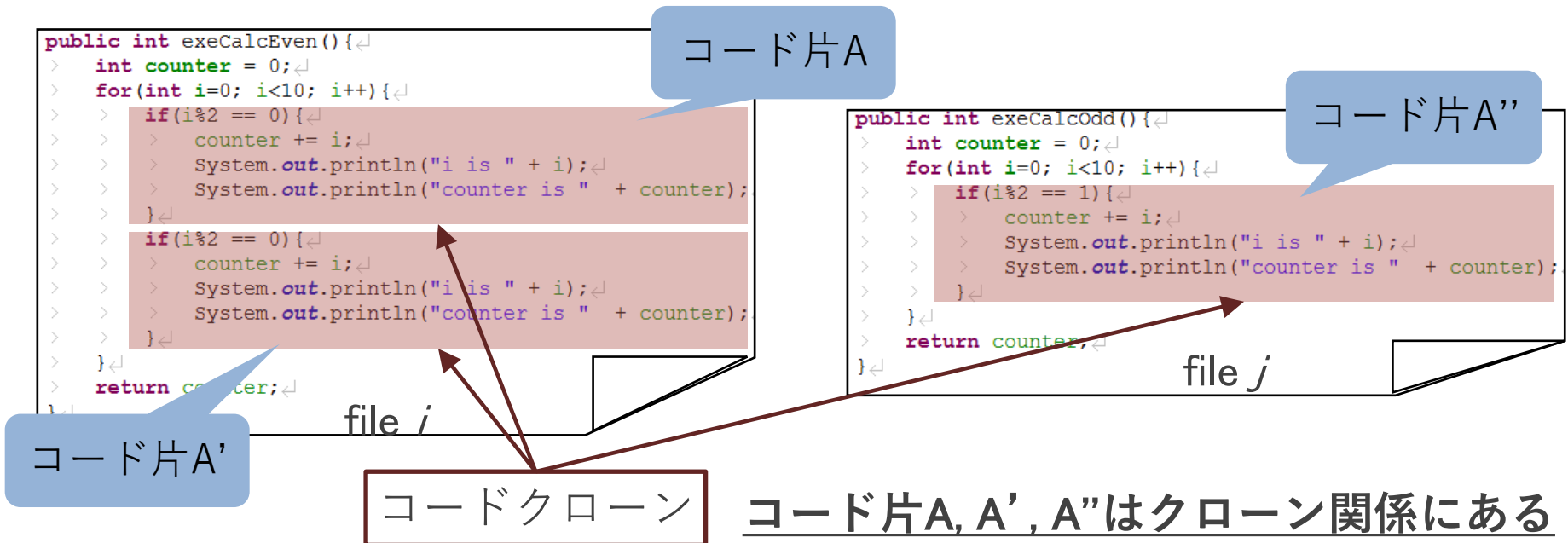


コードクローンの作成・利用過程と品質 の関係を分析するためのツールの構築

久木田 雄亮

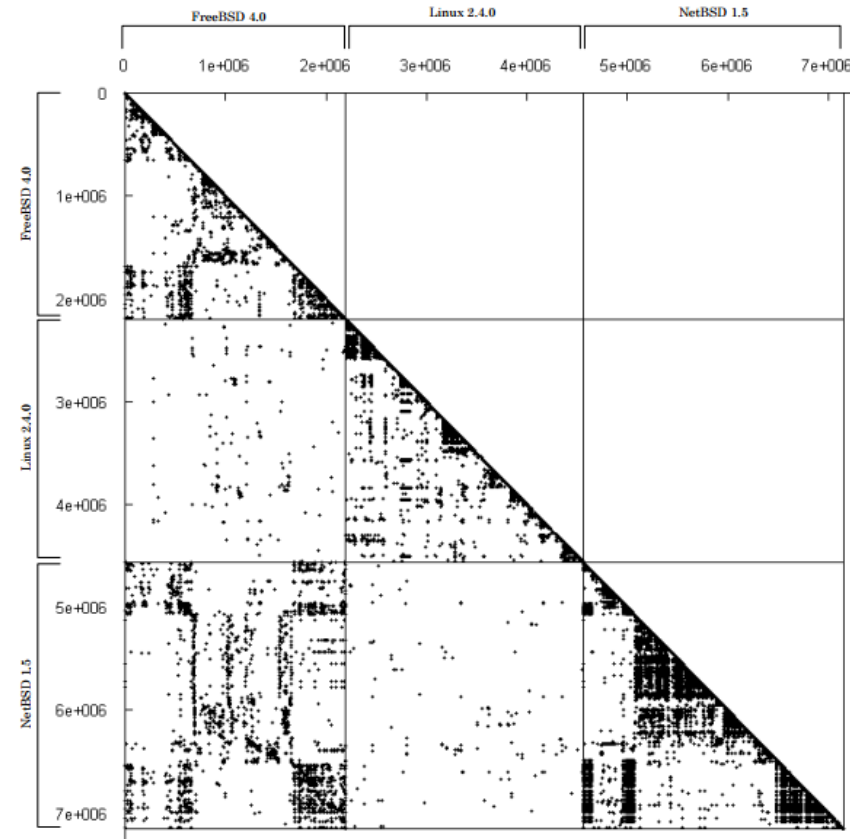
コードクローンとは

- ソースコード中に存在する「同一の」または「類似する」コード片のこと
 - 主に、既存コードを再利用（コピーアンドペースト）することにより作成される



研究の背景 (1/2)

- ソフトウェア開発・保守の過程では、生産性の向上を目的として既存コードが頻繁に再利用されることがある
- コードクローンがソースコード全体に遍在することが少なくない

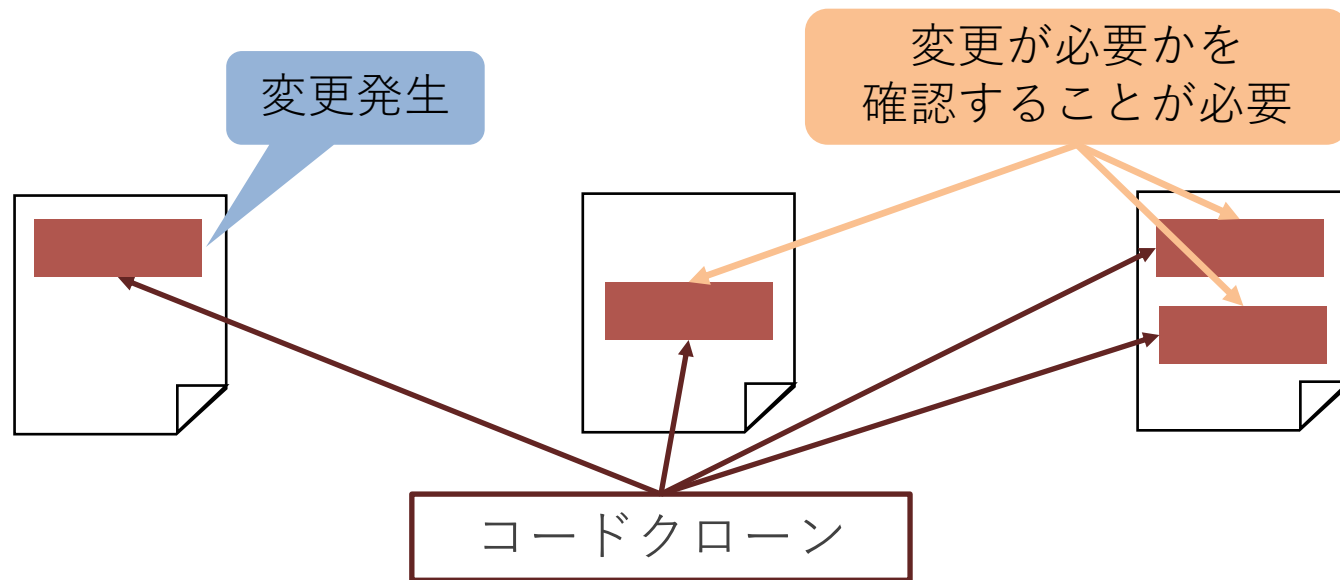


UNIX系OS, Linux間のコードクローン[1]

[1] Kamiya, T., Kusumoto S. and Inoue K.: CCFinder: A multilinguistic token-based code clone detection system for large scale source code, IEEE Transactions on Software Engineering (TSE), Vol.28, No.7, pp.654-670, 2002.

研究の背景 (2/2)

- 欠陥除去などの理由によりコードクローンに変更を加える際のリスク
 - 変更箇所が多岐に渡るため保守コストが増加する
 - 変更漏れにより新たな不具合を混入してしまう



これまでのコードクローン研究

- コードクロンの検出手法
 - 主眼：大規模なシステムのソースコードを高速に解析しコードクロンを検出すること
 - ソースコードをトークン（リテラル，括弧，演算子）の並びで表現し，一致する並びをクローンとして検出[1]
- コードクロンの分析手法
 - 主眼：コードクロンの特徴と品質（バグ発生確率）との関係を明らかにすること
 - コードクロンのタイプ（違いの度合い）ごとのバグ発生確率に関する分析[2]

[1] Kamiya, T., Kusumoto S. and Inoue K.: CCFinder: A multilinguistic token-based code clone detection system for large scale source code, IEEE Transactions on Software Engineering (TSE), Vol.28, No.7, pp.654-670, 2002.

[2] M. Mondal, C. K. Roy and K. A. Schneider, "A comparative study on the bug-proneness of different types of code clones," IEEE International Conference on Software Maintenance and Evolution (ICSME), pp. 91-100, 2015.

先行研究の問題点

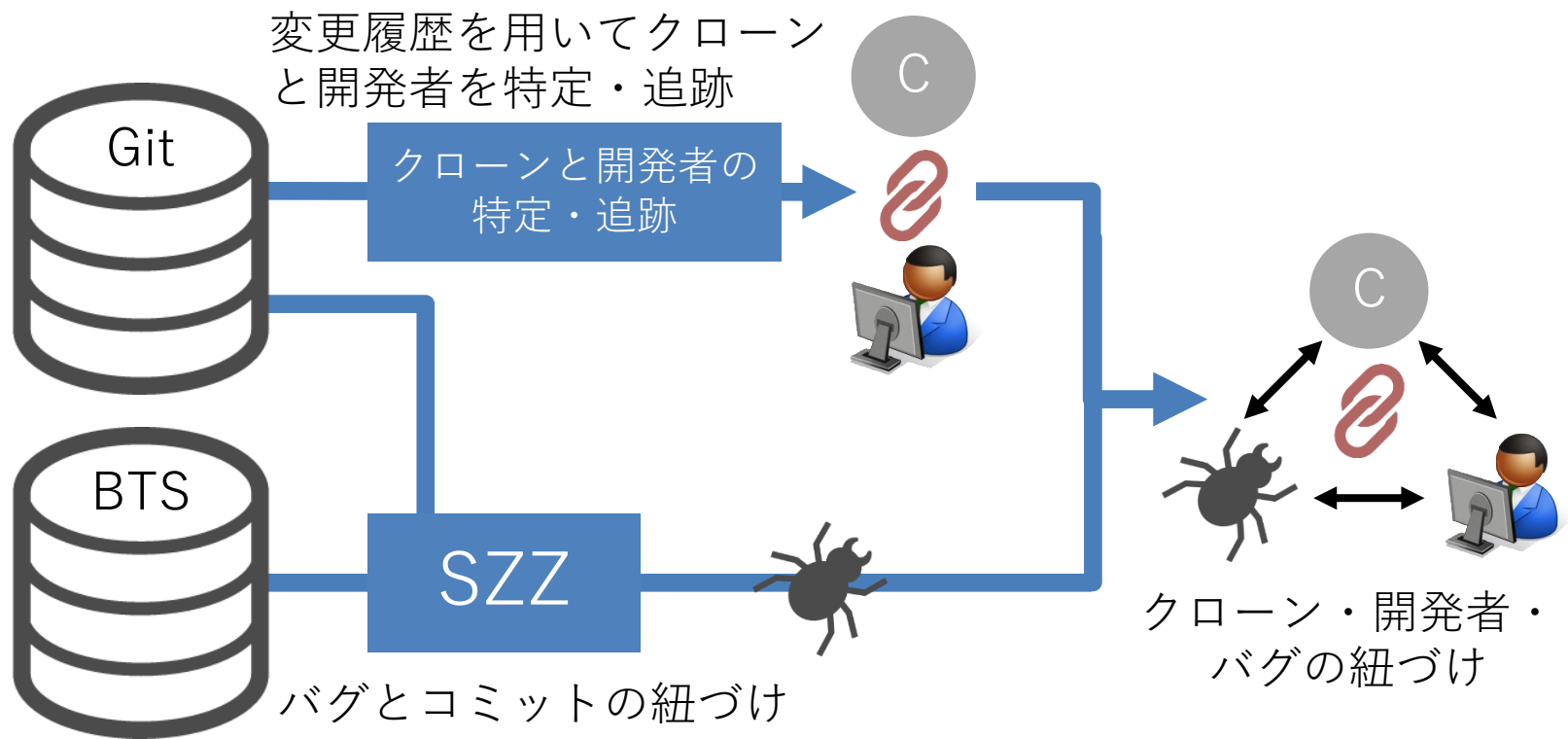
- コードクローンの作成・利用過程と品質の関係に着目した分析がおこなわれていない
 - 必要な理由（先行研究で明らかにできないこと）
 - クローンはソースコード全体に遍在しているので変更回数が多いと修正漏れの発生も増えるのではないか
 - 変更間隔が長すぎると作成した本人でさえクローンであるかを認識できなくなり、修正漏れやバグを埋め込むのではないか
 - 普段から共同で作業を行っている開発者同士であれば、開発をする上での意思疎通がとれているので、共同でクローンを作成してもバグの発生率が小さくなるのではないか
 - 開発者の経験年数が長いとプロジェクトに関する理解が深いので、作成・利用するクローンに埋め込むバグは少なくなるのではないか

研究の目的

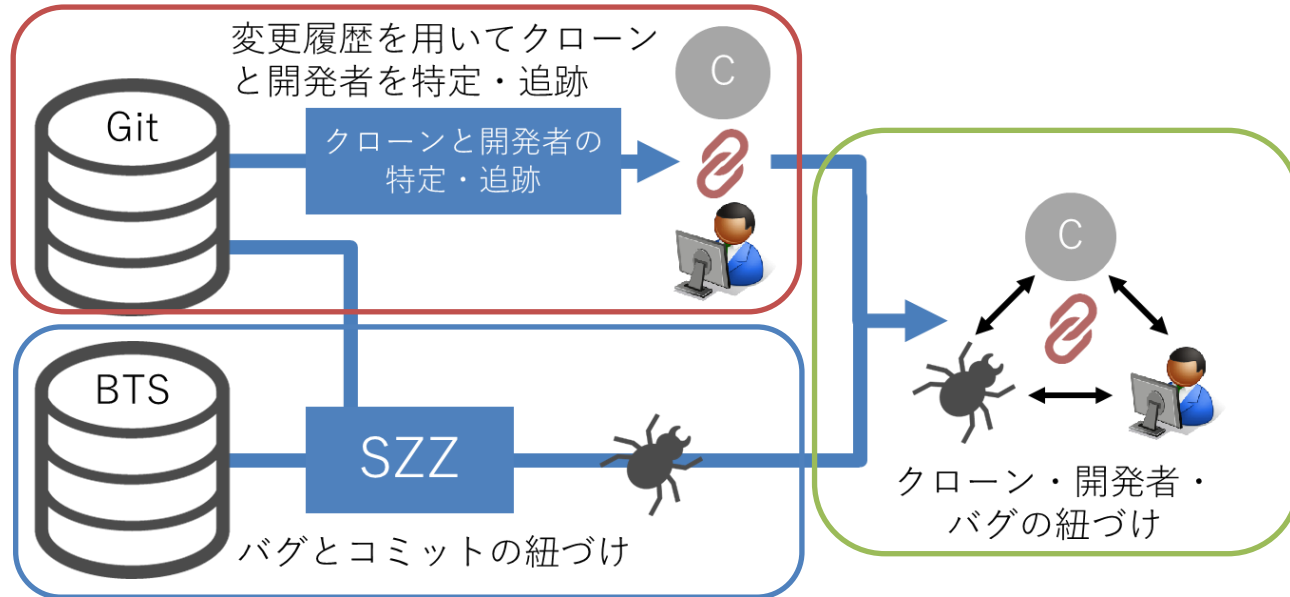
- 目的
 - コードクロンの長期的かつ安全な管理方法について、**コードクロンの作成・利用過程の観点**から有益な知見を提供すること
- アプローチ
 - コードクロンの作成・利用過程における人的影響を調査するためのツールを構築する
 - コードクロンの作成・利用過程と開発者の追跡
 - SZZアルゴリズムによるバグとクロンの紐付け
 - コードクロン作成・利用過程と品質の関係の分析

CCT: Code Clone Tracer

- コードクローンの作成・利用過程における人的影響の調査を支援するためツール



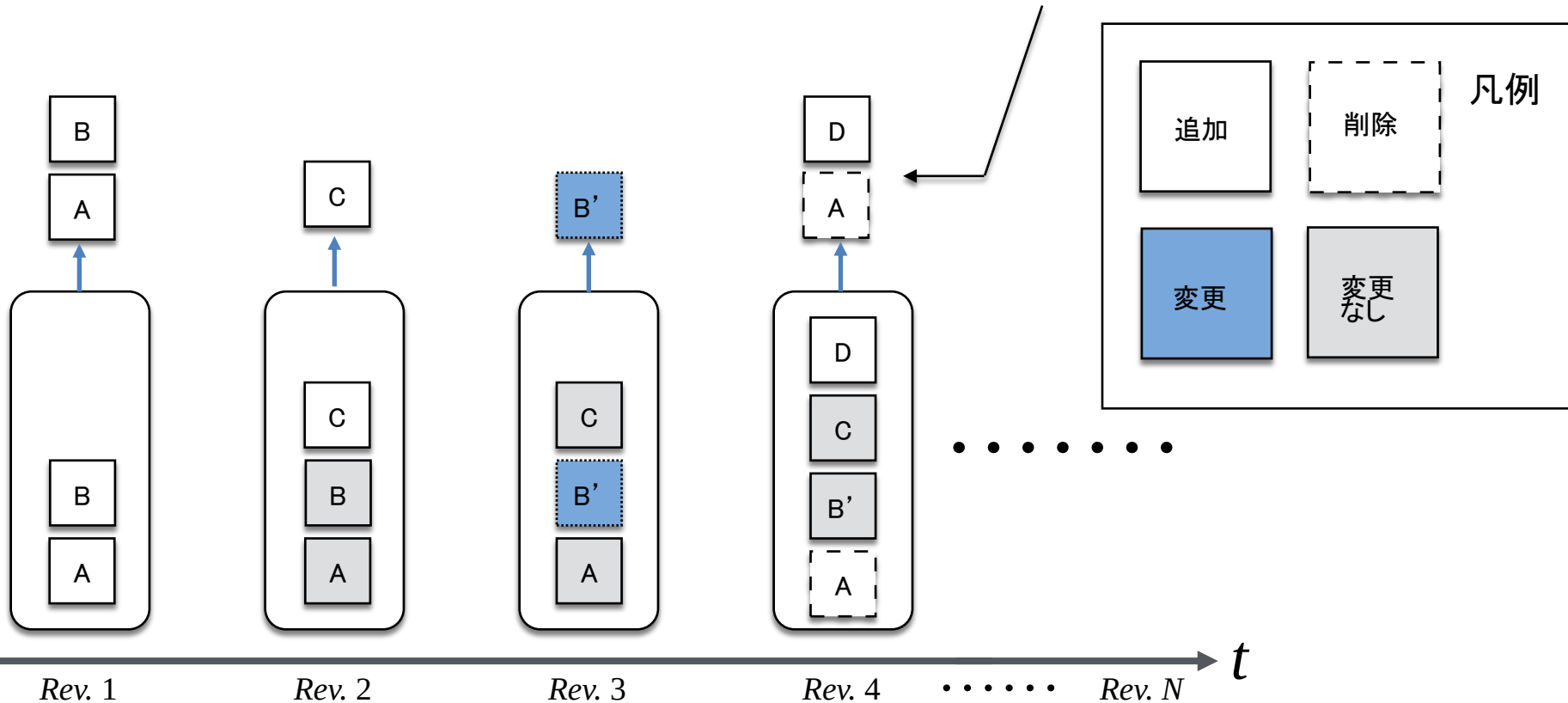
CCTの処理手順



- (1) コード片を特定
- (2) クローンを特定
- (3) クローンを追跡
- (4) クローンの作成・利用に関与した開発者を特定
- (5) バグが混入したコミットを特定
- (6) バグと開発者とクローンの紐づけ

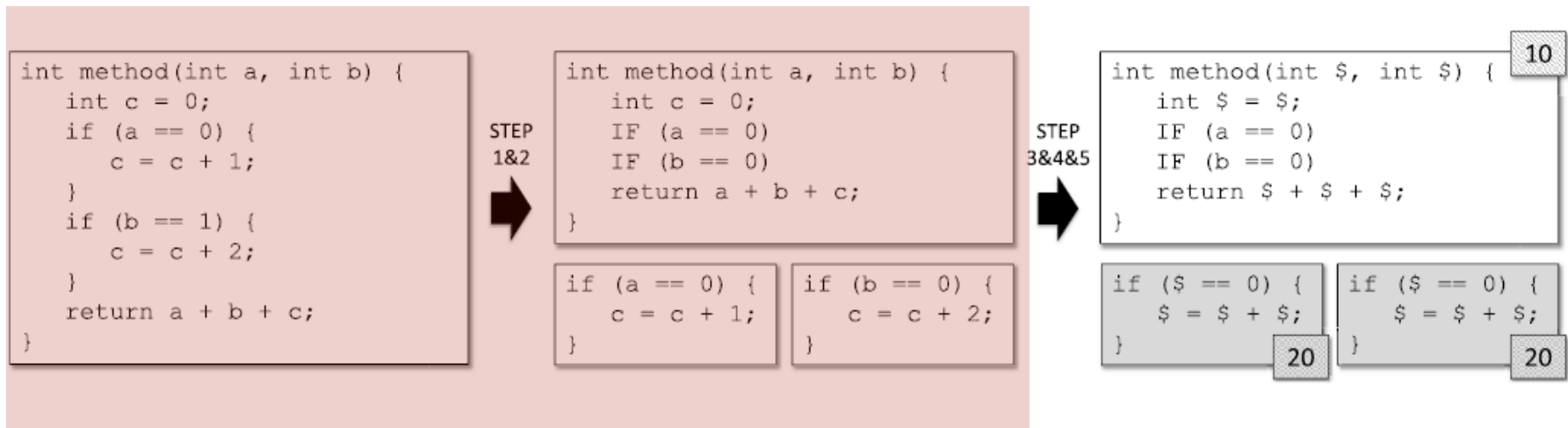
CCTの処理手順(1)：コード片特定

- 分析対象ファイル中に含まれるすべてのコード片を特定
 - 追加・変更・削除されたファイルをすべてのリビジョンに対して特定する(コード片特定対象となるファイルを抽出する)



CCTの処理手順(2)：コード片特定

- 分析対象ファイル中に含まれるすべてのコード片を特定
 - 2. 抽出したファイル差分を対象にブロック単位でコード片を特定
ブロック：クラスやメソッド、for文やif文などのブロック文

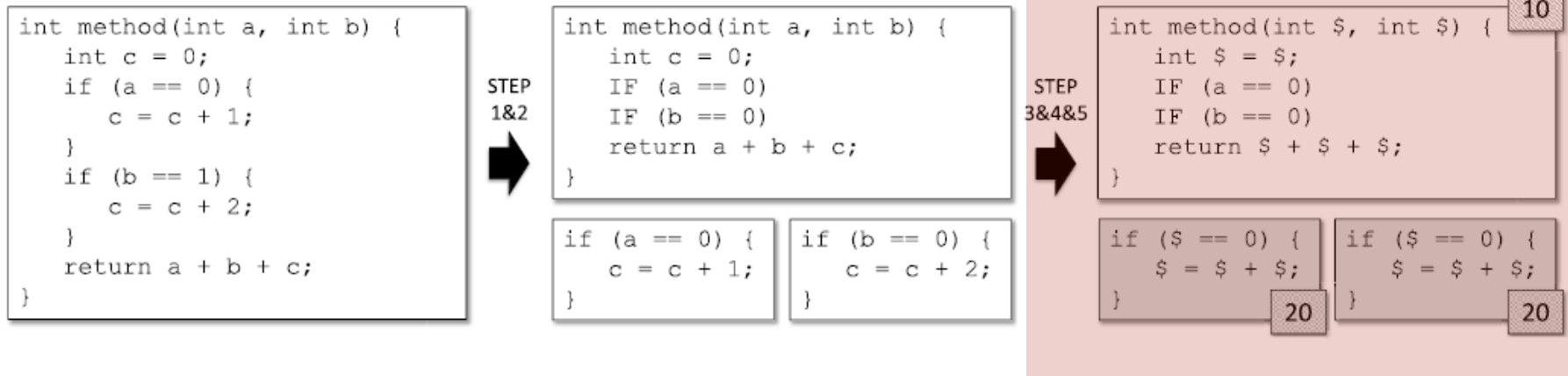


ブロックの検出・正規化・ハッシュ化の流れ([3]より抜粋)

[3] 堀田圭佑, 楊嘉晨, 肥後芳樹, 楠本真二: 粗粒度なコードクローン検出手法の精度に関する調査, 情報処理学会論文誌, Vol.56, No.2, pp. 580–592, 2015.

CCTの処理手順(2)：クローン特定

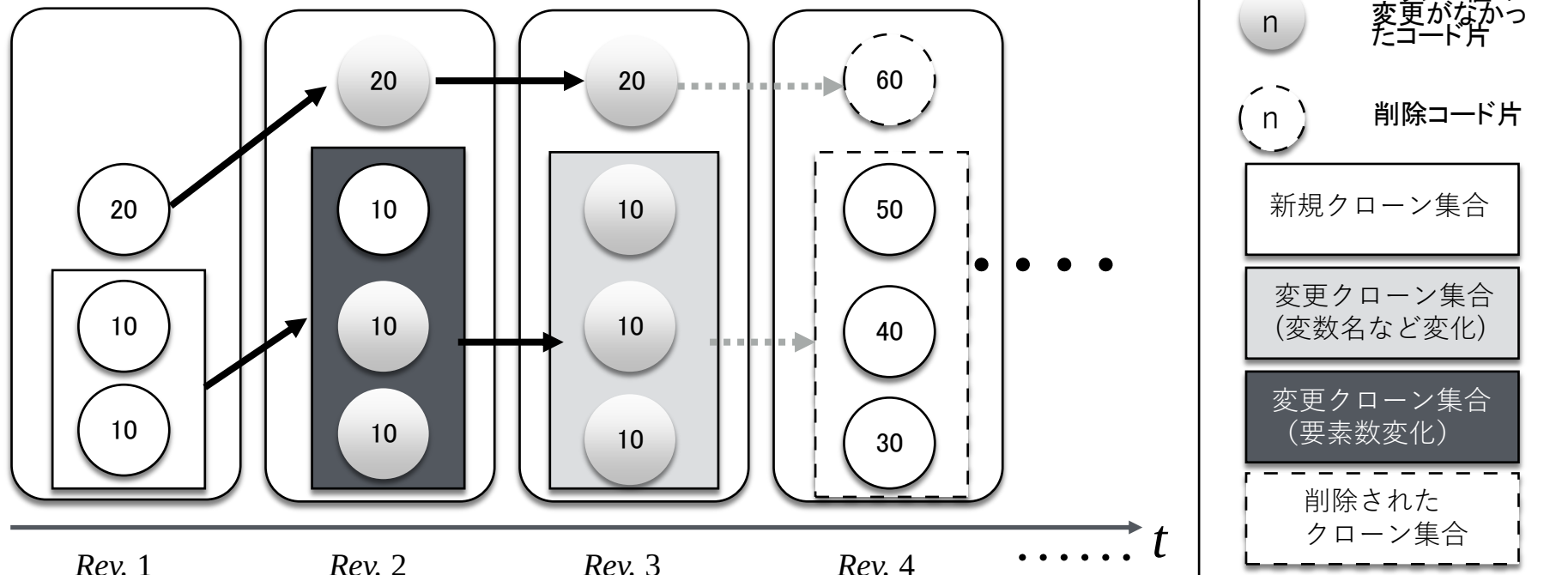
1. ブロックに含まれる変数名とリテラルを特殊文字で正規化
2. 正規化されたブロックの文字列をもとにハッシュ値を算出



3. ハッシュ値の一致するブロックをクローンと見なしクローンを特定する

CCTの処理手順(3)：クローン追跡

- リビジョン間でのクローン集合の前後関係を追跡できるようにする
 - 追加・変更・削除されたコード片のハッシュ値を調べ、直前のリビジョンのクローンのハッシュ値と比較する



CCTの処理手順(3)：クローン追跡

- クローン関係が解消されない小さな変更があった場合
CRD(Clone Region Descriptor)[4]を用いて追跡
 - ブロック内に複雑な変更（クラスの参照など）がなければ追跡が可能

```
public class DeleteManager {  
    ...  
    public void delete(int n) {  
        ...  
        for (int i = 0; i < delete.size(); i++) {  
            ...  
            if (delete.get(i) instanceof ElementNode) {  
                // some code  
            }  
        }  
        ...  
    }  
}
```

CRD

package: packageName.DeleteManager.java, DeleteManager⑤
delete(int)⑤
for, delete.size()④
if, delete.get(i) instanceof ElementNode②

```
<CRD> ::= <file> <class> <CM> [<method>]  
<method> ::= <signature> <block> *  
<block> ::= <btype> <anchor> <CM>  
<btype> ::= 'for' | 'while' | 'do' | 'if' | 'else' | 'switch'  
            | 'try' | 'catch' | 'finally' | 'synchronized'
```

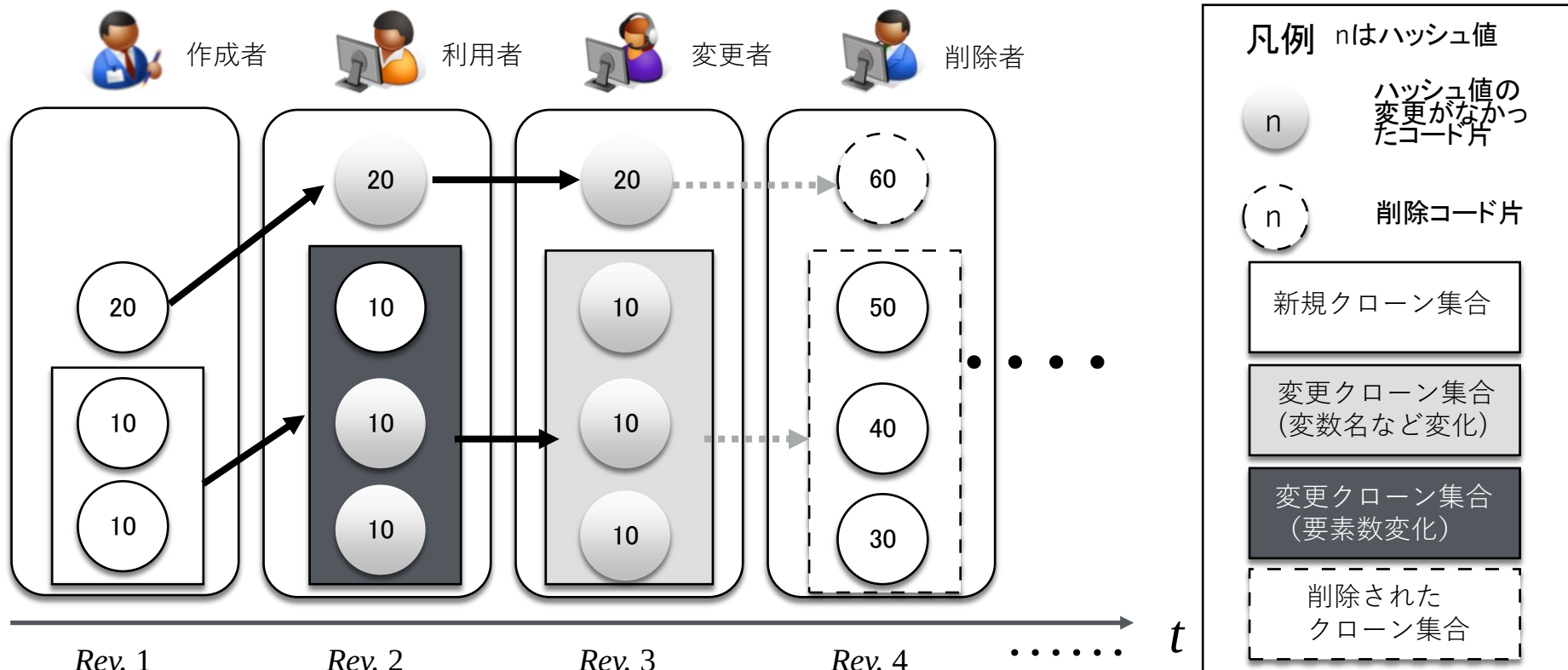
○ CM：各ブロックのメトリクス値
(サイクロマチック数 + fan-out)

CRDの文字列が一致・類似するものを同じブロックとして追跡する

[4] 堀田圭佑他, “CRDを用いたコードクローンの生存期間と修正回数に関する調査”, 情報処理学会論文誌, Vol.55, pp947-958, 2014年2月

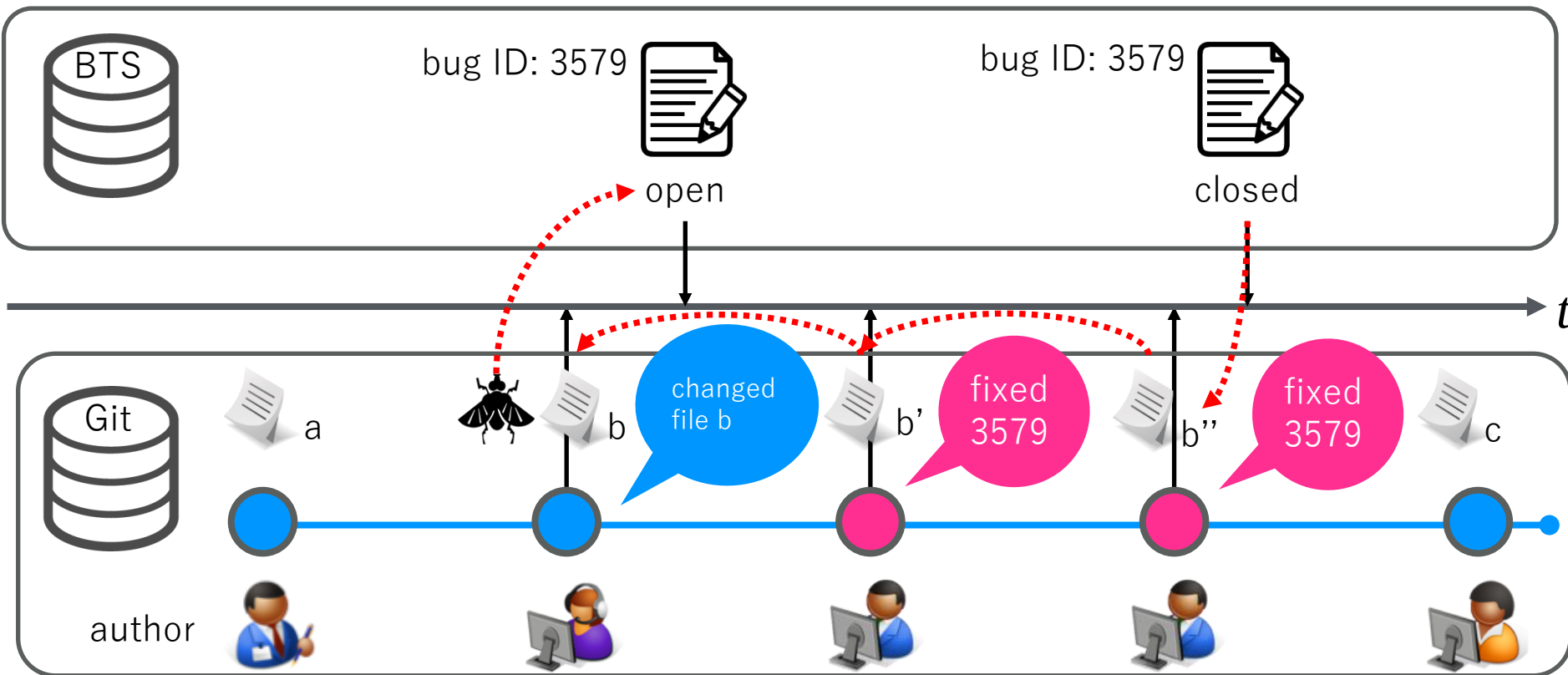
CCTの処理手順(4)：開発者の特定

- クロンの作成・利用過程に関与した開発者を4種類に分けて特定する



CCTの処理手順(5)：バグ混入コミットの特定制

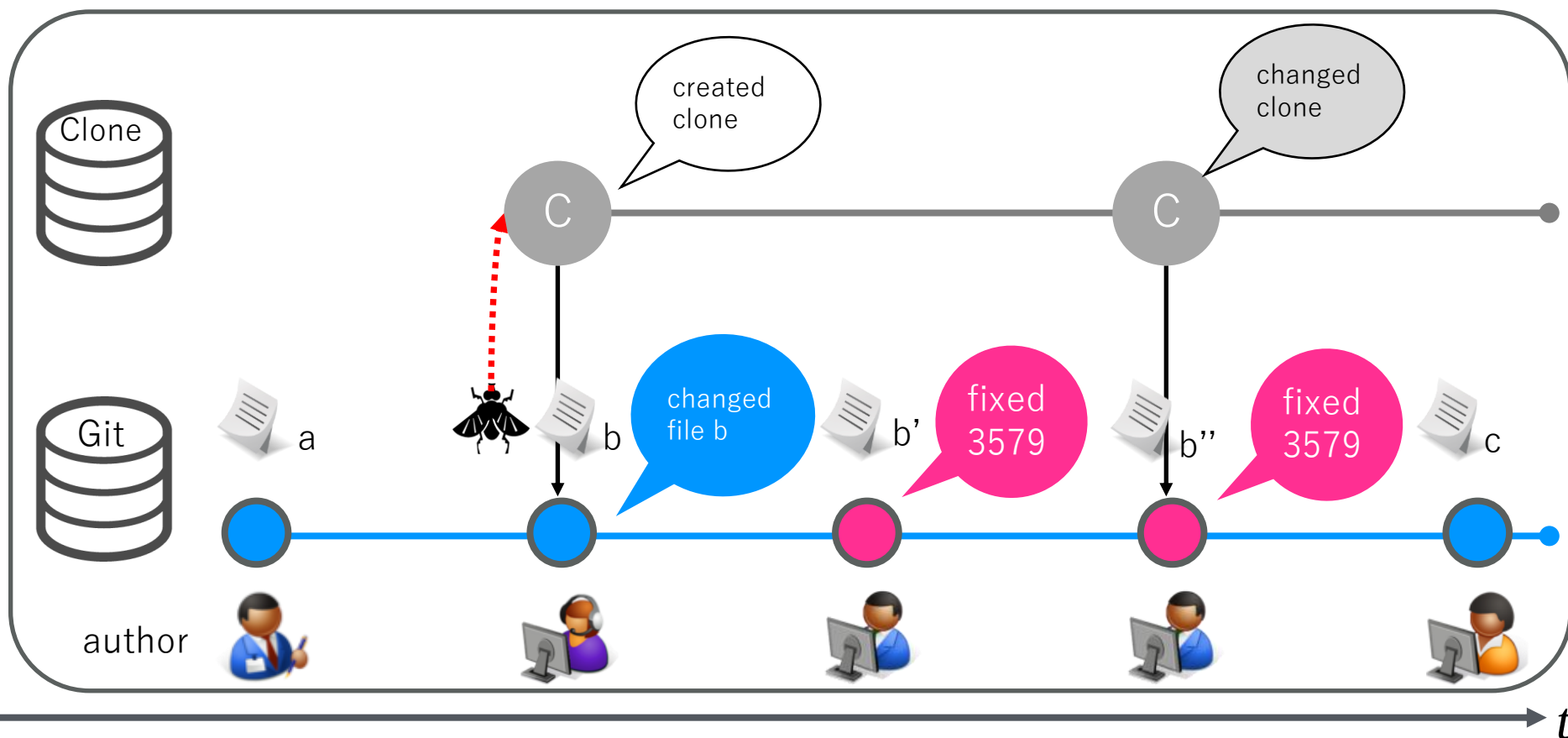
- SZZアルゴリズム[5]を用いてバグ混入コミットを特定し、バグを混入した開発者を特定する



[5] S 'liwerski, J., Zimmermann, T. and Zeller, A.: When Do Changes Induce Fixes?, Proceedings of the 2005 International Workshop on Mining Software Repositories (MSR2005), pp.1-5 2005.

CCTの処理手順(6)：バグと開発者とクローンの紐づけ

- どのリビジョンで開発者がクローンを作成・利用したかの情報とどのリビジョンでバグが混入したの情報を組み合わせて紐づけを行う



既存研究のツールとの違い

	CCFinder[1]	ECTEC[6]	CCT
コードクローン検出	◎	○	○
コードクローン追跡	×	○	○
クローンとバグの紐づけ	×	×	○

- CCFinder[1]
 - リテラルの並びが一致するものをクローンとして検出
 - 精度は良いが、検出速度が遅い
- ECTEC[6]
 - ブロック文を変換したハッシュ値の比較によるクローン検出
 - CRD(クローンの位置情報を示す表示形式)によるクローン追跡
- CCT
 - ハッシュ値とCRDに基づくクローン検出・追跡(ECTECの手法と同様)
 - SZZアルゴリズムを用いたクローンとバグの紐づけ

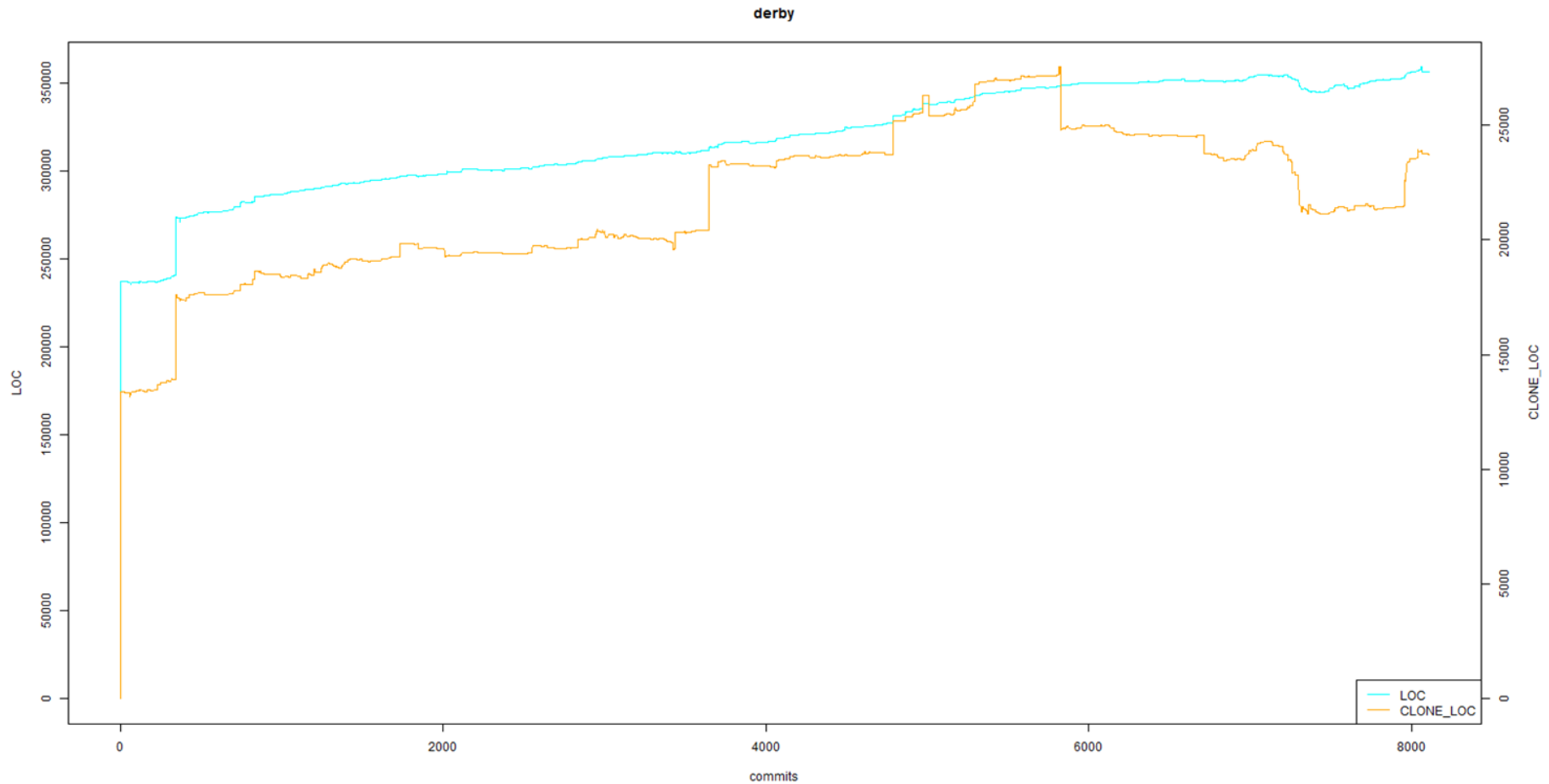
[1] Kamiya, T., Kusumoto S. and Inoue K.: CCFinder: A multilingual token-based code clone detection system for large scale source code, IEEE Transactions on Software Engineering (TSE), Vol.28, No.7, pp.654-670, 2002.

[6] Yoshiki Higo, Keisuke Hotta, and Shinji Kusumoto. Enhancement of CRD-based clone tracking, In Proceedings of the 2013 International Workshop on Principles of Software Evolution (IWPSE 2013), pp.28-37, 2013.

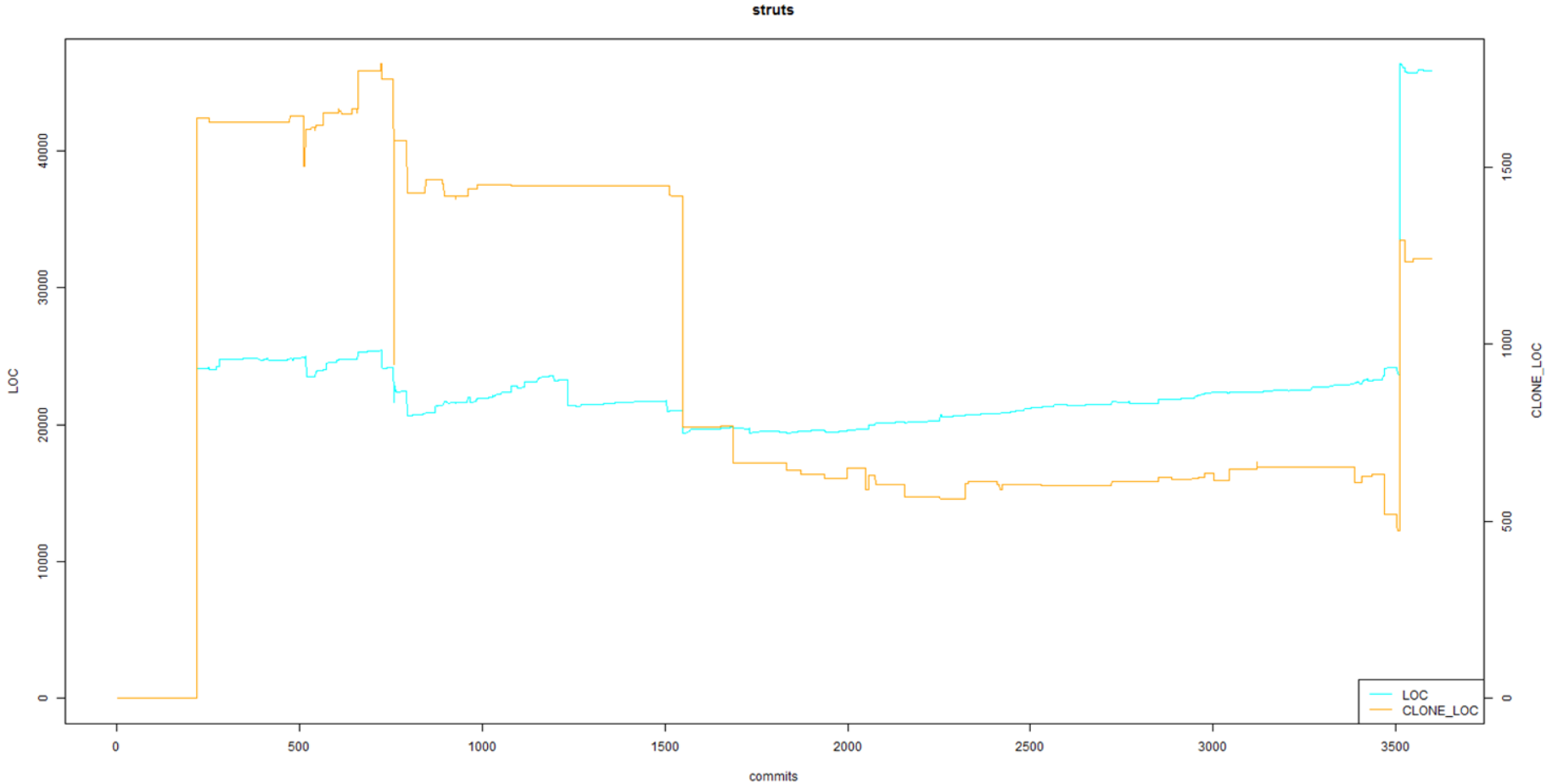
解析対象プロジェクト

プロジェクト名	期間	開発者数	ファイル数 (最新Rev)	クローン数 (最新Rev)	バグ有クローン数 ／ バグ無クローン数
Derby	2004/08 ~2016/06	37	1807	557	846/961
Struts	2006/02~ 2016/04	61	740	81	122/71

DerbyのLOCとCLONELOC



StrutsのLOCとCLONELOC



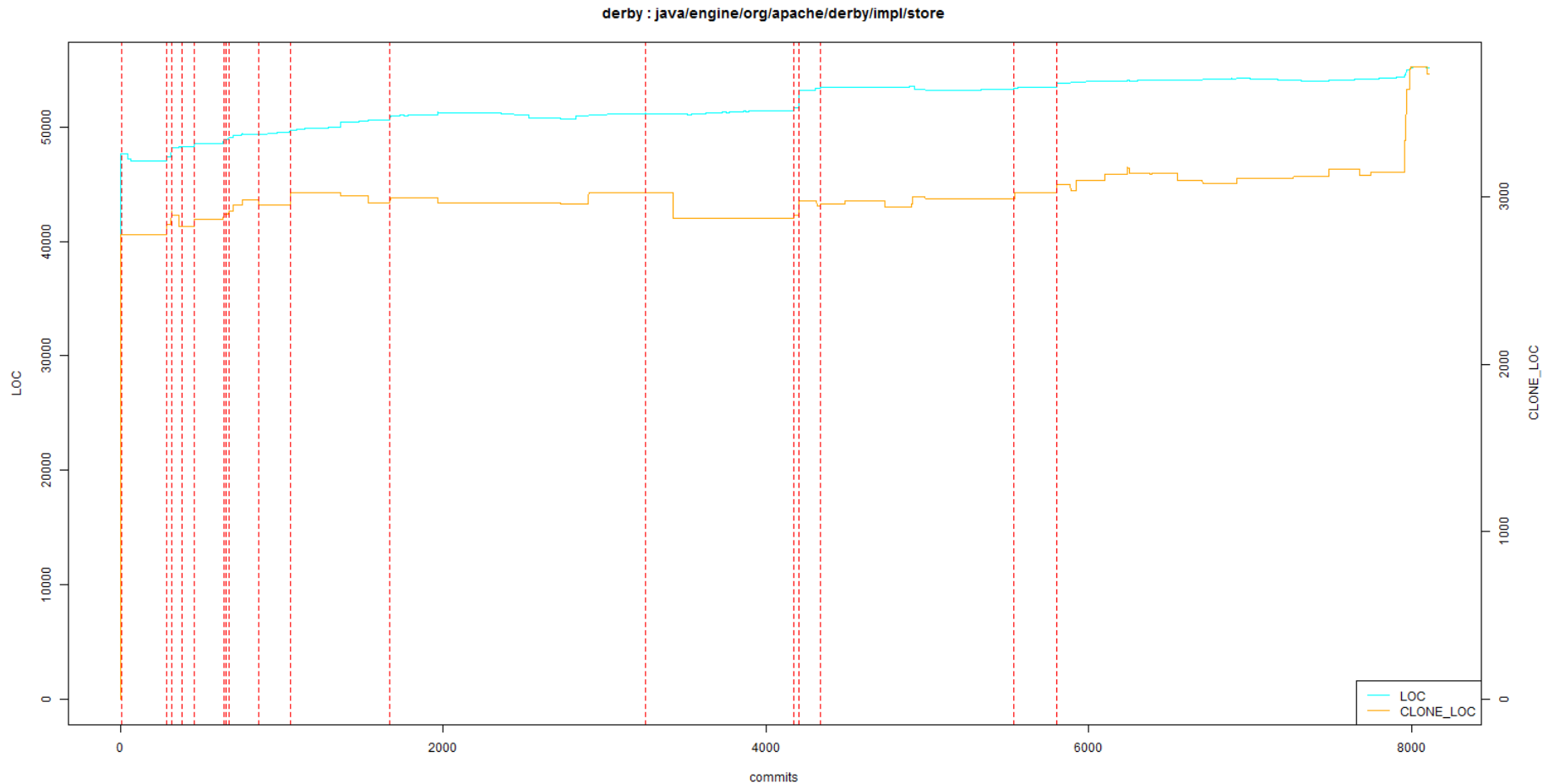
CloneとNonCloneのBug比較

- Derbyプロジェクト
 - java/engine/org/apache/derby/impl/store/以下のJavaファイルを対象

	Clone関係	全体
Issueの数	163	260
No Bug Issue 数	79	111
Bug Issue 数	84	149

	Clone数
Bug Issue	29
No Bug Issue	24
全体	108

Bug Issueの原因となる変更箇所



COMMITTER_NAME	CloneChange Count	BugIssueCloneChange Count
Mike Matrigali	22	10
Katherine Marsden	4	2
Knut Anders Hatlen	9	1
Dag H. Wanvik	6	1
Daniel John Debrunner	2	1
Suresh Thalamati	2	1
Oyvind Bakksjo	1	1
Richard N. Hillegas	6	0
Kristian Waagan	2	0
Mamta Satoor	2	0
Ken Coar	1	0
Samuel Andrew McIntyre	1	0
Satheesh E. Bandaram	1	0
David Van Couvering	1	0
Oystein Grovlen	1	0
Lily Wei	1	0

今後の分析

- 1つ1つのクローンの調査
 - だれが作成・変更し、Bugが埋め込まれ、修正していったのかを時系列で調査する