しない

1. 検出オプション グループの作成（1/2）

- CCFinderは以下の三種類のクローンの検出・非検出をそれぞれ設定することが可能（デフォルト設定では，全てのクローンを検出する）
  - ファイル内クローン
  - グループ内ファイル間クローン
  - グループ間クローン
- 対象ファイルを指定しただけでは，グループは設定されていない
  - ファイル内クローン，グループ内ファイル間クローンのみを検出している
- グループを設定することで，より有益な検出結果を得ることができる
  - グループを設定していない場合の「グループ内クローン」が「グループ内ファイル間クローン」と「グループ間クローン」に分けて検出される

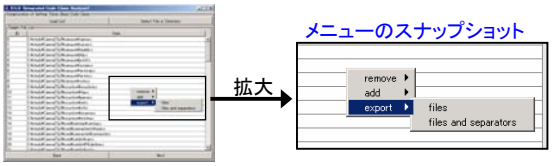
1. 検出オプション グループの作成（2/2）

- 適切なグループ設定の例
  - 一つのディレクトリ内に含まれるファイル群を一つのグループに
  - 一つのモジュールを構成しているファイル群を一つのグループに
  - 前者は「マウスの右クリック → add → separator → every directory」で簡単に行うことが可能

- ファイル間の類似度と共に，グループ間の類似度を得ることができる

1. 検出オプション 対象ファイル

- コードジェネレータが生成したコード（ファイル）はクローン検出対象とすべきではない
  - コードジェネレータが生成したコードは非常に多くのクローンを含む
- 何度も同じ対象ファイルからクローン検出を行う場合
  - ファイルリストをつくると便利
    - 「マウスの右クリック → export → files」
    - 「マウスの右クリック → export → files and separators」



2. 重要でないクローンのフィルタリング

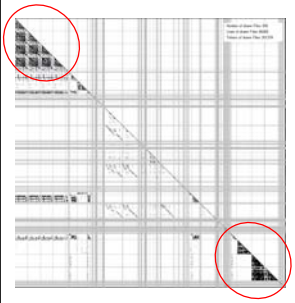
- CCFinderの検出するコードクローンはトークンの列であり，重要でないコードクローンを多数検出してしまう
  - switch文の各caseエントリ
  - 連続したimport文,printf文, scanf文 など
- フィルタリングメトリクス ***RNR(S)***
  - クローンセット S に含まれるコード片の非繰り返し度を表す
- 例 トークン列 <***x a b c a b c* a* b* c* y***>
  - CCFinder は以下の二つのコード片をコードクローンとして検出
    - *x a b c a b c*_{<F1>} a* b* c* y*
    - *x a b c a b c* a* b* c*_{<F2>} y*
    - F1はコード片の長さが6トークン，そのうち5トークンが非繰り返し
    - F2はコード片の長さが6トークン，そのうち2トークンが非繰り返し
  - $RNR(S_1) = (5 + 2)/(6 + 6) = 7/12 = 0.583$

3. 大まかな把握

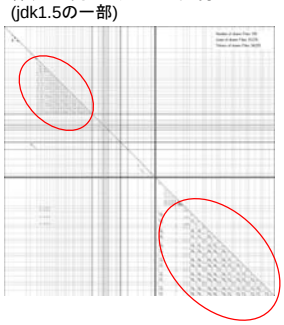
- 新規でクローン分析を用いる場合（分析の初期段階）に有効
  - クローンの量・分布状態をひと目で把握できる
- スキャタープロットで以下の二つの部分が目立ちやすい部分である
  - 一定の領域内にコードクローンが密集している部分
  - 同じようなパターンが繰り返し出現している部分
- スキャタープロットで目立つ部分に特徴的なクローンが存在するとは限らない
  - 複数種類のクローンが存在した結果，その場所が目立っている
- メトリクス *RNR* の値が閾値未満のコードクローンは青色，以上のコードクローンは黒色で描画
  - 閾値はユーザが自由に設定可能

3. 大まかな把握

クローンが密集している(Antlr)



繰り返し同じパターンが出現
(jdk1.5の一部)



4. 特徴的なクローンとその対処法 同形のコード片が多いクローン

- バグが検出された場合，多くの箇所に同様の修正を加えなければならない
  - 不安定（繰り返し修正が行われる）なコード
    - 修正コスト削減に向けての対策が必要（リファクタリングなど）
  - 安定したコード，定型処理部分などもこのようなクローンになりがち．
    - 例：データベースへのアクセス部分
  - プログラミング言語の文法上どうしてもクローンになってしまう．
    - 例：switch文（連続したcaseエントリ）
- ***RNR*** を用いることである程度の絞込みは可能
- 「6. 現在の取り組み」で上記のクローンへの判別手法を紹介

4. 特徴的なクローンとその対処法 トークン数の多いクローン

- コピーアンドペーストにより生成されたものではないかと思われる
  - ペースト後の変数名やメソッド名の修正漏れがバグに繋がる
  - 修正漏れのチェックを行うのは効果的な予防保守
- 実際のプロジェクトのコードからバグを検出
  - 単体テスト後のコードを分析
  - 見つかったバグ概要（検出された最もトークン数の多いクローン内）
    - ファイル A.cpp とファイルB.cppがクローンを共有
    - ファイルAではxxxAxxxというメソッドが呼ばれている
    - ファイルBではxxxBxxxというメソッドが呼ばれている
    - ファイルBの中で一箇所だけxxxAxxxというメソッドが呼ばれていた
      - ✓ ファイルAからファイルBへのコピーアンドペーストを行い，修正を忘れた

4. 特徴的なクローンとその対処法 多くのファイルを巻き込んでいるクローン

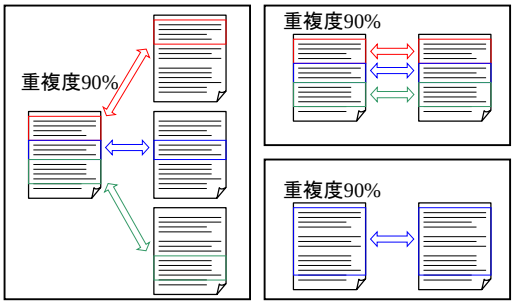
- 根本的な問題を表している可能性がある
  - 設計が悪いことを暗示
  - プログラミング言語に適切な抽象化機構が存在しない(横断的関心事)
- ***RNR*** を用いることである程度の絞込みは可能
- 「6. 現在の取り組み」で上記のクローンへの判別手法を紹介

5. 特徴的なファイルとその対処法 他グループと多くのクローンを共有しているファイル

- 特定のグループのファイルと多くのクローンを共有している
  - ファイルの位置と実装している機能にずれがある
    - 他の場所に移動させる
- 複数のグループのファイルと多くのクローンを共有している
  - 多くのことを行い過ぎている
  - ファイルを分割

5. 特徴的なファイルとその対処法
特定のファイルと非常に類似度が高いファイル

- 特定のファイルと非常に類似度が高いファイル
  - 本当にそれらのファイルは全て存在することが必要か？

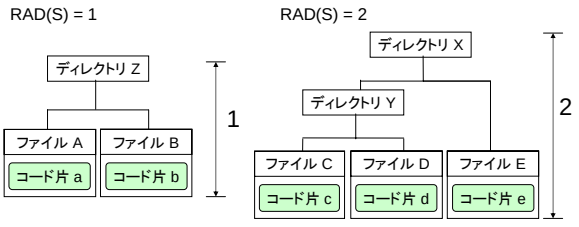


6. 現在の取り組み
クローン発生理由の予測

- 全てのクローンが「悪」ではない
- なぜクローンが発生したのか，どのような性質なのか，を知ることが重要
  - アドホックな開発
  - 安定コード，定型処理の使いまわし
  - プログラミング言語の機能的な制限
  - …
- 二つのメトリクス ***NIF(S)*** と ***RAD(S)*** を使って発生理由を予測
  - NIF(S)：クローンセット S に含まれるコード片を所有しているファイル数
  - RAD(S): クローンセット S に含まれるコード片のファイルシステム中での分散度

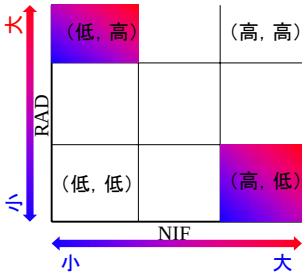
分散度メトリクス RAD(S)

- クローンセット S に含まれるコード片のファイルシステム中での分散度



6. 現在の取り組み
クローン発生理由・性質の予測

- 二つのメトリクスの「高・低」の組み合わせを用いて四つのカテゴリを作成
- それぞれのカテゴリに属するクローンの性質は異なるのではないか
  - (低，低)：特徴的な機能のクローン，共通化の対象
  - (高，低)：(低，低)と同様
  - (高，低)：機能の分割がうまく行われていない
  - (高，高)：プログラミング言語に依存したクローン
    - 文法上クローンになりがち
    - 抽象化機構が存在しない



7. 今後の取り組み

- クローンのブックマーク機能
  - 全てのクローンを自動的に正しく分類することは不可能
  - 人間が手動で分類する支援
  - 確認したクローンにチェックを入れる
    - 既に確認したという情報を残す
    - 必要でないクローンであればクローン情報から消す