Investigating Context Adaptation Bugs in Code Clones

Manishankar Mondal, Banani Roy
Chanchal K. Roy, Kevin A. Schneider

ICSME 2019 Proceedings of the 35th
IEEE International Conference on Software
Maintenance and Evolution
Pages 157-168

発表者：野口 耕二朗

背景

- コードクローンにバグが混入すると保守性の低下につながる
- バグの混入のしかたを明らかにし，効果的の混入防止につなげたい

コードクローンに関連するバグ

- バグ混入クローンからバグが混入したまま新たなクローン片が生成される場合[1]
  - バグ混入クローンの33%からバグが拡散

- バグが混入していないコード片を複製する際に，複製先で適切な変更を加えない場合
  - ➢本研究でContext Adaptation Bug(Context Bug)と呼称し，調査対象とする

[1] M. Mondal, C. K. Roy, K. A. Schneider, “Bug Propagation through Code Cloning: An Empirical Study”, Proc. *ICSME*, 2017, pp. 227 – 237.

目的

- Context Bugがどのような性質を持っているかを明らかにする
- Context Bugの特徴がコードクローンへのバグ混入防止に役立つかを明らかにする

Context Bugの例

Clone Fragment 2 (CF2) in Revision = 151			Clone Fragment 2 (CF2) in Revision = 152	
1	**private Object resolveObject(Object o) throws NamingException** {	Bug-fix change	**1**	**private Object resolveObject(Object o, Name name) throws NamingException** {
2	try { if (o instanceof IIOPRemoteReference) {		2	try { if (o instanceof IIOPRemoteReference) {
3	Reference objRef = ((IIOPRemoteReference)o).getReference();		3	Reference objRef = ((IIOPRemoteReference)o).getReference();
4	ObjectFactory objFact = (ObjectFactory)(Class.forName (objRef.getFactoryClassName())).newInstance();		4	ObjectFactory objFact = (ObjectFactory)(Class.forName (objRef.getFactoryClassName())).newInstance();
5	**return objFact.getObjectInstance(objRef,null,null,null);** }	Bug-fix change	**5**	**return objFact.getObjectInstance (objRef, name, this, iiopContext.getEnvironment());** }
6	else if (o instanceof IIOPRemoteResource) {		6	else if (o instanceof IIOPRemoteResource) {
7	return ((IIOPRemoteResource)o).getResource(); }		7	return ((IIOPRemoteResource)o).getResource(); }
8	else {return o;}}		8	else {return o;}}
9	catch(Exception e) {throw new NamingException("" + e); }		9	catch(Exception e) {throw new NamingException("" + e); }

Clone Fragment 1 (CF1) in Revision = 151		Clone Fragment 1 (CF1) was not changed in Revision = 152
1	private Object resolveObject(Object o) {	
2	try {if (o instanceof IIOPRemoteReference) {	**File path of Clone Fragment 1:** carol/src/org/objectweb/carol/jndi/iiop/IIOPContextWrapper.java
3	Reference objRef = ((IIOPRemoteReference)o).getReference();	
4	ObjectFactory objFact = (ObjectFactory)(Class.forName (objRef.getFactoryClassName())).newInstance();	**File path of Clone Fragment 2:** carol/src/org/objectweb/carol/jndi/iiop/IIOPReferenceContextWrapper.java
5	return (Referenceable)objFact.getObjectInstance(objRef,null, null,null);}	
6	else if (o instanceof IIOPRemoteResource) {	Clone Fragment 2 experienced context bug-fix as demonstrated above. However, Clone Fragment 1 was not buggy in its own context, and thus, it did not require the bug-fix changes.
7	return ((IIOPRemoteResource)o).getResource(); }	
8	else {return o;}}	
9	catch (Exception e) {TraceCarol.error ("IIOPContextWrapper.resolveObject()", e); return o; }	

Fig. 2: This figure shows an example of context bug-fix in a clone fragment denoted as Clone Fragment 2 (CF2) from our subject system Carol. The bug-fix commit was applied on revision 151. We can see the snapshots of CF2 both in revision 151 and 152. The changes (in Line 1 and Line 5) have been highlighted. The commit message says "*Bug 283 correction: IIOP Reference binding problem*". The figure contains another clone fragment called Clone Fragment 1 (CF1) in revision 151. Both CF1 and CF2 were created in revision 140 and they make a Type 3 clone-pair. None of these fragments was changed until revision 151 where CF2 experienced the bug-fix change as we have just described. CF1 never experienced a bug-fix. CF1 and CF2 follow the second bug-fix pattern described in Section IV-B. If we carefully look at CF1 and CF2 and the bug-fix changes experienced by CF2 we realize that the bug that was fixed in CF2 is a context-bug according to our definition. If we look at the lines 1 and 5 of CF1 and CF2 in revision 151, we see that the corresponding lines of the clone fragments are similar except that line 1 in CF2 throws an exception which is absent in line 1 of CF1, and line 5 of CF1 contains a type-casting (i.e., *(Referenceable)*) which is not present in line 5 of CF2. However, the bug-fix changes that occurred in CF2 indicate that CF2 was not properly adapted to its context at the time of its creation (possibly through copy/pasting from CF1) and the bug, *Bug 283*, was introduced to it. The bug-fix commit adapted it by adding an extra parameter called *name* of type *Name* to it and making corresponding changes in its fifth line.

Context Bugの検出

1. CTAGS[2]を用いてメソッドを検出
2. NiCad[3]を用いてクローンを検出
3. メソッドのクローンの系譜を抽出
4. バグ修正を含むクローンを検出
5. バグ修正されたクローンのうち，Context Bugを含むものを抽出

[2] CTAGS: http://ctags.sourceforge.net/
[3] C. K. Roy, J. R. Cordy, “NICAD: Accurate Detection of Near-Miss Intentional Clones Using Flexible Pretty-Printing and Code Normaliza- tion”, Proc. ICPC, 2008, pp. 172 – 181.

Context Bugsのパターン1

- 過去のリビジョンで生成されたコード片を後のリビジョンで複製

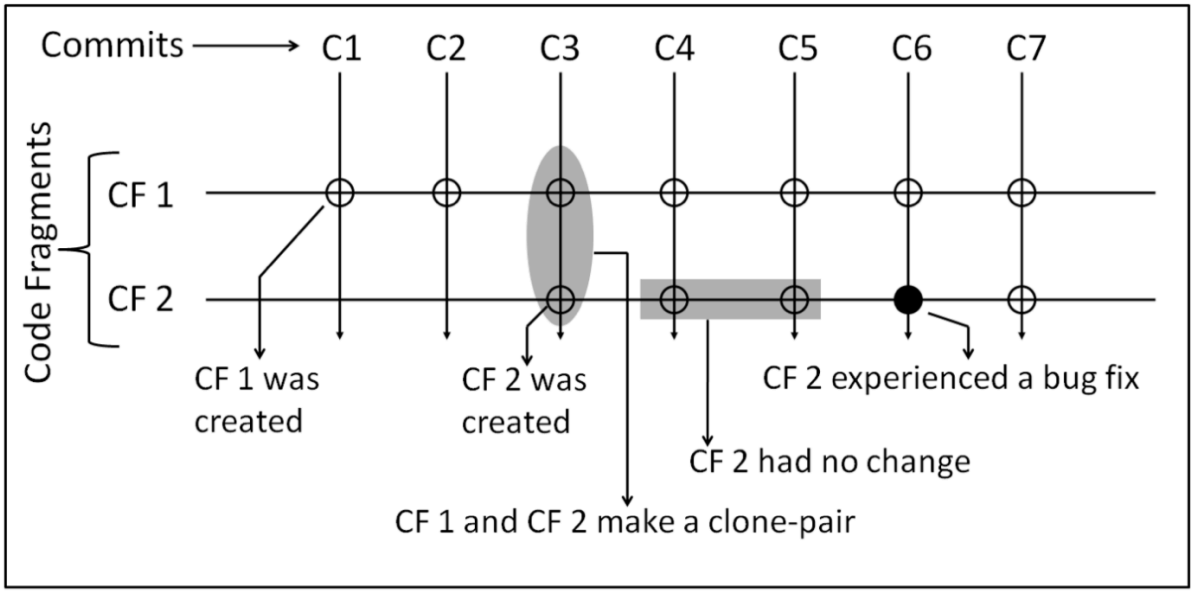

Fig. 3: Context bug-fix Pattern 1

Context Bugsのパターン2

- 同じリビジョンコード片を複製

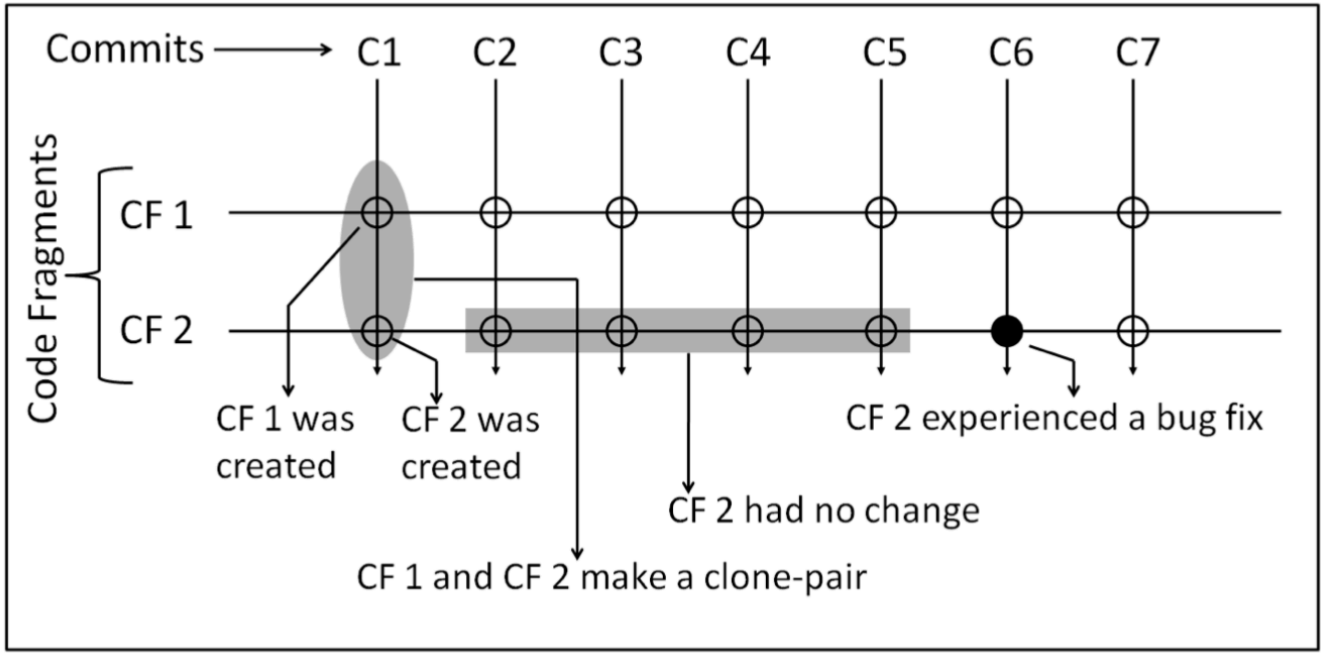

Fig. 4: Context bug-fix Pattern 2

リサーチクエスチョン

1. クローンに関連するバグのうちContext Bugはどれほどの割合であるか

2. どのような流れでContext Bugは発生するか

3. どのクローンタイプがContext Bugを引き起こしやすいか

4. Context Bugはファイル間をまたいだ方が混入しやすいか

データセット

- SourceForgeからJava, C, C#のSVNリポジトリをダウンロード

TABLE III. Investigated subject systems

Systems	Lang.	Domains	LLR	SRev	LRev
Ctags	C	Code Definition Generator	33,270	1	774
Carol	Java	Game	25,091	1	1700
Freecol	Java	Game	91,626	1000	1950
jEdit	Java	Text Editor	191,804	3791	4000
Jabref	Java	Reference Management	45,515	1	1545
MonoOSC	C#	Formats and protocols	14,883	1	355

LLR = LOC in the Last Revision
SRev = Starting Revision LRev = Last Revision

Context Bugの検出結果

TABLE II. Number of context bug-fixes in different types of code clones

	Ctags			Carol			Freecol			jEdit			Jabref			MonoOSC		
	T1	T2	T3	T1	T2	T3	T1	T2	T3	T1	T2	T3	T1	T2	T3	T1	T2	T3
Count of bug-fixes in clones	2	5	21	9	9	31	11	13	57	19	9	44	10	8	19	2	0	7
Count of context bug-fixes	1	4	8	2	1	13	6	5	29	16	5	32	5	2	5	1	0	2
Count of context bug fixes (Pattern 1)	0	0	5	1	0	3	0	0	5	4	1	2	0	0	1	0	0	1
Count of context bug fixes (Pattern 2)	1	4	3	1	1	10	6	5	24	12	4	30	5	2	4	1	0	1

T1 = Type 1 or exact clones T2 = Type 2 or syntactic clones T3 = Type 3 or gapped clones

RQ1:クローンに関連するバグのうちContext Bugはどれほどの割合であるか

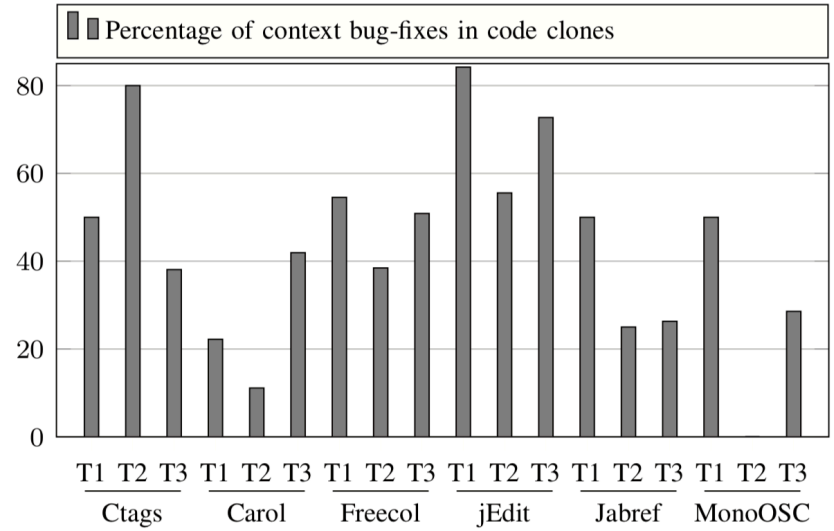

Fig. 5: Percentages of context bug-fixes in different types of code clones of different subject systems

- 50%のバグがContext Bug, 場合によっては87%になる
- Context Bugの発生を防ぐことでクローンに関連する大部分のバグを防ぐことができる

RQ2:どのような流れでContext Bugは発生するか

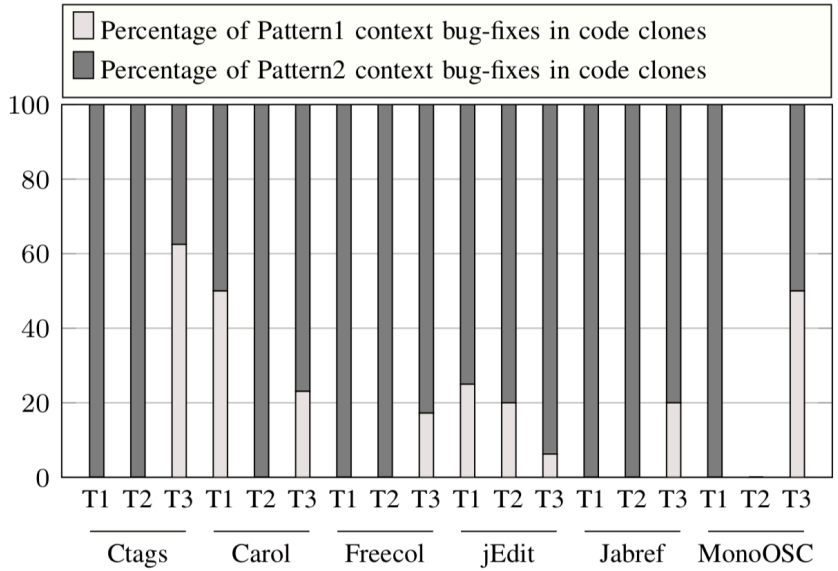

Fig. 6: Percentages of context bug-fixes of two patterns in different types of code clones of different subject systems

- パターン2の方が多い，割合は83.21%
- 新たに作成したコード片を複製する際はContext Bugの混入に気をつける必要がある

RQ3:どのクローンタイプがContext Bugを引き起こしやすいか

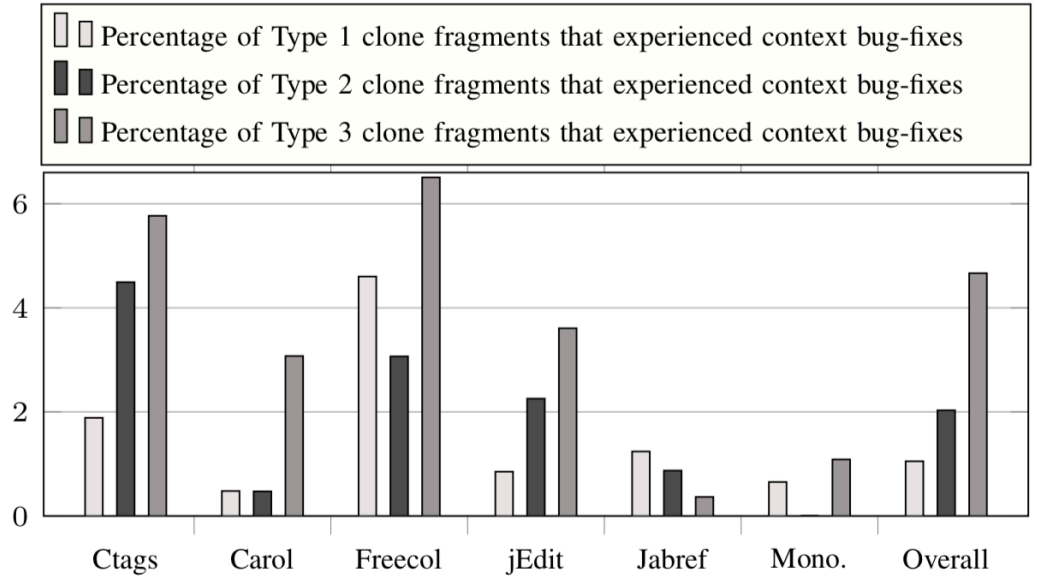

Fig. 7: Percentages of different types of clone fragments that experienced context bug-fixes

- タイプ3のクローンが最もContext Bugを引き起こしやすい
- プロジェクトによっては異なる場合もある

RQ4: Context Bugはファイル間をまたいだ方が混入しやすいか

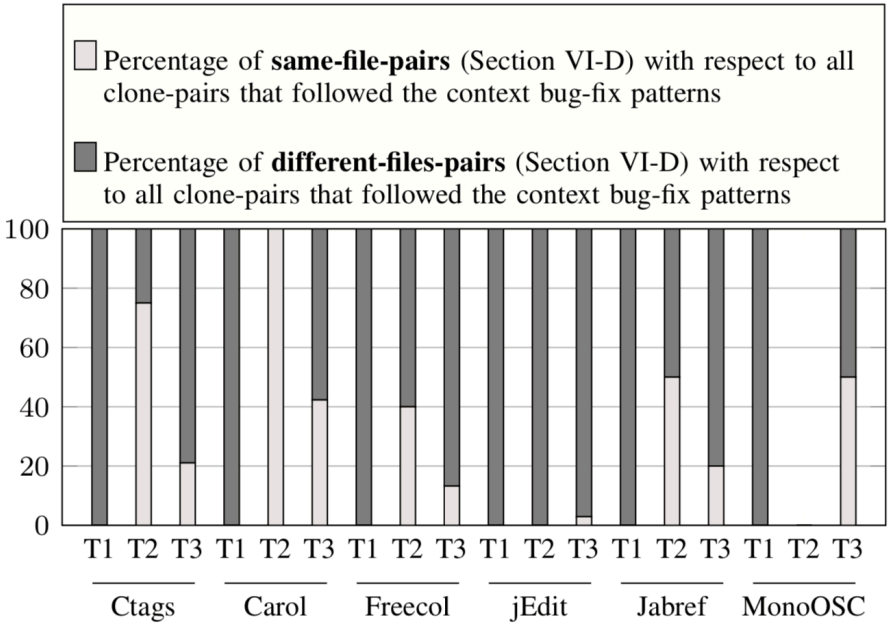

Fig. 8: Percentage of context bug-fix clone pairs containing clone fragments from the same or different files

- ファイル間をまたいでコード片を複製した方が混入しやすい，割合は89.92%

まとめ

- Context Bugはクローンに関連するバグの中で50％を占める
- 新たに生成されたコード片を複製する際はContext Bugの混入に注意が必要
- ファイルをまたいでコードを複製する際もContext Bugの混入に注意が必要

所感

- クローンへのバグ混入防止に有効な知見が得られたと思った
- クローンに関連するバグが少ないためもっとデータセットを増やして同じ分析をしてみたい