産学連携による携帯端末用ソフトウェアに対する品質改善の取り組み

吉田 則裕[1,a)]　崔 恩瀞[2,b)]　肥後 芳樹[2,c)]　楠本 真二[2,d)]　井上 克郎[2,e)]

概要：本稿では，著者らの研究グループがある企業と行った産学連携の取り組みについて紹介する．この共同研究では，ある企業が開発している携帯端末向けソフトウェアの品質を向上させるために，残存バグを効率的に検出するシステムを開発した．新しく支援システムを開発し，2 つの携帯端末用ソフトウェアに適用したところ，26 個のバグを検出できた．

An Approach to Quality Improvement on Mobile Software Based on Academic-Industrial Collaboration

NORIHIRO YOSHIDA[1,a)]　EUNJONG CHOI[2,b)]　YOSHIKI HIGO[2,c)]　SHINJI KUSUMOTO[2,d)]　KATSURO INOUE[2,e)]

Abstract: This paper introduces an activity of academic-industrial collaboration. The aim of this collaboration is to improve quality of mobile software systems by removing latent bugs in them. A new system for automatically discovering latent bugs has been developed. The system was applied to two mobile software, so that we discovered 26 unknown bugs.

1. はじめに

ソフトウェアシステムの大規模化・複雑化が原因となり，高品質なソフトウェアを安定して供給することが難しくなっているといわれて久しい．特にスマートフォンなどの携帯端末は，数ヶ月ごとに新機種が発売されている．各メーカーは Android 等の共通プラットフォームに独自の機能を付け加え他社との差別化を図っている．また，共通プラットフォーム上で動作する携帯端末用ソフトウェアも活発に開発されている．しかし，発売のサイクルに合わせて新機種を十分な品質で開発することは難しく，市場からの撤退や共同でソフトウェアを開発する企業が後を立たない．

本論文では，著者らの研究グループがある企業と行った産学連携の取り組みについて紹介する．連携先の企業（以降，企業 A とする）は，さまざまな携帯端末を開発しており，それらは類似の機能を持つため，端末間には多くの共通処理が存在している．これらの共通処理は，ソフトウェアのソースコード中には，類似コード（以降，コードクローン）として出現する．素早く共通の処理を実装するためにコピーアンドペーストによる再利用が頻繁に行われているが，そのあとに変数の修正漏れ等によるバグが多数発生して問題になっている．この産学連携は，そのようなコピーアンドペースト後の修正漏れを原因とするバグを効率的に検出するために実施された．

以降，本論文では，産学連携の取り組みとその結果，および産学連携を行うにあたり得た知見等について述べる．今後産学連携を行う企業や大学の一助となれば幸いである．

2. 産学連携の経緯

企業 A は，携帯端末用ソフトウェアの開発に力を入れている．しかし，開発時には頻繁にコピーアンドペースト等の再利用が行われること，ソフトウェアの規模が大きいこと，および市場に出すまでに期間が短いこと等から，ソースコード中に残存する類似バグの存在に悩まされていた．企業 A の品質保証部に所属する B 氏は，事業部が開発したシステムの品質をチェックする立場にある．B 氏は近年，

[1] 奈良先端科学技術大学院大学
Nara Institute of Science and Technology
[2] 大阪大学
Osaka University
[a)] yoshida@is.naist.jp
[b)] ejchoi@ist.osaka-u.ac.jp
[c)] higo@ist.osaka-u.ac.jp
[d)] kusumoto@ist.osaka-u.ac.jp
[e)] inoue@ist.osaka-u.ac.jp

```
        ......
127:  o_count = v_count;
128:  o_var = varse;
129:  o_names = v_names;
130:
131:  v_count += STORE_INCR;
132:  varse = (char **) malloc (v_count*sizeof(char *));
133:  v_names = (char **) malloc (v_count*sizeof(char *));
134:
135:  for (indx = 3; indx < o_count; indx++)
135:      varse[indx] = o_var[indx];
137:
138:  for (; indx < v_count; indx++)
139:      varse[indx] = NULL;
        ......
161:  o_count = a_count;
162:  o_ary = arrays;
163:  o_names = a_names;
164:
165:  a_count += STORE_INCR;
166:  arrays = (char **) malloc (a_count*sizeof(char *));
167:  a_names = (char **) malloc (a_count*sizeof(char *));
168:
169:  for (indx = 1; indx < o_count; indx++)
170:      varse[indx] = o_ary[indx];
171:
172:  for (; indx < v_count; indx++)
173:      lists[indx] = NULL;
        ......
```

図 1 オープンソースにおけるコードクローン部分の修正漏れの例
Fig. 1 A pair of code clones including wroing a variable reference

特に，携帯端末用ソフトウェアの開発では，テスト工程では取りきれない残存バグが多くなっていると認識していた．B 氏が事業部の開発プロセスと残存バグについて調査を行ったところ，コピーアンドペーストされたコード部分に残存バグが存在する場合があることを確認した．

図 1 はあるオープンソースソフトウェアの重複したコードを表している（企業 A のソースコードではない）．この例では，127 行目 ～139 行目のコードと 161 行目 ～173 行目のコードが重複している．172 行目の変数 v_count は誤った変数の参照であり，これは a_count でなければならない．このようにコピーアンドペースト後の修正ミスによりコードにはバグが入り込んでしまう [3].

B 氏は，著者らの研究グループがコードクローンに関する研究を行なっていることを，Web ページ等を通して知った，B 氏は著者らの研究グループに連絡をとり，それが産学連携を行うきっかけとなった．

以下，産学連携による取り組みの大まかな日程を記す．各項目の “＋ n ヶ月” というのは，その項目を実施したのが，B 氏が著者らに初めて連絡をしてきた何ヶ月後であるのかを表す．

(＋ 1 ヶ月) 企業 A（B 氏ら 2 名）が大阪大学を訪問し，現在直面している問題を著者らに説明した．また，著者らは現在までのコードクローンに関する研究の成果を紹介した．

(＋ 4 ヶ月) 企業 A から共同研究実施の依頼がある．著者らは企業 A の要求に関して著者らが支援できることを提示した．電子メールやテレビ会議システムで意見交換を行った．

(＋ 6 ヶ月) 著者らが所有する既存のツール CCFinder[2] や Gemini[1] 等を企業 A に利用して頂いた．しかし，修正漏れによるバグ検出支援には不十分であり，新しい支援システムを開発することになった．

(＋ 9 ヶ月) 企業 A が組織として大阪大学との共同研究を正式に認可した．

(＋ 10 ヶ月) 著者ら（3 名）が企業 A を訪問した．コードクローン検出に関する最新技術についての講演や，B 氏ら品質保証部のメンバーとの打ち合わせを行い，開発する支援システムについて議論した．

(＋ 12 ヶ月) 企業 A（B 氏ら 2 名）が大阪大学を訪問し，システムの実装について議論した．その後，B 氏らは支援システムを完成させ，携帯端末用ソフトウェアに対して適用した．

(＋ 13 ヶ月) 著者ら（4 名）が企業 B を訪問した．支援システムを用いた成果についての報告会を行った．

3. Clone Inspector

企業 A のソースコードに CCFinder と Gemini を適用したところ，非常に多くのコードクローンが検出された．しかし，修正漏れを含むコードクローンの数は，コードクローン全体の数に比べると少なく，コードクローンを検出するだけでは，修正漏れによるバグ検出支援としては不十分であった．よって，バグを含むと思われるコードクローンのみを出力するツール CloneInspector を開発した．

本節では, 共同研究により開発した支援システム, CloneInspector について説明する．

図 2 に示すように，CloneInspector は入力として下記の 3 つを受け取る．

- バグ検出対象ソースコード
- 最小一致字句数
- 非変化率のしきい値

入力のうち，最小一致字句数は，CloneInspector が検出するコードクローンの最小の大きさ（字句数）である．この大きさ以上のコードクローンを検出し，コードクローン間のユーザ定義名の非変化率を調べる．その値が入力として与えられる非変化率のしきい値以下であれば，そのコードクローン部分に修正漏れのバグがあるとして出力する．

CloneInspector は，コードクローンを検出するために CCFinder を利用している．現在のところ，CloneInspector は C/C++言語および Java 言語に対応している．CloneInspector は単一のワークステーション上で動作するスタンドアロンシステムであり，一千万行程度まで解析可能である．

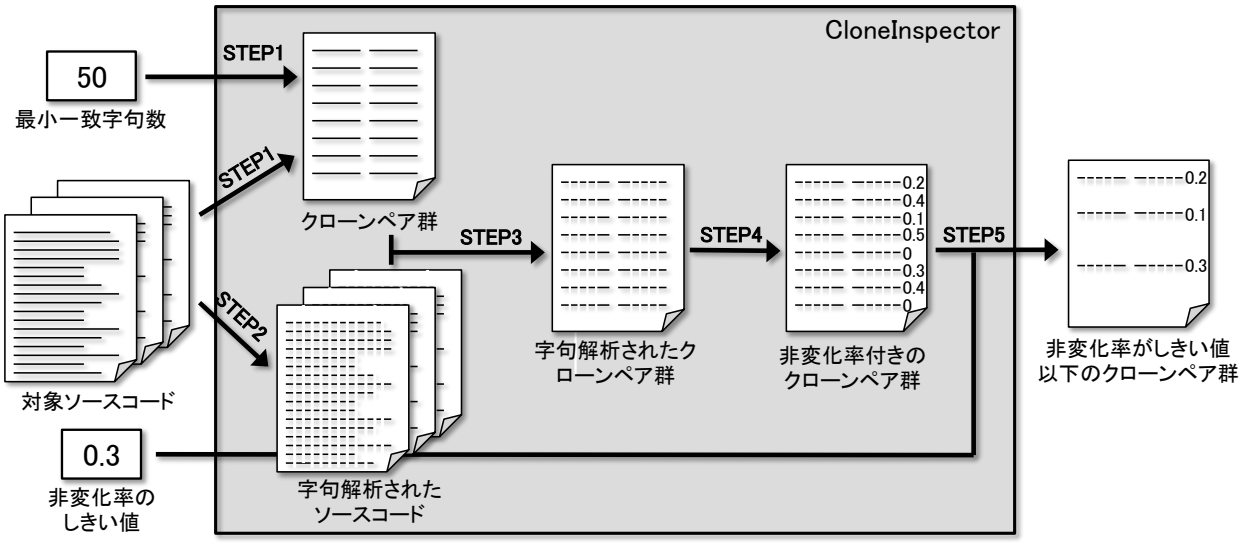

図 2　CloneInspector の処理の流れ
Fig. 2　Overview of CloneInspector

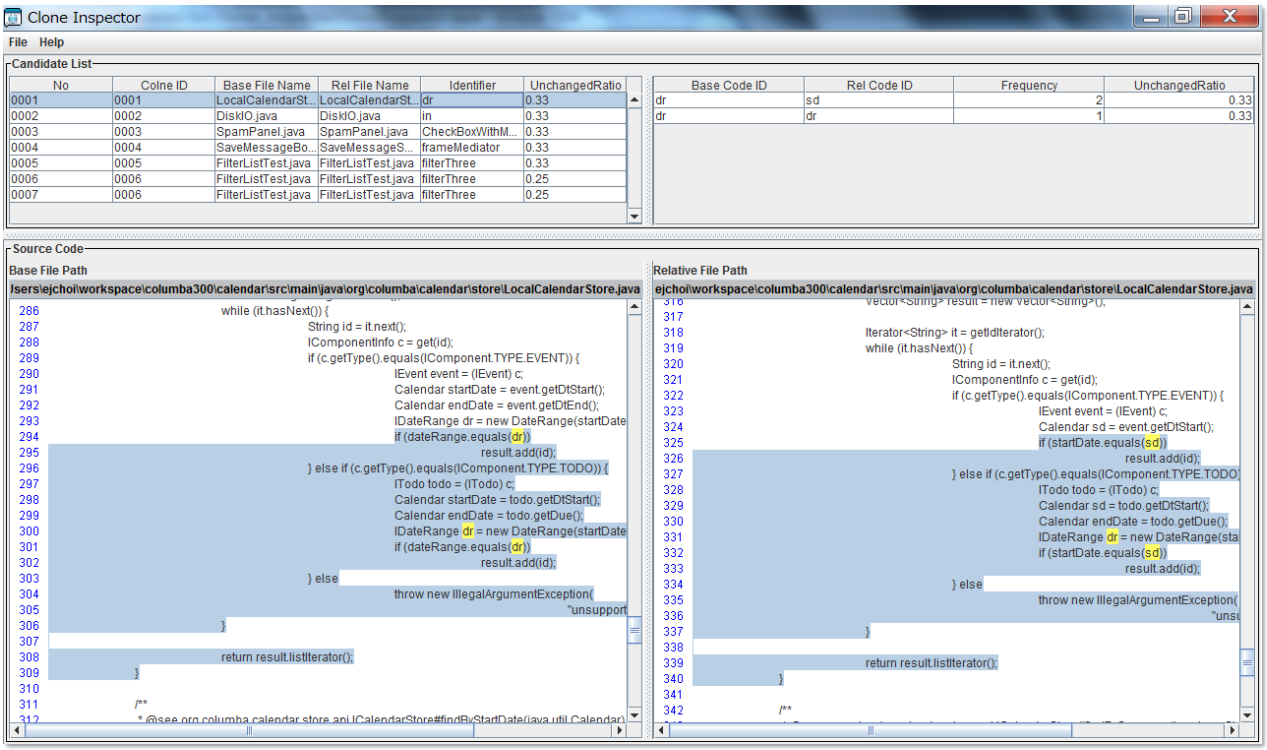

図 3　CloneInspector のスナップショット
Fig. 3　Snapshot of CloneInspector

図 3 は CloneInspector のスナップショットである．左側と右側にはコードクローンになっているソースコードが強調して表示されている．また，画面の上部には，それぞれのコードクローン中で利用されているユーザ定義名についての情報が表示されている．

CloneInspector の各ステップにおける処理を以下に示す．

STEP1　対象ソースファイル群からコードクローンを検出する．コードクローンの検出は CCFinder[2] を用いて行われる．

STEP2　ソースコードの字句解析を行う．全ての字句は，それを含むソースファイルのパス，開始位置（行と列），終了位置（行と列）の情報を保持している．

STEP3　STEP1 で得た情報と STEP2 で得た情報を突き合わせることにより，STEP1 で検出されたコードクローンに含まれている字句列を得る．

STEP4　各クローンペアの字句列を調査（以降，対応付け調査）することにより，その字句列に含まれている変数名の非変化率を計算する．非変化率の計算方法につ

いては，3.1 節で述べる．

STEP5 入力として与えられた非変化率のしきい値と各コードクローンの非変化率を比較して，しきい値以下の値を持つコードクローンのみを出力する．

以降，3.1 節と 3.2 節では，それぞれ STEP4 と STEP5 について詳細に説明する．

3.1 STEP4：非変化率の計算

コードクローンのペアとして検出されたコード片間におけるユーザ定義名の非変化率を算出する．図 4 に表すように，コード片内に出現するユーザ定義名を先頭から順番に対応付けていく．対応付けは，STEP3 で得られたコード片の字句列を用いて行われる．一方のコード片において，1 つの変数が複数箇所に表れているとする（図 4 におけるコード片 1 の v_count 等）．他方のコード片において，その変数と対応する位置に存在している変数が 2 種類以上の場合は，ユーザ定義名の不一致があるといえる．各コード片に出現するすべての変数に対してこの対応付けの調査が行われ，その結果はデータベースに保存される．表 1 は，図 1 のコードクローンに対して対応付け調査を行った結果を表している．この表より，コード片 1 で参照されている v_count と varse がコード片 2 において，複数の変数への参照になっていることがわかる．図 1 の 173 行目における変数 v_count の参照が，変化していない．一方，コード片 1 で 3 回参照されている変数 varse はコード片 2 では，3 つの変数の参照へと変化している．この例では，2 つの変数について不一致が発生していることになる．

この STEP4 では，各変数に対して**非変化率**（*UnchangedRatio*）を算出する．例えば，あるコード片において変数 x が m 箇所において出現しているとする．対応するコード片では，そのうちの n 箇所が他の変数で置換されているとする．このときに，変数 x の非変化率は下記の式で計算される．

$$UnchangedRatio(x) = \frac{m-n}{m} \quad (1)$$

図 1 において，不一致が発生している変数 v_count と varse については，非変化率は下記のように計算される．

$$UnchangedRatio(v\_count) = \frac{5-4}{5} = 0.2$$

$$UnchangedRatio(varse) = \frac{4-3}{4} = 0.25$$

非変化率は 0 以上 1 以下の値をとる．不一致が発生している変数では，非変化率は，0 と 1 以外の値になる．0 ではないが 0 に近いほど，少ない数の変数が変更されずに他のコード片に残っていることを表す．つまり，非変化率が低い値を指す場合は，その変数が修正漏れである可能性を示唆している．

表 1 図 1 のコードクローンに対する対応付け調査の結果
Table 1 Identifier mapping result for code clones in Figure

コード片 1	コード片 2	出現回数
malloc	malloc	2
indx	indx	8
o_count	o_count	2
o_name	o_name	1
o_var	o_ary	2
v_count	a_count	4
	v_count	1
v_name	a_name	2
varse	arrays	2
	lists	1
	varse	1
STORE_INCR	STORE_INCR	1

3.2 STEP5：フィルタリング

各コードクローンのペアについて，下記の両方を条件を満たす変数が存在するかを調べる．存在した場合は，その変数部分が修正漏れとなっているコード片であるとして出力する．

条件 1 非変化率の値が 0 ではなく，しきい値よりも小さい値である

条件 2 条件 1 を満たす変数が，高々 2 つの変数への参照に変化している

2 つ目の条件は，修正漏れの検索精度を挙げるために利用した経験則である．著者らのこれまでのコードクローン関係の研究に関する経験と，B 氏が実際に発見されたバグのソースコードを調査することによりこの条件を導き出した．

4. 携帯端末用ソフトウェアへの適用

CloneInspector を企業 A が開発している携帯端末用ソフトウェアに対して適用した．適用したシステムは 2 つであり，それらの概要を表 2 に示す．また，実験に用いたワークステーションや CloneInspector の設定を以下に示す．

CPU Intel Core i5

メモリサイズ 4GB

最小一致字句数 50

非変化率のしきい値 0.3

ソフトウェアは，約 400 万行の通信系のシステムである．CloneInspector はソフトウェア X から 63 個の修正漏れ候

表 2 実験対象の携帯端末用ソフトウェアの概要
Table 2 Target Mobile Software

	ソフトウェア X	ソフトウェア Y
用途	通信	アプリケーション
開発言語	C 言語	C 言語
行数	4,275,952	136,554

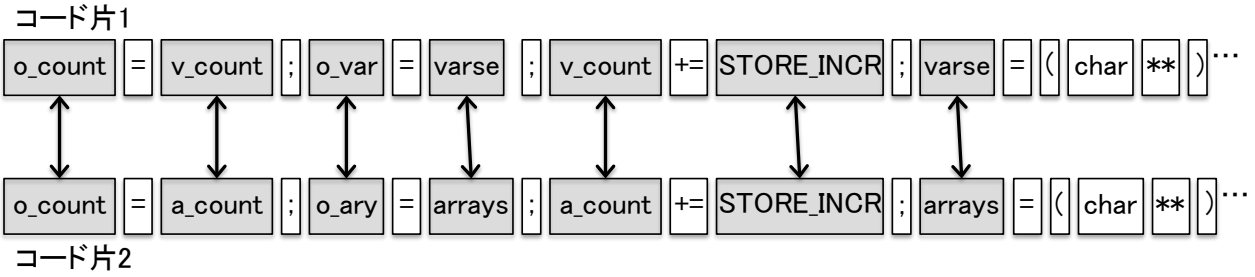

図 4 STEP4 の実行の様子
Fig. 4 Mapping user-defined names in STEP4

補を検出した．それらすべてをソフトウェア X の開発者が検証したところ，そのうちの 25 個が実際に修正漏れであった．同様に，ソフトウェア Y からは 5 個の修正漏れ候補が検出され，そのうちの 1 つが実際に修正漏れであった．今回の適用で検出された全ての修正漏れは，新規に見つかったバグ（開発者がその存在を知らなかったバグ）であり，次のバージョンにおいて全て修正が施された．

CloneInspector が検出した修正漏れ候補の数は，対象ソフトウェアに存在するコードクローンの数に比べると十分に小さく，人がひとつずつチェックできる数になっていることがわかる．CloneInspector がなく，CCFinder と Gemini を用いた場合には，大量のコードクローンの中から修正漏れを見つける必要があるが，数があまりに多いため，人手で行うことは現実的ではない．

この適用結果に企業 A は概ね満足していた．それは主に以下の理由による．

- CloneInspector が提示する修正漏れの候補数が多くなく，人手による確認作業が困難ではない．
- 大規模システムからでも高速に処理を行えた．

また，非変化率のしきい値を 0.4 にして修正漏れの候補を検出した．その結果として，CloneInspector はソフトウェア X から 193 個，ソフトウェア Y から 7 個の候補を検出した．しかし，非変化率のしきい値として 0.3 を用いた場合に検出できた実際の修正漏れ以外には，修正漏れは含まれてはいなかった．このことから，非変化率のしきい値としては，0.3 が適切な値と判断した．

表 3 実験対象に対する CloneInspector の適用結果
Table 3 Application Results

	ソフトウェア X	ソフトウェア Y
検出クローンセット数	38,192	4,053
修正漏れ候補数	63	5
実際の修正漏れ数	25	1
実行時間（CCFinder）	300 秒	40 秒
実行時間（CCFinder 以外）	143 秒	4 秒

5. 今回の産学連携の特徴，得た知見

この産学連携の特徴として，大学側は主に助言を行い，企業側が主にツール開発を行ったことが挙げられる．この特徴は，今回の産学連携を迅速に進めることができた要因になったと著者らは考えている．

上記の方式は，大学が持つ知識を企業に伝えることを目的として行われた．結果として，企業 A は今回の共同研究に必要な知識だけでなく，その背景となるコードクローンに関する知識も吸収することができた．また，自身らでツールを開発したために，ユーザインターフェース等，企業 A で利用しやすい形でツールを作成できた．大学側がツールを開発した場合は，その適用を企業が行うことが難しく，大学が企業に赴いて適用しなければならないことがある．しかし，このような連携は適用のオーバーヘットが大きい．

また，今回の産学連携は，契約時に具体的な目標が定まっていたことが迅速な成果につながった．契約時には広めの目標を設定し，産学連携を推し進めていく上で具体的な内容に掘り下げていく場合に比べて，早く成果を挙げることができた．しかし，今回の産学連携の形は，当初に定めた目標が，現在抱えている問題を真に捉えきれていないかもしれないという危険も含んでいる．

大阪大学と企業 A は地理的には大きく離れている．そのため，極力テレビ会議等を用いて打ち合わせを行った．

今回の産学連携では，問題が発生しているソースコードを著者らが閲覧することはできなかった．そのため，企業 A が所有するソースコードの特徴，および抱えている問題点を伝聞でしか理解できなかった．著者らが直接ソースコードを閲覧することができれば，その特徴をより詳しく把握できることができるため，問題を含むコード片を絞り込むためのよりよい条件を設定できたかもしれない．

6. おわりに

本稿では，著者らの研究グループとある企業の産学連携の取り組みについて紹介した．この産学連携では，ある企

業が開発している携帯端末用ソフトウェアに残存するバグの効率的な検出を目的として行われた．新しいシステム CloneInspector を開発し，2 つの携帯端末用ソフトウェアに対して適用したところ，合計で 26 個のバグを検出することができた．現在でも CloneInspector はその企業で利用されている．この産学連携は現在でも続いる．

参考文献

[1] 肥後芳樹，吉田則裕，楠本真二，井上克郎：産学連携に基づいたコードクローン可視化手法の改 良と実装，情報処理学会論文誌， Vol. 48, No. 2, pp. 811–822 (2007).

[2] Kamiya, T., Kusumoto, S. and Inoue, K.: CCFinder: A Multilinguistic Token-Based Code C lone Detection System for Large Scale Source Code, *IEEE Transactions on Software Engineering*, Vol. 28, No. 7, pp. 654–670 (2002).

[3] Li, Z., Lu, S., Myagmar, S. and uan Zhou, Y.: CP-Miner: Finding Copy-Paste and Related Bugs in Large-Scale Software Code, *IEEE Transactions on Software Engineering*, Vol. 32, No. 3, pp. 176–192 (2006).